С. Бобровский

Delphi 7

ББК 32.973.2-018я7
УДК 681.322.06(075)
Б72

Б72 **Delphi 7. Учебный курс** / С. И. Бобровский. — СПб.: Питер, 2004. — 736 с.: ил.

ISBN 5-8046-0086-9

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.

ББК32.973.2-018я7
УДК 681.322.06(075)

ISBN 5-8046-0086-9

Краткое содержание

Содержание

Введение

От автора

О стремительном темпе развития информационных технологий мы с вами читаем и слышим почти каждый день. Эти темпы действительно впечатляют. Однако в компьютерном мире остается одна самая важная область, изменения в которой происходят крайне медленно. Программирование, кодирование, составление исходных текстов — ключевой элемент в создании любого приложения сегодня происходит так же, как и сорок лет назад. Разработчик применяет крайне ограниченный набор логических конструкций (условный оператор и операторы присваивания и цикла) *и* небольшое число стандартных типов данных. Причем такой подход ничуть не изменился, хотя сменилось уже не одно поколение языков программирования. Например, на смену Си и Паскалю пришла *Java,* однако мои коллеги и знакомые в различных компьютерных фирмах по-прежнему запускают в консольной сессии *java*-компилятор командной строки и отлаживают сложные программы, вручную просматривая протоколы работы и игнорируя удобные и комфортные средства быстрой визуальной разработки и отладки типа *JBuilder* или *Delphi.* Это происходит не в 70-х годах прошлого века, а в середине 2002-го года!

Компьютерные издания, претендующие на звание профессиональных, нередке пропагандируют подобный полухакерский подход к созданию программ. Создается своеобразный имидж программиста-одиночки, способного за пару бессонных ночей написать нужную заказчику программу, которая героически уместится в ста килобайтах памяти. Программированию вообще присущ значительный консерватизм, так как в принципе можно создавать программы, ограничиваясь знаниями многолетней давности. Однако сегодня программирование безусловно превратилось из искусства в ремесло. Конечно, вряд ли можно стать профессиональным разработчиком, не изучив внутреннее устройство *Windows* или структуру компонентов *VCL* и принципы оптимизации программ. Однако такие знания сегодня отходят на второй или третий план. Работодателей интересует прежде всего скорость и качество создания программ в коллективе, а эти характеристики может обеспечить только среда визуального проектирования, способная взять на себя значительные объемы рутинной работы по подготовке приложений, а также согласовывать деятельность группы постановщиков, кодировщиков, тестеров и технических писателей.

Возможности *Delphi* полностью отвечают подобным требованиям и подходят для создания систем любой сложности. Система *Delphi* позволяет писать как крохотные

программы и утилиты для персонального использования, так и корпоративные системы, работающие с базами данных на разных платформах, Интернет-решения и коммерческие игры, распределенные *COM/CORBA/SOAP*-приложения и всевозможные *Web-службы.* При этом обеспечивается совместимость приложений при выходе новых версий *Delphi* — как друг с другом на уровне исходных текстов, так и с модифицированными версиями стандартных протоколов и технологий благодаря библиотеке независимых и легко настраиваемых компонентов.

В ходе совершенствования *Delphi* корпорация *Borland* внимательно анализирует мировые тенденции развития информационных технологий, добавлял в среду только те, которые действительно могут стать ключевыми в компьютерном бизнесе. Так произошло, в частности, с *Web*-сервисами *(Web Services)* — технологией дистанционного предоставления компьютерных услуг через Интернет-протоколы, которая была поддержана в шестой версии *Delphi,* а сегодня встраивается в свои продукты практически всеми ведушими корпорациями мира *(IBM, Microsoft, Oracle* и т. д.). Набор компонентов dbExpress позволил создавать системы, не зависящие от конкретного сервера баз данных и поэтому легко масштабирующиеся для задач произвольного объема. А новые средства *IntraWeb* теперь позволяют «нарисовать» мышкой Интернет-систему высокой сложности, вообще не набирая вручную никаких исходных текстов.

Очень важным, одним из ключевых нововведений *Delphi7* стало появление системы визуального моделирования *ModelMart.* До сих пор подобные системы, позволяющие описывать структуру, возможности и последовательность функционирования программной системы в виде набора графических диаграмм (как правило, на универсальном языке моделирования *UML*), выпускались без привязки к конкретной системе программирования. А система *ModelMart* интегрирована с *Delphi* и Паскалем и делает доступной всю мощь методологий моделирования не только постановщикам и системным аналитикам, но и программистам. Появление *ModelMart* отвечает самым передовым тенденциям развития методологий проектирования программного обеспечения.

В версии *Delphi 7 Studio Architect* имеется также набор компонентов *Bind,* дополняющий среду *ModelMart,* Этот во многом уникальный набор поддерживает популярную концепцию рефакторинга, активно применяемую при проектировании больших и сверхбольших приложений. Рефакторинг подразумевает создание многоуровневых систем, когда серверы баз данных и приложений отделены от клиентских мест и могут работать на разных платформах. При этом расширение функциональных возможностей системы возможно без внесения изменений в ее исходные тексты. Достаточно лишь откорректировать или перенастроить логику работы, описанную в виде независимых от операционной системы сценариев в промежуточных компонентах системы, причем делать это можно, даже не прерывая эксплуатацию системы.

В ближайшей перспективе *Borland* готовит новую систему для грядущей универсальной платформы *Microsoft .Net.* Она объединит в себе функциональность *Delphi* и *C++Builder* и библиотеки компонентов *VCL,* но станет поддерживать несколько языков программирования. В дополнению к объектному Паскалю (его название

было изменено в *Delphi 7* на «язык программирования *Delphi*») новая *.Net*-оболочка *Borland* позволит создавать программы на языках Си++, C# («Си шарп», язык, изобретенный *Microsoft* в противовес *Java*) и *VisualBasic*. Поэтому изучение *Delphi* позволит в дальнейшем без проблем осваивать новые платформы и операционные системы и переносить на них существующие приложения без существенных модификаций исходных текстов.

Основные понятия и принципы системы Delphi 7

Основные понятия и принципы программирования

Компьютерная *программа* — это набор элементарных команд процессора, представленных в файле в виде последовательности байтов (*машинный код*). Каждая команда может быть закодирована одним или несколькими байтами. Программы в таком виде можно составлять вручную, но подобная работа человеку просто не под силу из-за неудобства управления процессором напрямую с помощью простых команд. Поэтому программа пишется на одном из *языков программирования*, как обычный текст. Этот текст называется ***исходным*** *текстом* (или *исходным кодом*) программы.

Команды языка программирования понятны и наглядны. Например, следующий условный текст представляет собой небольшую программу сложения двух чисел:

```
взять значения, введенные пользователем в поля A1 и A2;
вычислить сумму этих значений;
поместить результат в поле A3.
```

Полужирным шрифтом выделены названия команд. Основные, наиболее часто используемые команды языка программирования называются *операторами* и обычно записываются с помощью специально предназначенных для этого символов или *ключевых слов*. Например, занесение результата вычисления выражения в некоторую ячейку памяти компьютера обычно производится с помощью *оператора присваивания* соответствующего значения.

Действия, выполняемые над имеющимися в программе значениями, называются *операциями*. Они отображаются в тексте с помощью специальных символов. Комбинации данных и операций над ними называются *выражениями*.

Алгоритмы и программы

Перед тем как начать составлять программу, надо предварительно придумать и продумать (лучше всего, записать на бумаге) *алгоритм* ее работы, представляющий собой строгое, формальное, не допускающее неоднозначностей и двусмысленностей

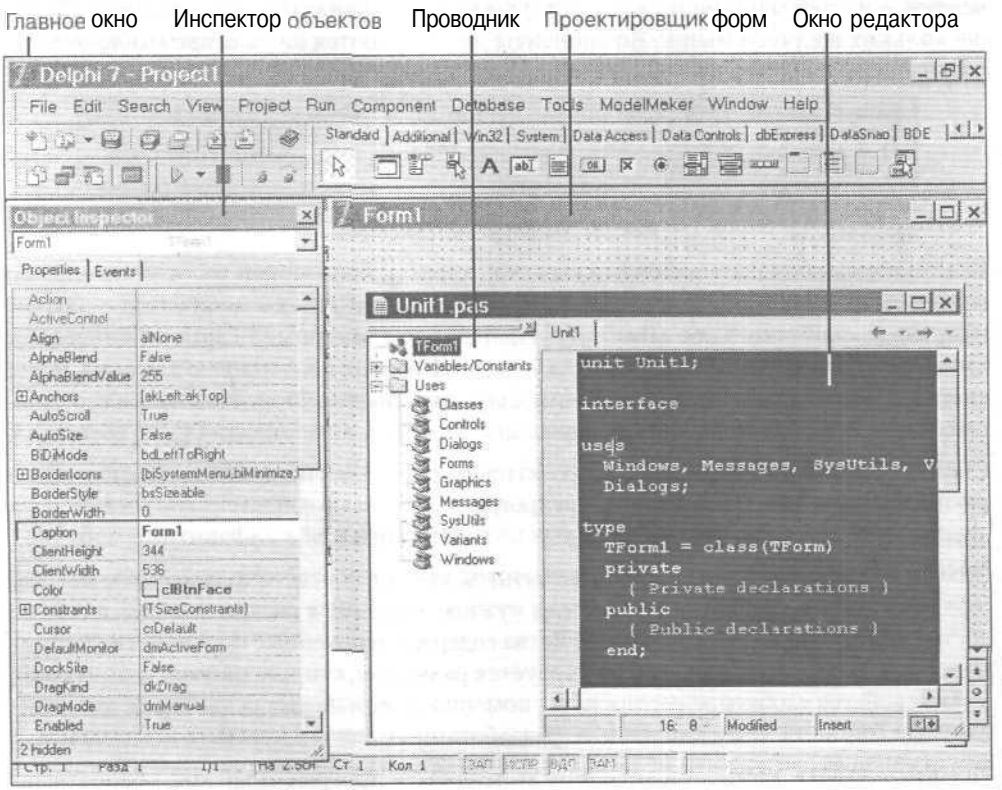
описание процесса решения задачи. После того как алгоритм готов, на его основе и составляется (*кодируется*) программа.

ВНИМАНИЕ Процесс создания алгоритма — самый важный. Если здесь допущены ошибки, то устранить их на этапе кодирования весьма трудно.

Исходный текст программы автоматически переводится в набор инструкций процессора с помощью специальной программы, называемой *компилятором*. В среде *Delphi 7* для этого достаточно выполнить всего одну команду или нажать одну клавишу. Процесс *компиляции* — перевода *(трансляции)* исходного текста в конкретные команды процессора выполняется очень быстро. За секунду программа-компилятор анализирует и транслирует тысячи строк исходного кода.

Среда Delphi 7 и ее составляющие

Среда *Delphi 7* представляет собой интегрированную оболочку разработчика, в которую входит набор специализированных программ, ответственных за разные этапы создания готового приложения (рис. 1),



РМС. 1. *Основные окна системы Delphi 7*

Исходный текст программы готовится в среде *Delphi 7* с помощью встроенного *редактора исходных текстов*. Этот редактор специализирован. Он отличается гибкими возможностями цветового выделения различных элементов текста программы (ключевых слов, названий, операций, чисел и строк) и предоставляет возможность быстрого ввода часто встречающихся конструкций.

Левая панель редактора представляет собой Проводник, позволяющий быстро перемещаться между частями исходного текста и по структуре создаваемой программы.

Важнейшая характеристика разрабатываемой программы — удобство ее пользовательского интерфейса, наличие и доступность необходимых элементов управления. В системе *Delphi 7* имеется специальный *проектировщик форм,* с помощью которого окна будущей программы подготавливаются в виде *форм*. Проектировщик позволяет подобрать оптимальные размеры окон, разместить и настроить всевозможные элементы управления и меню, добавить готовые изображения, указать заголовки, подсказки, подписи и так далее.

Компонентный подход

На этапе проектирования форм программа как бы составляется из готовых *компонентов* — частей машинного кода, которые можно добавлять к ней с помощью всего нескольких щелчков мыши. Компоненты располагаются на *палитре компонентов*, разделенной на несколько самостоятельных панелей (рис. 2).

***Рис.** 2. Палитра компонентов*

Компоненты обладают наборами *свойств,* характеризующими их отличительные особенности. Некоторые свойства имеются практически у всех компонентов — таково, например, свойство Name (Имя). Другие свойства, например Caption (Заголовок), имеются у большинства компонентов — ведь заголовок необходим и для окна, и для кнопки. Некоторые свойства уникальны для конкретных компонентов, например свойство SimpleText (Простой текст), содержащее текст для компонента Строка состояния.

Свойства компонентов в процессе проектирования формы настраиваются с помощью Инспектора объектов. Это специальная программа, показывающая список всех свойств данного компонента, отсортированных по категориям или в алфавитном порядке.

Значение любого свойства можно изменить, введя в соответствующее поле Инспектора объектов новую строку или выбрав нужное значение в раскрывающемся списке доступных значений. Некоторые свойства содержат вложенные подсвойства например, свойство Font (Шрифт) характеризуется размером, стилем, цветом, гарнитурой. Такие свойства удобнее редактировать с помощью специальных редакторов, как стандартных (шрифт), так и специально разработанных для конкретного компонента.

Помимо свойств, компоненты содержат *методы* — программный код, обрабатывающий значения свойств (например, устанавливающий переключатель в нужное

положение), а также *события* — сообщения, которые компонент принимает от приложения, если во время работы программы выполняется определенное действие (например, изменяется состояние флажка). Программист может самостоятельно формировать реакции программы на любые события каждого компонента.

Правильно подбирая компоненты и настраивая их совместную работу путем использования свойств, предназначенных для связи компонентов друг с другом, нередко удается создать приложение, не написав вручную ни строчки исходного текста. В системе *Delphi 7* существуют сотни готовых компонентов, и при решении многих задач бывает полезно предварительно поискать нужный компонент (например, в Интернете), вместо того чтобы выполнять работу по программированию, возможно, уже сделанную другими людьми. Компонентный подход к созданию программ позволяет *повторно использовать* готовые разработки и во многих случаях значительно повышает эффективность труда.

ЗАМЕЧАНИЕ С помощью системы Delphi 7 можно создавать не только обычные программы (EXE-файлы), но и динамически подключаемые библиотеки DLL [своеобразные хранилища кода и ресурсов), новые элементы управления, а также компоненты, отвечающие требованиям различных стандартов но компонентные технологии (COM, ActiveX, CORBA и так далее).

Потребность в ручном программировании возникает, только когда обойтись готовыми компонентами не удается. В вышеприведенном примере поля ввода A1, A2 и поле для вывода итогового результата A3 можно представить в виде стандартных компонентов *Delphi 7,* но чтобы выполнить сложение введенных чисел, необходим соответствующий оператор в тексте программы, срабатывающий, например, по щелчку на компоненте-кнопке.

Отличия системы Delphi 7 от предыдущей версии

Эта глава предназначена для тех, кто знаком с предыдущими версиями системы Delphi. Те, кто впервые приступают к знакомству с этой системой, могут ее пропустить.

Итак, что нового появилось в системе *Delphi 7?*

- Значительно изменена оболочка *Delphi.* Улучшен внешний вид меню, добавлены дополнительные средства настройки цветов выводимых сообщений, редактора кода (помимо Паскаля поддерживается синтаксис Си++, C#, *HTML* и *XML*)и быстрого завершения вводимых конструкций для Паскаля и *HTML.*

- Можно создавать приложения, внешний вид которых соответствует нормам *Windows 9x* и *WindowsXP*.
- Богатый набор компонентов *IntraWeb* дает возможность визуального создания Интернет-приложений любой сложности. Ручное кодирование при этом практически не требуется.
- Создание модулей для *Web*-серверов возможно для сервера *Apache* версий 1.X и 2.x.
- Библиотека импорта интерфейсов теперь поддерживает требования *Microsoft .Net*.
- Новый генератор отчетов *Rave* содержит гибкую среду формирования статических и динамических отчетов и множество компонентов для настройки внешнего вида отчета.
- Внесены новые функции и улучшен ряд существующих подпрограмм в стандартных модулях SysUtils, StrUtils, StdConvs, Hath. Дополнена поддержка данных типа Variant.
- Настройки компилятора позволяют задавать вывод конкретных классов сообщений. Кроме того, поддерживаются сообщения о небезопасных (*unsafe*) преобразованиях типов и данных, которые надо проверять при переносе программы в среду *Microsoft .Net*.
- Система *ModelMaker* позволяет создавать *UML*-диаграммы и автоматически генерировать па их основе программный код *Delphi* и, наоборот, строить диаграммы на базе исходных текстов существующего *Delphi*-приложения.

Это список лишь наиболее важных отличий новой версии системы *Delphi 7*. Многочисленные конкретные нововведения рассматриваются в соответствующих главах.

Язык Delphi (Object Pascal) и его использование

- Основы языка Delphi (Object Pascal)
- Определение собственных типов данных
- Подпрограммы
- Операторы
- Классы и объекты

Основы языка Delphi (Object Pascal)

Паскаль и Delphi (Object Pascal)

В системе *Delphi 7* используется специализированная, постоянно совершенствуемая версия языка программирования Паскаль, которая называется *Delphi* (в шестой и более ранних вариантах системы *Delphi* она называлась *Object Pascal,* «Объектный Паскаль»). Эта версия включает набор расширений, ориентированных только на применение в рамках среды *Delphi 7* и предназначенных для ускоренного создания приложений.

В комплект системы *Delphi 7* входит компилятор командной строки `dcc32.exe` для этого языка. Кроме того, выполнять компиляцию можно непосредственно из интегрированной оболочки.

ВНИМАНИЕ Далее, во избежание неоднозначностей, в книге под термином «Паскаль» понимается специализированная версия языка Delphi (Objecl Pascal), если не оговорено особо.

Запись программы

Программа на Паскале записывается с помощью набора символов, включающего латинские буквы (регистр не имеет значения), цифры, символ подчеркивания и стандартные знаки препинания. Элементы программы отделяются друг от друга с помощью произвольного числа пробелов и пустых строк.

Некоторые элементы языка записываются путем комбинации двух специальных символов, например:

```
.. // := <>
```

Программа содержит *ключевые* (или *зарезервированные*) слова, как стандартные, так и пользовательские (включаемые в программу разработчиком), а также *идентификаторы* и *выражения.*

В качестве идентификатора может выступать любая последовательность из букв, цифр и символа подчеркивания, начинающаяся не с цифры. Например:

```
Unit1
Integer
x
for
There_are_Dates
Go478
```

ЗАМЕЧАНИЕ В качестве пользовательских идентификаторов нельзя использовать зарезервированные слова и стандартные идентификаторы.

При анализе исходного текста программы компилятор не различает прописных и строчных букв, то есть можно написать, например:

```
begin
end;
```

а можно:

```
Begin
End;
```

Эти записи тождественны.

Правила записи команд Паскаля путем комбинирования ключевых слов и идентификаторов называются *синтаксическими правилами* или просто *синтаксисом записи*.

Типы данных

Все данные, используемые в программе, всегда относятся к конкретным *типам данных*. Например, число 32000 относится к типу Integer (целое), число 2,87 — к типу Real (число с десятичной запятой). Применяемые разработчиком значения должны укладываться в допустимый *диапазон значений* для имеющихся в Паскале типов.

Целые числа

Целые числа записываются в программе с помощью последовательности цифр, перед которой может стоять знак числа: символ «+» или «–». Если знак не указан, то считается, что число положительное. Например:

```
2
-3
3
+817
```

В следующей таблице перечислены стандартные типы целых чисел и соответствующие им диапазоны допустимых значений.

Таблица 1.1. *Стандартные типы целых чисел*

Название типа (стандартные идентификаторы)	Диапазон допустимых значений
Integer	-2147483648..+2147483647
Cardinal	0..4294967295
Shortint	-128..+127
Smallint	-32768..+32767
Longint	-2147483648..2147483647
Int64	$-2^{63}..+2^{63}-1$
Byte	0..+255
Word	0..+65535
Longword	0..+4294967295

ЗАМЕЧАНИЕ Целые числа могут записываться не только в десятичной форме, но и в шестнадцатеричной. Для этого перед началом числа ставится символ $, а в качестве цифр, соответствующих числам от 10 до 15, используются буквы A–F.

```
$FF
$00007a
```

Дробные числа

Дробные числа содержат дробную часть, которая отделяется от целой части десятичной точкой. В таких числах допускается также дополнительно указывать символ e (или E), за которым следует число, сообщающее, что левую часть дополнительно надо умножить на 10 в соответствующей степени. Например:

Запись 2e+5 означает 2, умноженное на 10 в степени 5 (200000);

Запись 31.4E–1 означает 31,4, умноженное на 10 в степени –1 (3,14).

Ниже приведены основные стандартные типы дробных чисел и соответствующие им диапазоны допустимых значений. В таблице для большинства типов указан только диапазон положительных значений, однако допустимым также является аналогичный диапазон отрицательных значений, а также число 0 (0.0).

***Таблица 1.2.** Основные стандартные типы дробных чисел*

Название типа (стандартные идентификаторы)	Диапазон допустимых значений
Real	5e–324 .. 1.7e+308
Real4S	2.9e–39 .. 1.7e+38
Single	1.5e–45 .. 3.4e38
Double	5e–324 .. 1.7e+308
Extended	3.6e–4951 .. 1.1e4932
Comp	-2^{63} .. $+2^{63}-1$
Currency	–922337203685477.5808 .. 922337203685477.5807

ВНИМАНИЕ Вычисления с дробными числами выполняются приближенно за исключением типа Currency (Финансовый), который предназначен для использования при создании бухгалтерских и финансовых программ и минимизирует ошибки округления.

Символы

Помимо чисел, в Паскале разрешается обрабатывать данные в виде *одиночных символов* и их последовательностей *(строк)*. Символы имеют тип Char и записываются в виде знака, взятого в одиночные кавычки:

```
'5'
'S'
'ж'
```

ЗАМЕЧАНИЕ В качестве символов допускается использование букв национального алфавита.

Иногда требуется обрабатывать символы, имеющие значения, которые невозможно отобразить на экране. В таких случаях символ записывается в виде числа, перед которым стоит знак # (в соответствии с кодами символов в кодировке *ANSI*). Например:

```
#0
#40
```

Полным аналогом типа Char является тип AnsiChar. Допустимый диапазон его значений (при записи с помощью чисел) — от #0 до #255. В Паскале имеется еще тип WideChar, соответствующий шрифтовой кодировке *UNICODE* (первые 256 символов этого типа соответствуют кодировке *ANSI).*

Строки

Последовательность символов, заключенная *в* одиночные кавычки, называется *строкой* (тип String — зарезервированное слово). Например:

```
'это текстовая строка Паскаля'
```

Если требуется поместить сам символ одиночной кавычки внутрь строки, его надо повторить дважды:

```
'это '' - символ одиночной кавычки'
```

Некоторые символы могут иметь значения, которые невозможно непосредственно отобразить на экране (символы, не соответствующие стандарту *ANSI).* В этом случае коды соответствующих символов можно прямо (без разделяющих пробелов) включать в состав строки.

```
''в этой строке'#10#13' имеются непечатные символы'#0
```

Строки в Паскале могут быть различной максимальной длины. Строка типа ShortString содержит до 255 символов (этот тип введен для совместимости со старыми версиями), строка типа AnsiString — 2^{31} (2 Гбайт) символов, относящихся к типу AnsiChar (данный строковый тип совпадает со стандартным типом string), строка типа WideString — 2^{30} символов типа WideChar.

Строка может быть пустой, не содержащей ни одного символа. Тогда она записывается как две идущие подряд одиночные кавычки — ''.

Строки с нулем в конце (null-terminated strings)

Структура строки в Паскале (в той версии языка, которая была реализована компанией *Borland* еще для системы *MS-DOS,* когда операционной системы *Windows* не существовало) отличается от структуры строки, которая обрабатывается системными вызовами *Windows.* Эта структура характеризуется тем, что отсчет символов в строке начинается с нуля, а завершается строка символом с кодом 0 (#0).

ЗАМЕЧАНИЕ Кроме Windows, строки с завершающим нулем применяются в широко распространенных языках программирования С (Си) и C++ (Си++). В Паскале описываемый тип строк введен для обеспечения совместимости с операционной системой Windows и кодом, написанным на C/C++.

Тип строки с нулем в конце в зависимости от типа составляющих ее символов называется PChar или PWideChar.

ПОДСКАЗКА В Паскале в большинстве случаев разрешается смешивать эти типы, но при программировании рекомендуется придерживаться, в основном, типа String, а к типу PChar прибегать только, когда без этого не обойтись.

Строки фиксированной длины

По умолчанию строка типа String может иметь размер до 2 Гбайт, а оперативная память для нее выделяется программой автоматически, в зависимости от текущей длины строки. В некоторых случаях бывает полезно ограничить длину строки небольшим фиксированным значением. Чаще всего это требуется при работе с файлами, которые содержат текстовую информацию в заранее известном формате.

Для явного указания длины после ключевого слова **string** *в* квадратных скобках задается число, определяющее эту длину.

```
string[50]
```

Для такой строки на этапе компиляции будет выделена область памяти в 50 символов. Строку большей длины (например, 51 символ) записать в нее нельзя (меньшей - можно, но объем зарезервированной для строки памяти останется неизменным).

Логические данные

Помимо чисел, символов и строк, в Паскале имеется тип данных Boolean, в диапазон значений которого входят всего две величины: True (истина, да) и False (ложь, нет). Их нельзя использовать в выражениях в качестве числовых или символьных величин.

ЗАМЕЧАНИЕ Значение False считается меньше, чем значение True.

Структура программы на Паскале

Модули

Программа на Паскале состоит из набора *модулей* (Unit), в каждом из которых содержится описание логически независимой части программы (например, описание работы конкретного окна или описание алгоритма вычисления сложной математической функции). Расширение имени файлов, содержащих модули — .PAS. Модули

программы часто создаются системой *Delphi* 7 автоматически, например при добавлении новой формы. При этом происходит *автоматическая генерация* исходного текста соответствующего модуля, что избавляет программиста от рутинной работы.

ВНИМАНИЕ Вносить изменения в исходный код программы, созданный автоматически, в большинстве случаев не разрешается. Это может привести к возникновению серьезных ошибок на этапе компиляции.

Модули могут иметь связь друг с другом, то есть из одного модуля разрешается обращаться к функциям других модулей. Применение модулей во время разработки программы напоминает применение компонентов во время проектирования экранных форм в том плане, что позволяет повторно использовать программный код, созданный ранее.

ЗАМЕЧАНИЕ В реальности исходные тексты компонентов Delphi 7 представляют собой обычные модули Паскаля, содержащие описание логики работы и способа отображения на экране соответствующих компонентов.

Главный файл

В программе может быть любое количество модулей (несколько сот или вообще ни одного), но только один *главный файл проекта.* Этот файл чаше всего невелик и содержит обращения к модулям. Он имеет расширение .DPR и создается средой Delphi 7 автоматически. Начинается этот файл с ключевого слова **program** (программа), за которым следует название программы и точка с запятой.

```
program DemoProgram;
```

Способы подключения модулей

Модули подключаются к главной программе и к другим модулям с помощью следующей конструкции языка:

```
uses список-модулей;
```

Список модулей представляет собой список названий модулей, перечисленных через запятую. Он может включать как модули, созданные разработчиком для текущей программы, так и стандартные, входящие в поставку *Delphi 7.* Все указанные модули должны существовать.

Например:

```
uses SysUtils, Forms, MyUnit;
```

В некоторых случаях местонахождение исходного текста модуля требуется задавать явно. Подобная потребность возникает в следующих случаях:

- О модуль расположен в отдельном каталоге, и в настройках *Delphi 7* этот каталог не указан;
- О модули из разных каталогов имеют одинаковые имена.

Для решения этих проблем в операторе **uses** после названия соответствующего модуля указывается ключевое слово in, а за ним в одинарных кавычках приводится путь к исходному тесту данного модуля:

```
uses Windows,
     MyUnit in 'c:\projects\games\dune5\MyUnit.pas', Main;
```

Логические блоки

Команды Паскаля принято группировать в логические блоки, представляющие собой своего рода модули в миниатюре. В логических блоках размещаются операторы, ответственные, например, за обработку нажатия пользователем кнопки, или операторы, выполняющие определенные действия в зависимости от некоторого условия. Главная часть программы (файл с расширением .DPR) всегда состоит из одного логического блока, в котором обычно происходит инициализация программы, затем выполняются операторы, ответственные за основную реализацию алгоритма работы, и в заключение выполняются действия по освобождению памяти и других ресурсов.

Логический блок начинается с ключевого слова **begin** (начало) и заканчивается ключевым словом **end** (конец). После ключевого слова end обычно ставится точка с запятой. Наиболее важным исключением является главная программа, в которой после слова **end** ставится точка.

```
program DemoProgram;
begin

end.
```

Стандартные функции и процедуры

Решение практически любой задачи можно запрограммировать самостоятельно от начала до конца. Однако при составлении программ очень часто возникает потребность выполнить какое-либо действие, которое уже использовалось в различных программах. Например, при математических расчетах нужно вычисление тригонометрических функций, а программированием этих вычислений наверняка уже не раз занималось множество программистов. Поэтому в систему *Delphi 7* входит обширный набор стандартных модулей, содержащих *стандартные функции*. Такие модули представляют собой готовый откомпилированный и оптимизированный код, предназначенный для решения самых разных задач. Чтобы вычислить значение синуса числа, не надо реализовывать алгоритм вычисления синуса заново, достаточно просто написать;

```
sin(3)
```

ЗАМЕЧАНИЕ Функции для вычисления синусо, косинуса и арктангенса входят в стандартный модуль System.

Все функции в Паскале (не только стандартные) записываются так: сначала следует название функции, потом в круглых скобках — список параметров через запятую (если параметров несколько).

Помимо функций, в Паскале имеются *стандартные процедуры*. Если функции используются для вычисления конкретных значений (как функция синуса), то процедуры предназначены для выполнения каких-то часто встречающихся действий (например для вывода информации на экран). И процедуры, и функции могут не требовать ни одного параметра — тогда круглые скобки за их названием не указываются.

```
ReadLn;
```

Переменные

Во время работы программы данные в ней могут храниться в неизменном виде, как *константы* (тогда они указываются в тексте программы явно), или же они записываются и обрабатываются как *переменные*. Переменные можно рассматривать как ячейки памяти компьютера, имеющие свои имена (идентификаторы). Содержимое переменных может многократно меняться. Каждая переменная имеет тип, определяющий, какого рода данные в ней хранятся. Паскаль не допускает использования переменных с неопределенным типом и не разрешает записывать в переменную одного типа данные другого типа.

Для того чтобы переменную можно было использовать в программе, ее предварительно надо *объявить* (*декларировать*, *описать*).

ЗАМЕЧАНИЕ Вся структура языка Паскаль такова, что в нем не разрешается использовать никакие пользовательские идентификаторы, если они не были описаны до момента обращения к ним.

Команда описания переменных в Паскале записывается так:

```
var имя-переменной: тип-переменной;
```

Слово **var** — ключевое. В качестве имени переменной выступает любой допустимый идентификатор, если он не был описан ранее и не является зарезервированным словом, а в качестве типа — одно из названий допустимых типов.

```
var X007: integer;
var Pi: real;
```

Если несколько описаний переменных следуют друг за другом, то ключевое слово var можно повторно не указывать.

```
var X007: integer;
    Pi: real;
```

Имен переменных может быть указано несколько, в таком случае они перечисляются через запятую, и эти переменные будут иметь одинаковый тип:

```
var A, B, X, Count, Delta2: string;
```

ЗАМЕЧАНИЕ Объявлять переменные разрешается только вне логических блоков **begin/end**.

Константы

В некоторых случая бывает удобно вместо явного указания конкретных значений (чисел или строк) использовать *константы* — фиксированные значения, для которых определено имя. Константы отличаются от переменных тем, что не могут менять свои значения. Они предназначены только для удобства программиста. Пусть, например, заранее неизвестно, какая потребуется точность при вычислении некоторой функции, а пороговое значение точности используется в исходном тексте программы в разных местах. Чтобы не пришлось для изменения этого значения выполнять трудоемкий поиск и замену конкретного числа во всем тексте программы, правильнее описать порог один раз как константу и в дальнейшем обращаться к нему только по имени. Тогда при необходимости внести изменение достаточно будет поменять всего одну строчку в программе.

ВНИМАНИЕ Константы, как и переменные, нельзя использовать до их первого описания.

Константы описываются способом, напоминающим описание переменных, только вместо ключевого слова var применяется ключевое слово **const,** а тип можно указывать, а можно и не указывать. Значение константы задается после знака равенства:

```
const PI = 3.14,-
      E: Real = 2.87;
```

ПОДСКАЗКА Названия констант принято записывать прописными буквами, чтобы легко отличать их от переменных.

Математические выражения

Для вычисления значений по формулам в Паскале применяются *выражения,* состоящие из *операндов* (данных, констант и переменных), связанных между собой арифметическими *операциями.* Каждая арифметическая операция имеет два операнда, расположенных слева и справа от знака операции.

Таблица 1.3. *Математические операции в Паскале*

Обозначение **операции** в Паскале	Назначение
+	Сложение
-	Вычитание
*	Умножение
/	Деление (результат имеет дробный тип)
div	Целочисленное деление (результат имеет целый тип, а остаток отбрасывается)
mod	Остаток от деления левого операнда на правый (для целых типов)

Каждая арифметическая операция имеет свой *приоритет* (очередность выполнения). Операции с более высокими приоритетами выполняются в первую очередь. Операции с равными приоритетами выполняются слева направо.

Более высокий приоритет имеют операции *, /, div, **mod,** меньший — операции + и -.

Чтобы изменить порядок вычисления выражения, используют круглые скобки. Скобки могут быть вложены друг в друга неограниченное число раз.

```
X + 1
(a + b) * (a - b)
Variable div 2 - 31
(Ы * (L2 + 45.8)) / 3.33
```

Логические выражения

Для манипулирования логическими величинами True и False в Паскале имеются четыре операции.

***Таблица** 1.4. Логические операции в Паскале*

Обозначение операции в Паскале	Назначение
And	Логическое И. Результат равен True, если оба операнда равны True, в противном случае результат равен False
Or	Логическое ИЛИ. Результат равен True, если хотя бы один из операндов равен True, в противном случае результат равен False
Xor	Исключающее ИЛИ. Результат равен True, если операнды не равны друг Другу, в противном случае результат равен False
Not	Отрицание. Имеет только один операнд, указываемый справа. Результат равен True, если значение операнда равно False, в противном случае результат равен False

Операция not имеет наивысший приоритет, операция **and** — более низкий, операции or и **xor** имеют самый низкий приоритет среди логических операций.

Как и В случае с арифметическими выражениями, порядок вычисления логического выражения можно менять с помощью круглых скобок.

```
X or Y
not One
(a1 or b1) and (a1 or c1)
```

Битовые выражения

В Паскале имеется возможность выполнять операции над отдельными битами числа (которое представлено в машинном коде программы в виде одного или нескольких

байтов). Если типы (длины) операндов битовых операций отличаются, то результат имеет тип, соответствующий типу данных самой короткой длины (в байтах).

Результат вычисляется путем применения битовой операции к соответствующим парным битам каждого операнда.

Таблица 1.5. Битовые операции в Паскале

Обозначение операции в Паскале	Назначение
And	Битовое И. Бит результата равен 1, если оба бита операндов равны 1, в противном случае итоговый бит равен 0
Or	Битовое ИЛИ. Бит результата равен 1, если хотя бы один из битов каждого операнда равен 1, в противном случае итоговый бит равен 0
Xor	Битовое исключающее ИЛИ. Бит результата равен 1, если соответствующие биты операндов не равны друг другу, в противном случае итоговый бит равен 0
Not	Битовое отрицание. Операция имеет один операнд, указываемый справа. Бит результата равен 1, если бит операнда равен 0, в противном случае итоговый бит равен 0
Shl	Битовый сдвиг влево (младшие, правые биты заполняются нулями). Левый операнд побитно сдвигается влево на число битов, заданное правым операндом
Shr	Битовый сдвиг вправо (старшие, левые биты заполняются нулями). Левый операнд побитно сдвигается вправо на число битов, заданное правым операндом

Операция **not** имеет наивысший приоритет, операции **and**, **shl** и shr — более низкий, операции or и **xor** имеют приоритет, самый низкий среди битовых операций.

Примеры выполнения битовых операций:

Выражение 11110000 **and** 10111101 имеет значение 10110000

Выражение 11110000 **or** 10111101 имеет значение 11111101

Выражение 11110000 **xor** 10111101 имеет значение 01001101

Выражение not 11110000 имеет значение 00001111

Выражение 00001111 **shl** *2* имеет значение 00111100

Выражение 11101111 **shr** 3 имеет значение 00011101

Строковые выражения

Для строк определена только одна операция — *сложение* или *сцепление*, обозначаемая символом «+». Результатом является строка, полученная сцеплением левого и правого операндов.

Выражение '* это' + ' строка' + '!' имеет значение '* это строка!'

Оператор присваивания

Результат вычисления выражения для его дальнейшей обработки (например, для вывода в поле на форме) надо предварительно сохранить, а именно записать в переменную, которая предварительно должна быть объявлена.

Чтобы поместить данные в переменную (в область памяти, соответствующую имени этой переменной; такое соответствие отслеживается компилятором автоматически), надо использовать оператор присваивания. Он записывается так:

```
имя-переменной := новое-значение;
```

ЗАМЕЧАНИЕ В языке Паскаль каждый оператор (за небольшими исключениями, о которых будет сказано ниже) заканчивается символом «;».

В левой части оператора присваивания указывается имя переменной, в которую будет занесено значение выражения, расположенного справа от обозначения оператора (символов :=).

Например:

```
X := 2;
a55 := (b20 + 120) / 3.14;
My Ping := X or $F0;
NameOfUser := 'имя пользователя:' + login;
```

ЗАМЕЧАНИЕ Оператор присваивания, как и остальные операторы Паскаля, можно выполнять только внутри логических блоков begin/end.

Комментарии

Программа на Паскале может снабжаться комментариями разработчика к своему исходному тексту. Желательно всегда сопровождать тексты своих программ комментариями, даже если алгоритм кодируется с помощью довольно простых операторов. Обычно трудно сразу понять, что делает участок программы из нескольких десятков операторов, особенно если реализуемая ими логика достаточно сложна.

Комментарии в Паскале бывают двух типов: многострочные и однострочные. Многострочные комментарии заключаются в фигурные скобки { и }. Разрешается также заключать комментарии в скобки, состоящие из пар символов: (* и *). Комментарий должен заканчиваться аналогично тому, как он начинался, то есть нельзя, например, «закрывать» фигурную скобку { парой символов *).

Комментарии разрешается вкладывать друг в друга.

```
(*
Это начало комментария.
```

```
Далее - вложенный комментарий
{ на одной строке }
комментарий заканчивается:
*)
```

Однострочный комментарий начинается с пары символов //. При этом весь текст до конца текущей строки считается комментарием. Закрывать его не надо.

```
X := 5; // присваивание
```

В тексте комментария можно использовать любые символы.

Создание простейших программ

Консольное приложение

С помощью системы *Delphi 7* можно создавать приложения *Windows* практически неограниченной сложности, использующие графический интерфейс. Однако для тех, кто только начинает знакомство с основными операторами Паскаля, имеется возможность создания простых программ в стиле *MS-DOS* (в качестве учебных). Это так называемые *консольные приложения*. Внешне они выглядят как программы с текстовым интерфейсом, но способны обращаться к большинству функций *Windows*.

Чтобы создать консольное приложение, надо дать команду File > New ➤ Other (Файл > Создать > Другое) и в диалоговом окне New Items (Создание программы) выбрать значок Console Application (Консольное приложение) (рис. 1.1).

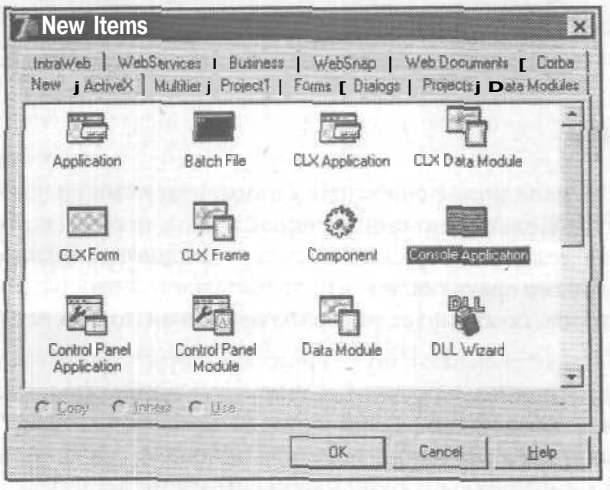


Рис. 1.1. Выбор категории, к которой относится создаваемая программа

Система *Delphi 7* автоматически сгенерирует в текстовом редакторе исходный код — заготовку будущего приложения.

```
program Project1;
{$APPTYPE CONSOLE}
uses sysutils;

begin
  {TODO -oUser -cConsole Main : Insert code here}
end.
```

Код начинается с заголовка программы, состоящего из ключевого слова **program** и автоматически сгенерированного названия Project1.

Далее в виде комментария следует *директива компилятора.* Она отличается от обычного комментария тем, что сразу за символом { следует символ **$**, а следом — слово, задающее определенную настройку компилятора. Хотя настройки компилятора можно задавать и непосредственно в среде *Delphi 7,* нужные настройки не всегда могут быть включены разработчиком вручную, поэтому лучше явно указывать их в тексте там, где они обязательно требуются.

Директива {$APPTYPE CONSOLE} говорит компилятору, что данная программа представляет собой консольное приложение.

Следующая строка задает подключение стандартного модуля *SysUtils* с помощью ключевого слова uses.

Далее идет собственно программа — для нее подготовлен логический блок, внутри которого система *Delphi 7* добавила комментарий Insert user code here (Вставьте сюда свой исходный текст).

Обмен информацией

Так как создаваемая программа — консольная, общаться с пользователем с помощью графического интерфейса она не может. Для этого нужны более простые средства обмена информацией с человеком.

В Паскале имеются две стандартные процедуры ReadLn (для ввода данных) и WriteLn (для записи данных), которые могут использоваться в консольных приложениях. Первая процедура выполняет ввод значения с клавиатуры и его передачу в переменную, которая указана в качестве параметра (можно указать и список переменных — тогда при вводе соответствующие им значения надо разделять пробелами). Вторая процедура выводит одно или несколько значений переменных или выражений в текущую строку экрана, разделяя их пробелами.

Процедура ReadLn требует, чтобы человек после окончания ввода нажал клавишу ENTER, а процедура WriteLn после окончания вывода значений осуществляет перевод курсора на следующую строку. Другая процедура вывода, Write, не выполняет такого перевода и продолжает вывод данных в текущую строку.

ВНИМАНИЕ Переход на следующую строку в процедуре WriteLn происходит не до, а *после* вывода значений параметров.

У процедуры ReadLn может не быть ни одного параметра. Тогда при ее выполнении программа просто ожидает, когда пользователь нажмет клавишу ENTER. Если не указан ни один параметр процедуры WriteLn, то на экране произойдет пропуск одной пустой строчки.

Для первого, пробного запуска в главный логический блок программы можно поместить один оператор ReadLn — тогда программа просто высветит черное консольное окно и закроется, когда будет нажата клавиша ENTER.

```
begin
ReadLn
end.
```

Обратите внимание: точку с запятой перед ключевым словом **end** можно не ставить.

Сохранение программы

Перед первым запуском программы ее исходный текст необходимо сохранить. Для этого щелкните на командной кнопке Save All (Сохранить все) на панели инструментов.

Система *Delphi 7* попросит указать место для сохранения главного файла Project1. Укажите любую подходящую папку.

Компиляция и запуск программы

Чтобы откомпилировать и сразу запустить данную программу, достаточно нажать клавишу F9. В папке, где был сохранен файл с исходным текстом, появится исполнимый файл программы — Project1.exe. Она будет тут же автоматически запущена из среды *Delphi 7*. На экране откроется окно консольного приложения.

После нажатия клавиши ENTER управление будет передано обратно в оболочку *Delphi 7*.

Программа сложения двух чисел

Чтобы продемонстрировать возможности Паскаля и среды *Delphi 7* на практике, реализуем небольшой пример. С клавиатуры будут вводиться два целых числа, а программа рассчитает их сумму.

Перед началом логического блока надо описать три переменные: две из них предназначены для хранения значений, вводимых пользователем, а третья будет содержать результат сложения. Назовем переменные X, Y и Z.

Исходный текст программы будет выглядеть так:

```
program Project1;
{$APPTYPE CONSOLE}
uses sysutils;

var X, Y, Z: integer;
```

```
begin
ReadLn(X, Y);
Z := X + Y;
WriteLn(Z);
ReadLn
end.
```

Сначала производится ввод двух чисел: их значения записываются в переменные X и Y, затем с помощью оператора присваивания сумма значений этих переменных помещается в переменную Z, а процедура WriteLn выводит значение переменной Z на экран. Последний оператор нужен, чтобы после выполнения всех действий программа не сразу закрывала свое окно, а ожидала нажатия клавиши ENTER — тогда мы сможем посмотреть на результат своей работы.

После нажатия клавиши F9 на экране появится окно созданной программы. В него можно ввести два числа, например:

```
2 2
```

Если теперь нажать клавишу ENTER, программа сразу же напечатает результат:

```
4
```

и будет ожидать еще одного нажатия клавиши ENTER, чтобы завершить свою работу (рис. 1.2).

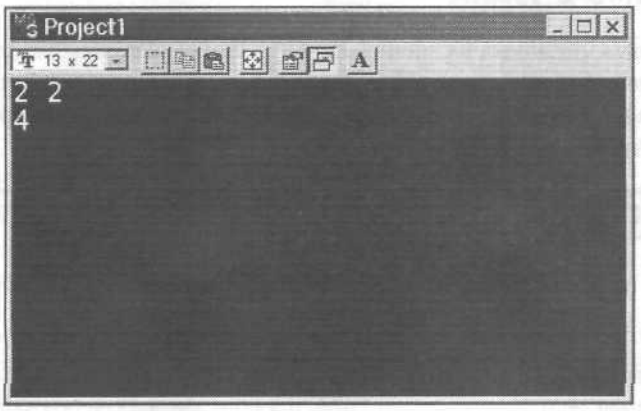


Рис. 1.2. *Созданное консольное приложение в действии*

ЗАМЕЧАНИЕ Окно консольного приложения после завершения работы может не закрываться. Чтобы закрытие происходило автоматически, надо в служебном меню приложения выбрать пункт Свойства и в открывшемся диалоговом окне свойств установить флажок Закрывать окно по завершении сеанса работы.

Определение собственных типов данных

Зачем нужны новые типы

При создании практически любой серьезной программы обойтись без дополнительных, более сложных, чем числа и строки, типов данных бывает довольно трудно. Гораздо удобнее работать, например, с типом данных Цвет и его значениями: красный, желтый, зеленый. — нежели просто с числами 1, 2, 3. Программа при этом получается значительно нагляднее, а это залог ее качества.

Описание нового типа

Чтобы описать (ввести в программу) новый тип данных, в Паскале имеется специальное ключевое слово **type:**

type название-типа = описание-типа;

Название типа — это произвольный идентификатор Паскаля. Описание типа может представлять собой описание перечислимого типа, описание сложного типа, описание массива и так далее.

Перечислимые типы

Помимо обычных числовых и строковых типов Паскаль позволяет создавать типы, диапазон значений которых — просто набор идентификаторов. Это удобно в тех случаях, когда в решаемой задаче имеется понятие, значения которого нагляднее описывать не числами, а словами.

ЗАМЕЧАНИЕ Хотя такие же значения можно создать с помощью констант, перечислимый тип представляет собой именно тип данных, с помощью которого можно описывать переменные и выполнять над ними различные операции.

Перечислимый тип записывается взятой в круглые скобки последовательностью идентификаторов — значений этого типа, перечисляемых через запятую. Первые элементы типа считаются *младшими* по сравнению с идущими следом.

Например, тип, описывающий названия футбольных команд, может быть сформирован так:

```
type TFootballTeam =
     (Spartak, CSKA, Dynamo, Locomotive, Torpedo);
var Team: TFootballTeam;
begin
Team := Locomotive;
...
```

В Паскале под *перечислимыми* типами обычно понимаются не только типы, представляющие собой списки идентификаторов, но и другие базовые типы, для которых можно формально определить последовательность значений и их старшинство. К таковым относятся:

- все *целочисленные* типы (Integer, Byte и так далее), для которых всегда можно сказать, какое следующее число будет следовать за числом N;
- символьные типы (Char): за символом 'a' всегда следует символ 'b', за символом '0' — символ T и так далее;
- логические типы — тип Boolean представляет собой не что иное, как перечислимый тип (False, True).

Типы поддиапазонов

Паскаль допускает возможность создания типов поддиапазонов. С их помощью на основе ранее сформированных или стандартных типов выделяются поддиапазоны значений, которые образуют новые типы. Такой подход хорош тем, что позволяет накладывать более жесткие ограничения на диапазоны значений переменных. Например, если заранее известно, что значение некоторой переменной на протяжении работы программы не должно выходить из промежутка 0-100, то такую переменную лучше описывать не как имеющую тип Integer, а как имеющую тип поддиапазона 0-100. Это позволит автоматически проконтролировать выход значений за пределы конкретного поддиапазона.

Диапазон данных в Паскале описывается как последовательность, включающая наименьшее значение в диапазоне, две точки без пробела между ними и наибольшее значение. Например:

- 0 .. 100

Соответствующий тип может быть описан так:

```
type TMySubset = 0..100;
var MySubset: TMySubset;
begin
MySubset := 0,- // все нормально
MySubset := 101; // возникнет ошибка
```

Помимо числовых значений в типах поддиапазонов можно использовать и значения других типов, например Char:

```
type TMyChar = 'a' .. 'z';
```

Переменная типа TMyChar сможет принимать значения только из набора символов от 'a' до 'z', в порядке возрастания значений (кодов *ASCII)* этих символов.

Можно использовать поддиапазоны перечислимых типов. Если тип TFootballTeam уже описан, то на его основе можно создать новый тип:

```
type TMyTeams = CSKA .. Locomotive;
```

Переменные типа TMyTeams могут принимать только одно из трех значений: CSKA, Dynamo или Locomotive.

Структурные типы данных

Обойтись только простыми — *линейными* типами в большой программе довольно сложно. Желательно, чтобы структура данных прикладной программы отвечала структуре данных решаемой задачи. Для этого в Паскале есть набор *структурных* типов данных.

Массивы

Массив — это структура данных, доступ к элементам которой осуществляется по номеру (или *индексу).* Все элементы в массиве имеют одинаковый тип. Индекс элемента массива может быть вычисляемым, что позволяет организовывать компактную и эффективную обработку больших наборов данных.

Описание массива имеет вид:

```
type имя-типа-массива = array[ диапазон ] of тип-элемента;
```

Слова array (массив) и of — ключевые. Диапазон определяет нижнюю и верхнюю *границы* массива и, соответственно, число элементов в нем. При обращении к массиву индекс должен лежать в этом диапазоне. Тип элемента определяет тип каждого элемента массива.

Массив из 100 элементов целого типа (первый элемент будет иметь номер 1, последний — номер 100) описывается в программе так:

```
type TMyArray = array[ 1..100 ] of integer;
```

При задании диапазона могут выступать не только числовые значения:

```
type TMyArray = array[ TMyTeams ] of string;
```

Таким способом определяется тип, описывающий массив из трех строк. При обращении к первой строке индекс должен иметь значение C5KA (самое маленькое значение из диапазона TMyTeams), а при обращении к третьей строке — значение Locomotive.

Доступ к элементу массива описывается путем указания имени переменной соответствующего типа вместе со следующим за ней в квадратных скобках индексом. В качестве индекса может выступать произвольное выражение Паскаля, значение которого должно укладываться в диапазон, указанный при описании массива.

Например:

```
type TMyArray = array[ 1..100 ] of integer;
var MyArray: TMyArray;
N: integer;
begin
N := 50;
MyArray[ 1 ] := 100;
MyArray[ K ] := MyArray[ N+1 ] * 2;
MyArray[ N+12 ] := N div 4;
```

Другой пример:

```
type TMyArray = array[ TMyTeams ] of string;
var Teams: TMyArray;
begin
Teams[ CSKA ] := 'это ЦСКА';
Teams[ Dynamo ] := 'это Динамо';
Teams[ Locomotive ] := 'это Локомотив';
```

Вместо создания в программе нового типа на основе описания массива можно явно декларировать переменную как массив с помощью ключевого слова var:

```
var Teams: array[ TMyTeams ] of string;
```

Хотя такой подход использовать не рекомендуется. Дело в том, что при этом методе описания, даже когда две переменные внешне описаны одинаково, в Паскале они считаются относящимися к разным типам. Б таком случае присваивать их значения (весь массив) друг другу в одном операторе присваивания не разрешается.

```
var T1: array[ TMyTeams ] of string;
   T2: array[ TMyTeams ] of string;
begin
...
T1 := T2; // неправильно!
```

Чтобы разрешить копирование массивов, надо переменные описать как имеющие один тип:

```
type TMyArray = array[ TMyTeams ] of string;
var T1: TMyArray;
   T2: TMyArray;
begin
...
T1 := T2; // правильно!
```

Массив может иметь несколько *измерении,* перечисляемых через запятую.

```
array[ -5..10, 0..2 ] of byte;
```

Каждое измерение характеризуется своим диапазоном, тип которого может не совпадать с типами диапазонов других измерений. Например, если требуется описать массив дробных чисел с тремя измерениями, первое из которых охватывает диапазон целых чисел от 1 до 10, второе — диапазон значений от False до True (Boolean), а третье — диапазон TMyTeams, то выглядеть это будет так:

```
type TNewArray =
   array[ 1 .. 10, boolean, TMyTeams ] of real;
```

При доступе к элементам массива значения индексов также перечисляются через запятую:

```
var AA: TNewArray;
begin
AA[ 5, true, CSKA ) = 3.14;
```

Строковые данные формально представляют собой массивы символов, и к элементам строк можно обращаться с помощью индексов. Это позволяет выделять конкретные символы и менять их значения. Отсчет символов для переменных типа string начинается с единицы.

```
var S: string;
begin
S := '54321';
S[5] := '0'; // в строке окажется значение '54320'
```

Однако явно описывать строки как массивы символов вместо использования ключевого слова string неправильно. Единственное отличие — строки, заканчивающиеся нулевым символом. Их можно описывать в программе как массивы символов, первый элемент которых имеет номер 0. При занесении значений в такие переменные в конце строки надо указывать символ с кодом 0:

```
var S0: array[ 0..50 ] of char;
begin
S0 := 'строка с нулем'#0;
```

Обычно размеры массива неизменны и поэтому должны описываться явно с помощью констант, задающих нижнюю и верхнюю границы диапазона, или названий типов. Такие массивы называются *статическими*. Паскаль позволяет определять также *динамические* массивы, размер которых заранее не известен и может меняться во время работы программы. Для описания динамических массивов указывать диапазоны значений его измерений не нужно.

```
type TDynArray: array of integer;
```

Первоначально в переменной, описанной как динамический массив, нет ни одного элемента. Ее размер задается с помощью стандартной процедуры SetLength(). В качестве первого параметра указывается имя переменной, в качестве второго — число элементов в массиве. Если новое число элементов больше старого, то новые элементы добавляются в конец массива, а их значения исходно будут не определены. Если новое число элементов меньше старого, то последние элементы массива отбрасываются и теряются. Отсчет элементов динамического массива всегда начинается с нуля.

```
var a : TDynArray;
begin
SetLength( a, 1 );
a[0] := 2;
```

Чтобы освободить память, выделенную динамическому массиву, и сделать его длину нулевой, надо соответствующей переменной присвоить значение nil:

```
a := nil;
```

ЗАМЕЧАНИЕ Ключевое слово языка Паскаль nil обозначает «отсутствие значения».

После этого в массиве а не окажется ни одного значения. В дальнейшем длину массива а можно изменить снова.

Как и статические массивы, динамические массивы могут иметь несколько измерений. Для каждого из измерений при описании массива надо повторить описание array of. Например, если требуется объявить динамический массив целых чисел с двумя измерениями, надо записать так:

```
type T3DArray = array of array of integer;
```

Далее надо объявить размер этого массива по каждому измерению. Реализация динамических массивов в Паскале позволяет, в отличие от описания статических массивов, задавать разные диапазоны для одного измерения. Например, когда описан двумерный массив (обычно это карта, график, поле), сначала задается размер по первому измерению (число столбцов):

```
var D: T3DArray;
begin
SetLength(D,4);
```

Далее нужно указать число элементов для *каждого* из четырех столбцов (при описании статических массивов это число считается одинаковым для всех столбцов). Длина соответствующего столбца задается указанием имени массива с номером столбца с помощью процедуры SetLength():

```
SetLength( D[0], 3 );
SetLength( D[1], 5 );
SetLength( D[2], 30 );
SetLength( D[3], 1 );
```

ПОДСКАЗКА Динамические массивы позволяют существенно экономить память и запрашивать ее у операционной системы по мере необходимости, однако работа с ними в программе происходит значительно медленнее, чем со статическими массивами.

Записи

Запись — это структура данных, доступ к элементам которой осуществляется по имени (названию элемента). Элементы записи могут иметь разный тип, поэтому при описании записи надо указывать и название каждого элемента, и его тип.

Описание записи имеет следующий вид:

```
type название-типа-записи =
   record
   название-элемента: тип-элемента;
   ...
   end;
```

В качестве названий элементов выступают обычные идентификаторы Паскаля — так же, как при описании переменных. Например, описание записи, характеризующей футбольную команду (содержащую название команды, города, страны и год создания) может выглядеть так:

```
type TFootballTeamRecord =
   record
   Name: TFootballTeam;
   City, Country: string;
   Year: 1800..3000;
   end;
```

Элемент записи Year (год) можно описать как имеющий тип Integer, но более корректно явно указать его допустимый диапазон значений.

Для обращения к элементу записи сначала указывается имя переменной, затем точка и название соответствующего элемента.

```
var T: TFootballTeamRecord;
begin
T.Name := Spartak;
T.City := 'Москва';
...
```

Когда надо задать значения большому числу элементов переменной-записи, каждый раз предварительно указывать ее имя неудобно. Б Паскале имеется ключевое слово **with,** которое позволяет опускать имя переменной в логическом блоке. При этом перед каждым упоминанием элемента в этом блоке имя переменной будет ставиться автоматически.

```
with T do
   begin
   Name := Spartak;
   City := 'Москва';
   end;
```

Если в логическом блоке, охваченном словом **with,** требуется использовать также переменную, имя которой совпадает с названием одного из элементов записи, перед ней необходимо указать название модуля (или программы), в котором эта переменная описана.

```
var T: TFootballTeamRecord;
Name: string;
begin
Name := 'Москва';
with T do
   begin
   Name := Spartak;
   City := Project1.Name;
   end;
```

Если в блоке **with** модуль для конкретной переменной не указан, то компилятор прежде всего ищет ее имя в списке названий элементов записи, и только потом - среди обычных переменных.

После ключевого слова **with** можно указывать но одно, а несколько имен переменных-записей:

```
with T, MyTeam, Foo do
begin
...
```

Множества

Множество напоминает перечислимый тип, но отличается от него тем, что элементы в нем не упорядочены (в множестве нет ни самого младшего, ни самого старшего элементов). В множество входят также все допустимые подмножества (все мыслимые комбинации его элементов), что приводит к огромному числу вариантов, поэтому количество элементов в множестве не может быть больше, чем 256.

Множество описывается так:

```
type тип-множества = set of диапазон-значений-множества;
```

В качестве диапазона указывается любой тип Паскаля, число элементов в котором не более 256: Char, Byte, 0..255, TFootballTeam и так далее.

```
type MySet = set of 0..255;
type MySet = set of Byte;
type MySet = set of (Spartak, CSKA, Dynamo);
```

Конкретные значения множества задаются перечислением списка значений через запятую, который берется в квадратные скобки.

```
MySet := [1, 2 , 5, 10];
```

В таком списке можно дополнительно указывать диапазоны значений:

```
MySet := [2, 4, 8, 12..21];
```

Дополнительные операции над множествами описаны в следующей таблице.

Таблица 1.6. Дополнительные операции над множествами

Название **операции**	**Действие**
+ (объединение)	Результирующее множество состоит из элементов обеих множеств, указанных в качестве операндов (одинаковые элементы не дублируются). Выражение [1,2,3] + [3,4,5] равно [1,2,3,4,5]
- (разность)	Результирующее множество состоит из тех элементов множества, указанного в качестве левого операнда, которые отсутствуют во множестве, указанном в качестве правого операнда. Выражение [1,2,3] - [3,4,5] равно [1,2]
* (пересечение)	Результирующее множество состоит из элементов, имеющихся в каждом из множеств, указанных в качестве операндов. Выражение [1,2,3] * [3,4,5] равно [3]

Указатели

С целью повышения эффективности создаваемых программ в первых версиях Паскаля было реализовано понятие *указателя* — переменной, которая хранит не значение конкретного типа, а адрес памяти (условно говоря, номер ячейки), где хранится значение соответствующего типа. Указатель только *указывает* на место в памяти, а получить доступ к данным в этом месте можно *с* помощью специальных операций.

ПОДСКАЗКА Указатели в некоторых случаях позволяют получить определенный выигрыш в быстродействии, однако их использование нередко приводит к серьезным ошибкам. Так как проконтролировать, правильный ли адрес памяти хранит указатель, трудно, в результате ошибок можно изменить или затереть участки системной памяти или области данных программы, вследствие чего сделать полностью неработоспособной как прикладную программу, так и всю систему Windows. Поэтому указатели, реализованные в языке Delphi (Object Pascal) в целях совместимости со старыми версиями, применять не рекомендуется.

Признаком того, что переменная содержит не данные, а указатель, служит символ ^, который ставится *перед* названием типа переменной.

```
var BytePtr: ^byte;
```

Переменная BytePtr описана как указатель на данные типа byte. Чтобы присвоить ей адрес конкретного места памяти, используется операция @.

```
var BytePtr: ^byte;
    ByteVal: byte;
    begin
    ByteVal := 255;
    BytePtr := @ByteVal;
```

В переменной BytePtr будет храниться адрес байта памяти, который связан с переменной ByteVal.

Чтобы получить доступ к содержимому памяти, на которую указывает переменная-указатель, используется знак операции ^. В данном случае он ставится *после* имени переменной.

```
N := BytePtr";
```

В переменную N запишется значение, которое расположено по адресу, хранимому в переменной BytePtr (фактически это адрес байта, отведенного компилятором для переменной ByteVal).

По адресу памяти можно поместить значение подходящего типа.

```
BytePtr^ := 50;
```

ВНИМАНИЕ В данном примере содержимое указателя BytePtr указывает на область памяти, отведенную для переменной ByteVal. Таким образом, при записи значения по адресу из переменной By!ePtr реально произойдет запись значения в переменную ByteVal. То есть после выполнения донного оператора физически изменится содержимое одного байта, а в программе с точки зрения программиста изменится как значение BytePtr^, так и значение переменной ByteVal. Подобная работа с указателями усложняет логику программы, поэтому использовать указатели нежелательно.

В некоторых случаях без указателей обойтись все же не удается, потому что главное их преимущество — возможность манипулировать не большими объемами памяти (например, массивом из тысячи элементов), а указателями на эту область (указатель занимает обычно 4 байта, и на его копирование лишние такты процессора не тратятся). При обращении к системным функциям *Windows* обычно требуется использовать именно указатели, поэтому в Паскале имеется несколько базовых типов указателей, объявленных заранее.

ЗАМЕЧАНИЕ В Паскале принято типы, связанные с указателями, начинать не с буквы T, а с буквы P (от английского Pointer — указатель).

```
type PByte = ^Byte;
```

Это, в частности, тип PChar — указатель на символ (в реальности — на область памяти, которая представлена как последовательность символов, заканчивающаяся символом с кодом 0).

Дополнительные операции над указателями

Указатели можно складывать (операция «+») друге другом и с целыми числами, а также вычитать (операция «–»). Смысл этих операций состоит либо в сдвиге (смещении) адреса памяти на заданное целое число единиц соответствующего типа вперед или назад, либо в определении разности (числа единиц данных) между двумя указателями. Складывать два указателя в языке Паскаль не разрешается.

Например, если переменная P1 имеет тип PChar и указывает на место в памяти, где расположена строка '12345', то после выполнения оператора

```
P1 := P1 + 2;
```

переменная P1 будет указывать на строку '345'. Если теперь указатель сдвинуть обратно на 1:

```
P1 := P1 - 1;
```

то переменная P1 укажет на строку '2345'.

ВНИМАНИЕ Смещение указателя происходит не на число байтов, а на число элементов того типа, к которому принадлежит переменная-указатель. Для переменной типа PInteger сдвиг будет происходить физически на 4 байта (размер элемента данных типа Integer), и указатель всегда будет корректно указывать на начало очередного элемента заданного типа.

Если переменная P2 указывает на последний символ строки '12345' (то есть, символ '5'), то разность P2 - P1 даст значение 3 (отличие в три символа типа Char).

Переменные, создаваемые динамически

По аналогии с динамическими массивами в Паскале имеется возможность создавать динамические переменные, память для которых выделяется во время работы программы по специальному запросу.

ПОДСКАЗКА Такая технология активно применялась в старых версиях Паскаля для MS-DOS, когда на счету был каждый байт оперативной памяти и приходилось прибегать к всевозможным ухищрениям для ее экономии. Сегодня подобными средствами лучше не пользоваться.

При создании динамических переменных в программу надо передавать адрес памяти, где находится новый выделенный операционной системой блок памяти. При этом не обойтись без указателей. Процедура New() выделяет оперативную память в соответствии с типом переменной-указателя, используемого в качестве аргумента, а процедура Dispose() освобождает эту память.

```
var P: ^Integer;
begin
// выделяется память для хранения числа типа Integer
// адрес этого участка памяти записывается в переменную P
New(P);
P^ := 12000;
// Память освобождается
Dispose(P);
// Далее обращаться к содержимому памяти,
// куда указывает переменная P (P^), нельзя
```

Варианты

Паскаль предъявляет очень строгие требования к типам используемых в программе данных, не допускает вольного преобразования типов, а применение указателей считается плохим стилем программирования. В то же время современные технологии создания приложений нередко требуют от разработчика реализации возможностей обмена данными сложной структуры между несколькими приложениями, рабо-

тающими как на одном, так и на нескольких компьютерах в локальной сети или Интернете. Такие приложения могут быть написаны на разных языках программирования, поэтому способы организации данных внутри программ обычно не совпадают.

Для поддержки межпрограммного обмена блоками данных в Паскале реализован специальный тип Variant, который позволяет менять тип соответствующих ему данных во время работы программы. Такой подход помогает повысить гибкость программы и ее совместимость с другими приложениями. В то же время он чреват серьезными ошибками и сбоями в работе в случае неверного преобразования типов. Поэтому применять тип Variant (так же, как и указатели) надо только там, где без этого не обойтись. Как правило, это случаи, связанные с обращением к внешним программам, запущенным в сети.

ЗАМЕЧАНИЕ Переменные типа Variant не могут хранить записи, статические массивы, файлы, классы (см. далее), указатели.

```
var V: Variant;
begin
V := 1;
V := 2.3;
V := true;
V := 'строка';
```

Переменная V последовательно будет хранить значения типа Integer (1), Real (2.3), Boolean (True) и String ('строка'). Допускается преобразование строки в число — значение '2,3' можно использовать как число:

```
var R: real;
    V: variant;
begin
V := '9,55';
R := V * 1.14;
```

ВНИМАНИЕ В качестве десятичного разделителя в вариантных типах не всегда используется точка. Эту роль играет символ, который установлен в качестве такого разделителя в операционной системе Windows (в русифицированной версии, как правило, запятая).

Разрешены также преобразования строки в логический тип (из 'True' — в True, из 'False' — в False), любого числа, не равного 0 (или строки, хранящей представление такого числа, например '100') — в True, нулевого значения (или символа '0') — в False.

После описания в программе переменным типа Variant автоматически присваивается начальное значение Unassigned. Специальное значение Null говорит об отсутствии данных или их неизвестном типе. Стандартная функция VarType() возвращает код, характеризующий тип аргумента.

```
V := 'строчка';
```

В данном случае функция VarType(V) вернет значение varString. Значения соответствующих разным типам и форматам данных констант приведены в справочном руководстве по *Delphi 7*.

Эти константы применяются также в стандартной функции VarAsType(), которая в качестве первого параметра получает значение типа Variant, а вторым параметром указывается константа, определяющая, во внутреннее представление какого типа первый аргумент надо преобразовать.

```
V2 := VarAsType( V, varDouble );
```

В Паскале имеется стандартная функция VarArrayCreate() для создания на базе типа Variant статического массива. Первым аргументом идет описание границ массива в квадратных скобках, вторым — константа, определяющая тип элементов массива.

```
V := VarArrayCreate( [1..100], varByte );
```

Сложные структуры данных

В Паскале допускается произвольное комбинирование структур данных и неограниченное создание новых типов на основе определенных ранее. В частности, разрешено описывать массивы структур:

```
type MyRec =
   record
   Krinteger;
   S: String;
   end;
   MyArr = array [1..10] of MyRec;
var A: MyArr;
```

К переменной А теперь можно обращаться так:

```
A[5].N := 1;
A[10].S := '333';
```

или так:

```
with A[1] do
   begin
   N := 0;
   S := '';
   end;
```

Сначала происходит выделение элемента массива, а затем — выделение элемента записи.

Возможен и противоположный подход.

```
type MyArr = array [1..10] of integer;
   MyRec =
      record
```

```
        A: MyArr;
        S: string;
        end;
    var R: MyRec;
```

К переменной R разрешается обращаться так:

```
R.A[5] := 1;
```

Уровней вложения структур в массивы и наоборот может быть сколько угодно. Например, допускается следующая запись:

```
A1[2].A2.A3[5].A4.A5.A6[1,2,88].A7
```

Присваивание значений сложных типов

Подобные длинные цепочки вложений конструкций разных типов друг в друга могут свободно использоваться в выражениях Паскаля наравне с обычными переменными. От них они, впрочем, ничем и не отличаются — разве что формой записи. При этом можно копировать не только значения самых простых (базовых) типов наподобие Integer или Real, но и значения массивов и структур в одном операторе присваивания, Надо только, чтобы типы и правой, и левой частей оператора присваивания совпадали.

```
type TAr = array[1..1000] of Byte;
   TRc=record
      Ar: TAr;
      Count: integer;
      end;
var A: TAr;
   R: TRc;
begin
...
A := R.Ar;
R.Ar :=A;
```

Упакованные типы

При описании массива (или других структурных типов) можно указать зарезервированное слово **packed,** которое означает, что переменные этого типа будут размещаться в памяти компьютера плотно и компактно. Такой подход позволяет сэкономить оперативную память, отведенную для данных, но значительно понижает скорость работы программы при работе с упакованными массивами.

```
type TpA = packed array[1..5] of integer;
type TpR = packed record
   N: integer;
   S: string[50];
   end;
```

Основные стандартные функции для работы с типами

В дальнейшей работе с Паскалем не обойтись без базового набора стандартных функций и процедур (табл. 1.7 и 1.8).

Таблица 1.7. *Стандартные функции*

Имя функции	Возвращаемое значение
Ord	Порядковый номер элемента для перечислимых типов, код ASCII для типа Char
Chr	Символ (тип Char), преобразованный из числового аргумента
Pred	Предыдущее по порядку значение данного типа. Например, значение Pred(5) равно 4
Succ	Следующее по порядку значение данного типа. Например, значение Succ(5) равно 6
Length	Длина строки или число элементов е массиве
High	Максимально допустимое значение (для типа). Например, значение High(Byte) равно 255. Верхняя граница (для массива). Для динамических массивов это значение всегда равно Length() - 1
Low	Минимально допустимое значение (для типа). Например, значение Low(Byte) равно 0. Нижняя граница (для массива). Для динамических массивов это значение всегда равно О
SizeOf	Размер элемента данных указанного типа в байтах. Например, значение SizeOf(Byte) равно 1, значение SizeOf (Integer) равно 4.

Таблица 1.8. Стандартные процедуры

Имя процедуры	Назначение
Inc	Увеличение аргумента на 1 (или переход к следующему по порядку значению, если тип аргумента не числовой). Например, если значение переменной X равно 3, то вызов Inc(X) приведет к увеличению значения X на 1 (значением переменной станет 4). Если переменная X относится к перечислимому типу (Spartak, CSKA, Dynamo) и содержит значение CSKA, то вызов Inc(X) приведет к присвоению переменной X значения Dynamo
Dec	Действие обратно предыдущей процедуре (уменьшение аргумента на 1)

Преобразование типов

В ряде случаев требуется гарантировать, что результат вычисления выражения принадлежит конкретному типу. Например, вычисления могут производиться над числами типа Longint, а результат должен принадлежать типу Byte. Такая операция называется *приведением типов*. Для этого применяют так называемые преобразователи типов, которые напоминают стандартные функции Паскаля с именами, совпадающими с именами базовых типов (Byte, Integer и другие). Результат, возвращаемый таким преобразователем, гарантированно лежит в диапазоне указанного типа. При этом, конечно, возможна потеря значащих цифр: берется остаток от деления значения аргумента на максимально допустимое значение соответствующего типа.

Например, значение Byte(300) равно 44

```
var X: Integer;
   L: Longint;
begin
X := Integer(L * 1234567);
```

Однако подобным способом невозможно преобразовать число типа Real в число типа Integer или любое число в строку. Для таких, более сложных, преобразований типов в *Delphi 7* имеется набор стандартных функций, упрощающих процесс преобразования.

Таблица 1.9. *Стандартные функции преобразования типов*

Имя функции	Назначение
Round	Округление дробного числа до ближайшего целого. Значение Round(3.74) равно 4
Trunc	Отбрасывание дробной части числа. Значение Trunc(3.74) равно 3
IntToStr	Преобразование целого числа в строку. Значение IntToStr(12987) равно '12987'
FloatToStr	Преобразование дробного числа в строку. Значение FloatToStr(3.74) равно '3.74'
StrToInt	Преобразование строки в целое число. Значение StrToInt('12985') равно 12985
StrToFloat	Преобразование строки в дробное число. Значение StrToFloat('3.14') равно 3.14

Приведение типов переменных

Помимо стандартных функций, в Паскале имеется еще одна возможность явно указывать, в какой тип должны быть приведены конкретные данные. Эта возможность используется в операторе присваивания, когда *переменная, стоящая* в левой части, «охватывается» названием типа, соответствующего типу выражения правой части оператора. Например:

```
var C: Byte;
...
Char(C) := 'A';
```

Переменной C будет присвоен код *ASCII* символа 'A'. При этом необходимо, чтобы длины (в байтах) переменной и присваиваемого значения совпадали.

Другая форма записи этого оператора:

```
C := Byte('A');
```

Инициализация констант сложных типов

Часто требуется использовать в работе константы не только базовых типов, но и константы сложных типов (массивы, записи). Например, таблицы всевозможных начальных значений и параметров удобно размещать в массивах.

Инициализация массивов

Значения массива располагаются последовательно в круглых скобках. Если измерений у массива более одного, то измерения вкладываются друг в друга, начиная

с младших измерений (справа налево, если смотреть на описание массива). Например, если создан тип

```
type T1 = array[1..2, 1..5] of byte;
```

то в первую очередь надо сформировать значения для диапазона 1. .5. Таких последовательностей из 5 чисел требуется две — в соответствии с размером диапазона первого измерения.

Записывается константа сложного типа так:

```
const T: T1 =
((1,2,3,4,5), (100,200,300,400,500));
```

ВНИМАНИЕ Число элементов при инициализации массива должно в точности равняться его размеру (произведению длин всех измерений).

Инициализация записи

Форма инициализации записи отличается от формы инициализации массива тем, что приходится дополнительно указывать названия элементов, а за ними через двоеточие — значения. Разделяются такие конструкции точками с запятой.

```
type T2 =
   record
   Name: string;
   Num: integer;
   end;
const TT: T2 = (Name:'первый'; Num:1);
```

Инициализация **указателей**

Указатель инициализируется значением адреса ранее описанной переменной.

```
var N: Integer;
const PI: ^Integer = @N;
const constPStr: PChar = 'нуль-строка';
```

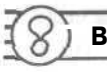
Подпрограммы

Ранее уже рассказывалось о стандартных подпрограммах: процедурах и функциях, — во множестве входящих в поставку *Delphi 7*. Далее описывается технология создания собственных подпрограмм.

Структура подпрограммы

Описание подпрограммы состоит из трех частей: заголовка подпрограммы, *локального описания и тела* подпрограммы. Заголовок используется, чтобы явно ввести в

программу новую подпрограмму и обозначить начало ее описания. Локальные описания представляют собой набор описаний типов, переменных, констант и других подпрограмм, которые действуют только в рамках данной подпрограммы. Тело подпрограммы — это логический блок begin/end, содержащий операторы и команды Паскаля и реализующий нужную логику работы.



ВНИМАНИЕ Описание подпрограммы — это только *описание*, которое никогда не выполняется само по себе и может располагаться в любом месте исходного текста (но обязательно до первого вызова подпрограммы). Вызывается процедура или функция только внутри логического блока begin/end с указанием конкретных значений для каждого из ее параметров.

Заголовок

Заголовок подпрограммы состоит из трех частей: ключевого слова, характеризующего тип подпрограммы (процедура — **procedure,** или функция — **function),** списка параметров, перечисленных через точку с запятой и взятых в круглые скобки, и (только для функций) типа возвращаемого значения, указываемого вслед за двоеточием.

Список параметров содержит параметры, каждый из которых представляет собой произвольное имя переменной и соответствующий ей тип через двоеточие. Если несколько параметров имеют одинаковый тип, то их можно перечислить через запятую.

```
procedure Compute(X: Integer; S: String);
function Compare(A1, A2 : Byte; Stroka: String): Real;
procedure ShowMap;.
function GameOver: Boolean;
```

Список параметров может быть опущен. Для некоторых параметров могут быть заданы значения по умолчанию (все параметры с такими значениями должны располагаться вместе и после параметров, для которых не задано значений по умолчанию).

```
procedure MyProc
    ( I: integer; i1: integer = 1; i2: integer = 2 );
```

В подобных случаях заключительные параметры при вызове процедуры можно опускать — они получат значения по умолчанию.

```
MyProc( 1, 2 );
```

Реально процедура вызывается с тремя параметрами, как если бы использовался следующий оператор.

```
MyProc( 1, 2, 2 );
```

Передача параметров по имени и по значению

Когда происходит обращение к подпрограмме, переменным, указанным в списке параметров в описании (*формальные* параметры), присваиваются значения, указанные в списке параметров в момент вызова *(фактические* параметры), после чего

выполняются необходимые вычисления. Это называется передачей параметров *по значению:* в подпрограмму передаются значения нужных типов. Формальные переменные играют роль локальных переменных.

В списке параметров перед любой переменной разрешается указывать ключевое слово **var.** Тогда считается, что данный параметр будет передаваться *по имени:* то есть копирования значения не происходит, а вместо формального параметра подставляется имя переменной — фактического параметра. (Отсюда следует, что в качестве фактических параметров, подставляемых вместо формальных параметров с ключевым словом **var,** можно применять только имена переменных.)

Например:

```
procedure MyProc( X: Byte; var Y: Byte );
```

Параметр X будет передаваться по значению, параметр Y — по имени. При вызове

```
MyProc( 2+2, N );
```

операторы в теле процедуры MyProc будут обрабатывать локальную переменную X, имеющую значение 4 (2+2), и переменную Y, которая на самом деле является переменной N из другой части программы. На «физическом» уровне произойдет просто подстановка адреса переменной N. Использование слова var в списке параметров по способу действия аналогично использованию указателя.

При этом, если в теле процедуры значение переменной X будет меняться, то по окончании работы MyProc эта локальная переменная будет удалена из памяти (до следующего вызова процедуры), а вот если произойдет изменение значения переменной Y, то изменится и значение переменной N.

Это потенциально чревато ошибками, так как разработчик может не рассчитывать на то. что во время работы подпрограммы произойдет изменение значения одного из ее парамегров. В данном случае переменная Y является **не чем** иным, как указателем на переменную N, только описан он немного иначе: не с помощью операции ^, а как обычная переменная.

Следует избегать передачи параметров по имени за исключением случая, когда требуется передать в подпрограмму данные большого объема. Например, если описан тип

```
type TBigArray = array [1..100000] of string[50];
```

то передавать переменную этого типа по значению очень неэффективно, особенно если вызов подпрограммы происходит часто, потому что при этом требуется копировать большие объемы данных. То есть описание

```
procedure Sum( A: TBigArray );
```

неудачно в плане эффективности (хотя компилятор может самостоятельно выполнить передачу параметра по имени в процессе автоматической оптимизации кода).

Правильнее написать так:

```
procedure Sum( var A: TBigArray );
```

При этом надо проследить, чтобы изменения элементов массива A внутри процедуры Sum не происходило (если этого не требует логика ее работы).

Параметры-константы

Если некоторый параметр будет использоваться в теле подпрограммы только для считывания данных (фактически, как константа), лучше не рассчитывать на возможности компилятора, а подсказать ему об этом явно. В этом случае присваивание данному параметру нового значения станет недопустимым.

Подобная «подсказка» осуществляется указанием зарезервированного слова **const** в списке параметров, что позволяет организовать эффективную обработку соответствующего параметра, не беспокоясь о возможных изменениях его значения.

```
procedure Sum( const A: TBigArray ) ,-
```

Менять значение элементов переменной A в теле процедуры Sum теперь нельзя: компилятор сообщит об ошибке.

Параметры-результаты

В Паскале имеется еще одна возможность передачи параметра по имени — с помощью зарезервированного слова **out.** Такой параметр может использоваться внутри тела подпрограммы только для присваивания значения (но не для считывания данных). По смыслу использование слова **out** противоположно использованию слова **const.**

Если заголовок процедуры описан так:

```
procedure Sum( out A: TBigArray );
```

то в теле процедуры Sum можно указывать операторы присваивания

```
A[10000] := 'num172855';
```

но нельзя считывать значения элементов массива A:

```
x := A[2] * 2; // нельзя!
```

ПОДСКАЗКА Для получения результатов от подпрограммы всегда правильнее использовать функции. Процедуры, возвращающие результат через свои параметры, лучше применять только тогда, когда без этого не обойтись (например, параметры-результаты требуются при создании приложений COM и CORBA).

Преобразования сложных типов

Ранее рассматривалась возможность явного преобразования типов путем использования названия нужного типа как функции (Byte(300)). Паскаль допускает подобный способ преобразования не только для базовых, но и для любых других типов, определенных разработчиком.

ЗАМЕЧАНИЕ В Паскале не делается различий между базовыми типами, исходно существующими в языке, и типами, созданными программистами.

Потребность в этом хоть и редко, но все же возникает. Например:

```
type T1 = array[1..2] of Byte;
var A1: T1;
    A2: array[1..2] of Byte;
```

Переменные A1 и A2 относятся к разным типам, хотя физически (в оперативной памяти) они представлены одинаково. Поэтому переменную A2, обычный массив, можно преобразовать (привести) к типу T1:

```
T1(A2)
```

Такая запись аналогична простому обращению к имени массива A2, только при этом считается, что он имеет новый тип T1. К элементам массива нового типа можно обращаться, как и раньше, с помощью квадратных скобок:

```
T1(A2)[1]
```

Параметры без типов

Паскаль позволяет указывать в заголовке подпрограммы параметры без типов (но с предшествующим словом **const, var** или **out).**

```
procedure Sum( const A );
```

Переменная A в таком случае считается не имеющей никакого типа, и ее нельзя использовать ни в каких выражениях или операторах, не выполнив предварительно преобразование — приведение к нужному типу.

Например, в теле процедуры Sum переменную A можно использовать так:

```
X := T1 (A) [5] * 2,-
```

Передача строк фиксированной длины

Строки можно описывать, как уже говорилось ранее, двумя способами: просто как тип **string**, или как тип **string** с конкретной длиной строки в квадратных скобках. Однако указывать строки фиксированной длины в качестве параметров подпрограмм нельзя. Их сначала надо описать как новый тип.

```
procedure Sum( S: string[50] ); // неверно!
```

Правильно написать так:

```
type string50 = string[50];
procedure Sum( S: string50 );
```

Передача массивов в качестве параметров

Стандартный прием Паскаля состоит в описании массива как типа данных и указании этого типа для параметра подпрограммы. Однако таким способом удается обрабатывать только динамические массивы и статические массивы заранее заданной длины. Часто требуется, чтобы подпрограмма не ограничивалась конкретными размерами массива. Например, функция, вычисляющая сумму всех элементов массива, должна уметь получать в качестве параметра массив любой длины. Динами-

ческие массивы использовать не всегда удобно и не всегда правильно, так как они ухудшают эффективность работы программы.

В Паскале разрешается указывать в качестве параметров массив с неопределенными границами.

```
procedure Sum( A: array of Byte );
```

ВНИМАНИЕ Данное описание похоже на описание динамического массива, но не является таковым. Это просто форма записи параметра подпрограммы для передачи статического массива заранее не известной длины.

Теперь массив, описанный как

```
var Ar: array [1..25] of Byte;
```

можно без проблем передавать в процедуру Sum:

```
Sum(Ar);
```

При этом внутри тела подпрограммы действуют следующие правила:

- Нумерация элементов массива начинается с нуля (число элементов можно определить с помощью стандартной функции SizeOf);
- Функция High возвращает верхнюю границу массива (равную SizeOf()–1);
- Копирование массива одним оператором присваивания не разрешается;
- Вместо массива в подпрограмму может быть передана обычная переменная соответствующего типа. Внутри подпрограммы она будет представлена в виде массива из одного элемента (с индексом 0).

Передача значений как массива

Вместо указания в параметрах при обращении к подпрограмме имени переменной-массива можно указать непосредственно содержимое этого массива: список значений через запятую в квадратных скобках.

```
Sum( [1, 5, X, a+b*2] );
```

Передача массива вариантного типа

Массив, описанный как

```
array [1..MaxNum] of Variant;
```

может быть передан в качестве параметра в подпро]рамму, если этот параметр описан как **array of const:**

```
procedure Sum( A: array of const );
```

Внутри тела подпрограммы надо анализировать перед обработкой тип каждого элемента этого массива.

Способы вызова подпрограмм

Подпрограммы с точки зрения прикладного программиста вызываются всегда одинаково, но машинный код, который создается компилятором, для разных подпрограмм может отличаться. Это зависит от целей применения конкретной подпрограммы. Она может использоваться:

- в рамках разрабатываемой прикладной программы;
- как функция, доступная из динамической библиотеки .DLL;
- как процедура, вызываемая из внешних программ или из *Windows* и т. д.

Для каждого из таких нестандартных случаев вслед за заголовком подпрограммы (за точкой с запятой) должно следовать одно из ключевых слов.

Таблица 1.10. Ключевые слова в заголовке подпрограмм

Ключевое слово	Способ передачи параметров
pascal	Стандартный (параметры помещаются в стек)
register	Способ, применяемый по умолчанию. Аналогичен использованию ключевого слова pascal, но параметры передаются с помощью трех регистров процессора, а не помещаются в стек (область оперативной памяти), что обычно приводит к повышению быстродействия программы
cdecl	В соответствии с соглашениями компиляторов для языков программирования С и С++. Применяется, когда происходит обращение к динамическим библиотекам DLL, написанным на этих языках
stdcall	В соответствии с соглашениями Windows
safecall	Используется при работе с компонентными технологиями

При использовании ключевых слов **register** и **pascal** вычисление параметров выполняется слева направо и располагаются они в оперативной памяти перед вызовом подпрограммы в таком же порядке. При использовании ключевых слов **cdecL,** stdcall и safecall параметры располагаются в обратном порядке (справа налево).

```
procedure Sum( A: array of const ); stdcall;
```

Существует еще одно зарезервированное слово Паскаля, **forward,** которое, при указании вслед за заголовком, говорит компилятору о том, что в данном месте расположен только заголовок подпрограммы, а все ее описание находится в исходном тексте далее. Такое описание обычно применяют, если в тексте имеется несколько подпрограмм, которые вызывают друг друга по кругу. Например, из процедуры P1 вызывается процедура P2, а из процедуры P2 — процедура P1.

Паскаль требует, чтобы любой идентификатор, будь то переменная или подпрограмма, был предварительно, до первого своего использования, описан. В таком круговом процессе (пример условный, потому что приведенная схема может привести к бесконечной цепочке вызовов) процедура P2, вызываемая в теле процедуры P1, расположенной выше в исходном тексте, еще не описана, следовательно, обращаться к ней нельзя и компилятор сообщит, что обнаружен неопределенный идентификатор.

Неправильно:

```
procedure P1;
begin
P2;
end;

procedure P2;
begin
P1;
end;
```

Правильно:

```
procedure P2; forward;

procedure P1;
begin
P2;
end;

procedure P2;
begin
P1;
end;
```

Теперь компилятор знает, как выглядит заголовок процедуры P2, и может корректно сгенерировать машинный код для обращения к ней.

ПОДСКАЗКА Без таких запутанных круговых ссылок и ключевого слова forward всегда можно обойтись. Лучше не усложнять структуру программы его использованием.

Перегружаемые подпрограммы

Хотя в Паскале не допускается использование одинаковых названий для переменных, констант и других идентификаторов, для локальных переменных и подпрограмм делается исключение. Так как в Паскале предъявляются строгие требования к типам данных, обращаться к подпрограмме, формальные параметры которой имеют тип Integer, с фактическими параметрами, *имеющими* тип Real, нельзя. Однако при решении задачи подчас бывает необходимо, чтобы подпрограмма с одним и тем же именем работала с разными типами данных.

Здесь есть два способа действий: либо использовать данные типа Variant (что чревато ошибками преобразования, снижает общую эффективность программы и требует от разработчика повышенной бдительности), либо применить *перегружаемые* подпрограммы. Они отличаются от обычных подпрограмм тем, что имеют совпадающие имена, а различаются только типами аргументов. Чтобы указать компилятору, что конкретная подпрограмма — перегружаемая, надо вслед за ее заголовком указать зарезервированное слово **overload**.

При вызове такой подпрограммы компилятор по типам параметров автоматически определит, какую же подпрограмму конкретно надо использовать в данном месте.

```
procedure Ov1 ( X: Real ) ; overload;
begin
...
end;

procedure Ov1 ( X: Byte ) ; overload;
begin
...
end;

Ov1( 1 ); // вызывается процедура Ov1( X: Byte )
Ov1( 1.0 ); // вызывается процедура Ov1( X: Real )
```

Необходимое требование к перегружаемым процедурам состоит в том, чтобы списки параметров совпадали во всем, за исключением типов переменных.

Надо особенно осторожно использовать перегружаемые подпрограммы с параметрами по умолчанию. Например:

```
procedure Ov1( X: Byte; Y: Real - 1 ); overload;
begin
...
end;

procedure Ov1 ( X: Byte ) ; overload;
begin
...
end;
```

При вызове

```
Ov1( 1 )
```

компилятор не сможет понять, какую из двух процедур ему вызывать, и сообщит об ошибке.

Локальное описание

Сразу за заголовком подпрограммы следует локальное описание типов, переменных и констант, локальных для данной подпрограммы и существующих только в ее границах. Такое описание подчиняется обычным правилам Паскаля. В нем разрешается использовать слова type, var и const. Локальное описание может быть опущено.

Вложенные подпрограммы

Помимо обычных описаний, внутри подпрограммы допускается объявлять также *локальные подпрограммы,* к которым можно обращаться (вызывать) только из тела «родительской» подпрограммы. При этом локальная подпрограмма может свободно

обращаться *к* любым локальным описаниям (переменным, типам), которые расположены до описания данной подпрограммы.

Такая возможность полезна, когда во время кодирования подпрограмма начинает непредвиденно разрастаться. Ее приходится делить на более мелкие фрагменты, которые в то же время желательно не выносить за пределы текущей подпрограммы, чтобы иметь возможность пользоваться ранее сделанными локальными описаниями.

Например:

```
procedure Demo;
type T1 = array [1..2] of Real;
var D, D1: T1;
    S: Real;

  procedure InDemo;
  begin
  D1 := D;
  S := D1[1] + D1[2]
  end;

begin
...
end;
```

Уровень вложенности локальных подпрограмм неограничен.

Тело подпрограммы

Тело подпрограммы заключается в логические скобки begin/end. В них располагаются только операторы и вызовы других подпрограмм.

Возврат значений из функции

Если описывается функция, то в ее теле надо определить, как значение будет возвращено в вызываемую программу. Для этого есть два способа.

1. Соответствующее значение присваивается переменной, имя которой совпадает с названием функции.

   ```
   function Sum( A, B: Integer ): Integer;
   begin
   Sum := A + B
   end;
   ```

   Имя функции для возврата значения разрешается указывать только в левой части оператора присваивания.

2. Соответствующее значение присваивается специальной локальной переменной Result (эту переменную описывать не надо).

```
function Sum( A, B: Integer ): Integer;
begin
Result := A + B
end;
```

Вызов подпрограммы

Когда в тексте программы указывается имя ранее описанной подпрограммы с фактическими параметрами, то выполнение основной части программы останавливается и управление передается подпрограмме, до тех пор пока в ходе работы не будет достигнут ее конец (зарезервированное слово **end).** После этого управление передается обратно в программу (или другую подпрограмму), вызывавшую данную подпрограмму.

Параметры должны следовать в "строгом соответствии с порядком их описания в заголовке подпрограммы. Типы их так же должны точно совпадать с указанными. Если параметров у подпрограммы нет, то записывается только название подпрограммы и следующая за ней точка с запятой.

```
Demo;
```

Функции, возвращающие значение, могут использоваться так же, как и процедуры. Например, описанную выше функцию Sum можно вызывать так:

```
X := Sum( 2,2 );
```

а можно и так:

```
Sum( 2,2 );
```

Б последнем случае значение, возвращаемое функцией, просто теряется.

Процедуры, играющие роль операторов

С развитием языка Паскаль в рамках среды *Delphi 7в* него добавлялось множество новых полезных возможностей, нередко заимствованных из других языков программирования. Эти возможности вводились в Паскаль не в виде новых операторов, что нарушило бы идеологию языка, а в виде стандартных подпрограмм, которые, хотя и не выделяются цветом наравне с другими ключевыми словами, тем не менее, фактически являются таковыми. И реализуются подобные подпрограммы не в виде обращений к машинному коду, хранимому в программной библиотеке. Компилятор не добавляет в генерируемый код ссылку, а превращает данную «процедуру», подобно обычным операторам, в небольшой набор машинных инструкций (а иногда и в одну такую инструкцию).

Одна из таких весьма полезных процедур — Exit (без параметров).

```
Exit;
```

При ее выполнении происходит немедленное завершение текущей подпрограммы и передача управления вызывающей программе. Такая возможность часто требуется, когда логика, реализуемая в подпрограмме, достаточно сложна и организовать линейный выход из подпрограммы (по достижении ее конца) затруднительно.

Полезна подпрограмма Exit и в тех случаях, когда при определенных значениях параметров вычислить значение функции удается сразу. Например, если при вычислении факториала числа параметр равен 1, можно сразу определить возвращаемое значение, также равное 1, и покинуть подпрограмму.

Процедурные типы

Процедуры в Паскале разрешается использовать при описании новых типов. На основании таких типов, обладающих равными правами с другими типами, можно описывать переменные, что позволяет передавать подпрограммы в качестве параметров. Подобный прием в некоторых случаях помогает эффективно решить сложную задачу, с трудом поддающуюся кодированию обычным способом, однако пользоваться им нежелательно, по крайней мере начинающим разработчикам.

Примеры:

```
type TSumFun = function( A, B: Integer ): integer;
     TEmptyProc = procedure;
     TMyProc = procedure ( X: Real ) ;

var SumP: TSumFun;
    EmptyProc : TEmptyProc;
    MyPr : TMyProc;
```

ВНИМАНИЕ При описании процедурных типов названия подпрограмм не указываются. Приводится только основная схема заголовка.

Далее, если в тексте описана подпрограмма, то ее можно присвоить переменной соответствующего типа:

```
SumP  := Sum;
EmptyProc  := Demo;
```

ВНИМАНИЕ В правой части оператора присваивания указывается только название подпрограммы без параметров.

Для обращения к нужной подпрограмме теперь можно указывать не только ее имя, но и имя переменной, хранящей «описание» этой подпрограммы:

```
X := SumP( 2,2 ),-
```

Реально здесь произойдет вызов функции Sum с аргументами 2 и 2.

ЗАМЕЧАНИЕ Переменные процедурных типов являются указателями. В них хранится только адрес ячейки оперативной памяти, где начинается соответствующая процедура.

Подпрограмму можно передавать в другую подпрограмму как параметр. Это удобно, когда над аргументами надо выполнять различные сложные действия в зависимости

от некоторых условий. В приведенном ниже примере функция MathAction будет вычислять или сумму, или разность параметров, в зависимости от того, какая функция указана в качестве параметра.

```
type TMathFun = function( X,Y: Integer ): Integer;

function Add( X,Y: Integer ): Integer;
begin
Add := X+Y
end;

function Sub( X,Y: Integer ): Integer;
begin
Sub := X-Y
end;

function MathAction( X,Y: Integer;
                     Proc: TMathFun ): Integer;
begin
Result := Proc( X,Y )
end;
```

Теперь к функции MathAction можно обращаться так:

```
WriteLn( MathAction( 10, 4, Add ) );
```

при этом будет напечатано число 14, или так:

```
WriteLn( MathAction( 10, 4, Sub ) );
```

В этом случае будет напечатано число 6.

Чтобы проверить, содержит ли переменная процедурного типа описание конкретной подпрограммы, используется стандартная функция Assigned(), которая в качестве аргумента получает процедурную переменную, а возвращает значение типа Boolean (если оно равно True, то переменная имеет корректное значение).

С переменными, хранящими указатели на функции, надо обращаться осторожно. Переменной, хранящей указатель на процедуру, можно присвоить значение такой же переменной. В то же время, если в левой части оператора присваивания стоит переменная типа, совпадающего с типом значения, возвращаемого функцией, то произойдет вызов функции. Когда надо выполнять копирование указателей, а когда — вызывать функции, решает компилятор в зависимости от контекста.

```
type TP = procedure;
     TF = function: integer,-

var p1, p2: TP;
    f1, f2: TF;
    N: Integer;
```

```
procedure A;
begin
end;

function B: integer;
begin
B := 0
end;

p2 := A;
p1 := p2;
// произойдет копирование указателя

f2 := B;
f1 := f2;
// произойдет копирование указателя

N := f2;
// произойдет вызов функции B
// и запись значения в переменную N
```

Указатели на подпрограммы

Хотя переменные, описанные как процедурные типы, фактически являются указателями, от программиста эта их особенность скрыта. Но Паскаль разрешает также явно описывать переменные-указатели на подпрограммы. Для этого в языке введен особый тип данных Pointer, представляющий собой указатель на информацию, не имеющую конкретного типа. Получение адреса начала подпрограммы выполняется с помощью операции @, так же, как и для получения адреса любых других данных.

```
var X: Pointer;
...
X := @MyProcedure;
```

Подобным способом в программе определяются и процедурные константы-указатели:

```
const P: Pointer = @MyFunction;
```

Операторы

С помощью оператора присваивания можно написать простые программы, преимущественно ориентированные на математические вычисления, но для создания приложений, реализующих сложную алгоритмическую логику, нужны средства

управления ходом работы программы: изменения порядка выполнения операторов в зависимости от различных условий и эффективной реализации часто повторяющихся вычислений.

Условный оператор

Условия

Один из важнейших операторов Паскаля — условный оператор. Он позволяет изменить порядок выполнения операторов в зависимости от некоторого условия, представляющего собой логическое выражение типа Boolean. Если это значение равно True, то выполняется одна группа операторов, если оно равно False, то выполняется другая группа операторов или не выполняется ничего.

Условия представляют собой логические выражения. В них происходит сравнение значений выражений, вызов функций, возвращающих значение типа Boolean, и комбинирование этих значений с помощью логических операций. Ниже приведены основные операции сравнения данных.

Таблица 1.11. *Основные операции сравнения данных*

Знак операции	Название операции
=	Равно
o	Не равно
>	Больше
<	Меньше
>=	Больше или равно
<-	Меньше или равно

Если используются логические операции or, and и так далее, то связываемые ими проверки заключаются в круглые скобки.

```
X > 5
(I >= 1) and (I <= 10)
(a+5 o b) or BoolFunc
Alf = 3
```

Для некоторых типов данных в Паскале имеются дополнительные операции, позволяющие сформировать более сложные условия. В частности, для множеств определена операция **in** (зарезервированное слово), которая проверяет, входит ли конкретное значение в множество:

```
X := [2,4,6,8,10];
```

Выражение 2 in X имеет значение True.

Выражение 5 in X имеет значение False.

Такой способ очень удобен тем, что позволяет выполнить проверки более наглядно и компактно.

Например, вместо того, чтобы писать

```
(I >= 1) and (I <= 10)
```

можно использовать операцию **in:**

```
I in [1..10]
```

Выполняются подобные проверки тоже значительно эффективнее.

Оператор if ... then ...

Условный оператор записывается в такой форме:

```
if условие then действие;
```

Слова !f (если) и **then** (то) — зарезервированные.

Действие выполняется только в том случае, если значение условия равно True. В противном случае ничего не происходит. Действие — это любой оператор Паскаля, или группа операторов, взятых в логические скобки **begin/end**, или вызов подпрограммы.

Например:

```
if X > 0 then X := 0;
if (N = 2) or (N = 3) then CallProc;
if Sim in ['a'..'я'] then
   begin
   WriteLn( Sim );
   Sim := Chr( 100 )
   end;
```

Оператор if ... then ... else ...

Нередко требуется выполнить определенные действия и в том случае, когда проверяемое условие ложно. Для этого можно использовать другую форму условного оператора:

```
if условие
then действие-1
else действие-2;
```

Действие-1 будет выполнено, если условие истинно (равно true), действие-2 выполняется, если условие ложно.

```
if X > 0 then X := -1 else X := +1;
```

ЗАМЕЧАНИЕ Перед ключевым словом else (иначе) точка с запятой не ставится.

Напишем небольшую программу, которая принимает ввод числа с клавиатуры и сообщает, четное оно или нет. Проверку на четность можно выполнить двумя способами: использовать стандартную функцию Odd(), которая возвращает значение True, если ее аргумент нечетный, или проверить, равен ли нулю остаток от деления числа на 2 (операция **mod).**

```
program Project1;
{$APPTYPE CONSOLE}
uses sysutils;
var N: Integer;
begin
ReadLn( N );
if Odd(N) then WriteLn('Число нечетно')
   else WriteLn('Число четно')
end.
```

Вложенные условия

Условные операторы могут быть неограниченно вложены друг в друга. Рассмотрим следующий пример. Надо ввести с клавиатуры два числа, и если их сумма меньше чем 100, то напечатать большее число, а в противном случае — напечатать меньшее число.

```
program Project1;
{$APPTYPE CONSOLE)
uses sysutils;
var X, Y: Integer;
begin
ReadLn( X, Y );
// если сумма меньше ста, то
if X+Y < 100 then
   // определить большее число
   if X > Y then Writeln(X)
   else WriteLn(Y)
// иначе определить меньшее число
else if X < Y then Writeln(X)
   else WriteLn(Y)
end;
```

При решении данной задачи использованы вложенные условные операторы,

В некоторых случаях бывает сложно разобраться, в каком порядке такие вложенные операторы будут выполняться. Например:

```
if a > b then
if a > e then c := 1
else c := 2;
```

Здесь непонятно, в каком случае выполняется оператор c := 2: когда ложно условие a > b или когда оно истинно, но ложно условие a > e.

В Паскале действует простой принцип: часть условного оператора **else** относится к ближайшему if. В данном случае, часть **else** c := 2 относится к ближайшему оператору **if a > e then,** а не к оператору if a > b **then,** тем самым правильно второе толкование.

Надо стараться записывать **подобные** вложенные операторы максимально наглядно, используя отступы из пробелов, избегая запутанных конструкций и не допуская слишком большого уровня вложенности.

Если в приведенном примере требуется, чтобы оператор c := 2 выполнялся, когда ложно условие a > b, то промежуточный условный оператор надо вынести в логический блок, чтобы он не влиял на порядок выполнения условных частей **then и else:**

```
if a > b then
   begin
   if a > e then c := 1
   end
else c := 2;
```

Оператор выбора

Когда требуется осуществить проверку множества условий, например выполнить один из пяти операторов в зависимости от того, чему равно значение переменной X, приходится записывать цепочки условных операторов наподобие следующей:

```
if X = 1 then a := 1 else
if X - 2 then a := 2 else
if (X = 3) or (X = 4) then a := 3 else
if X = 5 then a := 4 else
if (X = 6) or (X in [8..100]) then a := 5
else a := 0;
```

Подобная запись довольно громоздка и преобразовывается компилятором в не очень эффективный машинный код. В Паскале имеется более удобный оператор выбора **case,** позволяющий наглядно описать выбор выполняемого оператора или группы операторов в зависимости от ряда условий.

```
case выражение of
список-условий-1: действие-1;
...
список-условий-n: действие-n;
else действие-n+1;
end;
```

Тип выражения может быть одним из стандартных типов: целым числом, псречислимым типом, символьным типом и так далее. Список условий может содержать произвольные выражения, состоящие из констант и имеющие подходящий тип. В этом списке допускается использовать как обычные константы, так и символы и диапазоны значений. Результат выражения будет поочередно сравниваться с каждым из значений в списках, и при первом совпадении будет выполнено соответствующее действие (оператор или группа операторов, взятых в логические скобки **begin/end**), а все оставшиеся действия будут пропущены. В случае, когда ни одного совпадения результата выражения с заданными значениями не произошло, выпол-

няется действие, указанное за словом **else** (если оно имеется), или не выполняется ничего, если слово **else** внутри оператора выбора отсутствует.

Переписанный с помощью данного оператора вышеприведенный пример будет выглядеть так:

```
case X of
   1: a := 1,-
   2: a := 2;
   3, 4: a := 3;
   5 г a := 4;
   6, 8..100: a := 5;
else a := O
end;
```

Условное описание

Использовать оператор **case** в Паскале разрешается также в разделе описания переменных, когда создается новый тип на основе записи **(record).** Внутри записи допустимо указывать несколько форм ее представления, и описываются все эти формы «встиле» оператора **case** (это только правило описания, способ нужного представления, понятный компилятору и разработчику, а не реальный машинный код).

Например, программисту требуется использовать два типа данных: одну запись с полями:

```
Name: string[50];
Num: integer;
```

а другую — с полями:

```
TeamName: TFootballName;
Win: Boolean;
Gol: Byte;
```

При этом программист точно знает, что он не будет применять эти группы полей одновременно друг с другом. Тогда вместо описания двух новых типов программист может описать один, только разделив внутреннюю структуру записи на две:

```
type MyType =
   record
   case MyVar: Boolean of
   True: (Name: string[50];
   Num:integer);
   False: (TeamName: TFootballName;
   Win: Boolean;
   Gol: Byte);
   end;
```

Подобная оригинальная запись расшифровывается следующим образом. Слово **case** говорит, что далее следует несколько форм представления типа (записи) MyType. Сколько? Это определяется следующим за ключевым словом **case** описанием MyVar:

Boolean. Имя переменной MyVar реально больше нигде в программе не используется, это просто требование Паскаля. А тип этой переменной определяет возможный диапазон форм представления записи. Тип Boolean имеет два значения True и False, значит форм представления будет две. Можно было указать, например, тип Byte и сформировать не два, а пять или двадцать пять описаний внутри записи.

Далее следует последовательность значений указанного типа, и для каждого из них в круглых скобках приводится очередной вариант внутренней структуры.

ВНИМАНИЕ Закрывает описание case не собственное ключевое слово end, а слово end, завершающее описание всей записи.

Обращаться к полям записи в тексте программы можно как обычно:

```
var R: MyType;
...
R.Name := 'test';
R.Win := true;
```

Конечно, одновременно обращаться к полям из разных описаний не следует: ведь создаются подобные раздельные описания как раз исходя из того, что совместно использоваться они не будут.

Есть еще одна область применения подобных условных описаний. В памяти для них отведена одна общая область, поэтому одно и то же представление можно трактовать по-разному, в зависимости от размера (длины в байтах) значения конкретного типа. Например, поскольку тип Integer соответствует данным длиной в 4 байта, доступ к его отдельным байтам можно организовать так:

```
type IntDetail =
   record
   case V: Byte of
   1: (I: Integer);
   2: (B1, B2, B3, B4: Byte)
   end;
var ID: IntDetail;
begin
ID.I := 55555;
WriteLn(ID.B1, ' ', ID.B2, ' ', ID.B3, ' ', ID.B4),-
ID.B1 := $FF;
...
```

ПОДСКАЗКА Возможность розного представления структуры записи сохраняется в Паскале с времен его первых версий, когда остро стояли проблемы экономии памяти и приходилось прибегать к всевозможным ухищрениям. Сегодня такие возможности Паскаля неактуальны и только усложняют понимаемость исходных текстов, поэтому желательно обходиться без них.

Оператор цикла

Б программировании постоянно возникают задачи, требующие для своего решения многократного повторения одной и той же последовательности действий или однообразной обработки однородных объемов информации (массивов). Например, когда требуется определить сумму всех элементов массива, найти его элемент с максимальным значением, вычислить квадраты всех элементов и так далее.

В Паскале имеется несколько операторов, позволяющих организовать подобную работу наглядно и эффективно. Операторы, предназначенные для многократного (циклического) выполнения заданной последовательности команд, называются операторами *цикла*. Они всегда имеют *заголовок* цикла, определяющий число повторений, и *тело* цикла — повторяемое действие.

Оператор цикла записывается так:

```
for переменная-счетчик := выражение-1 to выражение-2 do
повторяемое-действие;
```

Переменная-счетчик должна быть объявлена перед логическим блоком, в котором этот оператор расположен, то есть если оператор цикла используется внутри подпрограммы, то в качестве счетчика должна выступать локальная переменная.

ЗАМЕЧАНИЕ Это сделано с целью максимально оптимизировать быстродействие оператора цикла, ориентированного на многократные интенсивные вычисления. Переменные-счетчики представляют собой, как правило, быстрые регистры процессора, а не обычные ячейки оперативной памяти.

Первоначально, перед первым выполнением тела цикла, счетчик получает значение, равное результату вычисления выражения-1. Переменная-счетчик должна обязательно относиться к одному из перечислимых типов. Выражение-2 определяет конечное значение, по достижении которого счетчиком тело цикла будет выполнено в последний раз.

После очередного выполнения тела цикла счетчик принимает значение, следующее за текущим (точнее, это будет значение, равное Succ() от текущего). Затем выполняется проверка, не превышено ли конечное значение выражения-2 (значения и начального, и конечного выражений вычисляются только один раз *перед* первым выполнением тела цикла). Если оно превышено, то работа оператора цикла заканчивается. В противном случае тело цикла выполняется еще раз.

Рассмотрим пример подпрограммы, вычисляющей факториал числа N. Для этого требуется последовательно перемножить все числа от 1 до N. Наиболее компактно такие вычисления реализуются с помощью оператора цикла.

```
function Factorial( N: integer ]: integer;
var i, R: integer;
begin
if N <= 0 then
   begin
```

```
    Result := 0;
    Exit
    end;
R := 1;
for i := 1 to N do
    R := R * i;
Result := R;
end;
```

Счетчик i будет последовательно принимать значения от 1 до значения параметра N, а в переменной R будет храниться промежуточное значение результата. Обратите внимание на условный оператор, который проверяет корректность значения параметра (оно должно быть положительным числом). Если значение параметра N меньше или равно нулю, то значение факториала принимается равным нулю без вычислений, после чего происходит выход из подпрограммы с помощью процедуры Exit.

В Паскале имеется еще одна форма записи оператора цикла. Она предназначена для использования в случаях, когда счетчик должен последовательно принимать не возрастающие, а убывающие значения.

```
for счетчик := выражение-1 downto выражение-2 do
тело-цикла;
```

Значение счетчика в этом случае будет не увеличиваться, а уменьшаться (как при вызове процедуры Pred()). Соответственно, необходимо, чтобы значение выражения-2 было меньше значения выражения-1. Тип счетчика — не обязательно целое число. Это может быть, в частности, перечислимый тип.

```
type TLoop = (MinVal, AveVal, MaxVal);
var L: TLoop;
begin
for L := MinVal to MaxVal do
...
```

Условный оператор цикла

Оператор **for** удобно применять, когда заранее известно, сколько раз требуется выполнить тело цикла (например, при обработке всех элементов массива). Но в большом количестве задач цикл приходится выполнять *неизвестное* число раз. Это происходит, если вычисление значения функции заканчивается по достижении заданной точности, если выполнение операторов зависит от информации, введенной пользователем, если надо найти в массиве элемент с конкретным значением и так далее. В таких случаях правильнее использовать условный оператор цикла:

```
while условие do
тело-цикла;
```

Тело цикла будет выполняться, пока истинно условие (логическое выражение, возвращающее значение типа Boolean). В отличие от оператора for условие окончания цикла каждый раз вычисляется заново. Это позволяет гибко управлять числом повторений цикла.

ВНИМАНИЕ Если перед выполнением оператора while значение условия равно False, тело цикла не будет выполнено ни разу.

Допустим, надо написать программу, которая вычисляет факториал вводимого с клавиатуры числа, но делает это не один раз, а многократно — до тех пор, пока человек не введет число 0, служащее условным признаком завершения работы программы. Используя ранее подготовленную подпрограмму Factorial, такую программу можно записать следующим образом:

```
program Project1;
{$APPTYPE CONSOLE}
uses sysutils;
// здесь следует
// описание подпрограммы Factorial
var N: integer;
begin
N := 1,-
while N <> 0 do
   begin
   ReadLn(N);
   WriteLn( Factorial(N)  );
   end;
end.
```

При выполнении тела цикла вводится начальное значение для вычисления факториала и печатается результат вызова функции Factorial(). Цикл повторяется до тех пор, пока введенное значение не окажется равным нулю, после чего работа программы завершится. Первый оператор присваивания N := \ нужен, чтобы войти в условный цикл (так как начальное значение переменной N не определено).

ПОДСКАЗКА Надо внимательно следить за вычислением условия завершения. Если в случае с оператором for число повторений цикла известно компилятору заранее, то в случае с оператором while контроль завершения цикла полностью возлагается на программиста. Иногда оператор while может повторять тело цикла бесконечное число раз, что приводит к *зацикливанию* и «зависанию» программы.

Условный оператор повторения

Исторически, в Паскале имеется еще одна форма условного оператора цикла, отличающаяся от формы записи оператора **while**. Это отличие состоит, во-первых, в том, что проверка условия выполняется не в начале цикла, а в его конце (что гарантирует как минимум однократное выполнение тела цикла), а во-вторых, в том, что завершение цикла происходит, когда условное выражение равно не False, а True (эта особенность часто вызывает ошибки у начинающих программистов).

```
repeat
тело-цикла
until условие;
```

ВНИМАНИЕ В отличие от всех остальных операторов Паскаля при использовании оператора repeat тело цикла, состоящее из нескольких команд, заключать в логические скобки begin/end не требуется. Компилятор определяет границы тела цикла по ключевым словам repeat/until.

С помощью данного оператора можно переписать программу вычисления факториала так:

```
begin
repeat
   ReadLn(N);
   WriteLn( Factorial(N) );
until N = 0,-
end.
```

Условие завершения цикла изменилось на противоположное — теперь для его окончания требуется, чтобы значение выражения N = 0 стало равно True (то есть, чтобы значение переменной N стало равным нулю), а лишний оператор инициализации переменной N не нужен.

Команда прерывания цикла

Условный оператор цикла позволяет остановить выполнение тела цикла, только когда все операторы, входящие в него, выполнены и достигнута проверка условия окончания. Такой подход иногда неудобен, особенно если тело цикла представляет собой длинную последовательность операторов и необходимость завершения цикла выясняется в середине этой последовательности. Для немедленного завершения текущего оператора цикла можно использовать подпрограмму Break без параметров (это подпрограмма, играющая роль оператора). Полезна данная команда и при использовании оператора **for,** например, когда в массиве с известными границами найдено нужное значение и дальнейшие вычисления выполнять не надо.

Например, если в строке S требуется найти номер первого пробела, можно применить следующие операторы:

```
N := 0;
for i := 1 to Length(S) do
   if S[i] = ' ' then
      begin
      N := i;
      Break;
      end;
if N > 0 then ...
```

В переменной N хранится номер подходящего символа (первоначально — 0). В цикле выполняется проверка каждого символа строки, при обнаружении пробела происходит запоминание номера символа и прерывание выполнения тела цикла. Затем значение переменной N сравнивается с нулем, чтобы определить, был ли найден нужный символ.

ЗАМЕЧАНИЕ В системе Delphi 7 имеется стандартная функция Pos(), которая, получая в качестве параметров подстроку и проверяемую строку, возвращает номер вхождения подстроки в строку или ноль, если совпадения не найдено:

```
Pos(' ', S)
```

Команда продолжения цикла

В Паскале имеется команда, по своему действию противоположная команде прерывания цикла. Она позволяет немедленно продолжить выполнение цикла, пропустив все оставшиеся операторы в теле цикла. Эта команда (подпрограмма без параметров, играющая роль оператора) записывается так:

```
Continue;
```

С ее помощью предыдущий пример можно записать следующим образом:

```
N := 0;
for i := 1 to Length(S) do
  begin
  if S[i] <> ' ' then Continue;
  N := i;
  Break;
  end;
if N > 0 then ...
```

При очередном выполнении тела цикла сначала произойдет проверка текущего символа на равенство пробелу, и если это не пробел, то выполнится команда продолжения цикла — все последующие операторы будут пропущены, а счетчик примет новое значение.

Вложенные циклы

При решении некоторых задач возникает потребность в организации вложенных циклов. Например, при анализе двумерного массива требуется выполнять цикл как по первому, так и по второму измерениям. В таких случаях используют вложенные циклы. Процедуры Break и Continue всегда воздействуют только на ближайший оператор цикла, поэтому прекратить выполнение всех циклов с их помощью невозможно.

Например, требуется написать программу, которая печатает все целые числа, сумма квадратов которых равна заданному числу. Проще всего сделать это с помощью двух вложенных циклов, которые последовательно перебирают все возможные

значения, а в теле внутреннего цикла проверяется, не равна ли сумма квадратов значений счетчиков заданной величине.

```
program Project1;
{$APPTYPE CONSOLE}
uses sysutils;
var i,j,N: integer;
begin
ReadLn(N);
for i := 1 to N div 2 do
for j := 1 to N div 2 do
   if i*i + j*j = N then
      WriteLn(i, ' , ' , jI;
end.
```

В каждом из циклов рассматривается диапазон значении от 3 до половины величины введенного числа (потому что сумма квадратов половин числа заведомо больше или равна этому числу). Можно найти дополнительные способы улучшить этот код, в частности, вынести вычисление произведения i*i из тела вложенного цикла, потому что это произведение не имеет смысла многократно вычислять во вложенном цикле, где оно всегда будет иметь одно и то же значение, а операция умножения для процессора достаточно дорогая (медленная).

Оператор перехода

Помимо условного оператора и операторов цикла, в Паскале имеется еще один оператор, позволяющий изменять последовательность выполнения программы. Это оператор перехода, который записывается так:

```
goto метка;
```

В качестве метки может выступать любой допустимый идентификатор или число в диапазоне от 0 до 9999. Для того чтобы метку можно было использовать в программе, ее надо предварительно объявить в разделе описания. Делается это не с помощью ключевого слова var, а в следующей форме:

```
label метка;
```

или

```
label список-меток;
```

Пока метка не объявлена, использовать ее в тексте программы нельзя.

Когда выполняется оператор перехода, то управление передается на оператор, следующий за меткой и отделенный от нее двоеточием:

```
flOto L;
Y ;= X;
L: X := 5;
```

При выполнении этой последовательности оператор присваивания Y := X пропускается.

Возможна передача управления на оператор, предшествующий оператору перехода:

```
var X: integer;
label M1;
begin
X := 0;
M1:
Inc(X);
if X < 10 then goto M1;
...
```

С помощью условного оператора и оператора перехода таким образом моделируется работа оператора цикла — переменная X играет роль счетчика.

Переходить на метку, расположенную вне подпрограммы, не разрешается. Нельзя также передавать управление на метку, расположенную внутри подпрограммы или цикла (в теле цикла).

Неверно:

```
var X: Integer;
procedure A;
label La;
begin
La: X := 1
end;
begin
goto La;
```

Неверно:

```
goto L;
for i := 1 to 10 do
begin
L: N := N + i;
...
```

ПОДСКАЗКА Применение оператора перехода считается плохим стилем программирования, так кок усложняет понимаемость исходных текстов программы. Профессиональные разработчики практически никогда не используют этот оператор. Математически доказано, что любой алгоритм можно закодировать только с помощью условного оператора, оператора присваивания и цикла. При этом структура программы остается линейной: все операторы в ней выполняются в одном направлении, сверху вниз, что делает ее простой для разбора.

Единственный случай, когда применение оператора goto может быть оправдано, — это выход из нескольких (двух и более) вложенных циклов, что невозможно сделать иными способами, с помощью процедуры Break.

Структура модуля

Ранее была рассмотрена структура простой консольной программы, содержащей в одном файле весь исходный текст. Полноценные приложения *Windows* строятся по другому принципу. В их главной части с расширением .DPR хранится только вызов нескольких команд, открывающих главное окно, а также выполняющих завершающие действия. Вся остальная логика содержится в файлах, хранящих описание дополнительных подключаемых модулей.

Каждый модуль имеет жестко заданную структуру, которая обычно автоматически генерируется системой *Delphi 7* при его создании. Модуль состоит из четырех частей: интерфейсной части, части реализации (обязательные части), части инициализации и части завершения (необязательные части). Сначала указывается заголовок модуля - ключевое слово **Unit**, за ним произвольное название модуля (оно должно совпадать с названием файла, в котором модуль хранится) и точка с запятой:

```
Unit TestUnit;
```

(Данный модуль будет храниться в файле testunit.pas.)

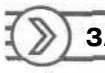
Интерфейсная часть описывает информацию, которая доступна из других частей программы: из других модулей и главной части. Часть реализации описывает информацию, которая недоступна из других модулей. Подобное разделение модуля на части позволяет создавать и распространять модули в откомпилированном виде (расширение .DCU), прикладывая к ним только описание интерфейсной части (наличия исходных текстов модуля, если имеется файл .DCU, не требуется). При этом внести изменения в такой модуль нельзя, а исходный код, реализующий описанные в интерфейсной части возможности, недоступен. Такой подход, во-первых, позволяет повторно использовать ранее написанные для других программ и уже отлаженные модули, во-вторых, разграничивает доступ к модулю нескольких программистов, в-третьих, позволяет разбивать программу на набор логически независимых модулей.

ЗАМЕЧАНИЕ Модули, применяемые при создании программ с помощью системы Delphi 7, могут быть созданы прикладным разработчиком, а могут быть стандартными: входящими в поставку Delphi 7 и включающими в себя множество самых разных полезных возможностей. К стандартным относится, в частности, модуль Sysfem, который содержит основные и постоянно встречающиеся подпрограммы Паскаля. Его не требуется подключать и явно указывать с помощью слова uses — он считается исходно подключенным к каждому модулю автоматически.

Интерфейсная часть всегда идет первой и начинается с ключевого слова **interface**, а часть реализации начинается с ключевого слова **implementation:**

```
Unit TestUnit;
interface
implementation
```

После заголовков этих частей можно дополнительно указать модули, подключаемые к данному модулю, с помощью ключевого слова uses, за которым следует список модулей через запятую, а в конце списка ставится точка с запятой.



ЗАМЕЧАНИЕ Модули, подключаемые в интерфейсной части, доступны из любого места модуля, а модули, подключаемые в части реализации, — во всем модуле за исключением интерфейсной части.

Части инициализации и части завершения необязательны. Указанные в них действия выполняются, соответственно, в самом начале и в самом конце работы программы и только один раз.

Если модулей в программе несколько, то последовательность выполнения их частей инициализации соответствует порядку их описания в модуле, где они подключаются с помощью ключевого слова uses, а последовательность выполнения частей завершения ей противоположна.

Например, если имеются модули A и B и в главной программе они подключаются так:

```
uses B, A;
```

то первой выполнится часть инициализации модуля B, а второй — часть инициализации модуля A. Когда программа завершит свою работу, первой выполнится завершающая часть модуля A, а второй — завершающая часть модуля B.

Часть инициализации начинается с ключевого слова

```
initialization
```

часть завершения — с ключевого слова

```
finalization
```

В конце модуля всегда ставится слово end и точка.

В самом общем случае структура пустого модуля будет такой:

```
Unit имя-модуля;
interface uses список-модулей;
Implementation uses список-модулей;
initialization
finalization
end.
```

ЗАМЕЧАНИЕ Если в модуль вносятся изменения, то при выполнении компиляции среда Delphi 7 автоматически проверяет все взаимосвязи между модулями с помощью ключевого слова uses и при необходимости выполняет также компиляцию других модулей, связанных с измененным.

Классы и объекты

Объект — основа Паскаля

До сих пор мы рассматривали типы данных, которые существовали в языках программирования еще тридцать лет назад. С их помощью можно разработать очень большую программу, однако потребует это значительных усилий группы профессиональных программистов. А в одиночку, используя только числа, строки, массивы и записи, можно создать программу объемом в лучшем случае в несколько тысяч строк исходного текста. Это является своеобразным пределом возможностей человека. Дело в том, что структура данных исходного алгоритма дробится на элементарные, слишком маленькие и явно не связанные друг с другом и с программным кодом частички. Они связаны только через операторы присваивания, разбросанные по разным модулям. По мере роста объема исходных текстов корректно обрабатывать переменные, не затрагивая уже нормально функционирующие части программы, становится невозможно.

В 80-х годах стали появляться первые коммерческие системы разработки, в которых была реализована новая парадигма программирования, так называемый *объектный подход*, что позволило резко повысить производительность труда программистов. Подход был основан на понятии *объекта*, типа данных, в котором сочетаются как *свойства*, сгруппированные данные (пример — поля в записи), так и *методы* их обработки (подпрограммы).

Фактически объект стал отражать реальные и даже абстрактные понятия окружающего мира. Например, автомобиль характеризуется такими свойствами, как марка, тип двигателя, наличие колес и руля, а файл — названием и размером. «Методы» автомобиля определяют его способность двигаться в нужном направлении в соответствии со значениями своих свойств: объемом бензина в баке, углом поворота руля. Из файла можно считывать данные, менять их и записывать обратно.

Благодаря этому теперь удается выполнять проектирование программ, основываясь на понятии объекта, что значительно проще и быстрее, чем раньше. Работать с привычными понятиями человеку легче, нежели с абстрактными числами. При этом специалистам удалось выделить большой набор объектов, которые нужны при создании самых разных программ. Эти объекты используются повторно, без расходования времени на их программирование. Именно такой подход и реализован в среде *Delphi 7*.

Понятие класса

В Паскале имеется четкое разграничение между понятиями *объекта* и *класса*. Класс — это тип данных (как `Integer` или `TMyRecord`), а объект — конкретный существующий в памяти компьютера *экземпляр* класса, переменная соответствующего типа. В ранних версиях Паскаля существовала некоторая терминологическая путаница, потому что первая реализация объектного подхода использовала для описания

объектного типа данных ключевое слово **object,** и в то же время объектом назывались экземпляры этого типа. Применять слово **object** можно и сейчас, однако подобная возможность поддерживается только для совместимости со старыми версиями системы *Delphi.* Вместо ключевого слова **object** правильно использовать ключевое слово **class.**

Классы имеют поля (как тип данных **record),** свойства (напоминающие поля, но имеющие дополнительные описатели, определяющие механизмы записи и считывания данных, что позволяет повысить строгость декларирования внутренней структуры класса) и методы (подпрограммы, которые обрабатывают поля и свойства класса).

ЗАМЕЧАНИЕ Поля, свойства и методы класса называются *членами* класса.

Когда описывается переменная типа **class,** для ее полей и свойств в памяти выделяется соответствующий объем (как и для записей), но машинный код, в который транслируются методы класса, наличествует в единственном экземпляре, так как меняться он не может, и хранить несколько одинаковых копий подпрограмм не имеет смысла. Когда объект создается, однократно вызывается специальный метод, называемый *конструктором.* В нем выполняются различные действия по начальной инициализации полей объекта. Когда объект уничтожается (например, он был описан внутри процедуры как локальная переменная и удаляется из памяти при ее завершении), вызывается другой метод — *деструктор,* который выполняет различные дополнительные действия по освобождению памяти, если это необходимо. Явно вызывать конструктор и деструктор из программного кода нельзя. Это происходит только автоматически.

Переменная, описанная как класс, фактически является указателем на экземпляр класса. Это сделано для повышения эффективности работы с ним. Однако при использовании таких переменных применять операции работы с указателями (^, @) не надо. Достаточно обычного обращения к ним как к обычным переменным, а к членам класса — как к полям записи, через точку.

ЗАМЕЧАНИЕ Описывать переменные, принадлежащие к типу class, внутри подпрограмм в виде локальных типов не разрешается.

Три принципа объектного программирования

Наследование

Чтобы наиболее эффективно повторно использовать ранее созданные классы, одного сочетания данных и методов в единой структуре недостаточно. Например, автомобиль может быть легковым и грузовым, и соответствующие классы будут иметь как общие поля и методы, так и отличия (например, дополнительное свойство «грузовой кузов» и связанный с ним метод «разгрузить»). Однако полностью заново определять новый тип данных, если требуется изменить или добавить несколько новых свойств к старому типу, нерационально. Это плохо еще и потому, что если в

метод, имеющийся в обоих классах, потребуется внести исправления, то их придется делать дважды, в двух одинаковых копиях подпрограмм.

Чтобы избежать ненужной работы, в объектном программировании был введен принцип *наследования* свойств и методов. Программисту достаточно описать один *базовый* класс (например, «автомобиль»), а классы «легковой автомобиль» и «грузовой автомобиль» основывать на этом базовом классе. При этом будут наследоваться *все* поля, свойства и методы базового (или *родительского*) класса, а дополнительно описывать их не требуется.

Цепочки наследования могут быть неограниченной длины. Так, у класса «грузовой автомобиль» могут быть классы-наследники (или *дочерние* классы) «МАЗ» и «КАМАЗ», обладающие дополнительными специфическими свойствами и методами (это классы, а не объекты; объектом будет конкретный грузовик МАЗ, а не марка этого автомобиля), у класса «кнопка* наследники — «графическая кнопка», «круглая кнопка» и так далее. При этом различные методы для каждого из наследников разрешается переопределять. Например, метод «двигаться» для классов «МАЗ» и «КАМАЗ» будет, хоть и немного, но отличаться: по-разному расходуется горючее (снижается значение соответствующего свойства), по-разному набирается скорость и так далее.

Полиморфизм

Когда будет происходить обращение к переменной, относящейся к классу «КАМАЗ», и вызов унаследованного метода «двигаться», программе придется решить, какой конкретно метод надо вызвать: метод класса «автомобиль», «грузовой автомобиль» или «КАМАЗ». В соответствии с принципом полиморфизма решение принимается в зависимости от типа переменной, вызывающей этот метод. То есть, если переменная описана как относящаяся к типу «КАМАЗ», будет вызван метод «двигаться», определенный именно для КАМАЗа.

Инкапсуляция

Инкапсуляция позволяет разграничить доступ разработчиков к различным полям и свойствам класса, примерно так, как это сделано в модулях *Delphi,* когда из других модулей видима только интерфейсная часть. Точно так же и внутри классов некоторые поля и методы можно сделать свободно доступными для использования (*видимыми) в* любом месте программы, а другие поля и методы сделать доступными только внутри текущего модуля и собственных методов класса. Это позволяет скрыть внутри описания различные характеристики и возможности класса, чтобы сосредоточить внимание разработчиков, повторно использующих этот класс, на его важнейших свойствах.

Кроме того, желательно не допускать бесконтрольного изменения значений свойств, так как это может привести к нарушению запланированной и сбалансированной взаимосвязи между этими свойствами. Например, нельзя бездумно изменить значение свойства «текущая скорость», просто записав в него новое число. Надо вызвать соответствующий метод, который учтет изменение потребления бензина и выполнит ряд дополнительных действий. Хорошим стилем программирова-

ния считается недоступность всех полей и свойств объекта для прямого изменения. Вместо этого создаются методы, позволяющие получить значение поля и занести в него новое значение. Сами поля помещаются в скрытую часть класса.

События

Помимо этих трех фундаментальных возможностей объектно-ориентированного программирования, в среде *Delphi* реализована новая характеристика объекта — возможность обработки так называемых сообщений (или событий), получаемых от системы *Windows* или самой программы. Этот принцип лежит в основе работы всех визуальных компонентов *Delphi,* которые обрабатывают различные события, возникающие в процессе выполнения программы.

Описание класса

Новый тип (класс) описывается в Паскале обычным способом, с помощью ключевого слова class.

```
type имя-класса =
class(имя-родительского-класса)
список-членов-класса;
end;
```

Имя родительского класса указываегся, если новый класс должен наследовать все его характеристики (и характеристики всех родителей этого класса). Исключать какие-то члены родительских классов из наследования нельзя. Если имя родительского класса опустить:

```
type имя-класса =
class
список-членов-класса;
end;
```

то новый класс по умолчанию будет наследовать характеристики базового класса TObject, который содержит встроенные конструктор, деструктор и общие свойства и методы классов в среде *Delphi,* предназначенные для создания объектов (экземпляров класса), инициализации и освобождения памяти, и так далее. Корпорация *Inprise* рекомендует всегда явно указывать базовый родительский класс TObject.

Б некоторых случаях возникает необходимость ввести в программу новый класс, пока не описывая его структуру, а только определив заголовок (по аналогии с описанием заголовка подпрограмм с ключевым словом forward). Эго допускается сделать, просто указав название класса и ключевое слово class:

```
type TMyClass = class;
```

В дальнейшем, конечно, класс TMyClass должен быть полностью определен.

Надо отличать такое предварительное упреждающее описание от реального описания класса следующего вида:

```
type TMyClass = class(TObject);
```

Таким образом описывается полноценный класс TMyClass, структурно совпадающий с классом TObject.

Список членов класса представляет собой список полей, свойств и методов, которые записываются как обычные поля записи или объявления подпрограмм в интерфейсном разделе, например:

```
type TMyClass = class
   Count: Integer;
   Name: String;
   procedure ShowMyClass( Dis: Boolean );
   function GetCount: Integer;
   end;
```

В классе TMyClass имеются два поля: Count и Name, — процедура ShowMyClass (которая, возможно, показывает данный объект па экране) и функция GetCount, возвращающая значение поля Count. Пока что поле Count доступно для изменения: по умолчанию считается, что все члены класса не имеют никаких ограничений на видимость.

ВНИМАНИЕ Описание полей и свойств в классе должно предшествовать описанию методов.

Теперь в программу можно ввести новую переменную, представляющую собой объект — экземпляр класса TMyClass:

```
var MyClass: TMyClass;
```

Присваивание объектов

Так как класс — это обычный тип данных Паскаля, то переменным можно свободно присваивать значения соответствующих объектов (реально копируются только значения полей и свойств), причем как в точности своего типа, так и всех классов, являющихся наследниками класса этой переменной.

Например, если в программе имеется следующее описание:

```
type T1 = class (TObject);
   T2 = class(T1);
var X: T1;
```

ТО в переменную X можно записывать как объекты типа T1, так и объекты типа T2.

Доступ к полям и методам объекта осуществляется в точности, как при доступе к полям записи, — указанием нужного поля или метода через точку.

```
var O: TMyClass;
...
O.Count := 0;
O.Name := 'Класс';
O.ShowMyClass(true);
```

Пять уровней инкапсуляции

Доступность любых членов класса определяется принадлежностью к одному из пяти уровней видимости класса, определяемых специальными ключевыми словами Паскаля, которые разделяют описание класса на секции путем простого их указания без дополнительных элементов типа точки с запятой. По умолчанию видимость родительских членов класса наследуется в точности, однако разрешено повышать видимость — делать поля, свойства и методы более доступными. Понижение видимости не допускается.

Раздел public. Члены класса, находящиеся в данном разделе, доступны из любой точки программы.

```
type TMyClass = class
public
Count: Integer;
Name: String;
procedure ShowMyClass( Dis: Boolean );
function GetCount: Integer;
end;
```

Раздел private. Этот раздел накладывает самые жесткие ограничения на видимость указанных в нем членов класса. Они доступны только в том модуле, где данный класс описан. Как правило, поля класса помещаются в эту секцию.

```
type TMyClass = class
private
Count: Integer;
Name: String;
public
procedure ShowMyClass( Dis: Boolean );
function GetCount: Integer;
end;
```

По умолчанию считается, что все поля класса расположены в данном разделе. Вышеприведенное описание аналогично следующему описанию (без ключевого слова private).

```
type TMyClass = class
Count: Integer;
Name: String;
public
procedure ShowMyClass( Dis: Boolean );
function GetCount: Integer;
end;
```

Раздел protected. Видимость членов класса, расположенных в этом разделе, совпадает с видимостью раздела **private** с единственным отличием. Члены класса раздела **protected** доступны также внутри методов классов, являющихся наследниками данного класса и описанных в других модулях.

Раздел published. В этом разделе располагаются свойства класса: поля, доступные для редактирования и изменения значений во время проектирования и из Инспектора объектов. По видимости свойства не отличаются от членов классов, расположенных в разделе **public**. Более подробно о свойствах и разделе **published** рассказывается в главе, посвященной созданию собственных компонентов.

Раздел automated. Правила видимости членов раздела **automated** совпадают с правилами видимости для раздела **public.** Описания разрешается размещать в этом разделе, только если класс является наследником стандартного класса TAutoObject, предназначенного для создания так называемых серверов автоматизации при использовании технологии *COM* (см. соответствующую главу).

Реализация методов

После того как в модуле описан новый класс, в нижеследующей части исходного текста необходимо описать реализацию всех его методов (если они есть). Чтобы компилятор понял, к какому классу относится конкретная подпрограмма, перед ее названием указывается имя соответствующего класса и точка:

```
function TMyClass.GetCount: Integer;
begin
Result := Count
end;
```

Заголовок подпрограммы должен в точности соответствовать заголовку, указанному в описании класса.

Указатель на себя

В Паскале имеется специальный идентификатор Self, который всегда указывает на текущий объект (однако используется переменная Self как переменная, имеющая тип, отличный от указателя, то есть применять операции ^ и @ не нужно):

```
function TMyClass.GetCount: Integer;
begin
Result := Self.Count
end;
```

В данном примере результатом функции GetCount становится значение поля Count текущего класса (указывать Self в данном примере, конечно, необязательно).

Применять переменную Self удобно, например, в тех случаях, когда в подпрограмме имеется локальная переменная, имя которой совпадает с названием одного из полей. В этом случае можно явно указать, какую переменную — собственную, предваряемую идентификатором Self с точкой, или локальную — надо использовать.

Создание и удаление объекта

Помимо обычных методов, в Паскале имеются так называемые *методы классов.* Их особенность заключается в том, что методы классов разрешается вызывать, используя для этого вызова не объект, а класс. Это означает, что подобные методы не

могут обращаться к полям класса — ведь поля существуют только когда существует объект (экземпляр класса), а методы в виде программного кода, созданного на этапе компиляции, присутствуют в памяти компьютера постоянно.

Однако вызывать таким образом можно только методы, описанные с помощью ключевого слова **class,** указываемого перед заголовком подпрограммы. Это нужно, чтобы компилятор проверил, насколько корректно реализован соответствующий метод, в частности, не происходит ли в нем обращения к полям и свойствам класса.

```
type TMyClass = class
   Item: integer;
   public
   class function Sozdanie: integer;
   end;
```

В заголовке реализации функции Sozdanie также надо указать ключевое слово class:

```
class function TMyClass.Sozdanie: integer;
begin
...
end;
```

Так как методы классов не привязаны к объектам, то их можно вызывать, указывая название класса, а не объекта:

```
N := TMyClass.Sozdanie;
```

Это одна из фундаментальных возможностей Паскаля, благодаря которой можно, в частности, динамически (во время работы **программы)** создавать объекты и записывать ссылки на них в переменные. Данный подход активно используется для задания переменным-классам начальных значений. У базового класса TObject имеется метод Create (в реальности это конструктор, выполняющий все необходимые действия по запросу памяти и начальной **инициализаций),** который практически всегда применяется при создании объектов.

```
var MyClass: TMyClass;
...
MyClags := TMyClass.Create;
```

Это стандартный способ создания объектов и инициализации указывающих на них переменных. Пока таким способом в переменную не записан указатель на объект, обращаться к этой переменной нельзя.

Когда работа с объектом полностью закончена, занятую им намять необходимо освободить. Для этого существует специальный метод Free, который уничтожает объект, автоматически вызывая его деструктор,

```
MyClass.Free;
```

Вызов родительских методов

Во многих случаях большая часть функциональности наследуемых методов уже реализована в родительских классах. Например, при отображении кнопки на экране

стандартный класс TButton уже выполняет основную работу по отрисовке ее изображения в окне. Если требуется создать кнопку, которая будет добавлять к своему изображению дополнительные декоративные элементы, то в методе, ответственном за рисование кнопки, не имеет смысла полностью переписывать всю функциональность соответствующего метода.

Чтобы вызывать одноименный метод ближайшего родительского класса, достаточно в нужном месте метода указать ключевое слово inherited:

```
inherited;
```

В момент, когда программа встретит это слово, работа метода временно прервется и вызовется одноименный метод родительского класса.

Если требуется вызывать другой метод, не совпадающий по названию с текущим, то его можно указать после данного ключевого слова явно:

```
inherited Click;
```

Типы методов

В Паскале каждый метод класса может иметь дополнительные характеристики, определяющие, как будет реализовываться этот метод в классах-наследниках.

Статические методы

Все методы по умолчанию считаются статическими. Это означает, что их вызов будет происходить в соответствии с принципом полиморфизма.

Например, описаны два класса.

```
type TCar = class(TObect)
   procedure Move;
   end;
   TMAZ = class(TCar)
   procedure Move;
   end;
var Car: TCar;
MAZ: TMAZ;
begin
MAZ := TMAZ.Create;
Car := TCar.Create;
MAZ.Move;
// вызовется метод Move класса TMAZ
Car.Move;
// вызовется метод Move класса TCar
```

Однако переменную Car можно инициализировать объектом типа TMAZ:

```
Car.Free;
Car := TMAZ.Create;
```

Если теперь обратиться к методу Move:

```
Car.Move;
```

то вызовется метод TCar.Move, так как переменная Car, реально храня экземпляр класса TMAZ, описана как TCar. Обращаться к методам и полям переменной Car, как к методам и полям типа TMAZ, можно, только явно выполнив приведение типа: TMAZ(Car). Об этом рассказывалось при описании оператора присваивания:

```
TMAZ(Car).Move;
```

При этом вызовется метод Move, относящийся к классу TMAZ.

Если при вызове метода Move класса TMAZ необходимо предварительно вызвать родительский метод, выполняющий общие для всех автомобилей действия, в реализации метода Move надо указать этот вызов с помощью ключевого слова **inherited.**

```
procedure TMAZ.Move;
begin
inherited Move;
...
end;
```

Виртуальные и динамические методы

Статические методы удобны, когда в программе заранее известно, какие типы объектов будут использоваться, и приведения типов, как правило, не требуется. Б ряде случаев, особенно в крупных проектах, часто приходится хранить в переменных объекты-наследники (как в примере с TCar и TMAZ), причем конкретный тип этих объектов может быть неизвестен. Поэтому в Паскале реализован новый тип методов — *виртуальные* методы (для их описания существует зарезервированное слово **virtual).** Описать виртуальный метод можно так.

```
type TCar = class(TObect)
   procedure Move; virtual;
   end;
```

Такие методы в классах-наследниках могут быть *перекрыты* методами с одноименными заголовками. Чтобы явно указать компилятору, что определенный метод перекрывает виртуальный метод родителя, надо использовать ключевое слово **override.**

```
type TMAZ = class(TCar)
   procedure Move; override;
   end;
```

Теперь в приведенном примере можно не выполнять приведение типов. Во время работы программы она самостоятельно определит тип хранимого в переменной Car объекта и вызовет нужный метод.

```
Car := TMAZ.Create;
Car.Move;
```

Вызовется метод Move класса TMAZ. Если же этот метод не перекрыть (не указать ключевое слово **override),** то вызовется метод Move класса Car.

Вместо ключевого слова **virtual** можно применять ключевое слово **dynamic,** выполняющее тс же функции и описывающее метод как динамический.

```
type TCar = class(TObect)
   procedure Move; dynamic;
   end;
```

Разница между виртуальными и динамическими методами заключается только в деталях программной реализации. Виртуальные методы оптимизированы для максимального быстродействия, динамические — для максимальной экономии памяти (создания компактного кода).

Абстрактные методы

В некоторых случаях не имеет смысла выполнять реализацию определенных методов базового класса, например, когда все реализации некоторого метода сильно отличаются друг от друга, а метод родительского класса не используется. Вместе с тем соответствующий метод обязан быть реализован в каждом из классов-наследников.

Такой метод надо объявить в родительском классе как *абстрактный*.

```
type TCar = class(TObect)
   procedure Move; virtual; abstract;
   end;
```

Теперь описывать реализацию метода TCar.Move не надо.

ЗАМЕЧАНИЕ Абстрактным может быть только динамический или виртуальный метод.

Перегружаемые методы

Ранее рассматривались перегружаемые подпрограммы, имеющие одинаковые имена, но различные типы параметров. Компилятор автоматически определяет, какую конкретно подпрограмму надо вызывать в зависимости от типов ее аргументов.

Имеется в Паскале аналогичная возможность и для методов.

```
type T1 = class(TObject)
   procedure Sum(X,Y: Integer); overload;
   end;
type T2 = class(T1)
   procedure Sum(X,Y: Real); overload;
   end;
var CT: T2;
...
CT := T2.Create;
CT.Sum(2,2);
// вызывается T1.Sum()
CT.Sum(2.0,2.0);
// вызывается T2.Sum()
```

Перегружаемым может быть и виртуальный метод. Чтобы у компилятора не возникало претензий к программисту по поводу несоответствия заголовков перегружа-

емых виртуальных методов, в наследуемом методе надо дополнительно указать ключевое слово **reintroduce**:

```
type T1 = class(TObject)
   procedure Sum(X,Y: Integer); overload; virtual;
   end;
type T2 = class(T1)
   procedure Sum(X,Y: Real); reintroduce; overload;
   end;
```

Конструкторы и деструкторы

Конструктор — это метод, который вызывается только один раз в момент создания экземпляра объекта соответствующего класса. Конструктор считается методом класса **(class procedure),** что позволяет вызывать его, указывая не только имя объекта, но и имя класса. Для конструктора выделено специальное ключевое слово **constructor.**

```
constructor CreateMyObject(MySize: Integer);
```

Конструкторов у объекта может быть сколько угодно, однако создаваться объект может с помощью только одного из них.

ПОДСКАЗКА В базовом классе TObject имеется готовый конструктор Create, который и рекомендуется использовать, если нет необходимости в конструкторах с особыми возможностями. В любом случае в реализации собственного конструктора желательно вызывать базовый конструктор Create с помощью директивы inherited.

Деструктор — это метод, который вызывается только один раз в момент уничтожения экземпляра объекта соответствующего класса. Класс TObject имеет стандартный деструктор Destroy, Разрешается создавать неограниченное число деструкторов с помощью ключевого слова **destructor:**

```
destructor MyDestructor;
```

ПОДСКАЗКА Рекомендуется создавать в классе только один деструктор, а в его реализации вызывать деструктор Destroy базового класса TObject с помощью директивы inherited.

Динамическое конструирование объектов

Тип «класс»

В Паскале имеется возможность объявлять новый тип, являющийся ссылкой на класс:

```
type имя-класса - class of тип-класса;
```

Например:

```
type TMetaClass = class of TObject;
```

Переменная, описанная как

```
var MetaClass: TMetaClass;
```

может хранить в себе объект любого класса, унаследованного от TObject. Это позволяет записывать в подобные переменные объекты произвольных классов во время выполнения программы, не зная конкретных типов этих объектов. Например, если имеется стандартная подпрограмма обработки объектов, тип которых будет известен только в процессе работы приложения, то в качестве типа соответствующего параметра надо указать тип, представляющий собой ссылку на класс. Тогда в момент вызова подпрограммы можно задавать объект, имеющий произвольный тип, унаследованный от TObject.

```
procedure UseObject( Obj: TMetaClass );
...
UseObject( TButton );
UseObject( TEdit );
```

Классы TButton и TEdit — стандартные классы *Delphi,* описывающие компоненты «кнопка» и «текстовое поле».

Проверка типа объекта

Так как в Паскале допускается применять объекты, тип которых на этапе компиляции не известен, то в языке должны быть и средства, позволяющие выполнять проверки типов таких объектов.

Специальная операция **is** (ключевое слово) позволяет проверить, соответствует ли тип объекта конкретному типу Паскаля.

```
объект is тип
```

В качестве типа должен указываться один из классов.

```
Obj is TButton
```

Это выражение возвращает значение логического типа (True, если тип объекта Obj соответствует типу, указанному в правой части, — TButton).

Приведение типа объекта

Если объект описан как ссылка на класс, то часто требуется применять этот объект как относящийся к конкретному типу, чтобы иметь доступ к его определенным полям и методам. Выполнить приведение к конкретному типу можно, применив операцию as:

```
объект as класс
```

Например, если в переменную MetaClass записан объект с типом TEdit, то получить доступ к полю Text класса TEdit можно с помощью следующей конструкции:

```
(MetaClass as TEdit).Text := 'содержимое';
```

Обработка сообщений

В Паскале для классов реализована дополнительная возможность — обработка сообщений, получаемых от других объектов программы и от *Windows,* Для этого применяется ключевое слово **message.** Процедура, использующая эту директиву,

будет вызываться, когда объект соответствующего класса получит сообщение с указанным идентификатором. Значение идентификатора должно лежать в диапазоне от 1 до 49151 в случае обычного обработчика сообщений или должно соответствовать одному из идентификаторов стандартных сообщений *Windows,* которые описаны в модуле Messages.

```
type T1 = class(TObject)
   procedure MyMsg(var Hsg: TMessage); message 12590;
   end;
```

Единственный параметр процедуры-обработчика сообщения должен быть описан с ключевым словом var (передаваться по ссылке).

При описании реализации обработчиков событий *Windows* можно сразу указывать название события и тип параметра:

```
procedure WMKeyDown(var Msg: TWMKeyDown);
   message WM_KEYDOWN;
```

Что нового мы узнали?

В этом уроке мы научились

- записывать программу на языке Паскаль;
- контролировать структуру программы;
- применять основные операторы языка Паскаль;
- создавать простейшие консольные приложения;
- определять собственные типы данных;
- использовать подпрограммы;
- работать с классами и объектами.

УРОК 2 Основы программирования в среде Delphi 7

- Создание программ для Windows
- Иерархия компонентов Delphi 7
- Управление проектом
- Новые стандартные действия

Создание программ для Windows

В предыдущих главах рассматривался язык программирования Паскаль. На примерах создания консольных приложений демонстрировались его возможности, использование операторов. Однако среда разработки *Delphi 7* ориентирована, прежде всего, на создание программ для *Windows,* хотя имеется возможность подготовки исходных текстов, на основе которых будет собрана программа, одинаково работающая в *Windows* и *Linux.* При этом особое внимание уделяется возможности визуальной разработки приложений с помощью большого набора готовых компонентов (стандартных классов), позволяющих избежать ручного кодирования. Эти компоненты охватывают практически все аспекты применения современных информационных технологий, однако для начала надо изучить базовые компоненты, которые требуются при подготовке практически любого приложения.

Использование визуальных компонентов

Компоненты первостепенной важности расположены на панели Standard (Стандартные) палитры компонентов (рис. 2.1).

Рис. 2.1. *Базовые визуальные компоненты*

Они соответствуют основным элементам управления *Windows,* без которых не обходится ни одна программа.

Рассмотрим процесс создания простейшей программы, которая по щелчку на кнопке выполняет сложение чисел, введенных в два текстовых поля, и показывает результат на отдельной панели.

Прежде всего надо закрыть все текущие файлы *Delphi 7,* которые имеют отношение к консольной программе. Это выполняется с помощью команды File ➤ Close All (Файл > Закрыть все). После этого надо создать заготовку будущего приложения *Windows,* выполнив команду File ➤ New ➤ Application (Файл ➤ Создать ➤ Приложение). На экране возникнет пустая форма — прообраз будущего главного окна программы, а в редакторе откроется файл Unit1.pas, соответствующий модулю Unit1, в котором хранится описание работы этой формы (она называется Forml). В разделе реализации выполняется подключение стандартных модулей:

```
Interface
uses
Windows, Messages, SysUtils, Classes,
Graphics, Controls, Forms, Dialogs;
```

затем объявляется тип TForm1, содержащий описание формы, и декларируется одна переменная этого типа:

```
var Form1: TForml;
```

ЗАМЕЧАНИЕ Все компоненты Delphi 7 хранятся в библиотеке визуальных компонентов Visual Component Library (VCL). Каждый из компонентов, а также форма (будущее окно) описаны соответствующими классами Паскаля: к названию компонента добавляется буква T; например, форма описывается классом TForm, кнопка (компонент Button) — классом TButton и так далее.

Когда создается новое приложение, на основании этих классов система Delphi 7 формирует новый тип данных, наследующий характеристики своего родителя:

```
type
TForm1 = class(TForm)
   private
    { Private declarations }
   public
    { Public declarations }
   end;
```

Форма Form1 представлена в программе типом TForm1, который исходно не имеет новых полей и методов, так как форма пуста. В дальнейшем члены класса TForm1 будут добавляться по мере необходимости, как автоматически (при размещении их на форме в Проектировщике форм), так и самим программистом.

Далее в модуле следует раздел реализации, в котором имеется всего одна директива компилятора

```
{$R *.DFM}
```

Она предназначена для связывания модуля с описанием соответствующей ему формы.

ЗАМЕЧАНИЕ Описание формы и всех размещенных на ней компонентов система Delphi 7 хранит в файлах с расширением .DFM. Эти файлы имеют обычный текстовый формат (в отличие от прежних версий Delphi, где они хранились в двоичном виде, что делало невозможным их анализ и ручное редактирование).

Создаваемая в среде *Delphi 7* программа состоит из нескольких файлов. Это файлы с исходными текстами на Паскале и файлы описаний форм. Все они связаны друг с другом. Кроме того, в среде *Delphi 7* имеется множество настроек, которые желательно сохранять на жестком диске в промежутках между сеансами работы с системой. Такой набор файлов, в которых содержатся исходные тексты и различные настройки, называется *проектом*. Разделение на проекты очень удобно, потому что позволяет выделить все файлы, относящиеся к конкретной задаче, в отдельную группу.

Чтобы сохранить текущий проект, надо выполнить команду File ➤ Save All (Файл ➤ Сохранить все) или щелкнуть на одноименной командной кнопке.

Сначала разработчику будет предложено сохранить файл с исходным текстом (Unit1.pas), а затем — файл проекта Project1 с расширением .DPR. Сохранять их лучше в

отдельном каталоге, специально отведенном для данного проекта. В этом же каталоге компилятор в будущем создаст и исполнимое приложение.

Теперь можно приступать к этапу *проектирования пользовательского интерфейса* — подготовке внешнего вида окна программы и размещению на нем элементов управления. Для этого надо переключиться в Проектировщик форм с помощью клавиши F12 или командной кнопки Toggle Form/Unit (Выбрать Форму/Модуль).

Проектировщик форм работает по принципу *WYSIWYG,* в соответствии с которым окно созданной программы будет выглядеть в точности так, как оно было подготовлено в Проектировщике форм. Например, изменить размер формы можно стандартным способом для системы *Windows* — протягиванием мыши.

ВНИМАНИЕ Создаваемая в Проектировщике форма — это только внешнее представление будущего окна, а не работоспособное приложение.

Любой объект Проектировщика форм доступен в исходном тексте программы по его имени. Имя (или название) объекта — это одно из его свойств, которые можно изменять и настраивать в Инспекторе объектов. Соответствующее свойство называется Name (Имя). Оно имеется у всех объектов без исключения и расположено в Инспекторе объектов в категории Miscellaneous (Дополнительные) (рис. 2.2).

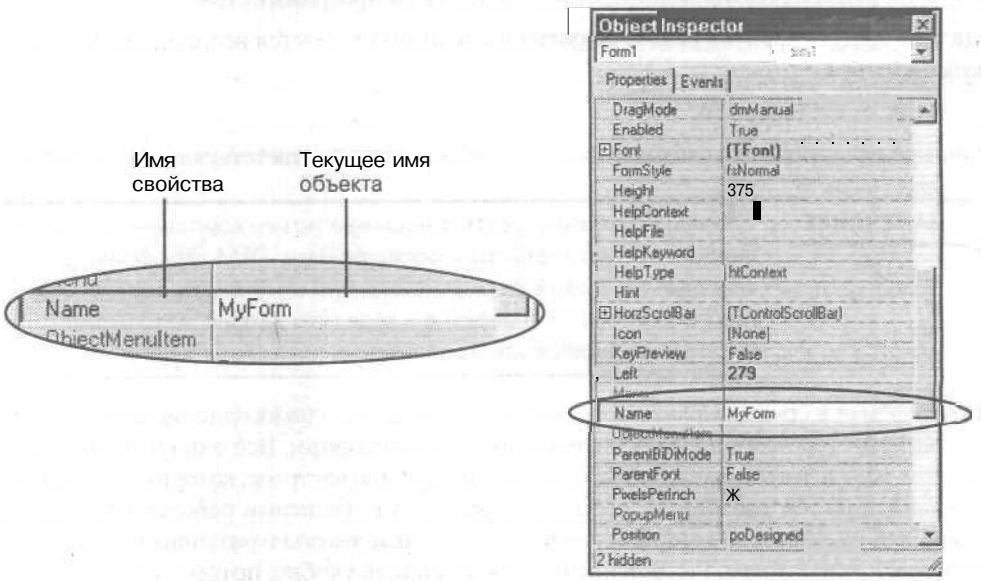


Рис. 2.2. *Задание имени объекта*

Первоначально значением этого свойства является строка Form1, сгенерированная системой *Delphi* 7 автоматически. Если ее изменить, например на MyForm, и нажать клавишу ENTER, то в проекте произойдут следующие изменения.

- Заголовок формы изменится с Form1 на MyForm. Система Delphi 7 считает, что заголовок окна совпадает с его названием, пока разработчик не изменит заголовок явно.
- Тип формы в файле Unit1.pas изменится с TForm1 на TmyForm.
- Имя переменной Forml изменится на MyForm.

Эти операции система *Delphi 7* выполнит автоматически. В дальнейшем при корректировке названий форм и других объектов в исходные тексты также будут вноситься изменения, не требующие вмешательства человека, что позволяет избежать неприятных ошибок.

Заголовок формы — это свойство Caption (Заголовок). В Инспекторе объектов оно находится в категории Visual (Внешний вид). Здесь можно ввести строку Пример. В свойстве Name (Имя) можно использовать только латинские буквы и цифры, так как содержимое этого свойства соответствует названию переменной Паскаля — идентификатору. Заголовок — это произвольная строка, не имеющая прямого отношения к программированию.

На форме размешаются объекты, соответствующие компонентам с палитры компонентов. Чтобы создать на форме текстовое ноле, надо:

- выбрать панель Standard (Стандартная);
- щелкнуть на кнопке Edit (Текстовое поле);
- щелкнуть на том месте формы, где требуется поместить текстовое ноле.

В этом месте появится новый элемент управления (рис. 2.3).

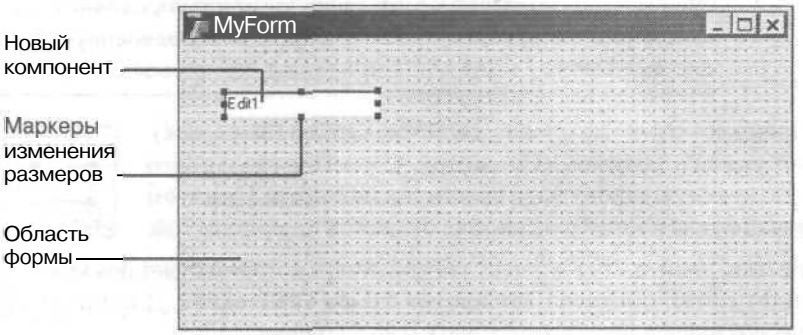


Рис. 2.3. *Добавление нового компонента на форму*

Черные маркеры по контуру объекта указывают, что он выделен. Эти маркеры предназначены для изменения размеров объекта с помощью мыши.

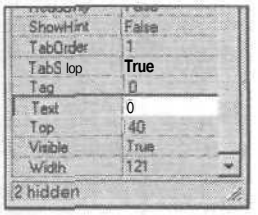
По умолчанию в системе *Delphi 7* принято, что название нового объекта совпадает с его заголовком. Это название создается средой *Delphi* по следующему принципу. Берется название компонента (для формы это Form, для текстового поля — Edit и так далее), и к нему добавляется порядковый номер, начиная с единицы. Если теперь на форму поместить еще одно текстовое поле, то его названием будет Edit2.

Если теперь взглянуть на исходный текст класса TMyForm, то окажется, что в разделе private появилось новое поле:

```
Edit1: TEdit;
```

Оно было добавлено средой *Delphi 7* в описание класса TMyForm автоматически.

ЗАМЕЧАНИЕ Общепринято определять название компонента в соответствии с названием класса. То есть, например, компонент «Кнопка» называется TButton, а не просто Button. Будем придерживаться этого принципа и мы.



У компонента TEdit (Текстовое поле) свойства Caption (Заголовок) нет. Вместо него активно используется свойство Text (Текст), относящееся в Инспекторе объектов к категории Localizable (Настраиваемые). Это свойство содержит введенные пользователем данные в текстовом виде (тип string). Первоначально это свойство содержит строку, совпадающую с именем элемента управления (Editl, Edit2). Текущее содержимое свойства Text (Текст) каждого текстового поля формы лучше удалить и ввести вместо этого строки «0». Аналогичным способом на форме размещается кнопка (компонент TButton).

ЗАМЕЧАНИЕ В Windows стандартной считается возможность обращения к конкретному элементу управления по «горячей клавише». Соответствующий символ выделяется в названии элемента управления подчеркиванием. Чтобы выделить таким образом определенную букву, в системе Delphi 7 надо поставить перед ней символ &.

Надпись на кнопке — это ее заголовок, свойство Caption (Заголовок). В нем можно указать подпись «Сложить». С учетом сказанного выше соответствующую строку надо ввести так: &Сложить. При этом на форме первая буква подписи на кнопке окажется подчеркнутой.

В заключение надо подготовить элемент управления, в котором будет показываться результат сложения чисел, введенных в поля Editl и Edit2. Для такой цели лучше всего использовать компонент TLabel (Надпись).

ВНИМАНИЕ Свойства Caption и Text имеют тип string. Если в них записываются числа, они будут храниться в текстовом виде.

В заголовке этого объекта (свойство Caption) можно указать строку *0». Размер объекта Labell при этом изменится. Это связано с тем, что некоторые компоненты *Delphi* 7 имеют свойство AutoSize (Автомасштабирование), относящееся к категории Visual (Внешний вид). Это свойство определяет, что размер объекта подстраивается под размер своего содержимого, в данном случае строки. Если это не нужно, значение

свойства AutoSize надо изменить с True на False, после чего задать нужный размер вручную.

ЗАМЕЧАНИЕ

После размещения объектов на форме их можно свободно перетаскивать при помощи мыши в другое место, причем не только по одному, но и группой. Для этого надо выделить группу элементов протягиванием (они помечаются серыми маркерами] и переместить их как один объект.

Теперь в описании формы MyForm в исходном тексте программы можно обнаружить четыре новые переменные:

```
Edit1: TEdit;
Edit2: TEdit;
Button1: TButton;
Label1: TLabel;
```

Они добавлялись в текст автоматически, по мере того как на форме размещались соответствующие элементы управления. Например, переменная Buttonl — это кнопка (класс TButton). Размеры каждого объекта на экране задаются и фиксируются с помощью свойств Width (Ширина) и Height (Высота).

Создание **работоспособной** программы

Хотя пока задан только внешний вид главного окна, уже можно получить работоспособную программу. Если ее запустить, она позволит вводить в текстовые поля различные значения, щелкать на кнопке, менять размеры окна. Правда, выполнять суммирование введенных чисел программа пока не будет — это действие надо запрограммировать вручную.

Чтобы контролировать ход компиляции и создания готового приложения, надо в настройках среды *Delphi 7* включить флажок Show compiler progress (Отображать ход компиляции) (рис. 2.4). Соответствующее диалоговое окно открывается командой Tools ➤ Environment Options ➤ Preferences (Сервис > Настройки среды ➤ Предпочтения разработчика).

Компиляция программы выполняется командой Project > Compile Project (Проект ➤ Компилировать проект) или нажатием комбинации клавиш CTRL+F9. При этом на экране появляется диалоговое окно, в котором отображаются:

- название главного файла проекта — в строке Project (Проект);
- имя компилируемого файла — в строке Compiling (Компиляция);
- номер текущей строки компилируемого исходного текста — в строке Current Line (Текущая строка);
- общее число строк с учетом других откомпилированных модулей — в строке Total lines (Всего арок);

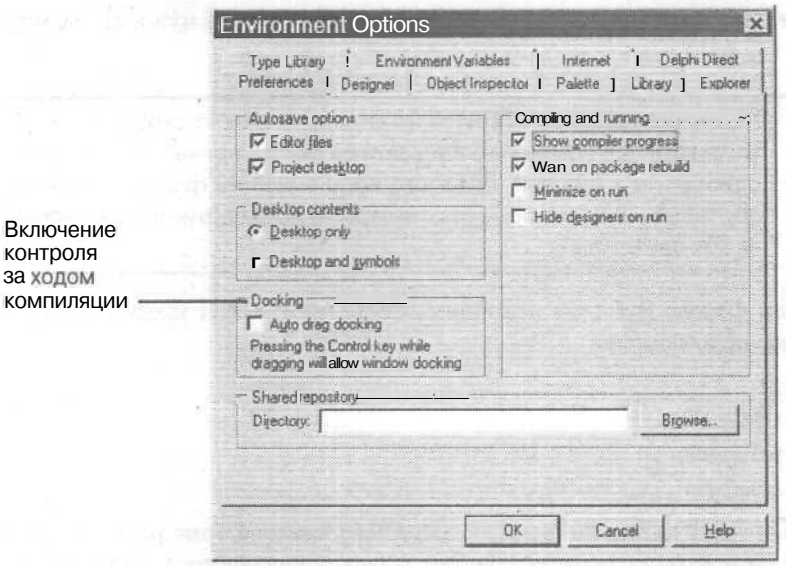


Рис. *2.4. Диалоговое окно настройки параметров рабочей среды*

- число советов, генерируемых компилятором для подсказки разработчику о двусмысленных местах в исходном тексте, например когда переменная перед ее первым использованием не имеет значения — в строке Hints (Подсказки);
- число предупреждений, выдаваемых компилятором при обнаружении мест в программе, которые могут служить потенциальным источником ошибок - в строке Warnings (Предупреждения);
- число ошибок, возникающих, когда компилятор не может определить, что означает некоторая строка исходного текста, — в строке Errors (Ошибки) (рис. 2.5).

***Рис. 2.5.** Информация о ходе компиляции*

Если в процессе компиляции встретились подсказки и предупреждения, работоспособная программа все равно будет создана. Если же встретились ошибки, этого не произойдет, а система *Delphi 7* во второй строке информационного окна сообщит об их обнаружении

Done: There are errors (Готово: Найдены ошибки).

Ошибки надо исправить и выполнить процесс компиляции заново.

В рассматриваемом примере ни ошибок, ни предупреждений, ни подсказок быть не должно, потому что весь исходный текст был сгенерирован системой Delphi 7 автоматически. Теперь созданную программу можно запустить непосредственно из среды *Delphi 7,* выполнив команду Run > Run (Запуск ➤ Запуск), нажав клавишу F9 или щелкнув на командной кнопке Run (Запуск).

ЗАМЕЧАНИЕ Созданную программу — она расположена в каталоге, где был сохранен проект, и называется Project1.exe — можно запустить и обычным способом, например, с помощью Проводника.

События и реакции на них

Внутренняя структура программ для *Windows* кардинальным образом отличается от структуры консольных программ, где операторы выполнялись *последовательно,* от начала программы, ключевого слова **begin,** до завершающего слова **end.** Операционная система *Windows* функционирует по другому принципу. Она обрабатывает возникающие в ней *события:* щелчок мыши на кнопке, выбор пункта меню, нажатие клавиши, достижение встроенным таймером заданного значения времени — и передает их выполняющимся в своей среде программам. Они, в свою очередь, обычно находятся в состоянии ожидания и активизируются только при получении от *Windows* сообщений о событиях — *реагируют* на них. Конечно, некоторые программы могут выполнять длительные вычисления, например в фоновом режиме, но это нетипично.

Сообщения *Windows* обрабатываются программой не одновременно, а последовательно (хотя некоторые сообщения имеют более высокие приоритеты, чем другие). Например, сообщение, требующее завершения работы приложения, не выполняется мгновенно, а ставится в *очередь сообщений,* и, пока не будут обработаны первые сообщения (это может быть информация о щелчках мыши, нажатиях клавиш, кнопок и прочее), работа программы продолжится.

Структура программы для *Windows* представляет собой набор подпрограмм, каждая из которых ответственна за обработку конкретного события и вызывается только при его получении. Программист сам решает, какие события в программе требуется обрабатывать. В нашем примере необходимо реагировать только на щелчок на кнопке. *Системные события:* выбор пункта системного меню, закрытие приложения — обрабатываются в программе, созданной с помощью системы *Delphi 7,* автоматически.

ЗАМЕЧАНИЕ Подобный подход к созданию программ называется *событийно-ориентированным*.

Обработчик щелчка на кнопке (метод класса TMyForm) создается на форме так: выполняется двойной щелчок на этой кнопке и *Delphi 7* генерирует заготовку кода подпрограммы, которая будет выполнять обработку щелчка.

```
procedure TMyForm.Button1Click(Sender: TObject);
begin
end;
```

Автоматически созданное имя процедуры — Button1Click. Ему предшествует имя класса TMyForm, к которому относится данная процедура. Параметр Sender определяет, какой объект программы вызывает данную подпрограмму. Хотя она будет вызываться автоматически при щелчке на кнопке Button1, ее можно вызывать и непосредственно из текста программы, как обычную подпрограмму. Например, чтобы проверить, является ли вызов данного обработчика реакцией на нажатие кнопки Button1, можно использовать следующий условный оператор:

```
if Sender = Button1 then ...
```

Обработчик должен функционировать следующим образом. Строки, введенные в поля Edit1 и Edit2, надо преобразовать в числа с помощью стандартной функции StrToInt(), затем сложить их, результат преобразовать обратно в строку и записать ее в свойство Caption объекта Label1.

Доступ к свойствам Text и Caption (и любым другим) происходит точно так же, как к полям записей. Сначала указывается имя переменной, определяющей «владельца» этих свойств, а затем через точку приводятся названия свойств:

```
Edit1.Text
Label1.Caption
```

Отображение нового содержимого надписи Label1 произойдет в момент завершения оператора присваивания свойству Caption нового значения. Таким образом, весь обработчик события «Щелчок на кнопке Сложить» (и вся логика работы программы) уместится в единственном операторе присваивания:

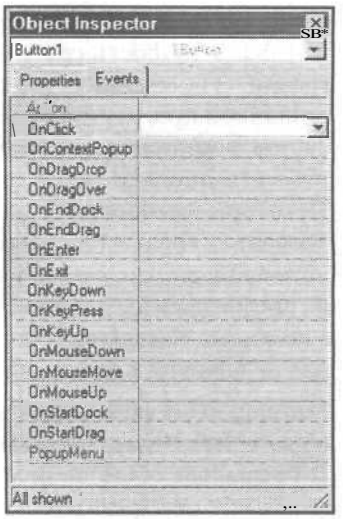
```
procedure TMyForm.Button1Click(Sender: TObject);
begin
Label1.Caption :=
IntToStr( StrToInt(Edit1.Text)+StrToInt(Edit2.Text) );
end;
```

Теперь можно выполнить компиляцию программы, запустить ее, ввести в поля ввода два произвольных числа и щелкнуть на кнопке Сложить. В том месте окна, где располагается надпись, появится результат суммирования.

Второй способ формирования обработчика события

Чтобы сформировать в программе обработчик щелчка на кнопке, достаточно выполнить двойной щелчок на этой кнопке в Проектировщике форм. Однако для большин-

ства компонентов Delphi 7 имеются события, обработчики которых невозможно создать подобным способом (например, «Нажатие клавиши» или «Изменение размера»). Подготовить обработчик требуемого события можно так.



- Выберите на форме нужный объект, к которому будет относиться рассматриваемое событие (например кнопку Button1).
- В Инспекторе объектов перейдите на вкладку Events (События).
- Найдите в списке событий строку с именем нужного события. В данном случае это событие OnClick (При щелчке). Так как для объекта Button1 обработчик уже создан, то в строке OnClick должно располагаться имя обработчика Button1Click.
- Дважды щелкните на правой части соответствующей строки.

Система *Delphi 7* автоматически сгенерирует нужный программный код и переключится в редактор.

Компонент Меню (TMainMenu)

Компонент TMainMenu предназначен для добавления к программе главного меню, без которого не обходится практически ни одно из приложений *Windows*.

Способ создания

Чтобы добавить к разрабатываемой программе меню, надо выбрать на панели компонентов Standard (Стандартные) компонент TMainMenu и поместить его на форме в произвольном месте (рис. 2.6).

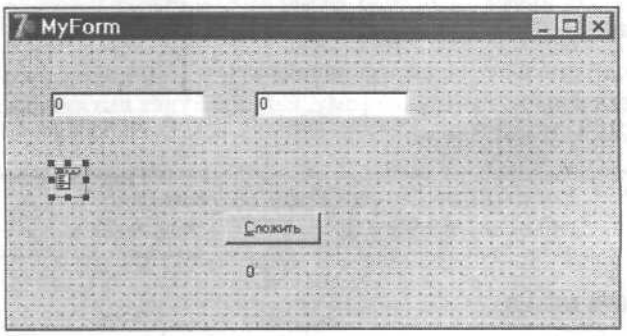
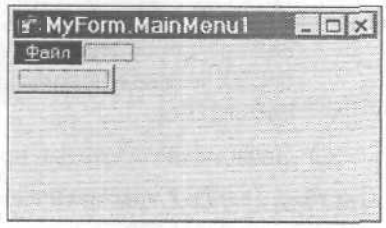


Рис. 2.6. *Добавление к форме компонента TMainMenu*

Компонент TMainMenu — *невизуальный*, в отличие от *визуальных* компонентов TEdit и TLabel, в точности соответствующих своему внешнему виду в работающей программе. Это означает, что хотя он виден на форме как небольшой квадрат, в окне созданной программы в таком виде компонент не появится. Представление его на форме в миниатюрном виде просто указывает на наличие в программе объекта, ответственного за меню. А создается меню с помощью специального редактора. Некоторые компоненты называются невизуальными потому, что, во-первых, ряд элементов управления невозможно разместить на форме без специальной подготовительной работы, а во-вторых, в системе *Delphi 7* имеется ряд компонентов, которые не предназначены для отображения на экране, хотя их свойства можно настраивать с помощью Инспектора объектов. Подобные компоненты используются, например, для обращения к базам данных, для установки связи с Интернетом и прочего.



Редактор меню вызывается двойным щелчком на объекте MainMenu1. Первоначально меню пустое. В Инспекторе объектов надо открыть категорию Localizable (Настраиваемые) и в свойстве Caption (Заголовок) ввести название первого пункта, например стандартную команду &Файл с указанной горячей клавишей, а затем нажать клавишу ENTER. Редактор меню переключится обратно в проектируемое меню, где уже появится первый пункт.

Теперь надо опять нажать клавишу ENTER, и система *Delphi 7* переключится к заголовку Caption для нового пункта. В него вводится очередное название (например, &Сложить), опять нажимается клавиша ENTER, и цикл формирования меню повторяется.

ЗАМЕЧАНИЕ Чтобы вставить линию-разделитель, надо в свойстве Caption в первой позиции указать символ - (дефис).

Самым последним обычно добавляется пункт &Выход.

Для вставки/добавления новых пунктов служит клавиша INSERT, для удаления — клавиша DELETE. По проектируемому меню можно перемещаться с помощью курсорных клавиш.

Когда меню подготовлено, редактор надо закрыть. При этом на форме появится меню, в точности соответствующее тому, как оно будет выглядеть в будущей программе (рис. 2.7).

ВНИМАНИЕ Редактировать структуру меню можно только обращением к объекту MainMenu1.

Использование меню

Чтобы создать обработчик события, возникающего при выборе одного из пунктов меню, надо в Редакторе меню дважды щелкнуть на соответствующем пункте, напри-

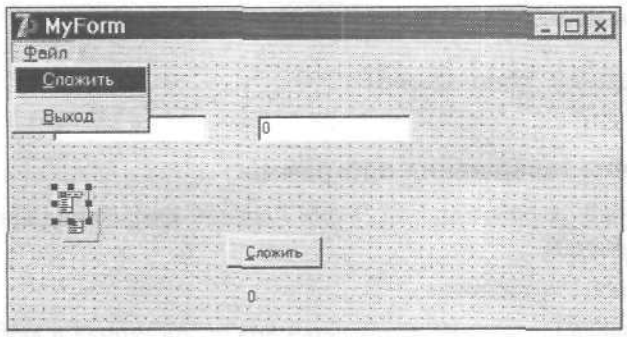


Рис. 2.7. *Законченное меню программы в режиме проектирования*

мер на пункте Выход. Как и в случае кнопки, система Delphi 7 автоматически создаст новый метод.

```
procedure TMyForm.N4Click(Sender: TObject);
begin
end;
```

N4 — это идентификатор пункта меню Выход внутри программы. В описании класса TMyForm он (наряду с другими пунктами: N1, N2, N3) декларирован так:

```
N4: TMenuItem;
```

Класс TMenuItem предназначен для объявления пунктов меню.

Чтобы закрыть приложение при выборе пункта Выход, надо в реализации метода N4Click вызвать метод Close класса TForm, наследником которого является класс TMyForm.

```
procedure TMyForm.N4Click(Sender: TObject);
begin
Close;
end;
```

Этот метод выполняет корректное закрытие текущей формы (окна).

ПОДСКАЗКА Если закрывать приложение щелчком на стандартной системной закрывающей кнопке, программа в некоторых случаях может завершаться некорректно. Лучше предусмотреть возможность явного завершения приложения средствами Delphi 7.

Если закрывается главная форма, то завершается и работа всего приложения.

Перед закрытием программы желательно проверить, можно ли ее в данный момент корректно закрыть (иногда закрытие окна не допускается, например когда запущено подчиненное приложение — *поток,* который должен завершиться в первую очередь). Подобную проверку выполняет функция CloseQuery, возвращающая значение типа Boolean.

```
procedure TMyForm.N4Click(Sender: TObject);
begin
if CloseQuery then Close;
end;
```

Другие способы завершения программы

Рекомендованный разработчиками *Delphi 7* способ завершения программы — это выполнение команды

```
Application.Terminate;
```

Переменная Application — глобальная переменная, доступная в любой программе, написанной в среде *Delphi 7*. Она имеет тип TApplication и содержит базовый набор полей и методов, характерных для любой программы *Windows*. В частности, вызов метода Terminate приводит к генерации сообщения *Windows*PostQuitMessage, которое ставится в очередь сообщений (то есть, программа завершит работу не сразу).

В Паскале также имеется подпрограмма Halt{), в качестве параметра которой указывается код завершения приложения в среде *Windows*. При выполнении данной подпрограммы работа приложения завершается немедленно. Применять эту процедуру не рекомендуется.

Вложенный вызов обработчика

Чтобы выполнить сложение чисел, введенных в поля Edit1 и Edit2, надо выбрать пункт меню Сложить. При щелчке на нем в Проектировщике форм будет создан соответствующий обработчик.

```
procedure TMyForm.N2Click(Sender: TObject);
begin
end;
```

Повторно писать ранее созданный код (метод Button1Click) не имеет смысла. Проще и правильнее вызвать метод Button1Click напрямую, передав ему параметр Sender.

```
procedure TMyForm.N2Click(Sender r TObj ect);
begin
Button1Click(Sender)
end;
```

Обработка щелчка мыши

Управление большинством программ *Windows* осуществляется с помощью мыши. Помимо стандартных действий с элементами управления, мышь также используется для самых разных дополнительных операций (например, для вызова контекстного меню при щелчке правой кнопкой). В системе *Delphi 7* имеется возможность обработки фактического щелчка мышкой на объекте. Такая возможность используется, когда важно просто среагировать на щелчок, не анализируя координаты указателя. Для этого обрабатывают событие OnClick, которое чаще всего используют

в таких объектах, как кнопка или переключатель, где знание точных координат указателя мыши не обязательно.

Такой подход далеко не всегда устраивает разработчика, поэтому в *Delphi 7* имеются еще два события: OnMouseDown (При нажатии кнопки мыши) и OnMouseUp (При отпускании кнопки мыши). Они содержат подробную информацию о параметрах щелчка: координаты указателя в рамках клиентской области объекта, на котором был выполнен щелчок, тип щелчка (одинарный или двойной), какая кнопка мыши была нажата/отпущена и состояние системных клавиш SHIFT, ALT и CTRL

Обработчики этих событий отличаются только названиями, списки их параметров полностью совпадают.

```
procedure TMyForm.Button1MouseDown
    (Sender: TObject; Button: TMouseButton;
    Shift: TShiftState,- X, Y: Integer);
```

Тип TMouseButton является перечислимым типом и содержит три значения: mbLeft, mbRight, mbMiddle (признаки использования, соответственно, левой, правой и средней кнопок мыши). Тип TShiftState представляет собой множество: так как можно выполнить, например, двойной щелчок, одновременно удерживая нажатыми клавиши ALT и SHIFT, необходимо проверять комбинации допустимых значений (табл. 2.1).

```
type TShiftState = set of
    (ssShift, ssAlt, ssCtrl,
    ssLeft, ssRight, ssMiddle, ssDouble);
```

***Таблица 2.1.** Назначение элементов множества*

Идентификатор	Значение
ssShift	Нажата клавиша SHIFT
ssAlt	Нажата клавиша ALT
ssCtrl	Нажата клавиша CTRL
ssLeft	Нажата левая кнопка мыши
ssRight	Нажата правая кнопка мыши
ssMiddle	Нажата средняя кнопка мыши
ssDouble	Выполнен двойной щелчок

ПОДСКАЗКА При обработке щелчка мыши правильнее анализировать событие OnMouseUp, а не событие OnMouseDown. При отпускании кнопки соответствующее событие генерируется всегда один роз, а при нажатии кнопки Windows генерирует сообщение о нажатии непрерывно, что приводит к многократному вызову обработчика события OnMouseDown.

В системе *Delphi* имеется еще одно полезное событие OnMouseMove, вызываемое при перемещении указателя мыши над объектом. В заголовке обработчика этого события

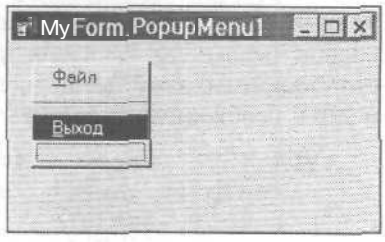
указываются только координаты текущей позиции указателя и состояние кнопок мыши.

```
procedure TMyForm.FormMouseMove(Sender: TObject;
   Shift: TShiftState; X, Y: Integer);
```

Как правило, обрабатывать это событие требуется при реализации действий, связанных с выделением области формы при нажатой кнопке мыши.

Компонент Контекстное меню (TPopupMenu)

Контекстное меню, вызываемое в любом грамотно сделанном приложении *Windows по* щелчку правой кнопки мыши, является стандартной и удобной возможностью многих программ. Компонент TPopupMenu (Всплывающее меню) предназначен для создания именно таких контекстных меню.



После того как компонент TPopupMenu размещен на форме (он почти не отличается от компонента TMainMenu и также является невизуальным), структура будущего меню формируется способом, описанным при создании строки меню. Единственное отличие состоит в том, что в контекстном меню не может быть нескольких разделов верхнего уровня, так как все его пункты располагаются в одном вертикальном столбце.

Дважды щелкнув на пункте Выход, в обработчик события можно скопировать ранее сформированный оператор завершения работы.

```
procedure TMyForm.N7Click(Sender: TObject);
begin
if CloseQuery then Close;
end;
```

Можно также напрямую вызвать обработчик N4Click(), но в данном примере это непринципиально, так как не приводит к заметной экономии труда программиста.

Аналогичным способом создается и обработчик события, возникающего при выборе пункта меню Сложить. В качестве его реализации можно вновь использовать ранее подготовленный метод Button1Click():

```
procedure TMyForm.N5Click(Sender: TObject);
begin
Button1Click(Sender)
end;
```

Теперь осталось вызвать созданное меню из программы. Для этого надо обработать щелчок правой кнопкой мыши на форме. Как уже говорилось выше, правильнее обрабатывать отпускание кнопки.

Чтобы добавить в программу обработчик такого события, надо в Проектировщике форм выделить главную форму и в Инспекторе объектов на вкладке Events дважды щелкнуть на строке с надписью OnMouseUp. Система *Delphi* 7 сгенерирует следующий текст.

```
procedure TMyForm.FormMouseUp(Sender: TObject;
   Button: TMouseButton;
   Shift: TShiftState,- X, Y: Integer);
begin
end;
```

В теле данной процедуры необходимо активизировать контекстное меню PopupMenu1. Это осуществляется с помощью метода Popup() с двумя параметрами — координатами верхнего левого угла меню. Только указываются эти координаты в экранной системе отсчета, а не в границах клиентской области программы. Чтобы указать корректное значение координат, надо добавить к ним смещение, определяющее начало клиентской части формы.

К сожалению, использовать свойства формы Left и Top, определяющие ее левую и верхнюю границы па экране (значения эгим свойствам задаются автоматически при перемещении формы на экране в Проектировщике форм) не удастся, потому что верхняя граница не учитывает дополнительной высоты строки заголовка и панели командных кнопок. Правильный подход состоит в том, чтобы пересчитать экранные координаты точки щелчка в координаты клиентской области с помощью метода формы GetClientOrigin. Эта функция обращается к стандартным функциям *Windows* и возвращает корректное значение смещения. Результирующее значение имеет тип TPoint (Координатная точка), представляющий собой запись из двух элементов: X и Y.

Итак, *чтобы* в рассматриваемом примере вызвать *меню* и показать его в месте щелчка мышкой (рис. 2.8), надо предварительно определить сдвиг клиентской области относительно верхнего левого угла экрана, проверить, былали нажата правая кнопка мыши, и обратиться к методу Popup:

```
procedure TMyForm.FormMouseUp(Sender: TObject;
   Button: TMouseButton;
   Shift: TShiftState; X, Y: Integer);
var P: TPoint;
begin
P := GetClientOrigin;
if Button = mbRight then PopupMenu1.Popup(P.X+X,P.Y+Y);
end;
```

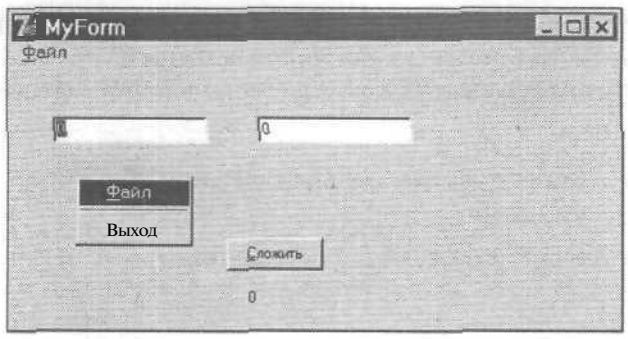


Рис. 2.8. Выдача контекстного меню по щелчку правой кнопкой

Стандартные классы системы Delphi 7

Многие компоненты имеют достаточно сложное внутреннее устройство, связанное с необходимостью обрабатывать сложные структуры данных, хранящиеся в различных элементах управления. Например, компонент Список должен уметь обращаться к любой строке списка по ее номеру, выполнять поиск, сортировку и другие операции. Подобные возможности нужны и в элементе управления Текстовая область и в ряде других. Многократно создавать для них методы, выполняющие одинаковую работу, нерационально, а помещать средства хранения и обработки строк в базовый класс неверно, потому что тогда эти возможности будут наследоваться и другими компонентами, не нуждающимися, например, в обработке текста.

Для решения этой проблемы в системе *Delphi 7* найден простой выход. В ней создан ряд стандартных абстрактных классов, не привязанных к компонентам. Объекты соответствующего типа просто включаются в конкретные компоненты в качестве свойств.

Список строк

Список строк (класс TStrings, описанный в модуле Classes) полностью соответствует своему названию и предназначен для:

- хранения строк;
- обращения к списку строк как к массиву и получения строки по ее номеру (индексация начинается с нуля);
- поиска строк;
- добавления или удаления строк в конкретные места списка.

```
for i := 1 to 10 do
   SList.Add('добавление в конец списка ' + IntToStr(i));
SList.Insert(5, 'добавление в 6-ю позицию списка');
SList[3] := 'изменение 4-й строки';
```

```
WriteLn(SList[1]); // вывод 2-й строки
SList.Delete(0); // удаление первой строки
```

Для удаления всех строк применяется метод Clear:

```
SList.Clear;
```

Для сравнения списков служит метод Equals:

```
if SList.Equals(MyList) then ...
```

Для перестановки двух строк местами предназначен метод Exchange:

```
SList.Exchange(0,3);
```

Переместить строку со старого места на новое позволяет метод Move:

```
SList.Move(1,8);
```

Поиск первого вхождения строки в список осуществляет метод IndexOf:

```
i := SList.IndexOf('строка 5');
```

Если строка в списке не найдена, то возвращается значение -1.

Сохранить весь список в файле можно при помощи метода SaveToFile:

```
SList.SaveToFile('C:\Strings.TXT');
```

Для чтения списка из файла служит метод LoadFromFile:

```
SList.LoadFromFile('C:\Strings.TXT');
```

Каждая строка текста в исходном файле должна быть отделена от следующей символом возврата каретки.

Все содержимое списка строк можно представить в виде одной длинной строки, обратившись к свойству Text:

```
WriteLn( SList.Text );
```

Список строк имеет определенную прикладную направленность. В системе *Windows* стандартным считается способ хранения настроек программы в виде файла .INI, когда название характеристики и ее значение разделяются знаком равенства. Вместо знака равенства можно использовать другие символы — они задаются свойством NameValueSeparator. В частности, собственные настройки *Windows* хранятся в файлах WIN.INI и SYSTEM.INI.

```
iCountry=7
```

Список строк позволяет автоматически выделять из таких строк название (часть текста до знака равенства) и значение (часть текста после знака равенства). Для этого служат соответственно свойства

```
Names[номер-строки]
```

и

```
Values[название]
```

Например, если выполнен оператор

```
SList[1] := 'Левая позиция окна=100';
```

то значением выражения Names[1] будет строка «Левая позиция окна», а значением выражения Values['Левая позиция окна'] — строка «100». Для получения значения по индексу можно воспользоваться свойством ValueFromIndex[номер-строки].

В строках подобного формата можно выполнять поиск с помощью метода IndexOfName, который в качестве параметра получает часть строки, относящуюся к названию, а возвращает номер первой строки с подходящим названием или число -1 при неудаче.

В описанном примере вызов функции

```
SList.IndexOfName('Левая позиция окна')
```

вернет значение 1.

Очень полезная особенность списка строк — возможность хранить связанные со строками объекты (класса TObject и унаследованных от него классов). Представьте себе, что пользователю предлагается выбрать строку из длинного списка (в элементе управления), который последовательно заполнялся значениями строк из массива, а затем был автоматически отсортирован. В такой ситуации невозможно определить соответствие выбранной строки номеру в исходном массиве обычными средствами без сравнения выбранной строки с каждым элементом массива (и даже это не гарантирует результат, если в массиве могут присутствовать одинаковые строки).

Возможность связывать объекты со строками пригодится вам и в том случае, когда в списке хранятся названия объектов и объекты желательно связать с этими названиями.

Если требуется заполнить список, который будет затем отсортирован автоматически, и сохранить возможность определения исходных номеров строк, то удобнее воспользоваться методом AddObject, который позволяет вместе со строкой указать сопутствующую информацию в виде объекта класса TObject.

```
for i := 1 to 10 do
  SList.AddObject('строка',TObject(i));
```

Хотя все элементы списка SList теперь имеют одинаковые значения, их можно без проблем различать по уникальным номерам от 1 до 10. Для этого надо воспользоваться методом IndexOfObject, который в качестве параметра получает ссылку на объект, а в качестве значения возвращает номер соответствующей строки в списке или -1 в случае неудачи:

```
i := SList.IndexOfObject(TObject(7));
```

У класса TStrings имеется также полезное свойство Count, хранящее число строк в списке.

Хотя в компонентах в качестве типа соответствующих свойств обычно указывается только что описанный класс TStrings, использовать его в программе явно нельзя. В частности, нельзя пытаться создать новый объект с помощью конструктора:

```
SList := TStrings.Create;
```

Но так как программисту вполне могут потребоваться возможности такого мощного класса, в системе *Delphi 7* реализован его наследник, который называется TStringList,

содержит всю функциональность родителя и допускает свое использование в программе. Чтобы включить в исходный текст приведенные выше команды, надо переменную SList описать как имеющую тип TStringList:

```
var SList: TStringList;
...
SList := TStringList.Create;
SList.AddObject('строка',TObject(12));
```

В классе TStringList добавилось несколько новых полезных методов и свойств. Метод Sort позволяет выполнить сортировку всех строк в списке в порядке возрастания, при условии, что свойство Sorted имеет значение False (в противном случае при добавлении новой строки она будет автоматически помещаться в список в соответствии с требованиями упорядоченности). В связи с тем, что класс TStringList поддерживает сортировку, немного изменилась работа ранее описанных методов. Теперь при добавлении, перемещении или изменении строк в списке выполняется проверка свойства Sorted, и при необходимости изменения вносятся в упорядоченный список с учетом значений строк. Кроме того, хранить в списке строки с одинаковыми значениями не разрешается. Чтобы при попытке добавить новую строку не возникало ошибок, надо предварительно проверить, не присутствует ли уже такая строка в списке. Для этого предназначен метод Find, который имеет следующий заголовок:

```
function Find(const S: string; var Index: Integer)
   : Boolean; virtual;
```

Проверяется, можно ли добавить строку в отсортированный список. Если строка S в списке уже есть, функция возвращает значение True, если нет — False, а в переменную Index записывается номер позиции в списке, куда будет добавлена строка S.

ВНИМАНИЕ Функцию Find можно применять только для отсортированных списков. Для несортированных списков надо использовать метод IndexOf.

Из новых свойств надо отметить свойство Objects, представляющее собой массив объектов, ассоциированных со строками. Получить доступ к нужному объекту можно с помощью индекса:

```
SList.Objects[4]
```

Свойство Sorted указывает, надо ли выполнять автоматическую сортировку списка (то есть, добавлять новые записи не в коней списка, а в соответствии с правилами упорядочивания символов для национальной версии *Windows).* Если это свойство имеет значение False и в него записывается значение True, то программа немедленно выполняет сортировку всего списка (для больших списков это может оказаться длительным процессом).

Свойство Duplicates определяет, что произойдет в программе, когда обнаружится попытка добавления в отсортированный список строки, которая уже содержится в нем.

Это свойство может принимать одно из трех значений.

Таблица 2.2. Значение свойства Duplicates

Значение свойства	Действие
dupIgnore	Попытка добавления строки игнорируется
dupError	Возникает ошибка
dupAccept	Производится добавление строки, совпадающей с одной из существующих

ЗАМЕЧАНИЕ Другие стандартные классы Delphi будут далее рассматриваться по мере необходимости.

Компонент Текстовая область (TMemo)

Обойтись простым текстовым полем удается не всегда. Если пользователь должен ввести большой объем информации (например, полный почтовый адрес или произвольный комментарий), ему может понадобиться несколько строк текста. В таком случае следует использовать компонент TMemo.

При вводе текста для перехода на новую строку (к новому абзацу) обычно используется клавиша ENTER. Однако в диалоговых окнах *Windows* эта клавиша часто применяется для завершения ввода. Способ использования клавиши ENTER определяется значением свойства WantReturns. Если оно имеет значение True, то клавиша ENTER позволяет переходить к новой строке внутри текстовой области, в противном случае она служит для завершения ввода и перехода к следующему элементу управления, а для перехода к новой строке применяется комбинация клавиш CTRL+ENTER.

Главное свойство данного компонента — Lines (Строки), имеющее тип TStrings. В нем хранится список строк, введенных пользователем. Эти строки можно обрабатывать всеми методами, доступными в классе TStrings, например сохранять в файле:

```
Memo1.Lines.SaveToFile('C:\Memo.TXT');
```

Наличие у текстовой области полос прокрутки задается в свойстве ScrollBars.

Таблица 2.3. Настройка свойства ScrollBars

Значение	Вид текстовой области
ssNone	Полосы прокрутки отсутствуют
ssHorizontal	Имеется горизонтальная полоса прокрутки
ssVertical	Имеется вертикальная полоса прокрутки
ssBoth	Имеются две полосы прокрутки

Если включена горизонтальная полоса прокрутки, значение свойства Wordwrap игнорируется. Это свойство определяет, будет ли выполняться автоматический перенос

слов на новую строку при достижении правой границы области (при этом никаких символов новой строки в текст не добавляется — перенос отображается только на экране).

При выделении фрагмента текста в текстовой области в свойство SelStart записывается позиция первого выделенного символа, а в свойство SelLength — число выделяемых символов. Выделенный текст доступен через свойство SelText (тип string). Для выделения всего текста применяется метод SelectAll, для удаления выделенного текста — метод ClearSelection.

Чтобы очистить содержимое текстовой области, используется метод Clear, чтобы отменить последние изменения — метод Undo, а чтобы очистить буфер, хранящий историю изменений, и сделать такую отмену невозможной — метод ClearUndo.

Группа методов предназначена для работы с буфером обмена *Windows*. Для копирования выделенного текста в буфер обмена применяется метод CopyToClipboard, для вырезания текста — метод CutToClipboard, для вставки текста из буфера — метод PasteFromClipboard.

Когда в текстовой области происходит изменение текста, генерируется событие OnChange.

Компонент Флажок (TCheckBox)

Данный компонент используется для фиксации включенного или выключенного состояния (одного из двух).

После размещения компонента Флажок на форме подпись к этому элементу управления можно задать в свойстве Caption. Расположение этой подписи определяется свойством Alignment: значение taRightJustify означает расположение подписи справа, а значение taLeftJustify — слева. Главное свойство флажка называется Checked. Оно доступно для изменения и на этапе проектирования, и на этапе работы программы. Это свойство принимает значение True, если флажок включен, и False, если он сброшен.

Некоторые флажки могут находиться в третьем, «частично установленном» состоянии, когда признак установки флажка отображается приглушенным цветом. Такая возможность нужна, когда требуется сообщить пользователю о неполном соответствии указанному статусу (например, в ходе установки программ таким образом сообщается, что для установки выбраны не все доступные средства).

Если для свойства AllowGrayed задано значение True, то флажок при последовательных щелчках на нем будет поочередно принимать состояния «сброшен», «установлен частично», «установлен». Определить текущее состояние или задать новое из числа доступных можно, проверив или изменив свойство State (табл. 2.4). Чтобы реагировать на изменение состояния флажка, надо создать обработчик события OnClick (При щелчке).

Таблица 2.4. Значения свойства State

Значение	Состояние флажка
cbUnchecked	Сброшен
cbGrayed	Установлен частично
cbChecked	Установлен

Рассмотрим пример, когда при изменении состояния флажка его текущее состояние выводится в надпись Label1. Для этого надо разместить на форме соответствующий компонент, установить значение свойства AllowGrayed равным True, сформировать обработчик события OnClick и записать в нем оператор выбора текущего состояния флажка (рис. 2.9).

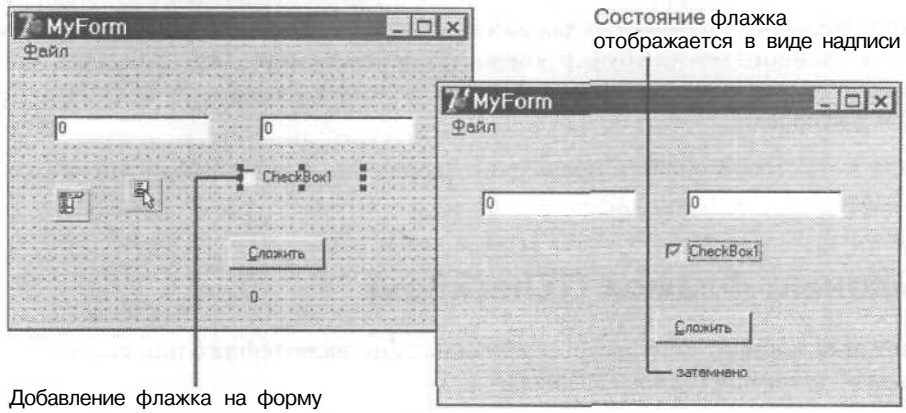


Рис. 2.9. Программа в работе

```
procedure TMyForm.CheckBox1Click(Sender: TObject);
begin
case CheckBox1.State of
   cbUnchecked: Label1.Caption := 'выключено';
   cbGrayed : Label1.Caption := 'затемнено';
   cbChecked : Label1.Caption := 'включено';
end
end;
```

Компонент Переключатель (TRadioButton)

В отличие от флажка, переключатель предназначен для выбора одного значения из ряда возможных. Переключатели всегда используются группами. Когда пользователь выбирает один из них, то выделение с текущего переключателя снимается. Таким образом, в группе выделен всегда ровно один переключатель.

Так как мы создаем группу переключателей, на форме надо расположить несколько компонентов TRadioButton. Программа поймет, что выделенным должен быть только один из них.

Свойства компонента Переключатель аналогичны свойствам компонента Флажок. Свойство Alignment определяет положение подписи справа или слева от переключателя, а свойство Checked — состояние объекта (True, если переключатель включен).

Методы GetChecked и SetChecked, позволяющие обращаться к свойству Checked и изменять его значение, в тексте программы явно не используются. Они предназначены для использования в классах-наследниках TRadioButton (конкретных реализациях переключателя) в соответствии с принципом инкапсуляции.

Для отслеживания состояния конкретного переключателя можно обрабатывать событие OnClick.

На форме достаточно разместить несколько переключателей, и после компиляции и запуска программы будет выделен всегда только один из них. Во время проектирования один из переключателей желательно включить, а все остальные по умолчанию оставить выключенными.

Если требуется отслеживать состояние переключателей динамически, надо создать обработчик события OnClick для каждого из них. О новом статусе переключателя можно сообщить, например, с помощью надписи Label.

```
procedure TMyForm.RadioButton1Click(Sender: TObject);
begin
   if RadioButton1.Checked
   then Label1.Caption := 'Включен первый'
end;
procedure TMyForm.RadioButton2Click(Sender: TObject);
begin
   if RadioButton2.Checked
   then Label1.Caption := 'Включен второй'
end;
```

Компонент Группа переключателей (TRadioGroup)

Если в программе требуется использовать несколько групп переключателей (например, одну для указания пола человека, а другую для выбора возрастной категории), можно применить один из двух подходов. Первый состоит в выделении для каждой группы специального объекта (панели), чтобы система *Delphi* могла понять, как объединяются переключатели. Второй подход состоит в использовании компонента TRadioGroup, который объединяет свойства и методы, обеспечивающие поддержку работы группы переключателей.

После размещения на форме компонента TRadioGroup входящие в него переключатели задаются перечислением их названий. Эти названия вводятся в свойство Items, имеющее тип TString. Так как требуется ввести не одну строку, а несколько, для их

ввода предусмотрен специальный редактор, который вызывается щелчком на специальной кнопке ..., расположенной справа в строке, описывающей свойство Items. Большая текстовая область окна редактора предназначена для ввода названий переключателей, по одному в каждой строке. Не следует забывать о поддержке возможности управления программой с помощью клавиатуры, поэтому перед некоторыми буквами в названиях надо указать символ &, чтобы сделать эти буквы «горячими». Затем щелкните на кнопке OK, и внешний вид объекта RadioGroup1 на форме сразу изменится (рис. 2.10).

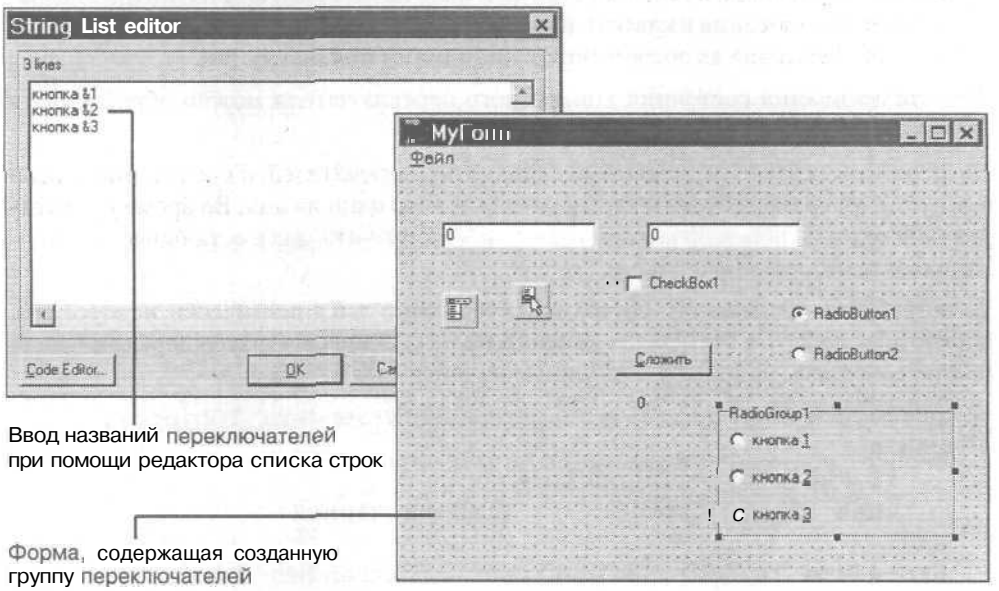


Рис. 2.10. *Создание двух групп переключателей*

Если теперь откомпилировать и запустить программу, то в ней будут независимо работать две группы переключателей. Первая создана ранее из отдельных элементов RadioButton1 и RadioButton2, а вторая, RadioGroupl, сформирована как единый объект (рис. 2.11).

Так как компонент TRadioGroup представляет единое целое со своими переключателями, использовать его в программе надо совсем не так, как компонент TRadioButton.

Так, свойство Caption определяет не подпись каждого переключателя (эти подписи теперь задаются в свойстве Items), а заголовок группы (исходно она называется RadioGroupl). Свойство Columns задает число столбцов (первоначально один), образованных переключателями. Свойство ItemIndex (исходное значение -1) содержит номер выделенного переключателя (число –1 показывает, что ни один переключатель не выбран). Значение этого свойства изменяется автоматически, когда пользователь выбирает один из переключателей группы. Разрешается менять его и программно: при занесении нового значения в свойство ItemIndex изменится и текущий выбранный переключатель на форме.

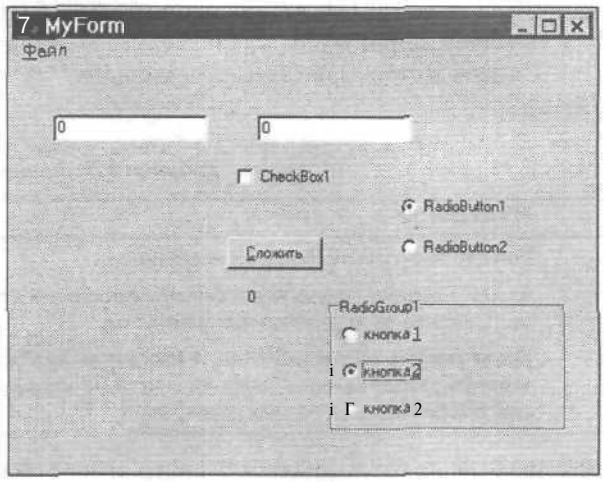


Рис. 2.11. *Программа в работе. Две группы переключателей созданы с использованием различных средств*

Динамически реагировать на выбор нового переключателя в группе можно с помощью обработчика события OnClick. Например, чтобы отображать с помощью надписи Label1 название текущего переключателя, надо обратиться к свойству Items (список строк) и выделить тот элемент, номер которого записан в свойстве ItemIndex. Предварительно следует проверить, имеется ли вообще выделенный переключатель (не равно ли значение свойства ItemIndex -1), или выбрать один из переключателей на этапе проектирования (например, присвоив свойству ItemIndex значение 0).

```
procedure TMyForm.RadioGroup1Click(Sender: TObject);
begin
   if RadioGroup1.ItemIndex > -1 then
   Label1.Caption := 'Выбран ' +
   RadioGroup1.Items[ RadioGroup1.ItemIndex ]
end;
```

Компонент Список (TListBox)

Компонент Список (TListBox) очень часто применяется в программах для *Windows.* Он позволяет выбрать одну или несколько строк в списке.

Первоначально компонент TListBox, размещенный на форме, изображается в виде пустого квадрата. Его размеры можно настроить протягиванием мыши.

Список может иметь несколько столбцов. Это не означает, что каждый столбец представляет собой отдельный список, просто при заполнении видимой части списка строками донизу очередная строка отображается в следующем столбце. В остальном работать такой список будет так же, как обычный список.

Число столбцов задается в свойстве Columns. Если оно больше 0, то на каждую колонку отводится часть общей ширины списка (значение свойства Width, деленное на число столбцов). Список может принадлежать к одному из трех возможных типов (свойство Style).

Таблица 2.5. *Значение свойства Style*

Значение	Стиль списка
lbStandard	Стандартный список (стиль по умолчанию)
lbOwnerDrawFixed	Каждый элемент списка имеет фиксированную высоту, но способ его отображения определяется программистом
lbOwnerDrawVariable	Кроме способа рисования в тексте программы необходимо явно задавать размер каждого элемента списка (что позволяет создавать списки с элементами разных размеров)

Если значение свойства Style не равно IbStandard, программист должен сформировать для списка обработчик события OnDrawItem, в котором самостоятельно организовать отрисовку содержимого. Такой список может содержать не только текст, но и другие объекты, например рисунки. Узнать высоту элемента в пикселах можно, обратившись к свойству ItemHeight.

Если высота элементов списка может быть различной, то перед обработкой события OnDrawItem надо обработать событие OnMeasureItem. В заголовке соответствующего обработчика имеется параметр, описанный так:

```
var Height: Integer;
```

В этот параметр, передаваемый по ссылке, надо записать размер очередного элемента (его номер указывается в параметре Index), а затем уже обработать событие OnDrawItem с учетом текущей высоты элемента.

Выбирать элементы в списке можно по одному, а можно и по нескольку одновременно. Свойство MultiSelect (тип Boolean) позволяет выбирать несколько элементов, если его значение равно True. Выделить сразу группу элементов можно, если при щелчке на последнем из них держать нажатой клавишу SHIFT. Такая возможность появляется, если в свойство ExtendedSelect занесено значение True.

Свойство SelCount (оно доступно только для чтения и не может быть изменено в тексте программы) содержит число выделенных строк списка. Чтобы определить, выделен ли конкретный элемент, надо, используя его номер, обратиться к свойству Selected, представляющему собой массив типа Boolean, Значение выражения

```
ListBox.Selected[4]
```

будет равно True, если пятый элемент (отсчет ведется с нуля) в списке выделен.

Если выделение нескольких элементов не разрешено, то узнать, какой элемент выделен, можно, обратившись к свойству ItemIndex, хранящему номер единственного выделенного элемента.

ПОДСКАЗКА Исходно выделенные элементы в списке отсутствуют. Это означает, что в свойстве ItemIndex записано значение -1- При попытке обратиться к какому-то свойству, допускающему индексацию (например, Items), используя ItemIndex в качестве индекса, возникнет ошибка выхода индекса за допустимые границы. Поэтому желательно после первого заполнения списка явно выделить один из элементов в тексте программы, занеся в свойство ItemIndex значение, отличное от −1.

Содержимое списка хранится в уже знакомом нам свойстве Items (список строк, класс TStrings). Строки можно задать на этапе проектирования, в специальном редакторе, так же, как задавалось содержимое группы переключателей, а можно и во время работы программы с помощью метода Add класса TStrings.

Например, если имеется потребность в занесении в список значений, получаемых в результате сложения, то на форму можно добавить новую кнопку Добавить (она автоматически получит название Button2), а в обработчике щелчка на этой кнопке (событие OnClick) вызывать метод Add (рис. 2.12).

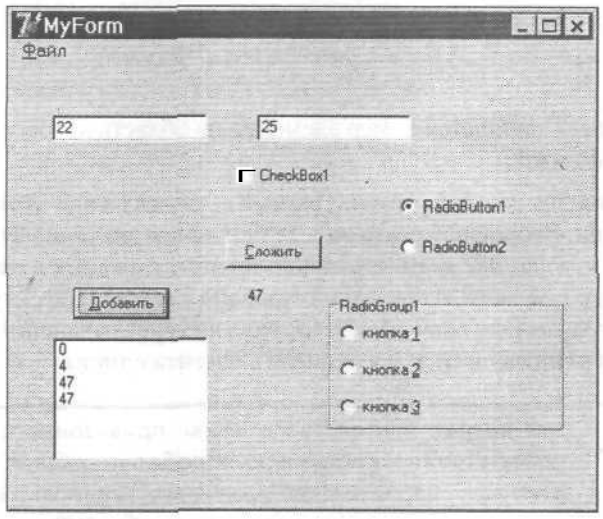


Рис. *2.12. В левом нижнем углу формируется список результатов*

```
procedure TMyForm.Button2Click(Sender: TObject);
begin
ListBox1.Items.Add( Label1.Caption );
end;
```

Элементы списка можно упорядочить в алфавитном порядке, занеся в свойство Sorted значение True. После этого при добавлении новых элементов они будут сортироваться автоматически.

Для очистки всего содержимого списка используется метод Clear:

```
ListBox1.Clear;
```

Для удаления конкретного элемента служит метод DeleteString:

```
ListBox1.DeleteString(4);
```

При этом, конечно, помимо перерисовки списка вносятся соответствующие изменения в свойство Items.

Еще один полезный метод, который часто используется для вызова контекстного меню конкретного элемента списка, называется ItemAtPos. **Он** переводит координату точки внутри списка в номер элемента, в рамках которого лежит эта точка. Его удобнее всего использовать в обработчике щелчка (отпускания) кнопки мыши для объекта ListBox.

```
procedure TMyForm.ListBox1MouseUp (Sender: TObject;
Button: TMouseButton;
Shift: TShiftState; X, Y: Integer);
var Point: TPoint;
Index: Integer;
begin
Point.X := X; Point.Y := Y;
Index := ListBoxl .ItemAtPos(Point, True) ;
end;
```

В переменную Index запишется номер элемента, на область которого внутри списка пришелся щелчок мыши.

У функции ItemAtPos два параметра. Первый — объект типа TPoint, содержащий координаты точки. Второй параметр определяет, какое значение будет возвращено, если щелчок выполнен вне границ реально присутствующих в списке строк (это может быть только в области за последней строкой). Если в такой ситуации в качестве второго параметра указано значение True, функция вернет значение –1, если False – увеличенный на единицу номер последнего элемента списка.

ЗАМЕЧАНИЕ Для вызова контекстного меню, привязанного к конкретному объекту, можно использовать обработчик события OnContextPopup (которое генерируется по умолчанию, если пользователь нажимает правую кнопку мыши над областью объекта).

Компонент Поле со списком (TComboBox)

Этот компонент представляет собой вариант списка с присоединенным дополнительным полем, в котором отображается выбранный элемент списка. Это же поле может использоваться для ввода новых элементов или для быстрого поиска элемента по начальным символам. Если на экране отображается

только присоединенное поле («раскрывающийся список»), то для раскрытия списка можно использовать клавиатурную комбинацию ALT+ВНИЗ.

Компонент Поле со списком может работать в трех разных режимах, определяемых значением свойства Style.

Таблица 2.6. Значения свойства Style

Значение	Механизм работы списка
csDropDown	В присоединенном поле можно указывать значения, отсутствующие в списке. Свойство MaxLength определяет максимально допустимое число символов, которое можно ввести в это поле (значение О указывает на отсутствие ограничений). Текст, введенный пользователем, доступен через свойство Text. Список раскрывающийся
csDropDownList	Допустим только выбор значений, уже имеющихся в списке. Список раскрывающийся
csSimple	Отличается от стиля csDropDown только тем, что список не является раскрывающимся

Как и в случае обычного списка, вместо режима работы csDropDown можно указать аналогичные режимы csOwnerDrawFixed и csOwnerDrawVariable, которые отличаются только необходимостью программной отрисовки содержимого каждого элемента (см. компонент TListBox).

Вводимый текст может автоматически преобразовываться к верхнему регистру (если свойство CharCase имеет значение ecUpperCase), к нижнему (ecLowerCase) или никак не преобразовываться (ecNormal, по умолчанию).

Максимальное число элементов, одновременно отображаемых в видимой части списка, задается в свойстве DropDownCount. Чтобы открыть список из программы, свойству Dropped Down (Раскрыт) надо присвоить значение True.

Понятия «выделенная строка* в раскрывающемся списке нет. В нем имеется только текущая выбранная строка (ее номер в списке хранится в свойстве ItemIndex). Соответственно, нельзя и выделять строки списка. Единственный метод, связанный с выделением данных, — это процедура SelectAll, которая выделяет весь текст, введенный пользователем в присоединенное поле. При работе раскрывающегося списка наиболее важными являются представленные ниже события.

Таблица 2.7. События класса TComboBox

Название	Условия генерации
OnChange	Пользователь изменил текст в присоединенном поле
OnDropDown	Список раскрывается. Это событие необходимо обрабатывать, если содержимое списка может меняться во время работы программы. Тогда в обработчике этого события можно заново сформировать содержимое списка (свойство Items)

Компонент Полоса прокрутки (TScrollBar)

Полосы прокрутки обычно используются как вспомогательные инструменты в других элементах управления: списках, текстовых областях и прочих. Однако их можно довольно эффективно применять и в качестве самостоятельных элементов управления, например для приблизительной, грубой настройки значений с помощью ползунка.

После того как компонент TScrollBar размещен на форме, надо определить его вид, который задается значением свойства Kind (sbHorizontal — горизонтальная полоса, sbVertical — вертикальная). Диапазон значений, охватываемых полосой прокрутки, указывается в свойствах Min (минимальное значение) и Max (максимальное значение). Текущая позиция ползунка определяется свойством Position (значение должно лежать в диапазоне от Min до Max). При перемещении ползунка значение в этом свойстве изменяется автоматически. Но можно выполнить такое изменение и программно — тогда ползунок сам переместится в нужную позицию.

Определить значения свойств Min. Max и Position можно «за один раз» с помощью метода SetParams:

```
procedure SetParams(APosition, AMin, AMax: Integer);
```

При щелчке на кнопках полосы прокрутки или при нажатии курсорных клавиш значение свойства Position изменяется на величину, указанную в свойстве SmallChange. Ползунок перемещается в соответствии с этим изменением. Если же щелчок выполняется на свободной области полосы прокрутки или происходит нажатие клавиш PAGE UP/ PAGE DOWN (листание страниц), то значение свойства Position (и соответствующее ему положение ползунка) изменяется на величину, указанную в свойстве LargeChange.

При изменении значения свойства Position всегда генерируется событие OnChange.

Полностью переписать функционирование полосы прокрутки можно, создав обработчик события OnScroll.

```
procedure TMyForm.ScrollBar1Scroll(Sender: TObject;
   ScrollCode: TScrollCode; var ScrollPos: Integer);
```

В переменную ScrollPos можно занести новое значение положения ползунка (которое потом автоматически скопируется в свойство Position). Новое значение надо сформировать на основе анализа действий пользователя, описываемых параметром ScrollCode.

***Таблица** 2.8. Значения параметра ScrollCode*

Значение	**Действие пользователя**
scLineUp	Нажата клавиша ВВЕРХ или ВЛЕВО или произведен щелчок на соответствующей кнопке полосы прокрутки
scLineDown	Нажата клавиша ВНИЗ или ВПРАВО или произведен щелчок на соответствующей кнопке полосы прокрутки
scPageUp	Нажата клавиша PAGE UP или произведен щелчок слева от ползунка (выше ползунка)

Таблица 2.8. Значения параметра ScrollCode (продолжение)

Значение	Действие пользователя
scPageDown	Нажата клавиша PAGE DOWN или произведен щелчок справа от ползунка (нижеползунка)
scPosition	Завершено перемещение ползунка перетаскиванием
scTrack	Идет перемещение ползунка перетаскиванием
scTop	Ползунок установлен в крайнее левое или верхнее положение
scBottom	Ползунок установлен в крайнее правое или нижнее положение
scEndScroll	Операция прокрутки завершена

Иерархия компонентов Delphi 7

Прежде чем перейти к рассмотрению остальных компонентов и возможностей системы *Delphi 7,* необходимо познакомиться с базовой иерархией классов этой системы, их наиболее важными свойствами и методами.

Класс TObject

Данный класс лежит в основе всей иерархии классов *Delphi 7.* Он обладает самыми общими методами, присущими любому объекту Паскаля. Класс TObject описывает основные принципы поведения объектов во время работы программы (создание, уничтожение, обработка событий и так далее).

Б практической работе необходимы представленные ниже методы класса TObject.

Таблица 2.9. Методы класса TObject

Метод	Назначение
class function ClassName: ShortString;	Возвращает название типа объекта (например, «Tlabel»)
class function ClassNameIs(const Name: string); Boolean;	Возвращает True, если обьект относится к указанному типу
constructor Create;	Конструктор. Выделение памяти для объекта
destructor Destroy;	Деструктор. Использовать напрямую не рекомендуется
procedure Free;	Удаляет объект и освобождает занятую им память

Класс TPersistent (наследник TObject)

На уровне этого класса реализованы основные методы копирования содержимого объектов.

Таблица2.10. *Основные методы класса TPersistene*

Метод	Назначение
procedure Assign (Source: TPersistent);	Копирование содержимого объекта-параметра е текущий объект. Например: Destignation. Assign (Sou rce);
procedure AssignTo(Dest: TPersistent);	Копирование содержимого текущего объекта в объект, заданный а качестве параметра
function GetOwner: TPersistent;	Возвращает «хозяина» объекта

Класс TComponent (наследник TPersistent)

Класс TComponent является основным родительским классом для всех классов, описывающих компоненты *Delphi 7.*

В этот класс входит набор самых общих свойств, имеющихся у каждого компонента *Delphi* 7 (табл. 2.11), и некоторые полезные методы (табл. 2.12).

Таблица 2.11*Свойства класса TComponent*

Свойство	Назначение
ComponentCount	Число объектов, подчиненных данному
ComponentIndex	Номер компонента в свойстве Components
Components	Список объектов, подчиненных данному
ComponentState	Текущее состояние компонента
ComponentStyle	Стиль компонента
Name	Название компонента в программе
Owner	Хозяин компонента
Tag	Вспомогательное свойство, хранящее число типа Integer, которое может использоваться для собственных целей разработчика

Таблица 2.12. *Методы класса TComponent*

Метод	Назначение
function FindComponent(• constAname: string): TComponent;	Поиск подчиненного объекта, подходящего по названию (параметр AName)
function GetParentComponent: TComponent;	Определение собственника текущего объекта
Procedure InsertComponent(Acomponent: TComponent);	Добавление объекта в конец списка Components
Procedure RemoveComponent(AComponent: TComponent);	Удаление объекта из списка Components

Класс TControl, наследник TComponent, служит основным классом для всех визуальных элементов управления. Если такой элемент управления является стандартным элементом *Windows,* то он базируется на еще одном промежуточном классе TWinControl (наследнике класса TControl).

Форма

Форма — это важнейший компонент *Delphi 7*, на котором основана вся работа этой системы по проектированию и разработке приложений. Форма (класс TForm) содержит богатый набор свойств (табл. 2.13), методов и событий (табл. 2.14), позволяющих легко настраивать и организовывать самые сложные алгоритмы ее функционирования.

Таблица 2.13. *Свойства класса TForm*

Свойство	Назначение
Active	Содержит значение True, если форма имеет фокус ввода
ActiveControl	Объект на форме, который имеет фокус ввода
BorderIcons	Список системных значков формы
BorderStyle	Вид границ формы
Canvas	Область рисования формы
CLientRect ClientHeight ClientWidth	Размеры формы
DropTarget	Содержит значение True, если форма может работать как приемник в операциях перетаскивания
Floating	Содержит значение True, если форма может пристыковываться к другим окнам
FormState	Текущее состояние формы
FormStyle	Стиль формы
HelpFile	Название файла справки для формы
Icon	Значок, обозначающий форму, когда она свернута
KeyPreview	Содержит значение True, если форма будет получать информацию о нажатых клавишах раньше, чем расположенные на ней объекты
Menu	Ссылка на главное меню формы (TMenu)
ModalResult	Значение, возвращаемое формой, если она работает как модальное диалоговое окно
Parent	«Хозяин» формы
PixelsPerInch	Число пикселов на дюйм. Применяется для настройки размера формы в зависимости от экранного разрешения
Position	Положение формы на экране в момент ее открытия в программе
PrintScale	Масштабирование формы при выводе на печать
Scaled	Содержит значение True, если размер формы будет подгоняться в соответствии со значением свойства PixelsPerInch
Visible	Содержит значение True, если форма будет видима во время работы программы
WindowState	Состояние формы (свернута, развернута, нормальный размер)

Таблица 2.14. События, поддерживаемые классом TForm

Событие	Условия генерации
OnActivate	Форма стала активной
OnClose	Форма закрывается
OnCloseQuery	Запрос на закрытие формы
OnCreate	Форма создается
OnDeactivate	Форма потеряла фокус ввода
OnDestroy	Форма уничтожается
OnHelp	Форма получила запрос на выдачу справочной информации
OnHide	Форма стала невидимой (значение свойства Visible установлено равным False)
OnPaint	Форма должна быть перерисована
OnShortCut	Пользователь нажал клавиатурную комбинацию, которая пока не обработана
OnShow	Форма стала видимой (значение свойства Visible установлено равным True)

Управление проектом

Как уже говорилось, вся информация, относящаяся к текущей разрабатываемой программе, объединяется в рамках системы *Delphi 7* в один *проект,* который хранит все необходимые настройки в специальных файлах. Проектом управляет Менеджер проекта — программа, входящая в состав системы *Delphi 7.* В функции Менеджера входит визуальное представление структуры проекта и его содержимого (это могут быть не только файлы с исходными текстами и файлы форм, но и любые другие файлы). Менеджер проекта дополнительно позволяет вести одновременно несколько проектов, объединенных в одну группу. Он вызывается командой View ➤ Project Manager (Вид х Менеджер проекта) и наглядно показывает структуру группы ProjectGroup1 (рис. 2.13).

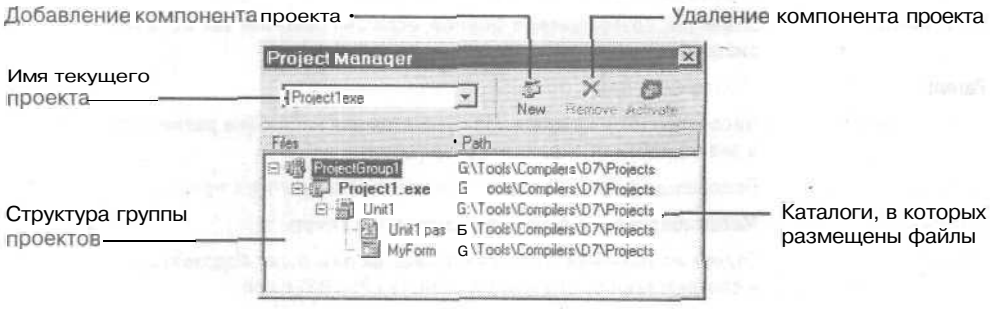


Рис. 2.13. Отображение сведений в Менеджере проектов

В эту группу пока что входит единственный проект Project1, состоящий, в свою очередь, из единственного модуля Unitl (он включает форму MyForm и файл Unit1.pas

с исходными текстами, описывающими работу этой формы). Тип результирующего приложения, которое будет получено в результате компиляции (это либо исполнимый код — файл .EXE, либо динамическая библиотека — файл .DLL), указан полужирным шрифтом. Для проекта, созданного командой File ➤ New > Application (Файл ➤ Создать > Приложение), по умолчанию считается, что он предназначен для получения исполнимого кода, поэтому в Менеджере проекта полужирной строкой выделено Project1.exe.

Любой компонент из проекта можно удалить, щелкнув на кнопке Remove (Удалить). С помощью кнопки New (Создать) можно добавить новый компонент как в проект, так и в группу. Свойства любого из объектов, доступных в Менеджере (от группы проектов до отдельной формы), можно изменить, щелкнув на значке объекта правой кнопкой мыши. Контекстное меню содержит набор пунктов, позволяющих выполнить и настройку, и компиляцию, и редактирование. Для быстрого перехода из Менеджера проекта к редактированию конкретного объекта (исходного текста или формы) достаточно дважды щелкнуть на значке этого объекта.

К группе проектов можно добавлять ранее созданные проекты. Это выполняется командой Add Existing Project (Добавить существующий проект) контекстного меню. Ранее созданные формы *и* соответствующие им файлы с исходными текстами добавляются к конкретному проекту командой Add (Добавить).

Структура проекта представлена в виде «дерева». Это стандартный подход к отображению иерархически организованной информации в *Windows*. Стандартны и основные приемы по работе с подобными «деревьями»: отдельные «листья» и «ветви» можно перемещать между узлами с помощью мыши.

Добавление новой формы

Продолжать изучение компонентов и возможностей системы *Delphi 7,* размещая объекты на одной форме, неудобно. Уже сейчас наша экспериментальная форма перенасыщена элементами управления. Поэтому надо либо создать новый проект (его лучите включить в текущую группу ProjectGroup1, чтобы сохранить целостность примеров), либо добавить к текущему проекту Project1 новую форму. Пока остановимся на последнем варианте.

Новая форма добавляется к текущему проекту одним щелчком мыши на командной кнопке New Form (Создать форму) или командой File > New ➤ Form (Файл > Создать ➤ Форма). При этом в Проектировщике сразу появится новая пустая форма. Называться она будет Form2, а соответствующий ей файл с исходными текстами добавится в редактор на новую панель Unit2. Теперь проект надо сохранить, при этом система *Delphi 7* поинтересуется названием нового модуля (пока что его лучше оставить без изменений — Unit2).

ЗАМЕЧАНИЕ Переключаться между имеющимися в проекте формами можно с помощью командной кнопки View Form (Отобразить форму) или комбинации клавиш SHIFT+F12.

У программы может быть только одна главная форма — это форма, которая показывается при запуске программы, — и неограниченное число подчиненных форм, вспомогательных окон, которые исходно на экране не появляются, а вызываются по команде из программы с помощью специальных методов.

ЗАМЕЧАНИЕ Исходно подчиненные формы не показываются по одной простой причине — значение их свойства Visible (Видимость) первоначально установлено в False. Свойство Visible имеют все без исключения компоненты Delphi 7, представляющие собой элементы управления. Изменяя значения свойства Visible во время работы программы, можно мгновенно делать любые объекты видимыми или невидимыми.

Добавим, например, к главной форме еще одну кнопку Button3 и назовем ее Окно.

При щелчке на ней должна отображаться форма Form2. Сделать это можно несколькими способами.

Показ формы как обычного окна

Чтобы форма отображалась как обычное окно, проще всего записать в свойство Visible формы Form2 значение True.

```
procedure TMyForm.Button3Click(Sender: TObject);
begin
Form2.Visible := true;
end;
```

Если теперь выполнить компиляцию проекта, то система Delphi 7 сообщит об ошибке: идентификатор Form2 в модуле Unit1 неизвестен. Одновременно система предложит включить в список подключаемых модулей новый модуль Unit2, где, по предположению системы, находится описание соответствующей переменной (рис 2.14).

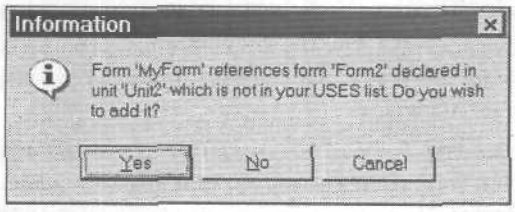


Рис. 2.14. Система предлагает включить в список подключаемых модулей новый модуль Unit2

В данном случае это действие оправдано, поэтому ответить надо Yes (Да) и выполнить компиляцию повторно — ошибок уже не будет. Однако лучше всего не забывать указывать ссылки на новые подключаемые к проекту модули самостоятельно.

Теперь, после запуска программы и щелчка на кнопке Окно на экране возникнет новое пустое окно (форма Form2). Она связана с родительским окном (главной формой MyForm): между ними можно свободно переключаться, а при закрытии главного окна автоматически закроются и все вспомогательные формы (но не наоборот!).

Если закрыть форму Form2, реально произойдет изменение значения ее свойства Visible с True на False, поэтому при щелчке на кнопке Окно форма появится опять. Если щелкнуть на кнопке Окно, когда форма Form2 видима, ничего не изменится.

Вместо оператора присваивания для показа формы лучше применять ее метод Show.

```
procedure TMyForm.Button3Click(Sender: TObject);
begin
Form2.Show;
end;
```

Он хорош тем, что показывает форму, перемещает ее на передний план экрана и делает активной.

Показ формы как модального окна

Между появившейся на экране формой Form2 и главным окном (MyForm), а также между другими подчиненными формами проекта, если бы они тоже были созданы и показаны, можно переключаться произвольным способом. Такой подход не всегда удобен, потому что не дает пользователю сосредоточиться на конкретном действии и позволяет, не закончив работу в одном окне, выполнять новые действия в другом окне. Подобный многооконный принцип при создании программ себя не оправдывает, так как требует от разработчика больших усилий по непрерывной координации состояний множества окон и только усложняет общение с человеком. Сегодня он применяется в основном в сложных системах, наподобие систем разработки типа *Delphi 7,* где одним окном не обойтись. А вспомогательные формы в обычных прикладных программах используются, как правило, для создания диалоговых окон, с которых невозможно переключиться на другие окна приложения, пока они не будут закрыты (такой режим работы окна еще называется *модальным*). Модальные окна хорошо подходят, в частности, для задания всевозможных настроек, выполнения ввода промежуточных значений, отображения результатов и других операций.

Чтобы вызвать форму в модальном режиме, надо использовать метод ShowModal.

```
procedure TMyForm.Button3Click(Sender: TObject);
begin
Form2.ShowModal;
end;
```

Теперь, когда после щелчка на кнопке Окно на переднем плане появится форма Form2, переключиться с нее на главное окно не удастся, пока она не будет закрыта. Переключаться на любые другие приложения *Windows,* конечно, можно без ограничений.

К оформлению модальных окон предъявляется набор негласных, но практически не имеющих исключений требований.

1. Диалоговое окно не должно позволять менять свои размеры. Для этого в свойстве BorderStyle (Стиль границы) надо выбрать любое значение, не допускающее изменения размера.

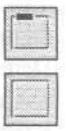
Таблица 2.15. Значения свойства BorderStyle

Значение	Вид границы **окна**
bsDialog	Размер окна менять не разрешается. Вид границ — как у стандартных диалоговых окон Windows
bsSingle	Размер окна менять не разрешается. Вид границ — тонкая полоса
bsNone	Размер окна менять не разрешается. Видимая граница отсутствует
bsSizeable	Стандартная граница, допускающая изменение размеров окна
bsToolWindow	Аналогично bsSingle, но высота заголовка окна уменьшена
bsSizeToolWin	Аналогично bsSizeable, но высота заголовка окна уменьшена

Лучше всего использовать значение bsDialog, специально предназначенное для оформления диалоговых окон.

2. Свойство BorderWidth (Ширина границы окна) определяет область, на которой не разрешается размещать элементы управления. Для этого свойства надо задать подходящее значение (например 2), потому что диалоговые окна имеют достаточно широкие границы.

Панели и декоративные элементы



На панели компонентов Standard имеются два компонента: TGroupBox и TPanel, — которые обладают, на первый взгляд, схожими областями применения. Оба компонента можно использовать для взятия части формы в рамку и для группирования элементов управления путем размещения их внутри области, охваченной этими объектами. В частности, компонент TGroupBox иногда используется для создания нескольких групп переключателей, когда обойтись стандартными возможностями компонента TRadioGroup не удается.

Панель обладает более общими свойствами. Это не только средство создал йя рамок разнообразного вида (выпуклых, вдавленных и так далее). Панель также предназначена для объединения произвольных элементов управления с возможностью их перемещения (перетаскивания) по форме вместе с родительской панелью и стыковки с другими панелями. Такая технология популярна в современных приложениях *Windows*.

После размещения панели на форме ее внешний вид можно настроить с помощью целого набора свойств (возможный пример оформления рамки показан на рис. 2.15).

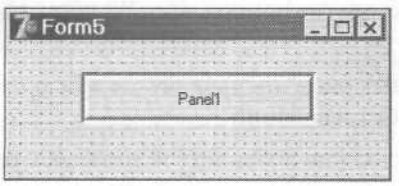


Рис. 2.15. Форма, содержащая панель с «вдавленной» рамкой

- Свойства BevelInner и BevelOuter задают стили оформления соответственно внутренней и внешней рамок панели. Они могут принимать одно из четырех значений.

Таблица 2.16. Значения свойств BevelInner и BevelOuter

Значение	Вид рамки
bvNone	Отсутствует
bvLowered	«Вдавленная» рамка
bvRaised	«Выпуклая»рамка
bvSpace	«Плоская» райка

- Свойство BevelWidth определяет расстояние между внутренней и внешней рамками (в пикселах).
- О свойствах BorderStyle и BorderWidth мы уже говорили выше.

Механизм перетаскивания

Панели активно используются при необходимости перетаскивания объектов по форме при помощи мыши. Для реализации таких возможностей в программе, создаваемой с помощью системы *Delphi 7,* поддерживаются два механизма. Один из них — это собственно перетаскивание (по-английски — *drag-and-drop*).Соответствующий метод заключается в следующем. После наведения указателя мыши на нужный объект нажимается левая кнопка мыши. Она удерживается нажатой, а объект в это время перетаскивается на новое место. После перетаскивания кнопка отпускается и объект получает новое фиксированное местоположение. Подобным способом перетаскивают папки в Проводнике *Windows,* выделенный текст в текстовых редакторах и объекты в других программах.

Рассмотрим работу данного механизма на двух примерах: перетаскивание текста из текстового поля на панель и перемещение кнопки по форме.

Примеры будут созданы в рамках формы Form2. На ней надо разместить три компонента (текстовое поле, панель и кнопку), которые получат названия Edit1, Panel1 и Button1 (рис. 2.16).

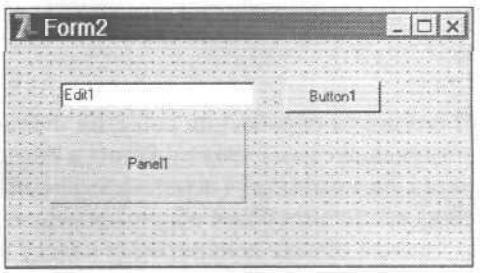


Рис. 2.16. Форма с подготовленными элементами управления

Процесс программирования механизма перетаскивания осуществляется в несколько этапов.

1. Фиксируется начало работы данного механизма с помощью метода формы BeginDrag (например, при нажатии левой кнопки мыши над соответствующим объектом). Указатель мыши принимает вид, указанный в свойстве DragCursor (значение по умолчанию — crDrag).
2. Происходит обработка процесса перетаскивания. Для объектов, которые могут принять перетаскиваемый объект, задается обработчик события OnDragOver. Этот обработчик вызывается, когда указатель мыши в режиме перетаскивания перемещается над принимающим объектом. Объект должен ответить, готов ли он послужить «приемником», занеся значение True (если готов) в параметр обработчика Accept, передаваемый по ссылке. В каких-то случаях от приема следует отказаться, например если в рамках одной формы поддерживается механизм перетаскивания разных объектов и надо выполнить дополнительную проверку на соответствие типов.
3. Когда перетаскиваемый объект вброшен» (кнопка мыши отпущена), генерируется событие OnDragDrop. В нем выполняются все необходимые действия по обработке информации, хранящейся в перетаскиваемом объекте.
4. По окончании перетаскивания генерируется событие OnEndDrag. Его имеет смысл обрабатывать только если требуется проверить, успешно или нет выполнено перетаскивание. Для этого один из параметров обработчика, Target (тип TObject), хранящий ссылку на принимающий объект, надо сравнить со значением nil. Если «приемник» отсутствует, значит, перетаскивание не выполнено.

Реализуем эти действия на практике.

1. Для объекта Edit1 создается обработчик события OnMouseDown. В нем проверяется, нажата ли именно левая кнопка, и вызывается метод BeginDrag с параметром False. Это означает, что процесс перетаскивания не начнется, пока пользователь не переместит мышь на некоторое расстояние. Это позволяет не путать простые щелчки на объекте с началом процесса перетаскивания.

```
procedure TForm2.Edit1MouseDown(Sender: TObject;
   Button: TMouseButton;
   Shift: TShiftState; X, Y: Integer);
begin
if Button = mbLeft then
   Edit1.BeginDrag(false);
end;
```

2. Для объекта PaneLl создается обработчик события OnDragOver путем двойного щелчка на соответствующей строке панели Events в Инспекторе объектов. В нем проверяется, является ли исходный (перетаскиваемый) объект полем ввода Editl. Можно, конечно, выполнить более общие проверки, например установить, принадлежит ли исходный объект конкретному классу и так далее. Результат проверки заносится в параметр Accept.

```
procedure TForm2.Panel1DragOver(Sender, Source: TObject;
   X, Y: Integer;
   State: TDragState; var Accept: Boolean);
```

```
begin
Accept := Source = Edit1
end;
```

Можно выполнить и более общую проверку.

```
Source is TEdit
```

3. Панель должна обрабатывать еще одно событие OnDragDrop, генерируемое, когда происходит «сброс» перетаскиваемого объекта. В нем производится итоговое действие: в заголовок панели записывается текст, находящийся в текстовом поле.

```
procedure TForm2.Panel1DragDrop(Sender, Source: TObject;
   X, Y: Integer);
begin
Panel1.Caption := Edit1.Text
end;
```

Теперь можно запустить программу, вызвать форму Form2, навести указатель на текстовое поле (в нем первоначально хранится текст Edit1), нажать левую кнопку мыши и, не отпуская ее, переместить мышь по направлению к панели. Указатель при этом примет вид перечеркнутого кружка, означающего, что в данном месте «сброс» перетаскиваемого объекта недопустим. Когда указатель окажется в области панели, он примет вид небольшой стрелки с квадратом. Теперь кнопку можно отпустить, и надпись на панели сразу изменится на Edit1.

Второй пример идеологически во многом напоминает первый. Для объекта Button1 создается обработчик нажатия кнопки мыши, в котором начинается процесс перетаскивания кнопки:

```
procedure TForm2.Button1MouseDown(Sender: TObject;
   Button: TMouseButton;
   Shift: TShiftState; X, Y: Integer);
begin
if Button = mbLeft then
   Button1.BeginDrag(false);
end;
```

Правом «принимать» кнопку должна обладать сама форма. Для этого она должна так обрабатывать событие OnDragOver, чтобы разрешить прием кнопок.

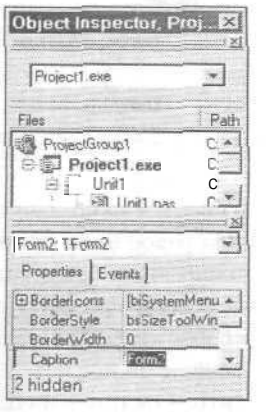
```
procedure TForm2.FormDragOver(Sender, Source: TObject;
   X, Y: Integer;
   State: TDragState; var Accept: Boolean);
begin
Accept := Source is TButton
end;
```

Необходимо также обрабатывать событие OnDragDrop. В ходе обработки для исходной кнопки (параметр Source, приведенный к типу TButton с помощью операции as) задаются новые координаты ее местоположения на форме (свойства Left и Top).

```
procedure TForm2.FormDragDrop(Sender, Source: TObject;
   X, Y: Integer);
begin
 (Source as TButton).Left := X;
 (Source as TButton).Top := Y;
end;
```

Теперь простыми движениями мыши кнопку можно свободна перемещать по форме.

Механизм стыковки



В ряде современных настольных приложений реализован новый подход к перемещению отдельных объектов и их групп. Элементы управления объединяются в стыкуемые палитры (Dock), которые могут свободно «плавать» внутри заданной области. Палитры могут пристыковываться друг к Другу, подстраивая размеры, а расположенные на них элементы управления можно перетаскивать между палитрами. Так, в частности, реализованы некоторые окна в самой системе *Delphi 7*. Например, Менеджер проектов и Инспектор объектов можно объединить вместе. Для этого в их контекстных меню имеется пункт Dockable (Стыкуемый). Если этот пункт включен (помечен галочкой), то он позволяет данному окну пристыковываться к другим окнам среды.

Для рассмотрения механизма стыковки (*drag-and-dock*) создадим новый проект в рамках текущей группы проектов. Для этого в Менеджере проектов надо выделить строку ProjectGroup1 и щелкнуть на кнопке New (Создать). В открывшемся диалоговом окне New Items (Создание программы) на вкладке New (Создание) надо выбрать значок Application (Приложение) и щелкнуть на кнопке OK. В Менеджере проектов появится новый проект Project2.exe. Он будет выделен шрифтом — это означает, что данный проект сейчас активен и при компиляции и запуске программы система *Delphi 7* будет обрабатывать именно его (рис. 2.17).

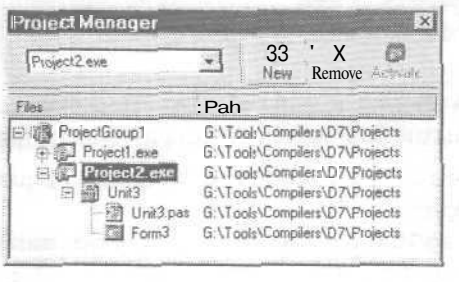


Рис. 2.17. Новое приложение входит в уже существующую группу проектов

ЗАМЕЧАНИЕ Чтобы в Менеджере проектов сделать активным нужный проект, надо выбрать соответствующий пункт и щелкнуть на кнопке Activate (Активизировать).

Одновременно в Проектировщике форм появится новая форма Form3, являющаяся главной формой нового проекта. На ней необходимо разместить две панели Panel1 и Panel2. На первой панели дополнительно помещаются две кнопки Button1 и Button2.

Обеспечить работу механизма стыковки можно и без программирования. Для этого сначала надо указать те объекты, которые будут выступать в роли стыкуемых палитр: Panell и Panel2. Свойства DockSite этих объектов определяют, может ли данный объект выполнять функции палитры. Значения этих свойств надо установить равным True. Чтобы автоматически поддерживать режимы стыковки, не обращаясь программно к методам этих палитр, надо для свойств DragMode (Режим перетаскивания) задать значение dmAutomatic. Кроме того, значения свойств DragKind (Вид перетаскивания) необходимо изменить с dkDrag (Свободное перетаскивание) на dkDock (Стыковка).

Кнопки не могут служить полноценными палитрами (свойства DockSite у них нет), поэтому для каждой из кнопок настраиваются только свойства DragMode (значение dmAutomatk) и DragKind (значение dkDock). Созданную программу можно откомпилировать и запустить. Теперь любую панель можно таскать по форме, помещать внутрь другой панели и перетаскивать кнопки, стыкуя их друг с другом самым причудливым образом.

В ходе перемещения кнопок и панелей для них автоматически создается небольшое охватывающее окно, позволяющее таскать эти объекты как в пределах формы, так и за ее границами (рис. 2.18).

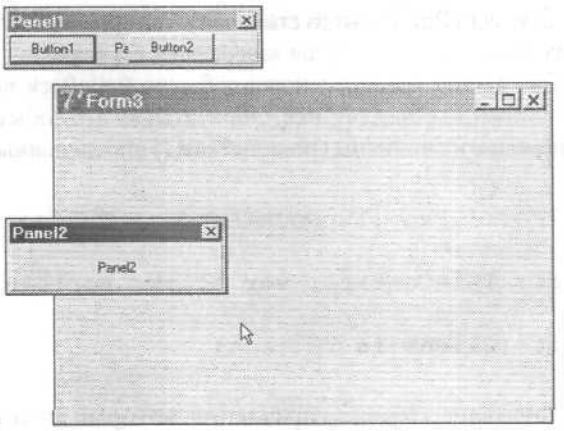


Рис. 2.18. *При работе программы стыкуемые панели можно размещать не только в пределах формы, но и автономно*

При этом стыковать объекты можно только друг с другом. Вернуть их на форму не удается, что естественно: ведь она сама пока что не является стыкуемой **палитрой.**

Чтобы позволить и кнопкам, и панелям «приземляться» на форме, надо закрыть программу-пример и в Инспекторе объектов изменить для формы Form3 значение свойства DockSite на True, а значение свойства DragKind установить равным dkDock. Теперь и кнопки, и панели в процессе их перемещения можно устанавливать в любом месте формы.

Программная поддержка процесса стыковки похожа на обработку процесса перетаскивания. Однако стыкуемой палитре необходимо предварительно обработать новое сообщение OnGetSiteInfo, с помощью которого передается информация об объекте, который хочет пристыковаться к данной палитре, и его размер (прямоугольник, хранящийся в параметре InfluenceRect). Аргумент CanDock, передающийся по ссылке, должен получить значение True, если разрешение на стыковку дается. При этом можно изменить область, отводимую для стыковки (прямоугольник InfluenceRect, который тоже передается по ссылке). Например, можно запретить панели Panel2 принимать кнопки.

```
procedure TForm3.Panel2GetSiteInfo(Sender: TObject;
   DockClient: TControl;
   var InfluenceRect: TRect; MousePos: TPoint;
   var CanDock: Boolean);
begin
CanDock := not (DockClient is TButton)
end;
```

Через параметр DockClient передается элемент управления (класс TControl), который собирается пристыковаться к панели. **Если** теперь перемещать кнопки по форме, то хотя они и могут располагаться на панели Panel2, но процесс автоподстройки **их** положения и размера под панель выполняться не будет (а объект Panell по-прежнему сможет пристыковываться к палитре Panel2).

Два события: OnDockOver (При попытке стыковки), генерируемое при перемещении объекта над палитрой, и OnDockDrop (При завершении стыковки) — аналогичны событиям OnDragOver и OnDragDrop. Еще одно новое событие, OnUnDock, посылается палитре при попытке вытаскивания объекта с нее. Обрабатывая это событие, можно, например, запретить перемещать с палитры (панели Panell) определенные элементы управления (кнопки).

```
procedure TForm3.Panel1UnDock(Sender: TObject;
   Client: TControl;
   NewTarget: TWinControl; var Allow: Boolean) ;
begin
Allow := not (Client is TButton);
end;
```

Параметр Client описывает элемент управления, который пользователь пытается забрать с палитры. Параметр Allow должен получить значение True, если разрешается забрать этот элемент, и False в противном случае.

Компонент Фрейм (TFrame)

Компонент Фрейм во многом напоминает форму и может использоваться как вместе с ней в виде отдельного элемента, так и сам по себе в качестве самостоятельного окна.

Фрейм предназначен для более гибкого и эффективного объединения и хранения групп объектов. Он может использоваться как заготовка (шаблон) для быстрого

создания новых фреймов нужной структуры. Допускаются неограниченные вложения фреймов друг *в* Друга. При этом если вносятся изменения в шаблон, то автоматически изменяются все фреймы, созданные на его основе и применяемые в разрабатываемых программах.

Фрейм может служить палитрой, поэтому с его помощью можно создавать сложные многооконные приложения с поддержкой режимов стыковки и перетаскивания для любых объектов.

Чтобы добавить к проекту новый фрейм, надо выполнить команду File ➤ New > Frame (Файл > Создать ➤ Фрейм). При этом в Менеджере проектов возникнет новый модуль Unit4, только вместо формы в нем будет фрейм.

Однако ссылка на переменную, описывающую фрейм, в модуле Unit4.pas не появится. Это связано с тем, что каждый фрейм обязательно должен быть привязан к конкретной форме.

Фрейм располагается на форме, когда на панели Standard выбирается значок Frames и выполняется щелчок на нужной форме. При этом отображается список всех доступных в программе (созданных ранее) фреймов, из которых надо выбрать подходящий.

Исходно фрейм можно сделать невидимым, установив значение свойства Visible равным False. Тогда отображать фрейм можно по щелчку на кнопке Button1 с помощью метода Show.

Новые стандартные действия

Начиная с шестой версии, набор стандартных действий значительно расширен. Среди наиболее полезных дополнений — новые стандартные действия по работе с файлами. Прежде всего, это возможность вызова стандартных диалоговых окон для открытия и сохранения файла без использования соответствуюших стандартных компонентов наподобие TOpenDialog.

Рассмотрим пример использования новых действий на практике. Допустим, при нажатии на кнопку нам требуется получить имя файла и полный путь, не прибегая к стандартному компоненту TOpenDialog и не вызывая его явно. Разместим на форме кнопку, а также компонент Список действий (TActionList) и добавим к нему новое стандартное действие в редакторе Action List Editor.

Для того чтобы получить название файла, для события OnAccept в Инспекторе объектов для действия FileOpen1 надо сформировать собственный обработчик. Среди автоматически добавленных к форме объектов на этом этапе можно найти вложенный в FileOpenl объект FileOpen1.OpenDialog — он относится к классу TOpenDialog и заполняется значениями, соответствующими параметрам, выбранным пользователем в диалоговом окне открытия файлов.

Событие OnAccept генерируется в тот момент, когда пользователь закрывает стандартное диалоговое окно с помощью кнопки OK. В случае, когда это окно закрывается щелчком на кнопке Cancel, генерируется сообщение OnCancel.

Для того чтобы получить доступ к имени файла, которое выбрал пользователь, надо обратиться к свойству FileName объекта FileOpen1.OpenDialog.

Помимо стандартных действий по открытию файла и сохранению данных в файле в Delphi 7 теперь доступны: действие по вызову стандартного диалогового окна настройки печати (TFilePrintSetup); стандартное действие TFileRun, позволяющее запускать внешнюю программу для той или иной **обработки** конкретного файла; а также стандартное действие TFileExit, при обращении к которому происходит закрытие программы.

Например, добавим стандартное действие TFileExit в имеющийся список действий и зададим в свойстве Action кнопки значение FileExit1 — стандартное действие закрытия окна. Теперь при нажатии на кнопку программа будет закрываться автоматически. Ранее для этого приходилось писать отдельный обработчик щелчка наподобие следующего:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Application.Terminate;
end;
```

Среди других новых стандартных действий:

- вызов стандартных диалоговых окон поиска и замены текстовой информации в документах (TSearchFind, TSearchFindFirst, TSearchReplace, TSearchFindNext);
- вызов стандартных окон настройки и форматирования текста в редакторе (TRichEditBold, TRichEditItalic, TRichEditUnderline, TRichEditStrikeOut, TRichEditBullets, TRichEditAlignLeft, TRichEditAlignRight, TRichEditAlignCenter);
- стандартные действия по перемещению к следующему (TNextTab) или предыдущему (TPreviousTab) корешку объекта TTabControl;
- вызов стандартного окна выбора цвета шрифта (TColorSelect);
- вызов стандартных окон открытия и сохранения графических изображений TOpenPicture и TSavePicture — они дополнены средствами предварительного просмотра выбранного изображения;
- вызов стандартного окна печати TPrintDlg;
- весьма полезное стандартное действие TBrowseURL, которое автоматически вызывает броузер, установленный в операционной системе *Windows* по умолчанию, и открывает в нем *Web*-страницу, передающуюся в виде строки в свойстве;

  ```
  URL: string;
  ```

- стандартное действие TDownLoadURL по загрузке файла, указанного в свойстве URL;
- стандартное действие TSendMail по отправке электронного письма; текст письма определяется в свойстве:

  ```
  Text: TStrings;
  ```

- набор стандартных действий (TListControlCopySelection, TListControlDeleteSelection, TListControlSelectAll, TListControlClearSelection, TListControlMoveSelection, TStaticListAction, TVirtualListAction) для работы с компонентами-списками (TListView и TListBox).

Склад объектов

До сих пор технология повторного использования объектов при создании программ была продемонстрирована на примере единственного проекта, В процессе его создания применялись готовые компоненты, позволившие быстро сформировать и настроить нужные элементы управления. В дальнейшем, при разработке новых проектов будут накапливаться готовые формы, которые подчас тоже желательно использовать повторно. Например, форму, выполняющую ввод информации по заданному шаблону (имя пользователя и пароль) или другие стандартные действия, удобно применять и в других проектах, где возникает аналогичная потребность.

Можно, конечно, просто скопировать файлы, хранящие описание формы и исходные тексты, в каталог с новым проектом, и затем добавить их в этот проект с помощью Менеджера проектов, однако подобный подход противоречит принципу объектно-ориентированного программирования. В систему *Delphi 7* встроена специальная технология, позволяющая храни гь подобные заготовки форм и фреймов на так называемом Складе объектов.

На Склад разрешается помещать как отдельные формы, так и шаблоны целых проектов, которые потом можно использовать в качестве полноценных заготовок. Чтобы добавить на Склад форму Form2, надо щелкнуть на ней правой кнопкой мыши и выбрать в контекстном меню пункт Add to Repository (Добавить на Склад).

В правой части возникшего диалогового окна надо выбрать форму, которая будет добавлена на Склад, ввести в коле Title (Имя) название, под которым она в качестве шаблона будет присутствовать на Складе, а в поле Description (Описание) — краткое описание фирмы. В раскрывающемся списке Page (Страница) надо выбрать страницу Склада, на которую будет помещена форма. Доступных страниц четыре: Forms (Формы), Dialogs (Диалоговые окна). Projects (Проекты), Data Modules (Модули данных). Форму следует разместить на странице Forms (Формы). В поле Author (Автор) указывается автор данного шаблона, С помощью кнопки Browse (Обзор) можно выбрать значок, отличающийся от стандартного, которым форма обозначается на Складе. После щелчка на кнопке OK форма Form2 копируется на Склад и становится доступной для других программистов, работающих с текущей копией программы *Delphi 7* (рис. 2.19).

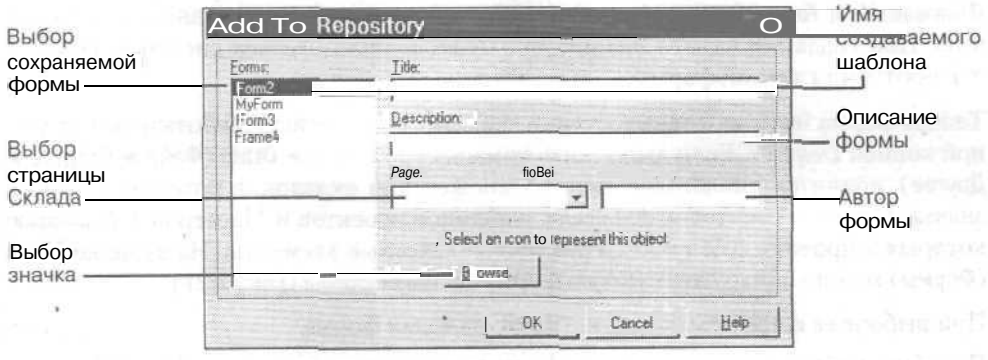


Рис. 2.19. Сохранение формы для последующего использования

Изменить введенные характеристики объекта можно, вызвав окно настройки Склада командой Tools ➤ Repository (Сервис ➤ Склад), На экране появляется диалоговое окно, в левой части которого располагается список доступных страниц. Для добавления новой страницы служит кнопка Add Page (Добавить страницу). При выделении одной из страниц в правой части окна показывается ее содержимое (рис. 2,20).

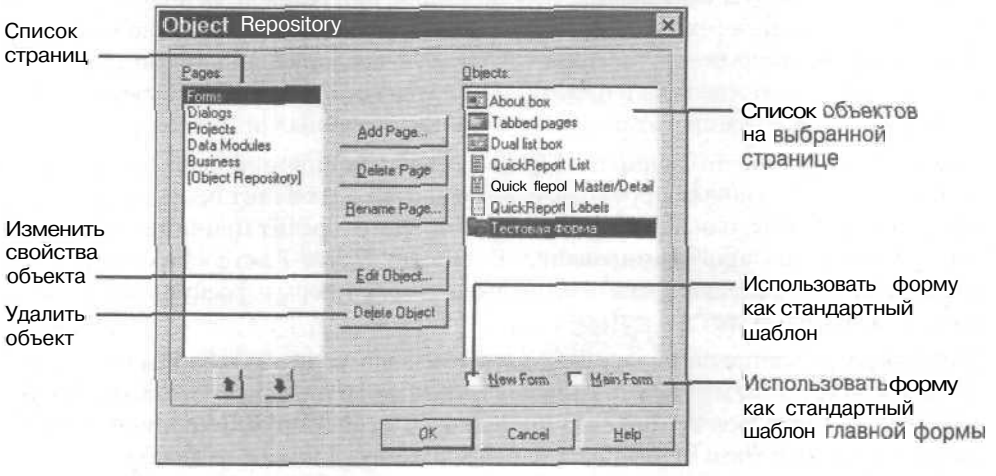


Рис. 2.20. *Изменение свойств объекта, хранящегося на Складе*

Кроме различных стандартных форм здесь имеется и только что добавленный объект Тестовая форма. Его можно удалить с помощью кнопки Delete Object (Удалить объект). Чтобы отредактировать сведения об объекте или сменить значок, щелкните на кнопке Edit Object (Изменить объект), после чего ранее введенные значения изменяются в уже знакомом окне.

Если включить флажок New Form (Новая форма), то в левом нижнем углу значка текущей выбранной формы появится небольшой прямоугольник. Это означает, что данная форма будет автоматически создаваться в Проектировщике при выполнении команды File ➤ New > Form (Файл > Создать ➤ Форма).

Флажок Main Form (Главная форма) делает выбранную форму главной по умолчанию. При создании нового проекта она будет использоваться системой *Delphi 7* как прототип главной формы.

Теперь форма Тестовая форма доступна любому программисту, работающему с данной копией *Delphi 7.* Если выполнить команду File ➤ New ➤ Other (Файл > Создать ➤ Другое), появится диалоговое окно со множеством вкладок, в котором хранятся значки объектов-заготовок со Склада, шаблонов проектов и Мастеров, с помощью которых к проекту добавляются различные сложные элементы. На вкладке Forms (Формы) можно обнаружить новую форму Тестовая форма (рис. 2.21).

При выборе ее в проекте появится еще одна такая форма.

Подобным образом можно очень эффективно настраивать систему *Delphi 7* на конкретные нужды групп разработчиков и отдельных программистов.

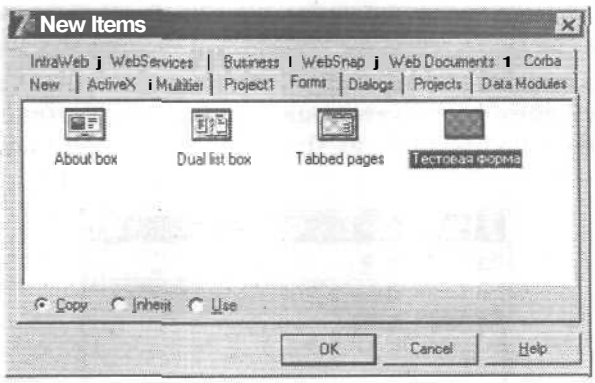


РИС. 2.21. *Выбор формы, хранящейся на Складе, для добавления в проект*

Компонент Список действий (TActionList)

Это последний компонент панели Standard (Стандартные). Он применяется, когда программа обрабатывает большое число всевозможных событий и одни и те же операции выполняются разными обработчиками, как, например, при щелчке на кнопке Сложить и при выборе одноименных команд строки меню и контекстного меню в первом примере (проект Project1). В таких ситуациях правильнее централизованно хранить все важнейшие события и формировать реакции на них в одном объекте. Для этого и предназначен компонент TActionList.

Вернемся к первому проекту (форма MyForm), сделав его активным в Менеджере проектов, и разместим на форме новый компонент TActionList. Много места для него не потребуется, так как он невизуальный (рис. 2.22).

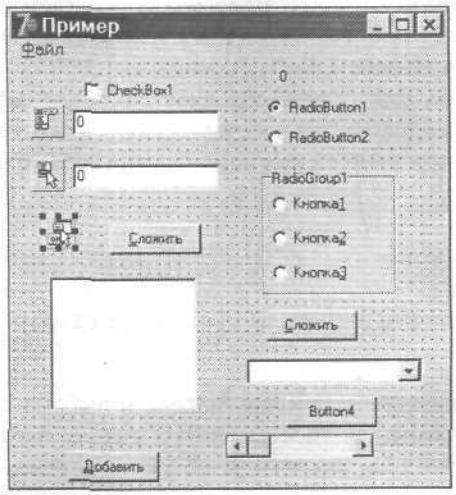


Рис. 2,22. *Размещение на форме компонента Список действий*

После двойного щелчка на объекте ActionList1 откроется окно Редактора. Чтобы добавить новое действие, необходимо щелкнуть на кнопке New Action (Создать действие). В разделе Categories (Категория действия) появится строка (None) (Отсутствует). В системе *Delphi* 7 имеются две категории действий: стандартные действия и действия, определяемые программистом. Пока что мы рассматриваем вторую категорию (рис. 2.23).

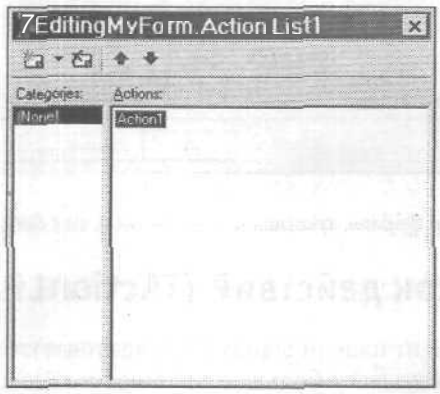


Рис. 2.23. *Редактирование списка действий*

ЗАМЕЧАНИЕ Этот редактор является стандартным редактором коллекции — набора объектов. О нем более подробно будет рассказано ниже.

В разделе Actions (Действия) имеется строка Action1. Она определяет новый объект класса TAction, для которого надо настроить ряд свойств в Инспекторе объектов.

Прежде всего, это свойство Caption (Заголовок). Значением заголовка будет строка &Сложить. Имя объекта — свойство Name (Имя) — можно изменить на более подходящее, например AddAction. Наконец, надо сформировать действие, которое будет выполняться данным объектом. Для этого необходимо сформировать обработчик события OnExecute. В нем указывается логика работы обработчика — она аналогична той, что ранее использовалась при щелчке на кнопке Button1 (сложение двух чисел).

```
procedure TMyForm.AddActionExecute(Sender: TObject);
begin
Label1.Caption :=
   IntToStr( StrToInt(Edit1.Text)+StrToInt(Edit2.Text) );
end;
```

Старые обработчики щелчка на кнопке Button1 и выбора пунктов меню Сложить можно удалить из программы, полностью уничтожив соответствующий код в редакторе. Вместо них с помощью Инспектора объектов достаточно указать действие AddAction в свойствах Action соответствующих объектов: кнопки Button1 и двух пунктов меню Сложить (рис. 2.24).

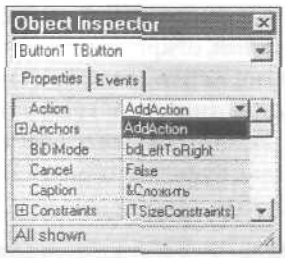


Рис. 2.24. Назначение действия элементу управления

Теперь программу можно откомпилировать и запустить. Она будет работать, как и раньше, только обработчик события Сложить, вызываемый из разных мест приложения, теперь находится в легкодоступном хранилище.

ВНИМАНИЕ Некоторые свойства каждого объекта класса TAction (в частности, AddAction) непосредственно влияют на внешний вид объектов, при обращении к которым вызывается соответствующее действие. Например, свойство Caption (Заголовок) определяет заголовок не для объекта AddAction, а для связанных с ним пунктов меню и кнопки. То есть, если сейчас это свойство изменить и ввести строку Суммирование, то при перезапуске программы автоматически изменятся и надпись на кнопке, и названия пунктов меню.

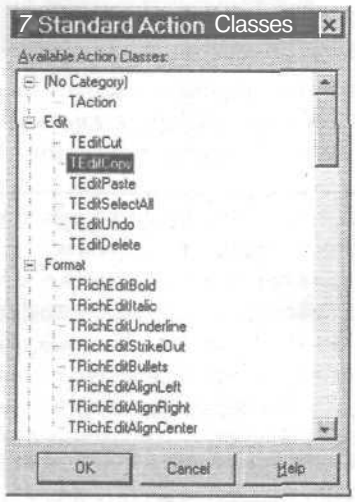
Еще несколько полезных свойств, позволяющих подобным способом централизованно и одновременно корректировать внешний вид нескольких элементов управления на форме.

***Таблица 2.17.** Свойства объекта TAction*

Свойство	Назначение
Checked	Признак пометки (используется для пунктов меню, переключателей, флажков)
ImageIndex	Номер картинки из элемента ImageList (Список картинок), о котором будет рассказано ниже. С его помощью можно единым способом создавать и пункты меню с картинками, и соответствующие им командные кнопки на панели управления. Элемент ImageList задается в свойстве Images объекта ActionList
Shortcut	«Горячая клавиша». Отличается от быстрых клавиш тем, что позволяет вызвать нужное действие, даже когда на форме нет соответствующих видимых объектов, например, когда действие привязано к пункту закрытого меню
Hint	Всплывающая подсказка

Событие OnUpdate посылается объекту класса TAction, когда программа находится в состоянии ожидания. В этот момент можно выполнить действия по изменению состояния связанного с этим объектом элемента управления (например, переключить флажок или пометить галочкой пункт меню).

Помимо действий, определяемых программистом, в список действий можно добавить одно из стандартных действий, открыв контекстное меню раздела Categories и выбрав в нем пункт New Standard Action (Создать стандартное действие). При этом появится большой список стандартных действий, которые не надо программировать. После добавления в список они становятся доступными из приложения (это, например, различные операции с буфером обмена *Windows,* управление расположением окон и прочее) (рис. 2.25).



Рис, 2.25. *Список стандартных действий, доступных для использования в программе*

Что нового мы узнали?

В этом уроке мы научились

- формировать окно приложения *Windows;*
- использовать визуальные компоненты;
- реагировать на события программы;
- использовать в программе классические средства управления;
- контролировать процесс создания программы.

УРОК 3 Отладка программ

- Что такое отладка
- Расширенные средства отладки
- Исключительные ситуации

Что такое отладка

Программ без ошибок не существует. *Синтаксические* ошибки, связанные с неверным вводом команд в редакторе, неверной записью идентификаторов и другими некорректными действиями, можно обнаружить простым анализом исходного текста, и они почти всегда фиксируются компилятором *Delphi 7.*

В некоторых случаях выдаются предупреждения или подсказки. Желательно устранять их источники, руководствуясь принципом «дыма без огня не бывает». Однако ошибки, связанные с неверной реализацией алгоритма (например, когда вместо символа > ошибочно введен символ <, что не является синтаксической ошибкой), могут привести к возникновению ошибок уже во время работы программы. Кроме того, неверная реализация исходного алгоритма не обязательно приводит к нарушению работоспособности приложения, но может повлечь за собой выдачу неверных результатов или выполнение ошибочных действий.

Рассмотрим следующий фрагмент исходного текста, выполняющий инициализацию массива Arr.

```
var Arr: array[1..10] of integer;
...
N := 11;
...
for i := 1 to N do
   Arr[i] := 0;
```

Он неверен, потому что в конце цикла произойдет обращение к одиннадцатому элементу массива, что приведет к выходу за границу массива и возникновению ошибки. Подобные вещи компилятор отследить и обнаружить не в силах, и процесс поиска и устранения ошибок такого рода, называемый *отладкой,* полностью возлагается на программиста.

Причины ошибок

Ошибки допускает любой программист: и начинающий, и имеющий тридцатилетний опыт. Различаются только причины этих ошибок. Начинающий разработчик редко берется за большие проекты, и его ошибки связаны, как правило, с плохим знанием операторов Паскаля. Профессиональный программист обычно не допускает ошибок на этапе кодирования. Но он может допускать ошибки на начальных этапах работы, когда будущая программа только проектируется и не все детали алгоритма до конца ясны. При этом очень сложно заранее предусмотреть скрытые взаимосвязи между различными модулями программы, насчитывающей сотни тысяч операторов. Поэтому приходится прибегать к различным программистским приемам, позволяющим повысить надежность приложения и выявлять ошибки как можно раньше. Это служит залогом значительного снижения трудозатрат. Если неверный оператор скрыт в «недрах» программы и уже написано много нового исходного текста, то изменение работы этого оператора может повлечь за собой лавинообразный эффект влияния на все последующие действия.

Система *Delphi 7* предоставляет средства отладки, которые ориентированы на программистов любой квалификации.

Синтаксические ошибки

Синтаксические ошибки обнаруживаются компилятором автоматически. Сообщения о найденных ошибках отображаются в нижней части редактора, в небольшом окне (рис. 3.1):

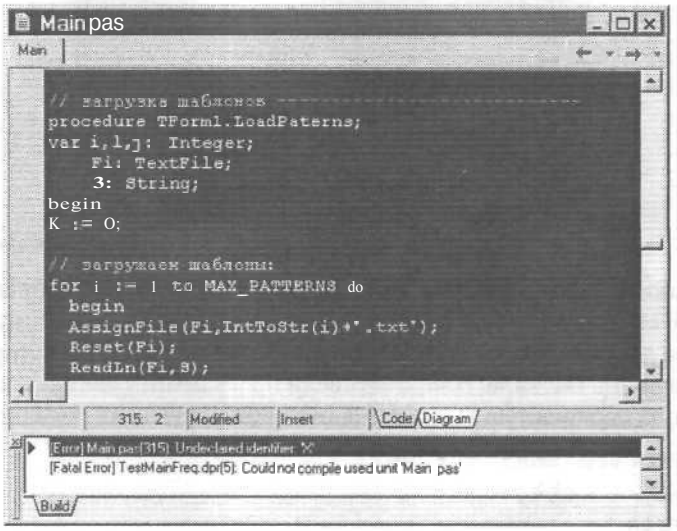


Рис. 3.1. *Информация о синтаксической ошибке, обнаруженной компилятором*

При двойном щелчке на строке с сообщением об ошибке система *Delphi 7* переключится в редактор, где подсветит строку, в которой эта ошибка обнаружена. Само сообщение (на английском языке) описывает ошибку достаточно подробно, чтобы можно было понять ее причину. Например:

```
Undeclared identifier: 'X'
```

В данном случае указано, что идентификатор X не объявлен.

Логические ошибки

Существует несколько способов предотвращения, выявления и устранения логических ошибок. Все они используются программистами, как правило, в комбинации друг с другом.

За наиболее часто встречающимися ошибками можно заставить следить саму программу. Для этого в настройках проекта — соответствующее диалоговое окно вызывается командой Project > Options (Проект ► Настройки) — на вкладке Compiler (Компилятор) надо выполнить следующие действия (рис. 3.2).

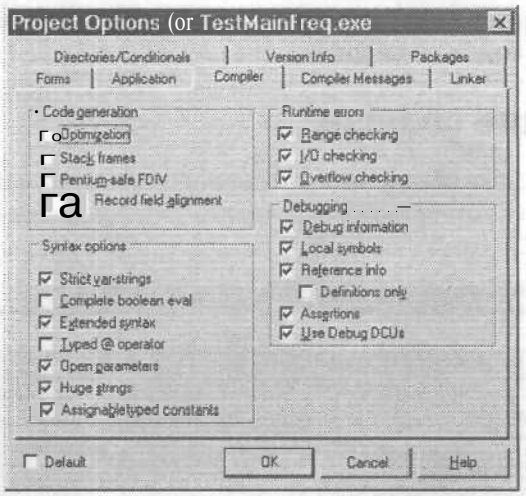


Рис. 3.2. *Настройка компилятора для максимального контроля ошибок*

- На панели Code generation (Генерация машинного кода) сбросьте флажок Optimization (Оптимизация). Когда компилятор создает оптимизированный код, он нередко вносит существенные улучшения в детали алгоритма, реализованного на Паскале. Например, если программист вводит в процедуре локальную переменную X для хранения промежуточных результатов:

```
function Sum: integer;
var X: integer;
begin
X := StrToInt( Edit1.Text ) + StrToInt( Edit2.Text );
Result := X
end;
```

  то после оптимизации этого кода программа вполне может не использовать промежуточную переменную, а результат вычислений условно будет представлен так:

```
Result := StrToInt( Editl.Text ) +
StrToInt( Edit2.Text );
```

  Это означает, что программист не сможет узнать значения локальной переменной X во время работы программы, потому что реально для нее не будет отводиться место в оперативной памяти.

Кок правило, для повышения эффективности вычисления подобных выражений компилятор активно использует регистры процессора.

- На панели Runtime errors (Ошибки времени выполнения) должны быть установлены флажки Range checking (Контроль выхода индекса за границы массива), I/O Checking (Контроль ошибок ввода/вывода) и Overflow checking (Контроль

переполнения при целочисленных операциях). Так как все типы Паскаля имеют строго ограниченные диапазоны, то, например, при сложении двух чисел, близких к максимально допустимому значению, получится число, которое не укладывается в этот диапазон. Последний флажок позволяет обнаруживать такие ошибки.

- На панели Debugging (Отладка) установите флажки Debug information (Добавление отладочной информации), Local symbols (Просмотр значений локальных переменных), Reference info (Просмотр структуры кода), Assertions (Включение процедуры Assert в машинный код) и Use Debug DCUs (Использование отладочных версий стандартных модулей библиотеки компонентов VCL).

  Без отладочной информации отладка программы в среде *Delphi 7* вообще невозможна. Процедура Assert выполняет отладочные функции. В заключительной версии программы она, как правило, не нужна, а удалять ее вызовы из исходного текста неудобно — их могут насчитываться сотни. Отключить генерацию машинного кода для этой процедуры можно с помощью флажка Assertions. Отладочные версии стандартных модулей содержат дополнительные режимы проверки корректности работы с компонентами *Delphi 7*.

Теперь, если в программе, запущенной непосредственно из среды *Delphi 7,* встретится ошибка (например, вышеупомянутый выход индекса за допустимые границы), выполнение программы прервется, а строка, Б которой встретилась ошибка (в данном случае — `Arr[i] :- 0;`), будет подсвечена.

При этом система *Delphi 7* позволит быстро определить, в чем причина ошибки. Например, наводя указатель мыши на различные переменные, можно сразу увидеть их значения в окне всплывающей подсказки (i = 11).

ЗАМЕЧАНИЕ 8 некоторых случаях, как в рассматриваемом примере, реальные значения переменных i и Arr могут не совпадать с отображаемыми, потому что произошла запрещенная попытка обращения к недоступной области памяти, в результате чего значения локальных переменных могли измениться.

Все же в большинстве ситуаций, когда работа программы, запущенной из *Delphi 7,* прервалась по ошибке, подобным образом можно посмотреть значения любых переменных и констант.

За более сложными ошибками разработчик должен следить из среды *Delphi 7* самостоятельно. Для этого применяется ряд стандартных приемов, однако требуется, чтобы отлаживаемая программа была запущена непосредственно из среды *Delphi 7.* Только тогда среда разработки сможет должным образом контролировать ход выполнения программы и изменение значений различных переменных.

Выполнение по шагам

Обычно разработчику приблизительно известно, в какой подпрограмме возникает ошибка, однако обнаружить ее быстро, просто рассматривая исходный текст, не всегда удается, особенно новичкам в программировании (хотя просмотр исходных

тестов признан наиболее эффективным средством обнаружения ошибок). Поэтому возникает необходимость выполнить эту подпрограмму по шагам: каждый оператор по очереди, с возможностью контроля значений используемых переменных.

Рассмотрим пример, связанный с проектом Project1 (предварительно в Менеджере проектов его надо сделать активным).

Добавим в обработчик списка действий AddAction следующие описание и оператор:

```
var Arr: array[1..10] of integer;
begin
for i := 1 to 11 do
    if i > 3 then Arr[i] := 0
    else Arr[i] := 1;
```

В данном месте программы скрывается ошибка. Чтобы перейти к выполнению этой подпрограммы по шагам, в ней надо создать *точку прерывания* (или *точку остановки*), встретив которую программа прервет свою работу и временно передаст управление системе *Delphi 7*.

Точки прерывания можно ставить не в любой строке исходного теста, а только там, где выполняются какие-то действия. Такие строки помечены на левом поле в редакторе синими круглыми маркерами, которые появляются после успешно выполненной компиляции.

В нашем случае точку прерывания можно поставить в строке с оператором цикла. Это делается нажатием клавиши F5 или щелчком мыши на синем маркере. При этом соответствующая строка выделяется красным цветом (рис. 3.3). Снимается точка прерывания аналогичным способом.

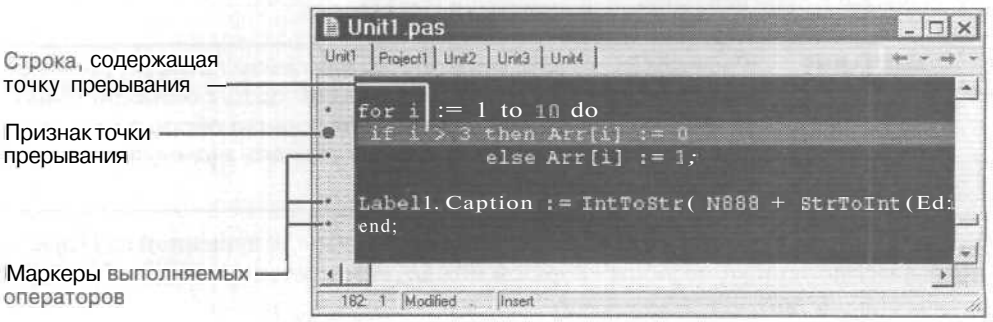


Рис. 3.3, *Подготовка подпрограммы к пошаговой отладке*

Если теперь запустить программу и выполнить операцию сложения (щелкнув па кнопке Сложить или выбрав соответствующий пункт в строке меню или в контекстном меню), то программа остановится и управление будет передано в систему *Delphi 7,* где строка с точкой прерывания помечается зеленой стрелкой на левом поле (рис, 3.4).

Рис. 3.4. *По достижении точки прерывания программа останавливает работу*

В заголовке главного окна системы *Delphi 7* появится информационное сообщение Project1 [Stopped] (Выполнение проекта Projectl остановлено).

Далее выполнение метода AddActionExecute можно продолжить по шагам. Для этого используются команда Run > Step Over (Запуск ➤ Перешагнуть), клавиша F8 или кнопка Step Over (Перешагнуть).

Если выполнить очередной шаг, то в редакторе подсвечивается голубым цветом и помечается зеленой стрелкой следующий оператор, ожидающий выполнения.

```
if i > 3 then Arr[i] := 0
```

Сейчас можно проверить значение переменной i, наведя на нее указатель мыши. Появится окно всплывающей подсказки с надписью i = 1. Если навести указатель на переменную Arr, то в круглых скобках будет показан список всех значений массива, однако, пока инициализация не закончена, значения, хранимые в массиве, непредсказуемы.

Продолжив выполнение метода по шагам, мы увидим, что следующим выполняемым оператором станет оператор

```
else Arr[i] := 1;
```

Предыдущий оператор присваивания Arr[i] := 0 пропущен, потому что условие i > 3 не выполнено. Наконец, на следующем шаге управление опять будет передано оператору цикла, который будет выполнять свое «тело» 11 раз, и на последнем шаге возникнет ошибка.

Полностью остановить работу программы можно с помощью команды Run > Program Reset (Запуск > Сброс программы) или комбинацией клавиш CTRL+F2. Теперь можно исправить параметр окончания цикла с 11 на 10 и вновь запустить программу. Вернуться к точке прерывания можно, например, щелкнув на кнопке Сложить.

Однако выполнять циклы по шагам не очень удобно. Если число повторений достигает сотен или тысяч, то постоянно жать на клавишу F8 бессмысленно. Чтобы пропустить выполнение группы операторов, можно использовать команду Run > Run to Cursor (Запуск ➤ Выполнение до курсора) или клавишу F4.

Предварительно надо выполнить следующие действия. Текстовый курсор надо установить в строку, начиная с которой требуется продолжить выполнение по шагам, например в начало оператора

```
Label1.Caption := IntToStr(
   StrToInt(Edit1.Text) +
   StrToInt(Edit2.Text) );
```

Затем следует убрать точку прерывания (так как при попытке выполнить группу операторов, в данном случае цикл, система *Delphi 7* встретит эту точку прерывания и вновь остановится на старом месте) и нажать клавишу F4. Работа цикла завершится (можно убедиться, что в массиве Arr хранятся только единицы и нули, наведя на его название указатель мыши) и подсветится оператор сложения чисел, введенных в поля. Продолжить выполнение программы (не по шагам) можно с помощью команды Run (Запуск) или клавиши F9.

Просмотреть список всех установленных точек прерывания можно с помощью команды View > Debug Windows ➤ Breakpoints (Вид ➤ Окна отладки ➤ Точки прерывания) (рис. 3.5).

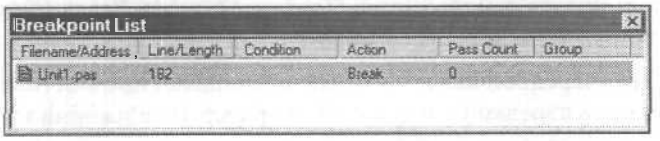


Рис. 3-5. *Список созданных точек прерывания*

ПОДСКАЗКА Если в момент, когда программа остановлена, выполнялись какие-то действия по просмотру файлов и текущее место остановки было потеряно, быстро вернуться к нему поможет команда Run ➤ Show Execution Point (Запуск ➤ Показать точку выполнения).

В некоторых случаях при выполнении программы по шагам требуется также следить за тем, как работают различные подпрограммы, вложенные в отлаживаемый модуль. Рассмотрим это на примере — добавим к форме MyForm новую кнопку и сформируем обработчик ее нажатия. В нем поместим обращение к методу AddActionExecute.

```
procedure TMyForm.Button4Click(Sender: TObject);
begin
AddActionExecute(Sender)
end;
```

Поставим на вызове этого метода точку прерывания. Запустим программу, щелкнем на новой кнопке, и выполнение прервется в нужном месте. Если теперь нажать на клавишу F8, то подпрограмма AddActionExecute() выполнится за один шаг (произойдет шаг *через* подпрограмму). Это не всегда удобно — ведь нам хочется выяснить, каковалогикаработысамогометодаAddActionExecute.Дляэтогонадоприменить команду Run ➤ Trace into (Запуск ➤ Войти внутрь), нажать клавишу F7 или щелкнуть на кнопке Trace into (Войти внутрь). В результате управление передается первой команде метода AddActionExecute.

ВНИМАНИЕ Можно входить внутрь только тех методов, которые написаны разработчиком. С помощью команды Trace into (Войти внутрь) удастся выполнить по шагам далеко не все стандартные функции и процедуры. Это связано с тем, что некоторые из них не существуют в отладочных версиях, а их исходные тексты в поставку системы не входят. Допустимо заходить внутрь методов объектов, включенных в состав библиотеки компонентов VCL, если используются отладочные версии ее модулей. Эта библиотека полностью поставляется в и сходных текстах.

Иногда сразу становится ясно, что делает подпрограмма, в которую разработчик вошел с помощью клавиши F7. В таких случаях можно быстро ее покинуть с помощью команды Run ➤ Run Until Return (Запуск ➤ Выполнять до выхода) или комбинации клавиш SHIFT+F8, в результате чего управление передается на оператор, следующий за вызовом данной подпрограммы. Б нашем примере, если программист вошел внутрь метода AddActionExecute, то быстро покинуть его и вернуться в обработчик Button4Click можно с помощью команды Run Until Return (Выполнять до выхода).

Довольно часто в программах встречаются ситуации, когда число вызовов вложенных подпрограмм весьма велико — оно может достигать десятков. Чтобы взглянуть на подобную последовательность подпрограмм с конкретными параметрами во время выполнения программы, надо дать команду View ➤ Debug Windows ➤ Call Stack (Вид > Окна отладки ➤ Стек вызовов) (рис. 3.6).

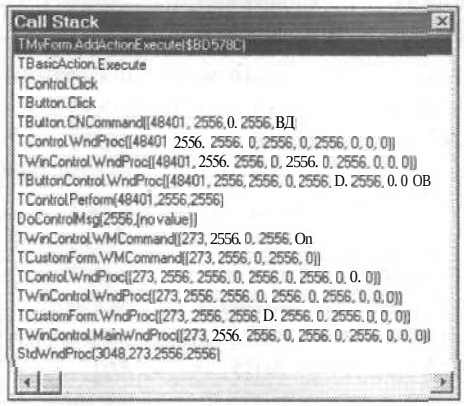
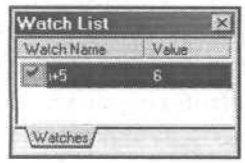


Рис. 3.6. *Стек вызовов подпрограмм*

Просмотр значений

Когда во время работы программы приходится контролировать множество значений разных переменных, использовать для этой цели мышь неэффективно. Система *Delphi 7* позволяет помешать переменные в специальное окно, где наряду с их именами показываются также и текущие значения. При выполнении программы по шагам переключаться между этим окном и окном редактора неудобно, поэтому в контекстном меню окна списка переменных имеется пункт Stay on Top (Поверх других окон). Если его включить, это окно будет во время отладки располагаться выше всех остальных окон, что позволяет следить за изменениями переменных при пошаговом выполнении программы. С помощью флажков (крайняя левая колонка) можно включать и выключать отслеживание конкретных переменных.



Добавление новых переменных в такое окно осуществляется с помощью команды Run ➤Add Watch (Запуск > Добавить слежение) или нажатием клавиш CTRL+F5. В появившемся окне Watch Properties (Своиава слежения) имя переменной вводится в поле

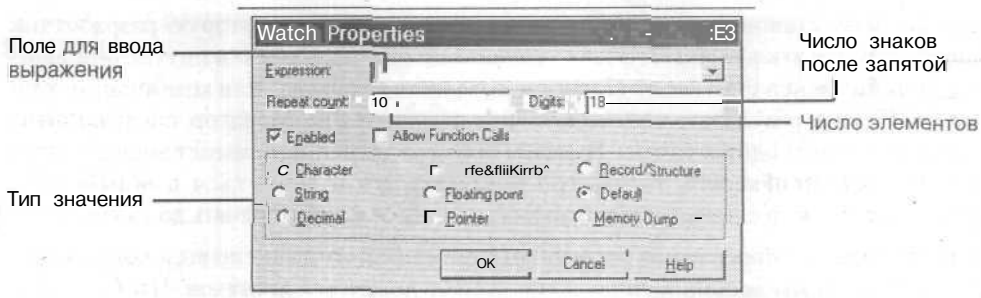
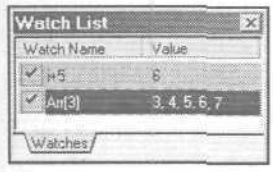


***Рис.* 3.7.** Ввод *выражения для постоянного контроля*

Expression (Выражение). Можно следить за изменениями не только отдельных переменных, но и за значениями целых выражений, например x+y или i-5 (рис. **3.7).**

В нижней части окна Watch Properties имеется ряд переключателей, с помощью которых можно указать тип результирующего значения (например, переменная может иметь тип Integer, а показывать требуется символ, соответствующий этому значению). По умолчанию считается, что показываться будет значение, имеющее тот же тип, что и переменная — переключатель Default (По умолчанию).



В поле Repeat Count (Число элементов) указывается, сколько элементов массива будут отображаться. Например, можно указать элемент массива Arr[3] и число элементов, равное 5. Тогда при выполнении программы по шагам в этом поле будут отображаться через запятую значения пяти элементов, начиная с Arr[3]: Arr[3], Arr[4], Arr[5], Arr[6], Arr[7].

В поле Digits (Разрядность) задается число значащих цифр после запятой, которые будут выводиться для чисел дробных типов.

С помощью переключателя Enable (Включено) можно временно отключать контроль значений некоторых переменных списка.

Флажок Allow Function Calls (Разрешить вызов функций) допускает использование в выражениях вызовов функций. Например, выражение IntToStr(Arr[3]) будет показывать текстовое представление значения, хранящегося в элементе Arr[3].

Содержимое переменной, тип которой основан на записи (record), отображается в виде, принятом для инициализации записей:

```
имя-поля : значение;
```

Значение точки P в методе FormMouseUp будет показано так:

```
P: (x: 129; y: 312]
```

Чтобы отредактировать выражение, ранее введенное в список, надо дважды щелкнуть на нем. Удаление выражение производится нажатием клавиши DELETE. В контекстном меню окна слежения имеются также пункты Delete All Watches (Удалить все элементы), Disable All Watches (Отключить все элементы), Enable All Watches (Включить все элементы), позволяющие, соответственно, удалить, временно выключить или включить все элементы. В *Delphi 7* появилась возможность объединять эле-

менты в несколько групп, каждой из которых соответствует своя закладка в нижней части окна просмотра. Этот удобный подход позволяет собирать отслеживаемые переменные программы в компактные группы по смыслу, а не показывать в одном длинном списке. Добавление новой группы выполняется командой локального меню просмотра значений Add Group (Добавить группу).

ЗАМЕЧАНИЕ В системе Delphi 7 имеется также специальное окно для показа значений всех локальных переменных текущей подпрограммы или метода. Оно открывается командой View ➤ Debug Windows > Local Variables (Вид ➤ Окна отладки ➤ Локальные переменные). Б этом окне отображается, в частности, значение переменной Self, указывающей на объект, которому принадлежит данный метод.

Просмотр и изменение значений

Помимо простого просмотра различных значений во время работы программы иногда требуется какое-нибудь значение изменить. Пусть, например, в процессе отладки по тагам обнаружена ошибка, но выполнение программы желательно продолжить. Чтобы обойти ошибку, неверное значение можно исправить вручную. Это делается в окне быстрого просмотра значений, которое открывается командой Run > Evaluate/Modufy (Запуск > Определить/Изменить) или комбинацией клавиш CTRL+F7 (рис. 3.8).

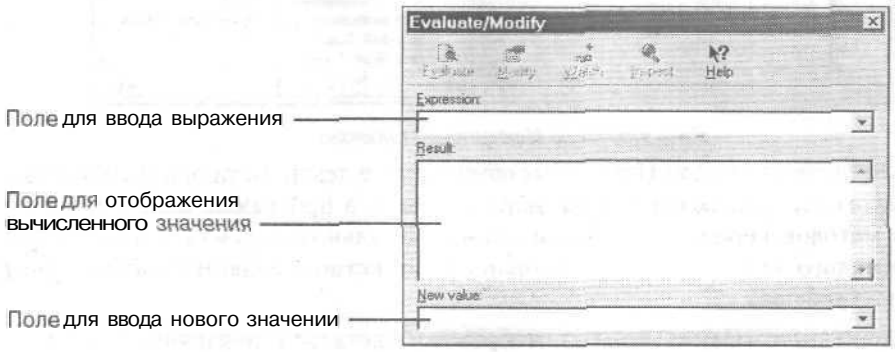


Рис. 3.8. *Окно быстрого просмотра значений*

В поле Expression (Выражение) вводится вычисляемое выражение. По щелчку на кнопке Evaluate (Вычислить) в поле Result (Результат) появится его значение. Это поле (панель) сделано таким большим, потому что в нем отображаются не только отдельные значения, но и массивы, и записи. В поле New value (Новое значение) выводится измененное значение. С помощью кнопки Watch (Следить) выражение, указанное в поле Expression (Выражение), можно добавить в окно слежения.

Просмотр и анализ кода

В системе *Delphi 7* имеются мощные средства для просмотра исходных текстов программ, позволяющие отображать структуру классов, типы переменных, их ме-

стоположение в программе, осуществлять навигацию по этим структурам и типам и быстро перемещаться по исходным текстам. Таких средств в системе несколько.

Навигатор проекта (Project Browser) вызывается командой View > Browser (Вид ➤ Навигатор) (рис. 3.9).

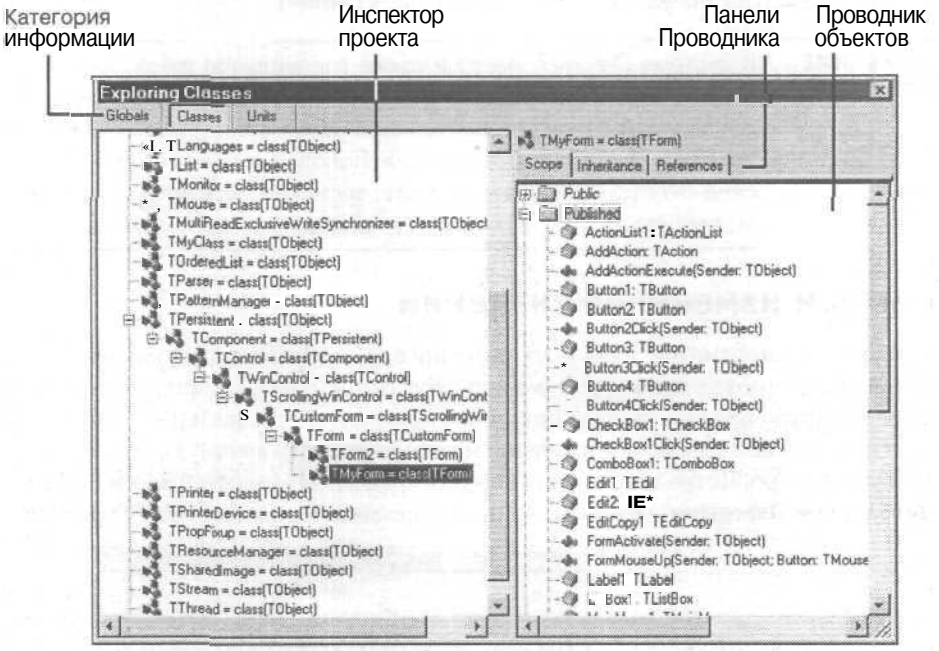


Рис. 3.9. *Окно Навигатора проекта*

Если нажата кнопка Globals (Глобальные объекты), то в левой части окна (это Инспектор проекта) отображаются списки используемых в программе классов, типов, свойств, методов, переменных и подпрограмм. Детально настроить способ отображения каждого элемента можно, выбрав в контекстном меню Навигатора пункт Properties (Свойства).

Если нажата кнопка Classes (Классы), отображается детальная иерархическая структура классов, используемых в программе.

Если нажата кнопка Units (Модули), отображаются все модули со взаимными ссылками и входящие в них переменные,

ЗАМЕЧАНИЕ Навигатор проекта также обычно присутствует в левой части редактора под именем Проводника по коду (Code Explorer), способного отображать структуру текущего модуля. Он вызывается командой контекстного меню редактора View Explorer (Открыть Проводник) или командой View ➤ Code Explorer (Вид ➤ Проводник по коду).

В Навигаторе каждый элемент структуры помимо названия содержит также небольшой значок (всего их восемь), поясняющий назначение этого элемента.

Таблица 3.1. Назначения элементов структуры

Значок	Элемент	Значок	Элемент
	Класс		Интерфейс
	Модуль		Переменная, константа
	Метод (процедура)		Метод (функция)
	Свойство		Тип

Нужный объект можно быстро найти, просто введя его имя. Навигатор автоматически перемещается по структуре, показывая объекты, подходящие по названию.

Если некоторый метод или подпрограмма только определены в интерфейсной части модуля, но не реализованы, их имена выделяются полужирным начертанием. Дважды щелкнув на таком имени, можно перейти к определению соответствующего метода в редакторе, а выбрав в контекстном меню (не убирая указатель с имени метода) пункт Complete class at cursor (Закончить реализацию класса), можно сразу автоматически сгенерировать пустую реализацию метода.

Для быстрой навигации по коду в системе *Delphi 7* имеются еще два метода.

1. Чтобы переключаться между описаниями подпрограммы в интерфейсном разделе и разделе реализации, надо, находясь внутри описания подпрограммы, использовать комбинации клавиш CTRL+SHIFT+UP и CTRL+SHIFT+DOWN.
2. Чтобы отметить некоторое место в тексте, надо сформировать для него закладку — нажать комбинацию клавиш CTRL+K и затем цифру от 0 до 9. При этом на внутренней кайме редактора появится зеленый прямоугольник с номером закладки.

```
for i := 1 to 10 do
 if i > 3 then Arr[i] := 0
          else Arr[i] := 1;
```

В дальнейшем вернуться к нужной закладке можно, нажав комбинацию клавиш CTRL+номер закладки (например, CTRL+1).

В правой части окна расположен Проводник объектов (Symbol Explorer). Его также можно вызвать командой Search > Browse Symbol (Поиск ➤ Выбор объекта) или непосредственно из редактора, наведя указатель на переменную и выбрав в контекстном меню пункт Browse Symbol at Cursor (Выбрать объект под указателем).

Этот Проводник показывает подробную информацию об объекте, выбранном на левой панели Навигатора проекта. На панели Scope (Область видимости) отображается список идентификаторов, объявленных внутри класса или модуля, на панели Inheritance (Наследование) — локальная иерархия в рамках текущего проекта для выбранного класса, на панели References (Ссылки) — список имен файлов и номеров строк в исходном тексте, где описан соответствующий идентификатор. Чтобы перейти к его описанию, надо выполнить двойной щелчок на нужном имени в Проводнике.

Навигация по коду. Порой разработчик забывает, что означает тот или иной класс или определение. Можно обратиться к справочной системе, но профессиональным

программистам обычно удобнее взглянуть на реализацию конкретного определения внутри системы *Delphi 7* (в стандартной библиотеке *VCL*). Для этого служит технология навигации по коду. Находясь в редакторе, надо нажать и удерживать клавишу CTRL, переместив при этом указатель к нужному определению. Курсор примет вид указательного пальца, а расположенный пол ним идентификатор, если он доступен для просмотра, выделяется синим цветом и подчеркиванием (рис. ЗЛО).

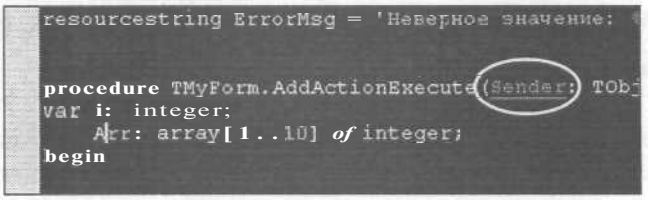


Рис. 3.10. *Выделение идентификатора в ходе навигации по коду*

Теперь достаточно щелкнуть на нем, и редактор переключится на файл, в котором хранится нужное определение. Например, если в модуле Unit1 в строке

```
type
TMyForm = class(TForm)
```

щелкнуть таким способом наслове TForm, то в редакторе будет открыт файл Forms.pas (реализация стандартного модуля Forms из библиотеки компонентов *VCL),* а курсор будет установлен в строку с началом определения TForm:

```
TForm = class(TCustomForm)
```

Автоподсказки. В системе *Delphi 7* реализовано несколько типов автоподсказок. Они включаются и выключаются в диалоговом окне настроек редактора, которое открывают с помощью пункта Properties (Свойства) в контекстном меню. Для этого используются флажки, расположенные на вкладке Code Insight (Анализ кода), как показано на рис. 3.11.

Флажок Code completion (Автозавершение) помогает при вводе членов класса. Когда в редакторе набирается название класса и ставится точка, система отображает список методов этого класса. Подходящий метод можно выбрать с помощью курсорных клавиш без ручного ввода (рис. 3.12).

Флажок Code parameters (Параметры вызова) упрощает ввод параметров для метода. Когда введено имя метода и поставлена открывающая скобка, система *Delphi 7* подскажет тип следующего параметра (рис. 3.13).

Флажок Tooltip expression evaluation (Быстрое вычисление значений) рекомендуется всегда держать установленным. В этом случае во время отладки значения переменных отображаются во вплывающем окне при наведении указателя.

Флажок Tooltip symbol insight (Информация об описании) управляет отображением всплывающей подсказки, которая поясняет, в каком модуле соответствующий идентификатор описан и какой тип имеет.

С помощью движка Delay (Задержка) устанавливается величина задержки перед появлением всплывающих подсказок (от 0,5 до 1,5 с).

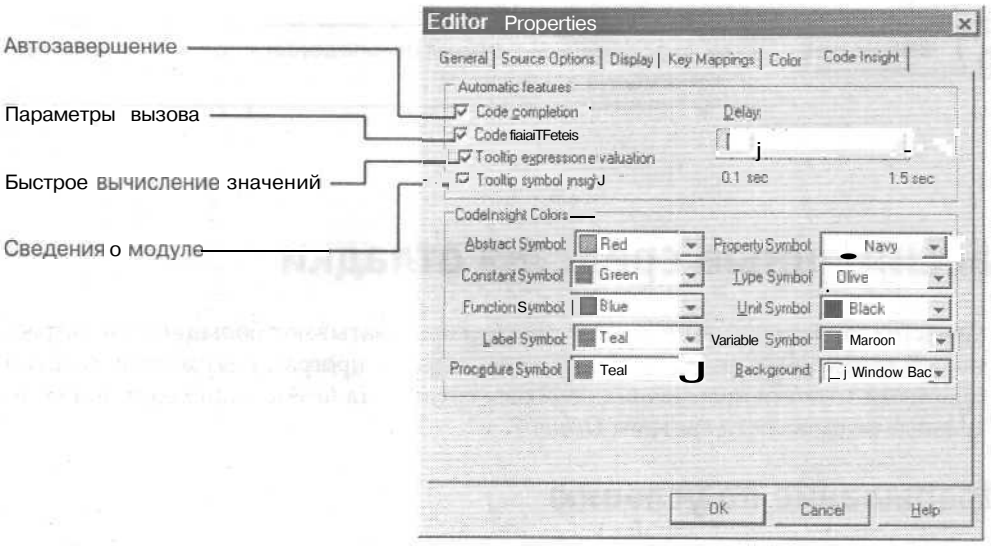


Рис. 3.11. Выбор средств автоматизации ввода кода

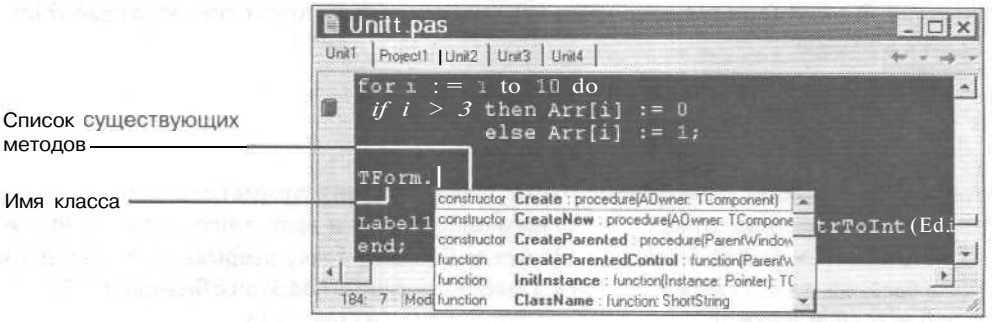


Рис. 3.12. Автоматизированный ввод имен методов

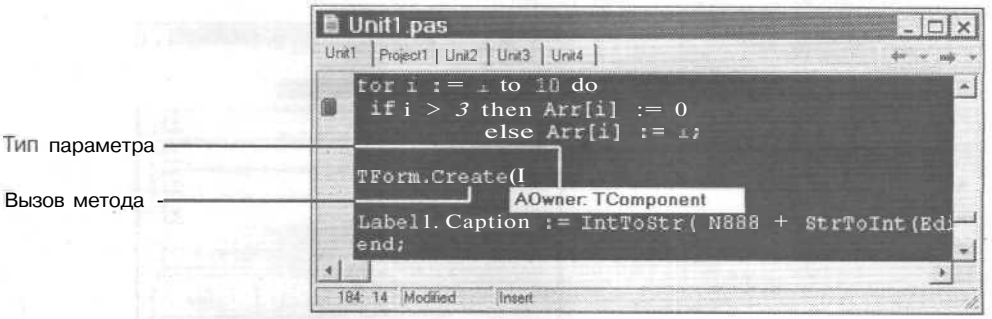


Рис. 3.13. Сведения о типе очередного параметра

ВНИМАНИЕ Если включены все режимы автоподсказок, работа редактора может заметно замедлиться.

Расширенные средства отладки

Средства пошагового выполнения программы охватывают большинство потребностей типичной разработки. Однако при создании программ со сложной логикой простыми точками прерывания обойтись трудно. На помощь приходят дополнительные возможности системы *Delphi 7*.

Прерывание по условию

Для пошаговой отладки обычно используется клавиша F8, но, например, когда выполняется длинный цикл, многократно нажимать клавишу F8 очень неудобно. Чтобы прервать работу цикла точно в нужный момент, к точке прерывания, установленной внутри цикла, надо добавить условие, при котором эта точка сработает. Если в примере

```
for i := 1 to 10 do
   if i > 3 then Arr[i] := 0
      else Arr[i] :- 1;
```

требуется прервать работу программы, когда счетчик примет значение 3, надо поместить курсор на строку с условным оператором и выполнить команду Run x Add Breakpoint ➤ Source Breakpoint (Запуск > Добавить точку прерывания x Исходная точка прерывания). На экране появится диалоговое окно Add Source Breakpoint (Добавление исходной точки прерывания), представленное на рис. 3.14.

В поле Condition (Условие) надо указать выражение i = 3 и щелкнуть на кнопке OK. Если теперь запустить программу и щелкнуть на кнопке Сложить, то ее работа пре-

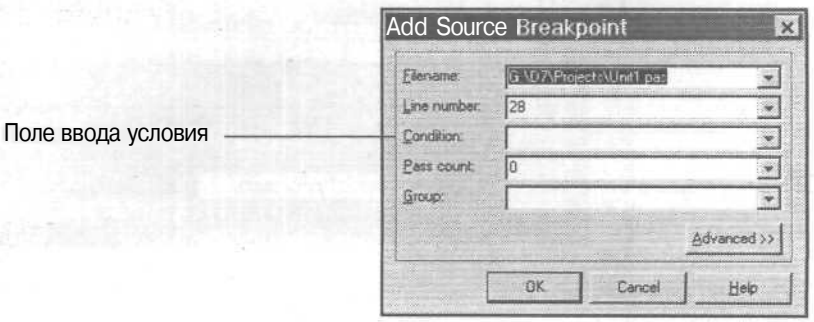


***Рис. 3.14.** Добавление условной точки прерывания*

рвется на данной точке, когда значение индекса i равно 3 (и, соответственно, первые два элемента массива Arr уже проинициализированы).

Можно также задать число повторов выполнения конкретной строки программы в поле Pass count (Число проходов). Перед первым проходом это число равно нулю, поэтому если оставить значение 0, предлагаемое по умолчанию, то программа прервется перед первым выполнением отмеченного оператора.

Группировка точек прерывания

Эта возможность *Delphi 7* позволяет объединить несколько точек прерывания в группу. Название группы вводится в поле Group (Группа) при установке точки прерывания. Подобный подход хорош тем, что позволяет выполнять схожие действия над целыми группами точек остановки: одновременно включать или выключать все точки, входящие в группу, с помощью раскрывающихся списков Enable Group (Включить группу) и Disable Group (Выключить группу). Это позволяет избежать путешествия по разным файлам в редакторе для поиска нужных строк и ручного переключения точек прерывания. Немаловажно, что при этом не происходит физического удаления точек.

Просмотр списка всех точек прерывания выполняется командой View ➤ Debug Windows ➤ Breakpoints (Вид ➤ Окна отладки ➤ Точки прерывания). На экране появляется окно, в котором отображается следующая информация:

- столбец Filename (Имя файла) указывает имя файла, в котором установлена точка прерывания;
- столбец Line (Строка) содержит номер строки, в которой установлена точка;
- столбец Condition (Условие) отображает условие активизации данной точки;
- столбец Action (Действие) показывает действие, выполняемое дополнительно (см. ниже);
- столбец Pass Count (Число проходов) указывает, сколько раз во время работы программы встретилась данная точка;
- в столбце Group (Группа) приводится название группы, к которой относится точка.

Быстро переключиться в редактор модуля к нужному месту исходного текста можно, выполнив щелчок на строке с соответствующей точкой прерывания. Для изменения состояния и характера работы точки выберите в контекстном меню этой точки пункт Properties (Свойства).

Действия

С каждой точкой прерывания можно связать некоторое действие, которое определяется на дополнительной панели в окне задания параметров работы выбранной точки. Эта панель открывается щелчком на кнопке Advanced (Дополнительно) и включает следующие элементы управления.

- Флажок Break (Прерывание) отвечает за обычное прерывание работы программы. Если он сброшен, то достижение точки прерывания по ходу работы программы не приведет к остановке приложения.
- Флажок Ignore subsequent exceptions (Игнорировать последующие исключительные ситуации) обеспечивает отключение всех исключительных ситуаций, возникающих при попытке выполнения ошибочных действий в программе. Это относится как к очевидным ситуациям, типа деления на ноль, так и к таким, как неверные преобразования типов и пр.
- Флажок Handle subsequent exceptions (Обрабатывать последующие исключительные ситуации), наоборот, включает поддержку всех исключительных ситуаций.

Ведение протокола работы

Выше упоминалось об отладке программы по шагам. Этот способ наиболее распространен среди программистов, однако при разработке больших систем такой отладкой обойтись удается не всегда, Основных причин две.

1. Иногда требуется выяснить, почему готовая система не работает у конкретного заказчика. Ехать к нему с исходными текстами и дистрибутивом *Delphi 7,* чтобы запустить программу из этой среды, неудобно и несолидно. Да и не всегда допустимо нарушать производственный процесс, отлаживая приложение таким способом в реальных условиях.
2. Отладка по шагам позволяет взглянуть на работу программы в статике: выполняется один оператор и программа «замирает», ожидая следующей команды. Сформировать глобальный взгляд на поток выполняющихся операторов таким образом невозможно.

Очень полезная возможность системы *Delphi 7* — это ведение протокола событий (*log*). Информация о состоянии различных объектов программы в определенных точках записывается в специальный протокол, который затем просматривается и анализируется. На его основе удается подчас обнаружить логические ошибки и некорректности, которые очень трудно выявить при пошаговой отладке и тестировании. Ведение протокола полезно и в готовых программах, когда пользоваться встроенным отладчиком *Delphi* 7 невозможно и остается только анализировать выдаваемый программой протокол работы, чтобы определить, где произошла ошибка,

ЗАМЕЧАНИЕ Пошаговые отладчики появились сравнительно недавно. Ранее, на больших ЭВМ, из-за крайне жестких требований к оперативной памяти и слабых вычислительных ресурсов программисты использовали технологию отладки, основанную только на ведении протоколов. В старых языках программирования были даже специальные операторы для выдачи «протокольной» информации. Спустя много лет в системе Delphi аналогичные средства появляются вновь, что говорит об их несомненной востребованности.

В протокол может записываться информация двоякого рода. Если в ходе задания параметров точки прерывания в поле Log message (Сообщение для протокола) введена какая-либо текстовая строка, то, когда программа во время своей работы встретит соответствующую точку прерывания, она запишет эту строку в протокол. Можно также указать произвольное выражение в поле Eval expression (Вычислить выражение). По достижении точки прерывания соответствующее выражение будет вычислено и, если установлен флажок Log result (Записать результат в протокол), занесено в протокол. Если флажок сброшен, то записи значения вычисленного выражения в протокол не произойдет.

ЗАМЕЧАНИЕ В ранних языках программирования встречались неточности в их описании или неверные реализации различных конструкций языка на уровне компиляторов. Это приводило к так называемым «побочным эффектам» и позволяло программистам использовать «черные ходы», добиваясь нужного результата неявным способом. Такая практика давно осуждена, и в современных реализациях популярных языков программирования подобных «дырок», конечно, нет. Все же в исключительных ситуациях, в особо тяжелых отладочных случаях полезна возможность, для примера, «на лету», без остановки работы откорректировать значение какого-то ошибочно измененного свойства, чтобы не мешать текущему выполнению программы. Именно для создания побочных эффектов и предназначена возможность системы Delphi 7 вычислять выражения без записи их в протокол.

Вернемся к вышеприведенному примеру и с помощью команды Run x Add Breakpoint ➤ Source Breakpoint (Запуск > Добавить точку прерывания ➤ Исходная точка прерывания) создадим точку прерывания на ключевом слове else. В поле Log message (Сообщение для протокола) укажем строку test. В поле Eval expression (Вычислить выражение) достаточно ввести только имя переменной i. Если теперь запустить программу, то ее работа не прервется, но будет сформирован протокол, который можно в любой момент просмотреть из оболочки *Delphi 7*. Для этого следует воспользоваться командой View ➤ Debug Windows ➤ Event Log (Вид ➤ Окна отладки > Протокол событий) или комбинацией клавиш CTRL+ALT+V.

Окна протокола можно гибко настраивать на вывод нужной информации. С помощью контекстного меню — пункт Properties (Свойства) — включается или отключается вывод следующей информации.

- Сообщения о встретившейся точке прерывания и выводимая ей информация относятся к категории Breakpoint messages (Сообщения точки прерывания). Если установлен флажок Display process info with event (Отображать также сведения о процессе), то дополнительно выводится информация о процессе *Windows*, который сгенерировал соответствующее сообщение.
- Сообщения об обращении отлаживаемого приложения к системным программам и библиотекам *Windows* относятся к категории Process messages (Сообщения процессов).

- Сообщения, выводимые в точках прерывания, — поля Log message (Сообщение для протокола) *и* Eval expression (Вычислить выражение) — входят в категорию Output messages (Сообщения вывода).
- Получаемые программой сообщения *Windows* входят в категорию Windows messages (Сообщения Windows). Их может быть очень много, поэтому злоупотреблять данной возможностью не стоит.
- Различные типы сообщений можно выделить разными цветами. Эти цвета настраиваются в разделе Use Event Log Colors (Использовать цвета сообщений протокола).

ПОДСКАЗКА Убрать все сообщения из окна протокола можно командой контекстного меню Clear Events (Удалить события). Сохранить протокол в текстовом файле позволяет команда Save Events to File (Сохранить события в файле).

Если в программе много точек прерывания, быстро получить информацию о функциях, выполняемых каждой из них, можно наведением указателя на красный круглый маркер в левой части окна редактора (на его рамке). При этом появится всплывающая подсказка с краткой информацией о назначении данной точки (рис. 3.15).

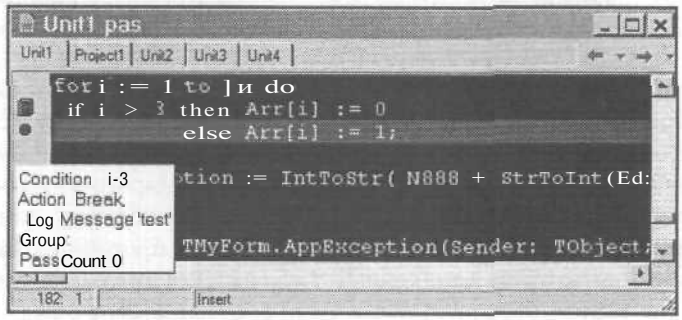


Рис. 3.15. *Получение сведений о точке прерывания*

Отладка внешних процессов

Отлаживать приложение, запущенное автономно, в том числе на другом компьютере, можно с помощью команды Run ➤ Attach to Process (Запуск ➤ Присоединить процесс). Для этого надо выбрать данное приложение в списке доступных процессов и нажать кнопку Attach (Присоединить). Конечно, присоединяемое приложение должно быть создано с помощью средств разработки компании *Borland* и содержать отладочную информацию (рис. 3.16).

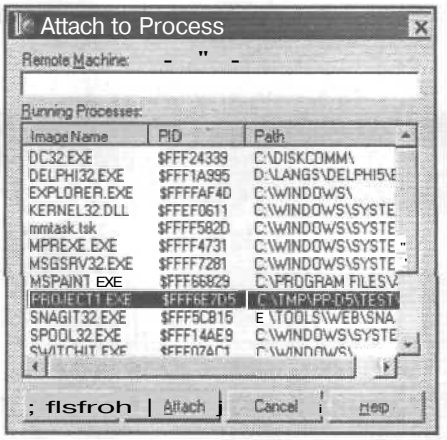


Рис. 3.16. *Присоединение внешнего приложения для отладки*

Машинный код

Даже прикладным программистам иногда приходится заглядывать в код, сгенерированный компилятором, чтобы выяснить причину той или иной ошибки. Обычно такая потребность возникает, если нюансы работы какой-то стандартной подпрограммы не ясны или требуется выяснить, во что транслируются конкретные операторы исходного текста, чтобы потом попробовать самостоятельно улучшить некоторые наиболее критичные по быстродействию участки кода.

Для этого надо во время работы программы вызвать окно машинного кода командой View ➤ Debug Windows ➤ CPU (Вид ➤ Окна отладки > Процессор) (рис. 3.17).

Конечно, чтобы разобраться в содержимом этого окна, надо понимать структуру оперативной памяти компьютера и хорошо знать ассемблер. В окне машинного кода после строки исходного текста программы следует соответствующий ей код, сгенерированный компилятором. Например, можно увидеть, что по адресу 00457F34 расположен отладочный вызов подпрограммы проверки нахождения индекса в допустимых границах:

```
call @BoundErr
```

В правой верхней части окна приводится состояние регистров процессора. Значения, которые были изменены на последнем шаге, подсвечиваются красным цветом. Под кодом программы расположено окно дампа («снимка» содержимого определенной области) памяти, внизу справа — окно стека программы (области памяти, отведенной для сохранения данных на время обработки вызовов подпрограмм и обеспечивающей возвращение к прерванной программе). При желании значения байтов памяти можно напрямую корректировать из среды *Delphi 7,* выделив нужный байт и выбрав в контекстном меню пункт Change (Изменить).

Рис. 3.17. Окно отладки в машинном коде

ЗАМЕЧАНИЕ Контекстные меню каждой панели окна машинного кода содержат вспомогательные пункты, позволяющие менять представление информации и выполнять быстрый поиск и навигацию по командам и данным.

В частности, при работе с машинным кодом пригодится команда Trace to Next Source Line (Выполнение до следующей строки исходного кода), которая немного напоминает работу клавиши F4 и запускается через строку меню — Run ➤ Trace to Next Source Line (Запуск ➤ Выполнение до следующей строки исходного кода) — или нажатием комбинации клавиш SHIFT+F7. При этом система *Delphi 7* выполняет инструкции машинного кода до первой команды, с которой начинается код, соответствующий очередному оператору Паскаля. Вернуться к исходному тексту можно с помощью комбинации CTRL+V.

ЗАМЕЧАНИЕ Система Delphi 7 позволяет устанавливать точки прерывания с расширенными возможностями не только на операторы Паскаля, но и на машинные инструкции. Для этого надо дать команду Run ➤ Add Breakpoint > Address Breakpoint (Запуск > Добавить точку прерывания ➤ Точка прерывания на машинному адресу).

Одно из существенных нововведений системы *Delphi 7* — *окно* состояния регистров математического сопроцессора, вызываемое командой View ➤ Debug Windows ➤ FPU (Просмотр ➤ Окна отладки ➤ Сопроцессор). Оно позволяет контролировать ход вычислений с плавающей запятой, следить за регистрами и флажками состояния сопроцессора, за информацией, специфичной для технологии *MMX*. Такая возможность позволяет вручную оптимизировать программный код, связанный с интенсивными вычислениями с плавающей запятой, которые выполняются не очень быстро.

Необходимо отметить такую полезную возможность системы *Delphi 7,* как точка прерывания, реагирующая не на выполнение инструкции, а на попытку изменить содержимое переменной или значение байта, расположенного по конкретному адресу оперативной памяти. Для этого служит команда Run > Add Breakpoint x Data Breakpoint (Запуск ➤ Добавить точку прерывания > Точка прерывания по изменению значения). Пусть у нас имеется следующий фрагмент программы:

```
var N888: integer;
...
for i := 1 to 10 do
   if i > 3 then N888 := 5;
```

Допустим, требуется прервать выполнение программы при попытке изменить значение переменной N888. Для этого выполните команду Run > Add Breakpoint ➤ Data Breakpoint и в поле Address (Адрес) введите имя переменной N888. Теперь при щелчке на кнопке Сложить в отлаживаемом приложении работа будет прервана на операторе

```
if i > 3 then N888 := 5;
```

в тот момент, когда значение индекса i станет больше 3,

ВНИМАНИЕ Команды Add Address Breakpoint (Точка прерывания по машинному адресу) и Add Data Breakpoint (Точка прерывания по изменению значения) не приводят к выделению ярким цветом строк исходного текста, содержащих точки прерывания, так кок привязываются не к операторам, а к значениям переменных. Установить такие точки можно *только* во время работы программы.

Инспектор отладки

В системе *Delphi 7* имеется еще один способ просмотра и изменения значений простых переменных и сложных структур: массивов, записей, классов — уже на уровне машинного кода. Команда Run > Inspect (Запуск > Инспектор), эквивалентная команда контекстного меню Debug > Inspect (Отладка > Инспектор) или комбинация клавиш ALT+F5 вызывают Инспектор отладки. Это средство либо сразу «инспектирует» содержимое переменной, па которой расположен курсор, либо предлагает ввести идентификатор, если курсор находится в ином месте.

ВНИМАНИЕ К Инспектору отладки можно обращаться только непосредственно во время отладки, когда программа запущена из среды Delphi 7.

Например, если вызвать Инспектор отладки при работе приложения Project1 и указать в качестве инспектируемого объекта переменную MyForm (главная форма), появится окно, структура которого напоминает структуру Инспектора объектов.

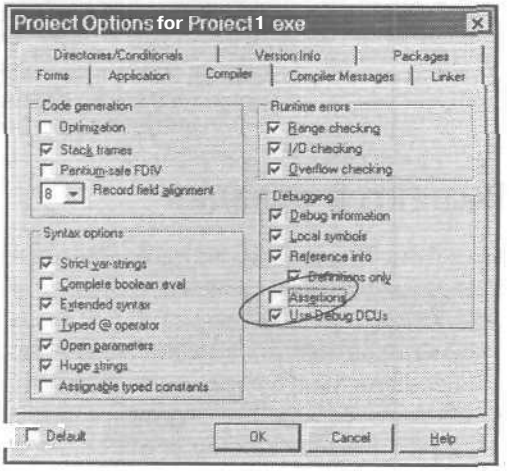
В верхней части окна указывается имя анализируемого объекта, его тип и физический адрес. Доступ к прочим данным осуществляется при помощи вкладок:

- вкладка Data (Данные) отображает поля, определенные пользователем, а также поля, скрытые внутри класса;
- вкладка Methods (Методы) содержит методы класса с указанием класса-родителя, модуля, в котором он описан, и физического адреса начала подпрограммы в машинном коде;
- на вкладке Properties (Свойства) приведены свойства класса и их текущие значения.

Все эти значения легко изменить, выделив нужную строку в Инспекторе отладки и выбрав в контекстном меню пункт Change (Изменить). Если какое-то поле слишком сложно (например, тоже представляет собой класс), то для него можно вызвать отдельный Инспектор отладки, выделив это поле и выбрав в контекстном меню пункт Inspect (Инспектор). Пункт Descend (Погрузиться) используется, чтобы отобразить новую информацию в текущем окне Инспектора отладки.

Средство отладки, ориентированное на программиста

Процедура Assert пришла в Паскаль из других языков программирования, потому что она активно и успешно используется для отладки уже несколькими поколениями разработчиков. У этой процедуры два параметра. Первый — логическое выражение (тип Boolean), второй — строка. Если значение выражения равно True, то ничего не происходит, в противном случае возникает искусственно сгенерированная ошибочная ситуация, а пользователю выдается сообщение о месте, где расположена данная процедура (имя файла, номер строки) и информационное сообщение (второй параметр).



Процедура Assert полезна тем, что записывается компактно и позволяет добавить промежуточную проверку принадлежности значений переменных

допустимым диапазонам в любое место программы. Подобные проверки желательно вставлять в исходные тексты как можно чаще, чтобы контролировать правильность процесса вычислений. Кроме того, в конечной версии приложения, где наличие подобных проверок не требуется, удалять вызовы процедуры Assert из текста не обязательно. Достаточно в любом месте файла, где применяется эта процедура, указать директиву компилятора

```
{$C-}
```

и компилятор не будет генерировать машинный код для вызова процедуры Assert. Этого же можно достичь, изменив отладочные настройки компилятора в визуальной оболочке (сбросив флажок Assertions).

```
Assert( I in [2..9], 'Счетчик вне диапазона' );
Assert( N o 0, 'Делитель не может быть равным нулю' );
```

Исключительные ситуации

В языке Паскаль важную роль играет обработка *исключительных ситуаций,* связанных с попыткой выполнения во время работы программы какого-то действия, приводящего к ошибке или просто нарушающего ее функционирование и делающего невозможным дальнейшее нормальное выполнение.

Исключительные ситуации контролируются специальным обработчиком исключительных ситуаций. Он перехватывает практически все возникающие в программе ошибки, приостанавливает программу, не давая выполниться разрушительной команде, и сообщает об этом пользователю (рис. 3.18) и программе, передавая ей информацию об обнаруженной ошибке в виде объекта, относящегося к специальной иерархии классов, описывающих исключительные ситуации. Базовым в этой иерархии является класс Exception. Он описан в модуле SysUtils.

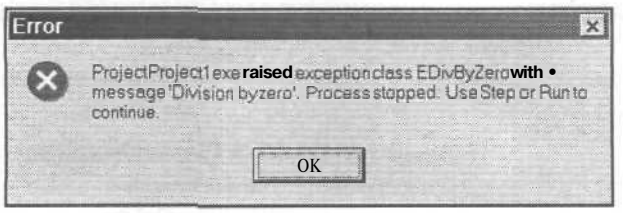


Рис. 3.18. *Уведомление о возникновении в программе исключительной ситуации (деления на ноль)*

Для оперативного вывода коротких сообщений можно использовать стандартную функцию ShowMessage, которая имеет один параметр-строку и отображает очень похожее диалоговое окно с единственной кнопкой OK:

```
ShowMessage('Это сообщение от программы');
```

Генерация исключительной ситуации

Классы иерархии Exception обладают несколькими видами полезных конструкторов, которые можно применять для искусственной генерации соответствующей исключительной ситуации.

Таблица 3.2. Конструкторы классов иерархии Exception

Конструктор	Параметры
Create(const Msg: string);	Параметр — строка, которая будет отображаться а диалоговом окне, информирующем о возникновении исключительной ситуации
CreateRes(ResStringRec: PresStringRec); overload;	Параметр — строка сообщения, которая загружается из ресурсов программы
CreateFmt (const Msg: string; const Args: array of const); CreateResFmt(ResStringRec: PresStringRec; const Args: array of const); overload;	Массив Args содержит значения, на основе которых с использованием строки форматирования Msg (см. функцию Format) или ResStringRec (загружается из ресурсов программы) формируется результирующее сообщение
CreateHelp(const Msg: string; AHelpContext: Integer);	Идентификатор раздела справки AHelpContext указывает, где пользователь сможет подробнее узнать о возникшей ошибке. Справочная система для текущей программы должна быть создана заранее

Создание объекта и вызов нужного конструктора осуществляется с помощью ключевого слова **raise:**

```
if N < 1 then
   raise Exception.Create('Значение переменной K меньше 1');
```

Стандартные классы исключительных ситуаций

В библиотеках системы *Delphi 7* имеется немало классов, ответственных за обработку различных исключительных ситуаций. В отличие от остальных типов Паскаля, названия которых принято записывать начиная с буквы T, имена этих классов начинаются с буквы E (Exception). Основные классы исключительных ситуаций приведены в табл. 3.3.

Таблица 3.3. Основные классы исключительных ситуаций

Имя класса	Когда возникает
EAbort	Данный класс предназначен для формирования и обработки «невидимых» для пользователя ошибок и используется разработчиками для управления ходом выполнения программы. Диалоговое окно с сообщением об ошибке не показывается
EAbstractError	Попытка выполнения абстрактного метода
EAccessViolation	Обращение к недоступной области памяти, например при выходе индекса за границы массива

Таблица 3.3. Основные классы исключительных ситуаций (продолжение)

Имя класса	Когда возникает
EAssertionFailed	Значение выражения в процедуре Assert равно False
EControlC	Нажата комбинация клавиш CTRL+C в консольном приложении
EConvertError	Попытка неверного преобразования типов, например при вызове StrToInt('aaa')
EDivByZero	Деление на ноль
EExternal	Неверное функционирование системы Windows
EExternalException	Невозможность распознать исключительную ситуацию средствами Delphi 7
EHeapException	Неверное динамическое распределение памяти или некорректная работа с указателями
EFileStreamError	Ошибка ввода/вывода при использовании файла, название которого указано в параметре FileName данного класса
EInOutError	Ошибка файлового ввода/вывода
EIntError	Базовый класс, на основе которого созданы классы исключительных ситуаций при работе с целыми числами
EIntOverflow	Слишком большой результат при операции с целыми числами
EIntfCastError	Неверное приведение типа объекта с помощью операции as во время обращения к интерфейсу
EInvalidCast	Неверное приведение типа с помощью операции as
EInvalidOp	Неверная операция над числами с плавающей запятой
EInvalidPointer	Неверная операция при работе с указателями
EMathError	Базовый класс, на основе которого созданы классы исключительных ситуаций при работе с числами с плавающей запятой
EOutOfMemory	Нехватка памяти
EOverflow	Переполнение при выполнении операции над числами с плавающей запятой
EPackageError	Некорректная работа с пакетами (см. далее). Возникает только на этапе проектирования в среде Delphi 7
EPrivilege	Попытка выполнить неразрешенную привилегированную инструкцию процессора
EPropReadOnly	Попытка занесения значения в свойство объекта, доступное «только для чтения»
ERangeError	Значение выражения целого типа выходит за допустимый для этого типа диапазон
EStackOverflow	Нехватка памяти в стеке. Возникает, когда используются слишком объемные локальные переменные (они создаются и хранятся в стеке) или когда слишком длинна последовательность вызовов вложенных подпрограмм
EUnderflow	Результатом выражения над числами с плавающей запятой является число, которое слишком мало для его представления в программе
EVariantError	Некорректное использование переменных типа Variant например при попытке неверного приведения типов
EWin32Error	Ошибка 32-разрядной"версии Windows
EZero Divide	Деление на ноль

ЗАМЕЧАНИЕ Иногда исключительные ситуации используют, чтобы изменить порядок выполнения операторов программы. Если, например, обнаружена ошибка при работе какого-либо метода, желательно не просто завершить его работу, а передать управление в часть программы, ответственную за исключительные ситуации. Для этого обычно применяют процедуру Abort (без параметров), генерирующую исключительную ситуацию класса EAbort.

Контроль над исключительными ситуациями

Для контроля над исключительными ситуациями в группе операторов Паскаля применяется следующая конструкция:

```
try
   операторы;
except
   обрабатываемые классы исключительных ситуаций;
else оператор;
end;
```

Ключевое слово *try* (попытка) обозначает начало блока контроля выполнения операторов, следующих до ключевого слова **except.** В случае возникновения исключительной ситуации происходит обращение к списку классов, перечисленных перед завершающим ключевым словом **end.** При этом выполняется действие, указанное для соответствующего класса, а затем управление передается первому оператору, следующему за завершающим ключевым словом **end.** Операторы, оставшиеся в части try. пропускаются. Если исключительные ситуации не встретились, то пропускаются все действия, следующие за ключевым словом **except.**

Если возникшая ситуация не относится ни к одному из явно обрабатываемых классов, то выполняется команда Паскаля, указанная после ключевого слова **else.** Часть **else** в блоке **try** указывать не обязательно.

Классы исключительных ситуаций, предназначенные для обработки, записываются в следующем формате:

```
on название-класса do операторы;
```

Таких классов может быть несколько, например:

```
try
   Assert ( Y o 5, ' ' );
   X := 100 div Y;
except
   on EZeroDivide do ZeroProc;
   on EAssertionFailed do
      begin
```

```
            ShowMessage('Ошибка # 22');
            X := 0;
        end;
else ShowMessage('Непонятно что');
end;
```

Если в операторе присваивания будет обнаружена попытка деления на ноль (для этого надо, чтобы значение переменной Y было равно 0), то выполнится процедура ZeroProc, которая должна быть определена ранее. Если же значением переменной Y окажется число 5, то обработчиком исключительных ситуаций будет сгенерирован объект класса EAssertionFailed и выполнится группа операторов в логических скобках. Оператор X := 100 div Y при этом будет пропущен. Если встретится какая-то другая исключительная ситуация, то выведется сообщение Непонятно что.

Поиск класса подходящей исключительной ситуации осуществляется в последовательном порядке. Если возникшую ошибку можно отнести к нескольким классам, то вызван будет обработчик для класса, расположенного первым, например:

```
try
   X := Y + Z;
except
   on EIntError do P1;
   on EIntOverflow do P2;
end;
```

Если при выполнении оператора X :- Y + Z возникнет ошибка переполнения EIntOverflow, то вызвана, тем не менее, будет подпрограмма P1, потому что эта ошибка относится также и к классу EIntError, расположенному первым в списке за ключевым словом **except.**

Можно создать единый обработчик для любой исключительной ситуации. Тогда блок try записывается так:

```
try
   операторы;
except
   действия;
end;
```

Например:

```
try
   X := 0;
except
   GlobalProc;
end;
```

В случае возникновения во время исполнения оператора X := 0 произвольной исключительной ситуации вызывается процедура GlobalProc.

Иногда реализованной программистом обработки ошибки недостаточно. Тогда, указав ключевое слово **raise** без параметра, можно передать управление стандартному обработчику *Delphi 7:*

```
try
   X := 100 div Y;
except
   on EZeroDivide do
      begin
         ShowMessage('Ошибка # 22');
         raise;
      end;
end;
```

После выдачи сообщения Ошибка # *22* исполнится стандартный обработчик ошибки EZeroDivide.

Из текста обработчика ошибки можно генерировать другие исключительные ситуации, например:

```
try
   X := Y + Z;
except
   on EIntError do
      raise EIntOverflow.Create('Возможно переполнение');
end;
```

Если при сложении целых чисел, хранящихся в переменных Y и Z, возникнет какая-то арифметическая ошибка (базовый класс EIntError), то в обработчике этой ошибки будет сгенерирована другая исключительная ситуация EIntOverflow, а обработка ошибки EIntError завершится.

Выполнение завершающих действий

В некоторых ситуациях программисту не нужен собственный обработчик ошибок, но требуется, чтобы программа гарантированно выполнила определенные действия, связанные, например, с освобождением ресурсов. В такой ситуации удобнее использовать следующий блок:

```
try
   операторы
finally
   заключительные действия
end;
```

Заключительные действия будут выполнены в любом случае, независимо от того, возникнет ли исключительная ситуация в операторах части try или, например, выполнится попытка выхода из подпрограммы с помощью процедуры Exit.

В следующем тексте освобождение памяти, выделенной для динамического массива DynArr (финальный оператор присваивания значения nil), произойдет независимо от того, успешно ли создан и обработан массив DynArr или памяти для него не хватило:

```
var DynArr: array of integer;
...
try
   SetLength(DynArr,100000);
...
finally
   DynArr := nil
end;
```

Передача объектов, связанных с исключительными ситуациями

Когда в программе возникает исключительная ситуация, специальный обработчик создает соответствующий ей объект. Чтобы получить доступ к этому **объекту,** в описании класса в части **except** надо предварительно указать произвольный идентификатор:

```
on E: EIntError do P;
```

В случае возникновения ошибки EIntError объект соответствующего класса будет сохранен в переменной E (описывать ее не надо), к которой можно обращаться, например для занесения новых значений в ее свойства.

У любого класса исключительной ситуации есть два свойства: Message, в котором хранится строка, выводимая в окне сообщения, и HelpContext, число-идентификатор раздела справочной системы для вывода более подробной информации об ошибке.

```
on E: EIntError do
   begin
      E.Message := 'Ошибка вычислений';
      raise;
   end;
```

Если доступ к объекту, описывающему ошибку, надо получить в части **else** блока **except,** где никакие классы явно не упоминаются, можно использовать функцию ExceptObject, которая возвращает такой объект.

Программный обработчик ошибок

По умолчанию при возникновении исключительной ситуации для ее обработки вызывается процедура HandleException. Она проверяет, обрабатывает ли приложе-

ние событие OnException. Если обработка отсутствует, выводится диалоговое окно с кратким сообщением об ошибке. Такое окно можно вызывать с помощью стандартной процедуры ShowException, которая получает в качестве параметра объект, имеющий тип исключительной ситуации:

```
on E: EIntError do
   ShowException(E);
```

Обработчик события OnException задается явно, с помощью оператора присваивания, например в методе создания или активации формы:

```
procedure TMyForm.FormActivate(Sender: TObject);
   begin
   Application.OnException := AppException;
end;
```

Процедуру AppException надо определить в классе TmyForm:

```
procedure AppException(Sender: TObject; E: Exception);
```

Затем ее надо описать в части реализации. В следующем примере при обнаружении исключительной ситуации отображается стандартное информационное окно, после чего работа приложения завершается:

```
procedure TMyForm.AppException (Sender: TObject; E:
Exception);
begin
   Application.ShowException(E);
   Application.Terminate;
end;
```

Что нового мы узнали?

В этом уроке мы научились

☑ выполнять программу по шагам;

0 определять и использовать точки прерывания;

0 контролировать значения переменных в ходе работы программы;

0 создавать протокол работы программы;

0 вести отладку в машинном коде;

☑ обрабатывать исключительные ситуации.

Современные компоненты интерфейса пользователя

- Основы интерфейса пользователя
- Работа с графикой
- Работа с файлами
- Стандартные диалоговые окна Windows
- Печать из программы
- Дополнительные компоненты Delphi 7
- Панель Win32
- Панель System (Системные компоненты)

Основы интерфейса пользователя

Составляющие пользовательского интерфейса

На самом высоком уровне пользовательский интерфейс основывается на четырех базовых классах *Delphi 7:*

- TApplication (фундаментальный класс, свойства и методы которого описывают основные характеристики приложений *Windows);*
- TClipboard (его средства определяют взаимодействие с буфером обмена *Windows);*
- TForm (описывает свойства и методы для создания окон, строительных блоков для размещения визуальных и невизуальных компонентов);
- TScreen (средства для задания системных настроек: разрешение экрана, доступные шрифты и прочее).

Приложение (класс TApplication)

На основе данного класса программист может обрабатывать системные сообщения *Windows,* предназначенные приложению, организовывать оперативные подсказки и вообще выполнять специфические действия, зависящие от операционной системы.

Каждая программа, создаваемая в системе *Delphi 7,* имеет доступ к глобальной переменной Application класса TApplication. К ней следует обращаться, когда необходимо воспользоваться возможностями этого класса.

Основные свойства, методы и события класса TApplication перечислены в табл. 4.1–4.3.

Таблица 4.1. *Основные свойства класса TApplication*

Свойство	Назначение
Active	Имеет значение True, если приложение активно (имеет фокус)
CurrentHelpFile	Имя текущего справочного файла (расширение .HLP)
ExeName	Полное имя файла программы вместе с путем поиска (например, c:\delphi\project1.exe)
Handle	Внутренний идентификатор программы в системе Windows (требуется для системных нужд)
HelpFile	Имя справочного файла, который используется по умолчанию (например, при отсутствии нужного раздела в файле, указанном в свойстве CurrentHelpFile)
Hint	Текст всплывающей подсказки, принятый по умолчанию
HintColor	Цвет окна всплывающей подсказки
HintHidePause	Временной интервал в миллисекундах, по истечении которого всплывающая подсказка будет убрана
HintPause HintShortPause	Временной интервал в миллисекундах, по истечении которого всплывающая подсказка появляется на экране. Свойство HintShortPause определяет этот интервал для случая, когда уже отображается другая подсказка

Таблица 4.1. *Основные свойства класса TApplication (продолжение)*

Свойство	Назначение
HintShortCuts	Имеет значение True, если в тексте всплывающей подсказки дополнительно будет отображаться информация о «горячей клавише». Обычно такая возможность применяется, когда указатель наведен на пункт меню или другой элемент управления, снабженный такой клавишей
Icon	Значок (класс TIcon), который будет использоваться системой Windows при идентификации данного приложения: при размещении значка на рабочем столе, при переключении между программами с помощью комбинации ALT+TAB и так далее
MainForm	Данное свойство имеет тип TForm и определяет главную форму программы
ShowMainForm	Имеет значение True, если главной считается форма, которая была таковой на этапе проектирования. Если требуется сделать главной другую форму приложения, надо задать данному свойству значение False, указать другую форму в свойстве MainForm, а также установить значение свойства Visible формы, которая была главной на этапе проектирования, равным False
Terminated	Имеет значение True, если приложение получило от Windows сообщение WM_QUIT, означающее, что приложение должно закончить работу
Title	Заголовок приложения, строка, которая отображается на кнопке Панели задач

Таблица 4.2. *Основные методы класса TApplication*

Метод	Назначение
procedure ActivateHint(CursorPos: TPoint);	Отображает всплывающую подсказку в заданной точке экрана
procedure BringToFront;	Перемещает последнее из активных окон на передний план экрана
procedure CancelHint;	Убирает всплывающую подсказку
procedure HandleMessage;	Прерывает работу программы для обработки очередного системного сообщения Windows, которое хранится в очереди сообщений. Данный метод полезен, когда приложение выполняет длительные вычисления и требуется периодически проверять, не поступила ли от операционной системы какая-либо команда
function HelpCommand(Command: Word; Data: Longint): Boolean;	Быстрый доступ к системной функции Windows, определяемой параметром Command и ответственной за работу справочной системы
function HelpContext(Context: THelpContext): Boolean; function HelpJump(const JumpID: string): Boolean;	Отображает указанный раздел справочной системы
function IsRightToLeft: Boolean;	Возвращает значение True, если в приложении используется работа элементов управления в режиме «справа налево»
function MessageBox(const Text, Caption: PChar; Flags: Longint): Integer;	Показывает стандартное диалоговое окно, содержащее кнопки, определяемые параметром Flags

продолжение

Таблица 4.2. *Основные методы класса TApplication (продолжение)*

Метод	Назначение
procedure Minimize;	Свертывает все окна приложения
procedure NormalizeAllTopMosts; procedure NormalizeTopMosts;	Переводит все окна приложения (при использовании метода NormalizeTopMosts — за исключением главной формы) из состояния «всегда поверх» в обычный режим
procedure ProcessMessages;	Работает аналогично методу HandleMessage, но обрабатывает не одно, а все системные сообщения из очереди Windows
procedure Restore;	Восстанавливает все свернутые окна программы до нормального размера
procedure RestoreTopMosts;	Переводит все окна из нормального состояния в состояние «всегда поверх», если эти окна ранее уже находились в таком состоянии, измененном, например, с помощью метода NormalizeAllTopMosts
procedure ShowException(E: Exception);	Выводит диалоговое окно с сообщением, описывающим исключительную ситуацию (параметр E)
procedure Terminate;	Завершает работу приложения
function UseRightToLeftAlignment: Boolean; function UseRightToLeftReading: Boolean; function UseRightToLeftScrollBar: Boolean;	Возвращает значение True, если режим работы «справа налево» используется, соответственно: для выравнивания объектов, для вывода текстовой информации, для отображения полос прокрутки (с левой стороны от элементов управления)

Таблица 4.3. *Основные события класса TApplication*

Событие	Условие генерации
OnAction Execute OnAction Update	События, не определенные разработчиком в списке событий
On Activate	Приложение становится активным
OnDeactivate	Приложение становится неактивным (теряет фокус)
OnException	Возникает исключительная ситуация, которая не контролируется программно (с помощью блока try)
On Help	Приложение получает запрос на выдачу справочной информации
OnHint	Пользователь переместил указатель на элемент управления, способный выдавать всплывающую подсказку
OnIdle	Приложение находится в состоянии ожидания, не выполняя какой-либо программный код (например, ожидается ввод информации от пользователя)
OnMessage	Приложение получило системное сообщение от Windows
OnMinimize	Приложение свернуто
OnRestore	Приложение восстановлено до нормального размера из свернутого состояния

Таблица 4.3. Основные события класса TApplication (продолжение)

Событие	Условие генерации
OnShortCut	Пользователь нажал клавишу. Данное сообщение генерируется ранее всех остальных сообщений, связанных с обработкой нажатий клавиш (OnKeyDown, OnKeyPress, OnKeyUp)
OnShowHint	Приложение готовится вывести всплывающую подсказку

Вот несколько примеров использования класса TApplication.

1. Предположим, выполняется длинный цикл с большим объемом вычислений.

```
for i := 1 to 1000000 do
  begin
  // счет
  end;
```

Прямой подход не совсем корректен, потому что цикл монопольно захватывает ресурсы процессора. Чтобы позволить пользователю переключиться на другую программу, а системе *Windows*— оптимально распределять нагрузку, в цикл надо добавить команду обработки очереди системных сообщений *Windows*.

```
for i := 1 to 1000000 do
  begin
  // счет
  Application.ProcessMessages;
  end;
```

2. Чтобы начать работу с файлом справочной системы, надо в тексте программы указать его название:

```
Application.HelpFile := 'PROGHELP.HLP';
```

Выполнить команду поиска можно с помощью метода HelpCommand.

```
Application.HelpCommand(HELP_FINDER, 0);
```

Вызвать конкретный раздел справочной системы позволяет метод HelpContext.

```
Application.HelpContext(DATANOTFOUND);
```

3. Если требуется завершить работу программы при возникновении необрабатываемой исключительной ситуации, можно подменить стандартный обработчик таких ситуаций:

```
// собственный обработчик
procedure TForm1.AppException(Sender: TObject;
  E: Exception);
begin
// вывод описания исключительной ситуации
// в диалоговом окне:
Application.ShowException(E);
// завершение приложения
Application.Terminate;
end;
```

```
procedure TForm1.FormCreate(Sender: TObject);
begin
// в момент создания формы
// указываем собственный обработчик
// исключительных ситуаций:
Application.OnException := AppException;
end;
```

Класс Буфер обмена (TClipboard)

Данный класс позволяет работать со стандартным буфером обмена ***Windows.*** Он описан в модуле Clipbrd. Специально создавать объект типа TClipboard не требуется. В системе *Delphi 7* имеется готовый объект Clipboard, к которому можно обращаться для использования буфера обмена.

Свойства и методы класса TClipboard приведены в табл. 4.4 и 4.5. Некоторые стандартные форматы данных для буфера обмена *Windows* описываются значениями-константами, указанными в табл. 4.6.

Таблица 4.4. *Свойства класса* ***TClipboard***

Свойство	Назначение
AsText	Содержимое буфера обмена в виде строки. Данное свойство позволяет также заносить информацию в буфер с помощью обычного оператора присваивания. Clipboard.AsText := Edit1.Text;
FormatCount	Число форматов данных буфера обмена, поддерживаемых системой
Formats	Массив доступных (зарегистрированных) форматов для буфера обмена

Таблица 4,5. *Методы класса TClipboard*

Метод	Назначение
procedure Assign(Source: Tpersistent);	Копирование объекта, указанного в качестве параметра, в буфер обмена, например: Clipboard.Assign(Bitmap1); (копирование точечного изображения е буфер обмена). Получить содержимое буфера можно аналогичным способом: Bitmap1.Assign (Clipboard);
procedure Clear;	Удаление текущего содержимого буфера обмена Windows
procedure Close;	Закрытие буфера обмена. Выполняется, когда запись данных в буфер происходит в несколько приемов, и очередное занесение информации не должно стирать прежнего содержимого буфера обмена. После того как буфер обмена закрыт, запись в него новых данных уничтожит старую информацию
function GetComponent(Owner, Parent: TComponent): TComponent; procedure SetComponent(Component: TComponent);	Запись данных в буфер обмена (SetComponent) и выполнение вставки из буфера (GetComponent). Работа с буфером выполняется на уровне компонентов Delphi 7 (класс TComponent). Параметр Owner определяет объект, принимающий содержимое буфера, а параметр Parent — родителя объекта Owner (обычно это форма). В следующем примере расположенная на текущей форме кнопка Button1 будет скопирована в буфер обмена и затем вставлена на панель GroupBox1

Таблица 4.5. Методы класса TClipboard(продолжение)

Метод	Назначение

	//задаем исходной кнопке имя 81 Button1.Name := 'B1';
	//копируем кнопку в буфер обмена
	Clipboard. SetComponent(Button1);
	//задаем исходной кнопке новое имя B2
	Button1.Name := 'B2';
	//вставляем кнопку с именем B1
	// из буфера обмена на объект GroupBoxl
	Clipboard.GetComponent(Self, GroupBoxl);

function GetTextBuf(Buffer: PChar; BufSize: Integer): Integer; procedure SetTextBuf(Buffer: PChar);	Получение данных из буфера обмена Windows в программный буфер (указатель Buffer) или запись данных из программного буфера Buffer в буфер обмена Windows
procedure HasFormat(Format: Word): Boolean;	Возвращает значение True, если данные в буфере обмена хранятся в формате, описанном параметром Format
procedure Open;	Открытие буфера обмена Windows. Требуется, если данные будут записываться в буфер обмена в несколько приемов

Таблица 4.6. Константы, описывающие некоторые форматы данных для буфера обмена Windows

Значение	Формат
CF_TEXT	Текст, разделенный символами новой строки (стандартный текстовый формат)
CF_BITMAP	Формат точечного изображения Windows
CF_METAFILEPICT	Формат графического метафайла Windows
CF_PICTURE	Объект типа TPicture
CF_COMPONENT	Другой стандартный объект

Виртуальный экран в Delphi 7

С помощью класса TScreen можно определить, какие формы имеются в приложении, узнать, какая форма и какой элемент управления являются активными, получить сведения о доступных шрифтах, указателях мыши. В системе *Delphi 7* имеется глобальная переменная Screen — объект этого класса. Следует обращаться именно к ней, а не создавать экземпляры TScreen самостоятельно.

Основные свойства, методы и события класса TScreen приведены в табл. 4.7-4,9.

Таблица 4.7. Основные свойства класса TScreen

Свойство	Назначение
ActiveControl	Элемент управления, имеющий фокус
ActiveForm	Форма, имеющая фокус. Свойство доступно только для чтения, однако можно изменять значения свойств этой формы. Например, фоновый цвет активной формы можно изменить так: Screen.ActiveForm.Color := clRed;

продолжение

Таблица 4.7. Основные свойства класса TScreen (продолжение)

Свойство	Назначение
Cursor	Текущая форма указателя (тип TCursor), используемого в приложении. Вид указателя можно менять с помощью набора стандартных констант. Например, чтобы указатель принял форму песочных часов, надо использовать следующий оператор: `Screen.Cursor := crHourglass;` Указатель в виде песочных часов задает константа crHourglass, стандартная форма указателя соответствует константе crDefault, при использовании константы crNone указатель невидим
Cursors	Массив указателей, доступных в приложении
FormCount	Число форм программы, отображенных на экране
Forms	Массив видимых форм программы. С его помощью можно менять любые свойства этих форм
DataModuleCount	Число модулей данных (специальных типов форм)
DataModules	Массив модулей данных
DesktopWidth Desktop Height	Ширина и высота Рабочего стола в пикселах
Desktop Left DesktopTop	Левая и верхняя координаты Рабочего стола
Fonts	Массив имен шрифтов (тип TStrings), предназначенных для вывода текста на экран (в этот массив не попадают шрифты, используемые для работы с принтером). Эти имена можно использовать для динамического изменения шрифта конкретного элемента в программе: `Edit1.Font.Name := Screen.Fonts[0];`
Width Height	Ширина и высота экранного пространства в пикселах
HintFont	Шрифт, которым отображается всплывающая подсказка
IconFont	Шрифт, которым выполняются подписи под значками в диалоговом окне выбора файлов
MenuFont	Шрифт, который используется для надписей в пунктах меню
MonitorCount	Число мониторов, используемых для представления Рабочего стола
Monitors	Массив мониторов (класс TMonitor), доступных в системе
PixelsPerInch	Число пикселов на один дюйм экрана монитора (разрешение экрана)

Таблица 4.8. Основные методы класса TScreen

Метод	Назначение
procedure EnableAlign; procedure DisableAlign;	Разрешает или запрещает выравнивание форм по размерам экрана (в зависимости от значения их свойства Align)
procedure Realign;	Переупорядочивает формы на экране в зависимости от свойства Align
procedure ResetFonts;	Обновляет список текущих шрифтов

Таблица 4.9. Основные события класса TScreen

Событие	Условие генерации
OnActiveControlChange	Фокус переместился на новый элемент управления в текущей форме
OnActiveFormChange	Фокус переместился на новое окно (форму) программы

Работа с графикой

Понятие холста

Вся технология вывода графической информации на экран основывается на понятии *холста* (класс TCanvas). Этот класс обладает всеми возможностями для отображения такой информации, и на его основе создано большинство компонентов *Delphi 7,* которые представляют собой элементы управления и должны отрисовываться на экране с использованием средств этого класса.

По умолчанию область холста совпадает с *клиентской областью* формы или элемента управления. Эта область представляет собой ту часть объекта, которая не занята какими-то вспомогательными деталями оформления (например, заголовком окна, строкой меню, панелью командных кнопок, границами объекта), недоступными для вывода на них графической информации.

Класс TCanvas имеет набор стандартных свойств и методов, позволяющих выполнять простейшие графические операции. Работа большинства этих методов основана на понятии *графического курсора* — виртуальной точки, определяющей начало выполнения очередной операции (например, точки, из которой будет рисоваться линия).

Принцип отрисовки изображений в Windows

После того как на холст выведена графическая информация, она отображается в рамках формы до тех пор, пока область холста не закроется другим окном или приложение не будет свернуто, то есть пока область холста не окажется закрытой (возможно, частично). При новом появлении этой области наэкране графические данные, выведенные на нее ранее, не восстановятся, и их придется выводить снова. В какой момент это лучше делать? Когда область холста становится видимой, *Windows* генерирует системное сообщение WM_PAINT, описывающее ту часть формы, которая требует перерисовки. В системе *Delphi 7* такое сообщение обозначается OnPaint. Оно автоматически обрабатывается формами и элементами управления при необходимости их отображения и обычно не требует вмешательства со стороны программиста. Исключением являются те случаи, когда форма используется именно для вывода графической информации: всевозможных графиков, мультимедийных данных и т. п.

Перед тем как перейти к рассмотрению конкретных свойств, остановимся на таком важном классе, как TGraphics.

Класс TGraphics

Данный класс является абстрактным и сам по себе не применяется. На его основе созданы классы, предназначенные для использования в программах конкретных графических объектов (точечное изображение, значок и прочие).

От TGraphics такие объекты наследуют свойства, приведенные в табл. 4.10. Методы класса TGraphics имеют характеристики virtual и abstract и определяются в конкретных классах-наследниках. Они приведены в табл. 4.11.

Таблица 4.10. Наследуемые свойства класса TGraphics

Свойство	Назначение
Width Height	Ширина и высота объекта в пикселах
Modified	Имеет значение True, если объект был изменен (например, отредактирован)
Palette	Идентификатор цветовой палитры Windows
Transparent	Имеет значение True, если объект будет рисоваться в «прозрачном» режиме. Цвет, определяющий уровень прозрачности, задается в конкретном классе

Таблица 4.11. Абстрактные методы класса TGraphics

Метод	Назначение
procedure LoadFromFile(const FileName: string); procedure SaveToFile(const FileName: string);	Загрузка и сохранение графической информации в файле
procedure LoadFromClipboardFormat; procedure SaveToClipboard Format;	Загрузка и сохранение графической информации в буфере обмена Windows
procedure LoadFromStream(Stream: TStream); procedure SaveToStream(Stream: TStream);	Загрузка и сохранение графической информации в потоке

Свойства и методы класса TCanvas

В качестве свойств, в первую очередь, используются классы, описывающие цвет и способ заполнения областей формы, цвет и толщину линий, стиль и размер шрифта и другие. Дополнительные методы предназначены для вывода на экран изображений и рисунков.

Класс Карандаш (TPen). Карандаш — свойство Реп класса TCanvas, определяется основными свойствами, перечисленными в табл. 4.12.

Таблица 4.12. Основные свойства класса TPen

Свойство	Назначение
Color	Цвет карандаша
Mode	Режим рисования. Определяет, в частности, способ комбинирования своего цвета с текущим цветом холста; например, значение pmXor позволяет рисовать линии, которые при повторной отрисовке на том же месте исчезают. Это удобно, когда надо динамически отображать постоянно меняющиеся линии вслед за движущимся указателем
Style	Стиль линии определяет, будет ли она сплошной или пунктирной. Возможные значения: psSolid (сплошная линия); psDash (пунктирная) и другие
Width	Толщина линии в пикселах

Класс Кисть (TBrush). Кисть — свойство Brush класса TCanvas, предназначена для заполнения сплошных областей клиентской части формы в соответствии с заданным шаблоном. Помимо свойств Color и Style, совпадающих с аналогичными свойствами класса TPen, в класс TBrush добавлено новое свойство Bitmap, которое позволяет заполнять область не только сплошным цветом или пунктирными линиями, но и заранее подготовленным точечным изображением.

Класс Шрифт (TFont). Шрифт — свойство Font класса TCanvas, служит оболочкой ресурса *Windows*, определяющего текущий шрифт. Содержит множество стандартных свойств, описывающих характеристики шрифта. Наиболее важные из них приведены в табл. 4.13.

Таблица 4.13. *Основные свойства класса TFont*

Свойство	Назначение
Color	Цвет
Charset	Набор символов, определяемый используемой кодировкой. Например, для шрифта с русскими буквами могут существовать различные наборы: CP-1251, КОИ-8 и другие
Height	Высота шрифта в пикселах. Реально эта высота вычисляется по специальной формуле и может принимать отрицательные значения. Подробнее об этом можно узнать в справочном руководство го программированию в Windows. Вместо данного свойства лучше использовать свойство Size
Name	Название шрифта, под которым он зарегистрирован в Windows, например Times New Roman, Courier и прочие
Pitch	Профиль шрифта, определяющий, будет ли расстояние между символами фиксированным (fpFixed) или переменным (fpVariable), как это имеет место в шрифтах Courier и Times New Roman соответственно. Если для шрифта явно задано значение, не соответствующее реальному профилю, система Windows автоматически подберет шрифт, все символы которого наиболее точно соответствуют указанным параметрам
Size	Высота шрифта в пикселах
Style	Стиль шрифта. Возможно е значения: fsBold (полужирный); fsItalic (курсив); fsUnderline (подчеркнутый); fsStrikeOut (зачеркнутый)

Свойства холста

Выше были перечислены самые важные свойства холста, которые активно используются в процессе вывода графической информации. Однако помимо них необходимо отметить некоторые свойства самого класса TCanvas, приведенные в табл. 4.14. Класс TCanvas содержит большое количество методов. Их краткое описание приведено в табл. 4.15.

Таблица 4.14. *Некоторые свойства класса TCanvas*

Свойство	Назначение
CanvasOrientation	Данное свойство, доступное только для чтения, определяет позицию начала координат. Значение по умолчанию —coLeftToRight (отсчет ведется от левого верхнего угла клиентской области). В некоторых случаях используется значение coRightToLeft (когда в соответствии с национальными требованиями текст пишется справа налево). В этом случае отсчет идет от верхнего правого угла
ClipRect	Область холста, которая реально отрисовывается. Обычно эта область совпадает с клиентской областью, что означает вывод всей графической информации. 8 ряде случаев удается значительно повысить скорость отображения информации благодаря ограничению области вывода небольшим прямоугольником

продолжение

Таблица 4.14. *Некоторые свойства класса TCanvas (продолжение)*

Свойство	Назначение
CopyMode	Режим копирования графического образа на холст. С помощью данного свойства удается создавать самые разные графические и анимационные эффекты путем выполнения операций логического сложения, умножения инвертирования битов исходной и результирующей графических областей. Например, значение cmSrcInvert (логическая операция XOR) активно применяется при перемещении спрайтов
PenPos	Текущая позиция графического курсора (тип TPoint)
Pixels	Двумерный массив, хранящий цвета каждого пиксела изображения. Это свойство очень полезно при поточечной обработке рисунка, однако им нельзя злоупотреблять, так как обработка отдельных пикселов — весьма медленная операция. Пример использования: `Canvas.Pixels[123,50] := clRed;`
TextFlags	Способ вывода текста на холст. Возможные значения: — ETO_CLIPPED (обычный вывод, по умолчанию); ETO_OPAQUE (вывод текста с заливкой фона, что ускоряет процесс вывода, но перекрывает фоновый рисунок)

Таблица 4.15. *Краткое описание методов класса TCanvas*

Метод	**Назначение**
`procedure Arc( X1, Y1, X2, YZ, X3, У3, X4, Y4: Integer);`	Рисование части эллипса
`procedure BrushCopy(const Dest: TRect; Bitmap: TBitmap; const Source: TRect; Color: TColor);`	Копирование заданной части графического изображения на холст. При этом цвет, указанный в качестве параметра, трактуется как прозрачный (не отображаемый на экране)
`procedure Chord( X1, Y1, X2, Уг, X3, Y3, X4, Y4): Integer;`	Рисуется замкнутая фигура, созданная пересечением эллипса и отрезка прямой линии (хорды)
`procedure CopyRect(Dest: TRect; Canvas: TCanvas; Source: TRect);`	Копирование на данный холст области с другого холста. Способ вывода определяется значением свойства CopyMode
`procedure Draw(X, Y: Integer; Graphic: TGraphic);`	Вывод графического изображения в заданной точке холста
`procedure DrawFocusRect( const Rect: TRect);`	Рисуется прямоугольник в стиле, принятом в Windows для отображения элементов, имеющих фокус. Повторный вывод такого прямоугольника в том же месте приводит к его исчезновению (логическая операция XOR)
`procedure Ellipse(const Rect: TRect);`	Рисуется эллипс
`procedure FillRect(const Rect: TRect);`	Рисуется прямоугольник, который заполняется в соответствии со значением свойства Brush
`procedure FloodFill(X, Y: Integer; Color: TColor; FillStyle: TFillStyle);`	Заполнение указанным цветом области холста, прилегающей к точке, заданной в качестве параметра, и имеющей цвет, совпадающий с цветом этой точки
`procedure FrameRect(const Rect; TRect);`	Рисуется прямоугольник заданного размера с толщиной границы в 1 пиксел. Вид линии определяется значением свойства Brush

Таблица 4.15, *Краткое описание методов класса TCanvas (продолжение)*

Метод	Назначение
procedure LineTo(X, Y: Integer);	Рисует линию от точки, определенной свойством PenPos (текущая позиция графического курсора) до точки, указанной в качестве параметра
procedure MoveTo(X, Y: Integer);	Устанавливает текущую позицию графического курсора в точку, заданную в параметрах метода
procedure Pie(X1,Y1, X2,Y2,X3,Y3,X4,Y4: Longint);	Рисуется сектор эллипса, расположенный внутри заданного прямоугольника
procedure PolyBezier(const Points: array of TPoint);	Рисуется кривая Безье — гладкая линия, соединяющая заданные точки. Точки передаются в динамическом массиве, состоящем из элементов типа TPoint. При рисовании очередного фрагмента линии учитываются три последовательные точки
procedure Poly8ezierTo(const Points: array of TPoint);	Метод аналогичен предыдущему, но по окончании отрисовки линии графический курсор перемещается в ее последнюю точку. Значение свойства PenPos, в отличие от метода PolyBezier, когда графический курсор остается на старом месте, обновляется
procedure Polygon(Points: array of TPoint);	Рисуется сложная фигура, состоящая из отрезков, последовательно соединяющих точки, представленные в виде динамического массива элементов TPoint. Последняя точка соединяется с первой. Внутренняя часть фигуры заполняется в соответствии со значением свойства Brush
procedure Polyline(Points: array of TPoint)	Метод аналогичен предыдущему, но заполнения внутренней части фигуры не происходит
procedure Rectangle(const Rect: TRect);	Рисуется прямоугольник. Его внутренняя область заполняется в соответствии со значением свойства Brush
procedure RoundRect(X1, Y1, X2, Y2, X3, Y3: Integer);	Метод аналогичен предыдущему. Рисуется прямоугольник со скругленными углами
procedure StretchDraw(const Rect: TRect; Graphic: TGraphic);	Вывод графического изображения в область, заданную параметром-прямоугольником. Изображение масштабируется в соответствии с размерами этого прямоугольника
function TextExtent(const Text: string): TSize;	Возвращает ширину и высоту в пикселах строки, выведенной текущим шрифтом
function TextHeight(const Text: string): Integer;	Возвращает высоту в пикселах строки, выведенной текущим шрифтом
procedure TextOut(X, Y: Integer; const Text: string);	Вывод строки в конкретную позицию холста
procedure TextRect(Rect: TRect; X, Y: Integer; const Text: string);	Вывод строки в указанную позицию в рамках прямоугольника Rect. Часть строки, не попавшая в указанную область, на холст не выводится
function TextWidth(const Text: string): Integer;	Ширина в пикселах строки, выведенной текущим шрифтом

Наследники класса TGraphics

Класс Точечное изображение (TBitmap)

Это специальный класс, с помощью которого можно хранить, загружать из файла или буфера обмена *Windows*,сохранять в файле или буфере обмена графические точечные изображения в формате битовой карты (расширение .BMP), а также выполнять над ними ряд вспомогательных операций. Использование класса TBitmap неразрывно связано с понятием холста.

Свойства и методы класса TBitmap приведены в табл. 4.16 и 4.17.

Таблица 4.16. *Свойства класса TBitmap*

Свойство	Назначение
Canvas	Область изображения (холст), на которой можно выполнять рисование. Данное свойство используется, когда надо подготовить изображение, нарисовав на нем, например, спрайты в скрытом режиме, а затем быстро отобразить рисунок на экране. Такой подход применяется, в частности, при создании анимационных эффектов
Empty	Имеет значение True, если объект не содержит никакого изображения
PixelFormat	Число битов на пиксел (глубина цвета)

ScanLine	Массив указателей на каждую строку точечного изображения. Число элементов в массиве равно значению свойства Height. С помощью указателя можно получить доступ к конкретному пикселу. Пример использования:
	`var BitMap : TBitMap; P : PbyteArray; ... P := BitMap.ScanLine[y];`
	`for x := 0 to BitMap.Width-1 do`
	`P[x] := Color;`

TransparentColor	Цвет, который будет считаться прозрачным (не отображаемым) при выводе изображения на экран
TransparentMode	Способ определения прозрачного цвета. Цвет определяется по пикселу в левом верхнем углу точечного изображения или задается программно

Таблица 4.17. *Методы класса TBitmap*

Метод	Назначение
`procedure FreeImage;`	Уменьшение объема памяти для хранения точечного изображения путем уменьшения глубины цвета
`procedure LoadFromResourceID( Instance: THandle; ResID: Integer);`	Загрузка изображения из ресурсов программы
`procedure Mask( TransparentColor: TColor);`	Установка конкретного цвета изображения в качестве прозрачного

Класс TBitmap можно использовать для создания несложной спрайтовой мультипликации следующим образом. В память компьютера загружается заранее подготовленное фоновое изображение и набор небольших картинок-спрайтов. Затем исполь-

зуется динамически созданный объект класса TBitmap. Б него сначала копируется фон, а затем на него накладываются отдельные спрайты. Данные заносятся в область объекта, доступную для рисования (свойство Canvas). Можно использовать такие методы холста, как Draw, CopyRect и другие. После этого сформированный рисунок выводится на экран (свойство Bitmap свойства Picture компонента TImage) и становится видимым.

Промежуточный объект класса TBitmap необходим, потому что если выполнять вывод спрайтов сразу на фоновое изображение, то при последующих циклах создания итогового рисунка па этом изображении останутся предыдущие спрайты.

Класс Значок (TIcon)

Этот класс предназначен для работы с изображениями в формате значка *Windows* (расширение файла .ICO). Его свойства и методы не отличаются от свойств и методов класса TBitmap за исключением того, что значок всегда имеет определенный прозрачный цвет, а масштабировать его с помощью метода StretchDraw нельзя.

Класс Метафайл (TMetafile)

В *Windows* имеется специальный тип графических данных, называемый *метафайлом* (файлы с расширениями .EMF и .WMF). Он отличается от точечного изображения тем, что хранит не пикселы, а специальный код, который при выводе интерпретируется как набор команд типа «провести линию из точки А в точку В красным цветом» или «закрасить прямоугольник».

Соответствующий класс имеет несколько специфических свойств.

Таблица 4.18. Свойства класса TMetafile

Свойство	Назначение
Description	Внутренний комментарий
Enhanced	Имеет значение True, если метафайл хранится в формате .EMF
Inch	Число точек на дюйм с учетом системы координат метафайла
MMWidth, MMHeight	Ширина и высота изображения в условных точках (0,01 миллиметра)

Класс Изображение в формате JPG (TJPEGImage)

Данный класс предназначен для работы с изображениями JPEG, представленными в специальном формате, позволяющем компактно хранить большие рисунки.

ВНИМАНИЕ При работе с этим классом использовать свойство Canvas, то есть рисовать на холсте, нельзя. Класс TJPEGImage применяется только для отображения на экране.

Характерные свойства и методы класса TJPEGIrnage приведены в табл. 4.19 и 4.20.

Таблица 4.19. Некоторые свойства класса TJPEGImage

Свойство	Назначение
CompressionQuality	Соотношение между качеством изображения и размером файла, в котором это изображение хранится. Свойство может принимать значения от 1 до 100. Большее значение соответствует худшему качеству, но меньшей величине файла
Grayscale	Имеет значение True, если изображение будет выводиться на экран в серой шкале (255 оттенков), что существенно повышает скорость распаковки рисунка
Performance	Более высокое качество (jpBestQuality) или более высокая скорость распаковки (jpBestSpeed) при считывании изображения из файла
PixelFormat	Указывает формат рисунка jpeg (8-битный или 24-битный)
ProgressiveDisplay	Имеет значение True, если выводимый рисунок будет показываться на экране постепенно, по частям
Progressive En coding	Имеет значение True, если разрешается выводить рисунок по частям
Scale	Масштаб изображения. Возможные значения — jsFullSize (полный размер), jsHalf (в половину размера), jsQuarter (в четверть размера), jsEighth (в 1/8 размера)
Smoothing	Имеет значение True, если во время вывода изображения по частям оно будет постепенно повышать свою четкость. В противном случае изображение будет выводиться небольшими порциями построчно

Таблица 4.20. Некоторые методы класса TjPEGImage

Метод	Назначение
procedure Compress;	Сжатие изображения в соответствии со значением *свойства Compression Quality*
procedure DIBNeeded;	Распаковка изображения JPEG в битовый формат BMP
procedure JPEGNeeded;	Создание изображения JPEG на основе внутреннего битового формата BMP

Компонент Область рисования (TPaintBox)

Этот компонент расположен на панели System. Он не имеет никаких отличительных особенностей и обычно используется для выделения на форме нескольких областей рисования. Такой подход удобен, когда в программе происходит активный вывод графической информации на экран и желательно разделить этот процесс на независимые части.

Компонент TPaintBox может охватывать произвольную прямоугольную область формы. Он содержит единственное главное свойство Canvas, имеющее собственную систему координат. Единственное обрабатываемое событие OnPaint генерируется системой *Windows* автоматически при необходимости перерисовать одну или несколько областей (или их части). Разработчик должен только определить, что будет изображено в каждом объекте класса TPaintBox.

Этот компонент также может самостоятельно вызывать процесс перерисовки области холста с помощью метода Paint.

Работа с файлами

Способы работы с файлами в системе Delphi 7

При работе с файлами в системе *Delphi 7* возможны два принципиально разных подхода. Первый состоит в использовании стандартных подпрограмм (они имелись еще в классической версии Паскаля тридцатилетней давности), позволяющих записывать содержимое переменных в файлы и считывать их обратно из файлов в переменные. К этим средствам добавились также библиотеки стандартных функций по работе с файлами, основанные на системных функциях *Windows*.

В связи с появлением версии *Object Pascal* и реализации понятия класса в языке появились средства объектной работы с данными. Это второй подход к работе с файлами в системе *Delphi 7*. В свою очередь в рамках каждого из подходов применяются также существенно различающиеся приемы. Например, при классическом подходе в работе с файлами могут использоваться прямые обращения к функциям *Windows* или обращения к функциям *BIOS*.

Общая технология работы с файлами в Delphi 7

Несмотря на все различия, независимо от используемого подхода технология работы с файлами в системе *Delphi 7* требует определенного порядка действий.

1. Прежде всего файл должен быть *открыт*. Это означает, что операционная система «дает добро» на внесение изменений в данный файл (например, на запись данных) и следит, чтобы обращения других пользователей и программ к этому файлу (если компьютер подключен к сети) выполнялись корректно. Так, считывание данных из файла, в который другой пользователь в этот момент вносит изменения, невозможно.

   При открытии файла системе управления файлами обычно сообщается, в каком *режиме* файл будет открыт: планируется ли вносить изменения в его содержимое или же файл открывается только для считывания из него данных. В последнем случае к файлу, как правило, могут обращаться и другие пользователи. Обычно указывается также, какова внутренняя структура открываемого файла — это требуется, чтобы выполнять операции с ним максимально быстро.

   После того как файл успешно открыт, в программу возвращается его идентификатор — переменная, которая будет использоваться для идентификации этого файла во всех процедурах обработки.

2. Начинается работас файлом. Это может быть считывание из него данных, запись, поиск и другие операции.

3. Файл *закрывается*. Это означает, что он снова доступен другим приложениям без ограничений. Кроме того, закрытие файла гарантирует, что все внесенные в него изменения не пропадут, потому что для повышения скорости работы результаты промежуточных действий обычно сохраняются в специальных буферах операционной системы.

Стандартный подход к работе с файлами в системе Delphi 7

Типы файлов

В системе *Delphi 7 имеется* стандартный тип **File** (**ключевое** слово), на основе которого можно создавать новые файловые типы для работы со структурированными данными. Если переменная описана так:

```
var F: File;
```

то она считается *нетипизированной* файловой переменной, позволяющей работать с файлами на низком уровне (структура файла неизвестна). При этом данные будут считываться и записываться блоками по 128 байт (значение по умолчанию). Размер блока можно изменить в момент открытия файла. Рекомендуется назначать этот размер равным 1 байту, чтобы корректно обрабатывать файлы любой структуры.

Чаще всего в программах используются файлы, состоящие из последовательности одинаковых записей. Для работы с ними применяется следующая форма описания.

```
var имя-переменной: File of тип;
```

Б качестве типа файла должен быть указан тип, для которого точно известен фиксированный размер в байтах. К таковым относятся все базовые типы (за исключением типа **String,** если для него явно не задан размер), структуры, статические массивы и прочие, например:

```
type TMyFile = record
   Name: string[20];
   Number: integer;
   end;
var F: File of TMyFile;
```

ВНИМАНИЕ Файловые типы нельзя использовать в качестве элементов массивов и полей структур.

Открытие файлов

Перед тем как выполнить физическое открытие файла, программе надо сообщить, где он расположен. Для этого файловая переменная должна быть связана с именем файла с помощью процедуры AssignFile. Первый параметр этой процедуры — имя переменной, а второй — строка, содержащая название файла. Если полный путь поиска не указан, то файл будет разыскиваться *в* текущем каталоге.

```
AssignFile( F, 'test.dat');
AssignFile( F, 'c:\projects\test.dat' ];
```

Процедур открытия файлов в Паскале две. Первая из них, Rewrite, используется для открытия файла в режиме записи (при этом происходит полное уничтожение его содержимого, а размер файла становится равным нулю), а вторая, Reset, — для

открытия файла в режиме чтения (при этом вносить изменения в содержимое файла не разрешается). Процедура Rewrite может также применяться для создания нового файла.

Каждая из этих процедур может иметь второй необязательный параметр, который определяет длину записи нетипизированного файла в байтах.

```
var F: File;
...
Rewrite( F, 1 );
```

ЗАМЕЧАНИЕ В программе одновременно может быть открыто большое число файлов (до нескольких сотен). Конкретное значение определяется настройками Windows.

Запись в файл

Для записи данных в файл, имеющий определенную структуру (описанный с помощью ключевых слов **File of** ...), применяется процедура Write. В качестве первого параметра указывается файловая переменная, а далее следует список переменных типа, соответствующего типу файла.

```
var F: File of TMyFile;
   A, MyData, N5: TMyFile;
...
Rewrite( F );
Write( F, N5, MyData );
```

Значения из этих переменных будут последовательно записаны в конец файла.

Считывание из файла

Чтобы считать данные из файла, надо использовать процедуру Read. Она записывается аналогично процедуре Write.

```
var F: File of TMyFile;
   A, MyData, N5: TMyFile;
...
Reset( F );
Read( F, MyData, A );
```

Начиная с текущей позиции в файле F (исходно это начало файла) из него будут последовательно считаны блоки данных, соответствующие размерам экземпляра класса TMyFile, и записаны в переменные MyData и A. По окончании операции ввода текущая позиция в файле F сдвинется на два элемента.

Специальная процедура

```
procedure Truncate(var F);
```

позволяет отсечь (удалить) все содержимое файла, начиная с текущей позиции до его конца.

Закрытие файла

По завершении работы с файлом его надо закрыть. Это выполняется вызовом процедуры CloseFile:

```
CloseFile( F );
```

Работа с нетипизированными файлами

Так как при работе с нетипизированными файлами данные считываются и записываются побайтно, допускается обрабатывать такие файлы, как последовательность байтов, не имеющую строгой внутренней структуры. Для этого применяют процедуры блочного ввода/вывода BlockRead и BlockWrite. Они имеют одинаковый список параметров и отличаются только названиями.

```
procedure BlockWrite(var f: File; var Buf;
   Count: Integer; var AmtTransferred: Integer);
```

Параметр Buf — это произвольная переменная, параметр Count — число блоков считываемой или записываемой информации. Если при открытии файла размер блока не был указан явно, считается, что он равен 128 байтам. Однако нетипизированные файлы применяют, как правило, для побайтной обработки данных, поэтому длину блока обычно задают равной 1 байту и в параметр Count записывают просто число байтов. Параметр AmtTransferred — необязательный. По окончании выполнения процедуры в нем будет храниться число реально считанных или записанных блоков.

Установка новой позиции в файле

Считывание и запись информации в файл происходит последовательно, блок за блоком. Если же, например, требуется считать пятую запись из файла, в котором ранее было сохранено 10 записей, или изменить ее, не перезаписывая все остальные, то можно воспользоваться процедурой Seek.

```
procedure Seek(var F; N: Longint);
```

Первый параметр — файловая переменная (определенного типа или нетипизированная), второй параметр — номер записи в файле, начиная с которой будет выполнена следующая операция ввода/вывода. Этот номер обычно называется *позицией в файле*. Нумерация записей в файле начинается с нуля. В случае нетипизированного файла второй параметр определяет номер байта, с которого начнется запись или чтение информации.

С помощью этой процедуры можно легко выполнять редактирование содержимого любых файлов. Рассмотрим примеры создания и редактирования нетипизированных файлов и файлов, состоящих из набора записей.

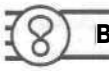
```
var f: File;
   b: Byte;
   i: Integer;
begin
// создание нетипизированного файла:
AssignFile(F,'C:\A.DAT');
Rewrite(F,1);
```

```
b := $41;
for i := 1 to 100 do
   BlockWrite(F, b, 1);
CloseFile(F);
```

В результате выполнения данного участка кода на диске C: (в корневом каталоге) будет создан файл A.DAT, содержащий 100 символов 'A' (значение байта $41 в шестнадцатеричном формате соответствует символу 'A').

Следующий текст изменит одиннадцатый байт в этом файле, который получит новое значение $42 (символ 'B'). Хотя в процедуре Seek указано смещение 10, изменен будет именно одиннадцатый байт, потому что отсчет блоков внутри файла ведется с нуля.

```
AssignFile(F,'C:\A.DAT');
b := $42;
Reset(F,1);
Seek(F,10);
BlockWrite(F,b,1),-
CloseFile(F);
```



ВНИМАНИЕ Для редактирования содержимого файла с использованием процедуры Seek его надо открывать с помощью процедуры Reset (которая обычно используется для открытия файла в режиме «только для чтения»), а не с помощью процедуры Rewrite, которая уничтожает всю информацию в файле.

Аналогичный пример демонстрирует формирование файла, хранящего набор записей сложного типа, и редактирование одной из записей.

```
type TF = record X,Y: integer end;
var F: Pile of TF;
   i: Integer;
   FF: TF;
begin
FF.X := $42; FF.Y := $42;
AssignFile(F,'C:\A.DAT');
Rewrite(F);
for i := 1 to 100 do
   Write(F,FF),-
   CloseFile(F);
```

Модификация четвертой записи файла выполняется следующим образом.

```
FF.X := $45; FF.Y := $45;
AssignFile(F,'C:\A.DAT'),-
Reset(F);
Seek(F,3),-
Write(F, FF);
CloseFile(F);
```

Текстовые файлы

В Паскале имеется еще один тип файлов, занимающий промежуточное положение между типизированными и нетипизированными файлами. Он называется Text и предназначен исключительно для обработки строк, которые, с одной стороны, описываются базовым типом String, ас другой стороны, не имеют фиксированной длины. В таких файлах считывание и запись происходят построчно, причем символы перевода строки и возврата каретки используются как управляющие. Для этих файлов дополнительно реализованы две процедуры, явно осуществляющие ввод/вывод с новой строки: ReadLn и WriteLn. При этом размер считанной строки определяется автоматически, по наличию управляющих символов, которые в строку не записываются. Если применять процедуры Read и Write без элемента Ln, означающего переход на новую строку, то считывание и запись текста производятся сплошным потоком, без разделения на строки.

Специально для работы с текстовыми файлами в системе *Delphi 7* имеется набор стандартных подпрограмм, приведенных ниже.

Таблица 4.21. *Стандартные* ***подпрограммы*** *для работы с текстовыми* ***файлами***

Подпрограмма	Назначение
procedure Append(var F: Text);	Открытие текстового файла в режиме записи. Отличается от процедуры Rewrite тем, что не стирает все содержимое, а устанавливает текущую позицию в самый конец файла, что позволяет добавлять информацию
procedure AssignPrn(var F: Text);	Вся информация, записываемая в файл, перенаправляется на принтер. Файл должен быть открыт с помощью процедуры Rewrite
function Eoln(var F: Text): Boolean;	Возвращает значение True, если текущая позиция в файле расположена либо в конце файла, либо в конце строки. Такая проверка может понадобиться, если ввод выполняется с помощью процедуры Read, не переходящей к началу следующей строки автоматически
procedure Erase(var F: Text);	Удаление файла. Он должен быть определен с помощью процедуры AssignFile, но не должен быть открыт
procedure Flush(var F: Text);	Информация, которая была записана в файл из программы, но находится во временном буфере, физически перемещается в файл на диске
function SeekEof(var F: Text): Boolean;	Возвращает значение True, если текущая позиция расположена в конце файла
function SeekEotn(var F: Text): Boolean;	Возвращает значение True, если текущая позиция расположена е конце строки
procedure SetTextBuf(var F: Text; var Buf; Size: Integer);	Устанавливает размер буфера для операций ввода/вывода с текстовыми файлами (параметр Size). Этот буфер располагается внутри программы. Он указывается в качестве второго параметра. Это может быть, например, массивсимволов

Конец файла

В процессе обработки файла часто возникает потребность обнаруживать его конец. Например, когда файл необходимо прочитать от начала до конца при поиске нужной записи. Для подобного контроля в Паскале имеется функция Eof, единственный параметр которой — файловая переменная. Эта функция возвращает значение True, если после выполнения последней операции ввода/вывода текущая позиция оказывается в самом конце файла и дальнейшее считывание данных невозможно. Как правило, данная **функция** применяется при вводе, потому что запись в файл, открытый с помощью процедуры Rewrite, выполняется последовательно и текущая позиция всегда расположена в конце файла.

ВНИМАНИЕ При использовании функции Eof надо помнить, что если она вернула значение True, то попытка выполнить считывание из файла приведет к ошибке.

В следующем примере выполняется подсчёт числа строк в текстовом файле.

```
AssignFile(F,'C:\A.TXT');
Reset(F);
i := 0;
while not Eof(F) do
   begin
   ReadLn(F,S);
   inc(i);
   end;
CloseFile(F);
```

Значение переменной i будет увеличиваться при каждой операции ввода строки.

Контроль ошибок

Большинство упомянутых подпрограмм было реализовано еще в самых ранних версиях Паскаля корпорации *Borland,* когда в языке не существовало понятия «объект», а механизм обработки исключительных ситуаций еще не был создай, Поэтому для контроля правильности выполнения операций ввода/вывода использовалась другая техника программирования. Когда в диалоговом окне свойств компилятора, открываемом командой Project v Options (Проект ➤ Параметры), включен флажок I/O Checking (Проверка ввода/вывода), при возникновении подобных ошибок генерируется исключительная ситуация EInOutError. Ее можно обрабатывать, а можно временно отключать подобный автоматический контроль непосредственно в тексте программы с помощью управляющих директив компилятора. Например, в следующем тексте контроль ошибок ввода/вывода отключается в момент открытия файла.

```
AssignFile(F,'C:\A.TXT');
{$I-}
Reset(F);
{$I+}
```

Директива $I управляет процессом контроля ошибок. В случае невозможности открытия указанного файла исключительная ситуация не возникнет, а проверить

результат последней файловой операции можно, обратившись к стандартной функции IOResult. Она возвращает число 0, если ошибок не было, и ненулевой код ошибки в противном случае:

```
if IOResult <> 0 then
  ShowMessage('Ошибка ввода/вывода');
```

Другие стандартные подпрограммы для работы с типизированными файлами и каталогами

В табл. 4.22 приведен большой список подпрограмм для работы с файлами и каталогами. Эти подпрограммы хранятся в двух модулях: System и SysUtils — и в некоторых областях дублируют друг друга. Это связано с необходимостью поддерживать совместимость со старыми версиями системы *Delphi* и языка Паскаль, а также с появлением новых мощных и гибких средств, более полно отвечающих потребностям разработчиков. В частности, функции модуля SysUtils в большинстве своем позволяют контролировать результат работы по возвращаемому значению (если это True, то операция выполнена успешно).

Таблица 4.22. *Стандартные подпрограммы для работы с файлами и каталогами*

Подпрограмма	Назначение
procedure ChDir(S: string); function SetCurrentDir(const Dir: string): Boolean;	Устанавливает текущий каталог в соответствии с путем поиска, заданным параметром 5 (Dir)
function GetCurrentDir: string;	Возвращает имя текущего каталога
function CreateDir(const Dir: string): Boolean; procedure MkDir(S: string);	Создает новый каталог, путь поиска которого указан в качестве параметра. Все промежуточные каталоги (кроме самого последнего) должны существовать
function DeleteFile{ const FileName: string): Boolean;	Удаляет файл
function DirectoryExists(Name: string): Boolean;	Проверяет, существует ли указанный каталог
function DiskFree(Drive: Byte): Int64;	Определяет число свободных байтов на диске, который задан параметром Drive. Значение 0 соответствует текущему диску, 1 — диску A, 2 — диску B, 3 — диску C и так далее
function DiskSize(Drive: Byte): Int64;	Возвращает объем указанного диска в байтах
function FileExists(const FileName: string): Boolean;	Проверяет, существует ли указанный файл
function FileGetAttr(const FileName: string): Integer; function FileSetAttr(const FileName: string; Attr: Integer): Integer;	Получает или устанавливает атрибуты указанного файла. Набор атрибутов представляет собой число типа Integer, разные биты которого определяют разные характеристики файла. faReadOnly — только для чтения; faHidden — скрытый; faSysFile — системный; faVolumeID — метка диска; faDirectory — каталог; faArchive — архивный; faAnyFile — произвольный
function FilePos(var F): Longint;	Возвращает номер текущей записи в файле. Нумерация начинается с нуля

Таблица 4.22. Стандартные подпрограммы для работы с файлами и каталогами (продолжение)

Подпрограмма	Назначение
function FileSize(var F): Integer;	Возвращает число записей в файле или его размер в байтах
function ForceDirectories(Dir: string): Boolean;	Создает каталог. Если указаны несуществующие промежуточные каталоги, то они также будут созданы
procedure GetDir(D: Byte; var S: string);	Для диска, заданного параметром D (0 — A, 1 — C, 2 — D и так далее), записывает в переменную S полный путь поиска текущего каталога этого диска
procedure RmDir(S: string); function RemoveDir(const Dir: string): Boolean;	Удаляет пустой каталог, полный путь поиска для которого задан параметром S (Dir)
procedure Rename(var F; Newname: string); function RenameFile(const OldName, NewName: string): Boolean;	Переименовывает файл, ассоциированный с переменной F при помощи процедуры AssignFile. Файл получает имя, определяемое строковым параметром Newname

Обработка имен файлов

Во многих случаях разработчику приходится анализировать структуру полного имени файла вместе с его путем поиска. Это требуется, например, при создании всевозможных программ установки (инсталляции), поиска информации, составления каталогов ресурсов, при решении системных задач. В системе *Delphi 7* имеется богатый набор подпрограмм, позволяющих автоматизировать большинство аспектов подобного анализа. Эти подпрограммы входят в стандартный модуль Fi(eCtr[и приведены в табл. 4.23.

Таблица 4.23. Стандартные подпрограммы для работы с именами файлов

Подпрограмма	Назначение
function ChangeFileExt(const FileName, Extension: string): string;	Изменяет расширение имени файла FileName на новое, заданное параметром Extension
function ExcludeTrailingBackslash(const S: string): string;	Удаляет завершающий символ \ если он присутствует в строке
function ExpandFileName(const FileName: string): string;	Формирует полный путь поиска для файла FileName
function ExpandUNCFileName(const FileName: string): string;	Формирует полный путь поиска файла в формате UNC (\\имя-сервера\имя-ресурса)
function ExtractFileDir(const FileName: string): string;	Выделяет имя каталога и диска в формате, пригодном для непосредственного использования в функциях CreateDir, GetCurrentDir, RemoveDir и SetCurrentDir
function ExtractFileDrive(const FileName: string): string;	Выделяет имя локального диска или сетевого устройства

продолжение

Таблица 4.23. Стандартные подпрограммы для работы с именами файлов (продолжение)

Подпрограмма	Назначение
function ExtractFileExt(const FileName: string): string;	Выделяет расширение имени файла
function ExtractFileName(const FileName: string): string;	Выделяет имя файла, отделяя его от пути поиска
function ExtractFilePath(const FileName: string): string;	Выделяет путь поиска (с завершающим символом \) из полного имени файла
function ExtractRelativePath(const BaseName, DestName: string): string;	Преобразует полный путь поиска файла (параметр DestName) в строку, которая указывает относительный путь поиска из каталога, заданного параметром BaseName. Элементы пути поиска могут включать промежуточные переходы на уровень вверх, например, ..\
function ExtractShortPathName(const FileName: string): string;	Преобразует путь поиска в формат коротких имен (не более 8 символов на имя файла или каталога). Например, путь поиска C:\Program Files\A.TXT будет преобразован как C:\Progra~1\A.TXT
function IncludeTrailingBackslash(const S: string): string;	Добавляет в конец строки символ \, если он там отсутствует
function IsPathDelimiter(const S: string; Index: Integer): Boolean;	Определяет, находится ли в позиции строки, определяемой параметром Index, символ \
function MatchesMask(const Filename, Mask: string): Boolean;	Возвращает значение True, если файл Filename соответствует строке Mask, в которой задана маска имен файлов, содержащая подстановочные символы
procedure ProcessPath(const EditText: string; var Drive: Char; var DirPart: string; var FilePart: string);	Выделяет имя диска (один символ), путь поиска и имя файла из параметра EditText

Поиск файлов

Процесс поиска файлов выполняется в системе *Delphi 7* в три этапа.

1. Сначала находится первый файл, удовлетворяющий заданной маске. Этот поиск осуществляется с помощью функции

```
function FindFirst (const Path: string; Attr: Integer;
  var F: TSearchRec): Integer;
```

Параметр Path содержит путь доступа для каталога, в котором производится поиск. Путь доступа должен завершаться маской имен файлов, например:

```
C:\TMP\*.*
*.txt
```

Кроме маски можно указать и набор атрибутов, учитываемых при отборе файлов. Эти атрибуты были указаны при описании функций FileGetAttr/FileSetAttr.

Результат поиска сохраняется в переменной F, имеющей тип TSearchRec.

```
type TSearchRec =
   record
   Time: Integer;
   Size: Integer;
   Attr: Integer;
   Name: TFileName;
   ExcludeAttr: Integer;
   FindHandle: THandle;
   FindData: TWin32FindData;
   end;
```

Среди ее полей надо отметить следующие:

- поле Time содержит время создания файла в формате *DOS;*
- поле Size содержит размер файла в байтах;
- поле Name содержит имя файла.

2. Вызывается функция

```
function FindNext(var F: TSearchRec): Integer;
```

Переменная типа TSearchRec, использованная в функции FindFirst, Передается в качестве параметра. На основании записанной в нее информации будет продолжен поиск следующего подходящего файла.

3. Процесс поиска завершается вызовом процедуры

```
procedure FindClose(var F: TSearchRec);
```

Эта процедура освобождает память, которая была выделена для проведения процесса поиска.

Работа с файловой системой Windows

В предыдущем разделе были рассмотрены способы работы с файлами, которые имелись еще в самых первых версиях Паскаля. Хотя эти способы иногда считаются устаревшими, их можно весьма эффективно использовать при создании программ.

При создании программ для ***Windows*** более корректно применять стандартные функции этой системы. Для работы с ними не требуется создавать специальные переменные файлового типа: в системе ***Windows*** каждый файл имеет уникальный цифровой идентификатор (тип Integer). По-английски он называется Handle, и подтаким названием присутствует в описании многих функций.

ВНИМАНИЕ Ни в коем случае нельзя смешивать работу с файлами, открытыми с помощью выше описанных подпрограмм Reset и Rewrite (для их обработки применяются процедуры Read, Write, BlockRead, BlockWrite), и работу с файлами, открытыми с помощью функций Windows.

Файл создается с помощью функции

```
function FileCreate(const FileName: string): Integer;
```

В *Windows* режимы создания и открытия файла различаются.

Функция возвращает идентификатор файла (целое положительное число) или значение -1, если создать файл не удалось. Значение –1 обозначает ошибку для большинства функций *Windows*. Параметр FileName содержит имя файла, возможно, вместе с полным путем поиска.

Открытие файла выполняется функцией

```
function FileOpen(const FileName: string;
   Mode: LongWord): Integer;
```

Режим открытия определяется параметром Mode. Чаще всего применяется одно из трех следующих значений:

- fmOpenRead — открытие только на запись;
- fmOpenWrite — открытие только на чтение;
- fmOpenReadWrite — открытие и на запись, и на чтение (режим изменения файла).

Функции ввода и вывода данных напоминают функции BlockRead/BlockWrite. Чтение данных:

```
function FileRead(Handle: Integer;
   var Buffer;
   Count: Integer): Integer;
```

Параметр Count указывает число считываемых байтов, параметр Buffer — это переменная, в которую эти байты записываются. Как правило, в качестве такой переменной выступает массив из элементов типа Char или Byte. Аналогично выглядит и функция записи данных:

```
function FileWrite(Handle: Integer;
   const Buffer; Count: Integer): Integer;
```

Данные для записи содержатся в параметре Buffer. Параметр Count задает число записываемых байтов. Закрытие файла происходит с помощью процедуры

```
procedure FileClose(Handle: Integer);
```

Следующий пример создает новый файл и записывает в него содержимое текстовой строки.

```
var FileHandle: Integer;
   S: String;
begin
S := 'текстовая строка';
FileHandle ;= FileCreate('C:\TMP\S.TXT');
FileWrite(FileHandle, S, SizeOf(S));
FileClose(FileHandle);
end;
```

Так как размер строки S заранее обычно не известен, следует использовать функцию SizeOf, возвращающую размер объекта-параметра в байтах.

Имеется ряд дополнительных файловых функций *Windows*, предоставляющих расширенные возможности для работы. Для поиска файла можно использовать функцию

```
function FileSearch(const Name, DirList: string): string;
```

Параметр Name содержит имя искомого файла, параметр DirList — список каталогов, в которых будет вестись поиск. Каталоги отделяются друг от друга точкой с запятой. Функция возвращает имя файла вместе с полным путем поиска или пустую строку, если файл не найден.

```
S := FileSearch('MyDoc.doc',
   'C:\Мои документы;D:\Info\Texts');
If S <> '' than ShowMessage(S);
```

Данный код выполняет поиск файла MyDoc.doc в каталогах C:\Мои документы и D:\Info\Texts. Если файл найден, то выводится сообщение, содержащее имя этого файла вместе с полным путем поиска.

Выше рассказывалось о возможности произвольной установки текущей позиции в файле с помощью процедуры Seek. При использовании средств *Windows* применяется функция

```
function FileSeek(Handle, Offset, Origin: Integer): Integer;
```

Параметр Offset задает сдвиг в байтах относительно позиции, определяемой параметром Origin. **Он** может принимать одно из следующих значений.

Таблица 4.24, *Значения параметра Offset*

Значение параметра	Способ отсчета
0	От начала файла
1	От текущей позиции
2	От конца файла

Объектный подход к работе с файлами

После того как в Паскале появилось понятие объекта, на его основе был создан ряд новых типов, позволивших абстрагироваться от конкретного понятия «файл». С помощью объектного подхода можно одинаково работать с любым внешним хранилищем данных (дисковые или ленточные накопители, различные типы памяти и т. п.). Способ хранения данных перестает играть роль.

Эти новые типы основаны на базовом классе TStream (поток). Базовый класс имеет набор виртуальных методов записи и считывания информации и установки конкретной позиции внутри набора данных.

В данном разделе будет рассмотрен класс TFileStream, предназначенный для работы с файлами на жестком диске.

Класс TFileStream

Этот класс является наследником класса THandleStream, берущего свое начало непосредственно от класса TStream. На нем основываются классы, предназначенные для считывания и записи информации при работе с объектами (всевозможными коммуникационными ресурсами). Обращаться к этим объектам можно через уникальный идентификатор *Windows*— handle.

ЗАМЕЧАНИЕ Ток как при объектном подходе файл представляется простой последовательностью (потоком) байтов, то он может обрабатываться только как двоичный.

Процесс работы с файлом, представленным в программе в виде объекта, аналогичен обычному процессу работы с файлом. Сначала файл открывается (для объекта эту функцию выполняет конструктор), затем происходит запись или считывание данных, в заключение файл закрывается (вызывается деструктор).

Конструктор имеет два параметра: имя открываемого файла и режим открытия,

```
constructor Create(const FileName: string; Mode: Word);
```

Параметр Mode принимает одно из следующих значений.

Таблица 4.25. *Значения параметра Mode*

Значение	Способ открытия файла
fmCreate	Создается новый файл. Если указанный файл уже существует, он открывается в режиме записи
fmOpenRead	Для чтения
fmOpenWrite	Для записи
fmOpenReadWrite	Для чтения и записи

Если открыть файл не удается, возникает исключительная ситуация. Чтобы закрыть файл, надо вызвать метод Free. Для считывания данных из потока применяется метод

```
function Read(var Buffer; Count: Longinti : Longint;
```

В переменную Buffer, начиная с текущей позиции в файле, записывается число байтов, указанное в параметре Count. Функция возвращает реально считанное число байтов. Запись данных осуществляется с помощью метода

```
function Write(const Buffer; Count: Longint): Longint;
```

Число байтов, указанное в параметре Count, записывается из переменной Buffer в текущую позицию в файле. Функция возвращает реально записанное число байтов. Метод

```
function Seek(Offset: Longint; Origin: Word): Longint;
```

дает возможность установить текущую позицию в файле в зависимости от параметра Offset (число байтов, на которое перемещается эта позиция) и параметра Origin, который может принимать одно из следующих значений.

Таблица 4.26. Значения параметра Origin

Значение параметра	Способ отсчета
soFromBeginning	От начала файла
soFromCurrent	От текущей позиции в файле
soFromEnd	От конца файла (значение параметра Offset должно быть меньше или равно 0)

Собственных свойств у класса TFileStream нет. Он наследует два свойства класса THandleStream: сам идентификатор Handle и свойство Size, определяющее длину файла в байтах. Еще одно свойство унаследовано от класса TStream — это текущая позиция файлового указателя, отсчитанная в байтах от начала файла (свойство Position).

В следующем примере демонстрируется открытие файла в виде потока, выполняется установка текущей позиции на его середину, считывается 50 байтов, которые выводятся в поле Edit1, после чего файл закрывается.

```
var Stream: TFileStream;
Buf: array[0..50] of char;
begin
try
Stream := TFileStream.Create('c:\a.dat',fmOpenReadWrite);
Stream.Seek(Stream.Size div 2,soFromBeginning);
Stream.Read(Buf,50);
Edit1.Text := StrPas(Buf);
finally
Stream.Free;
end;
end;
```

Класс TFileStream наследует также ряд методов вышестоящих классов. От класса THandleStream унаследован метод

```
procedure SetSize(NewSize: Longint);
```

который позволяет изменить текущий размер файла новым. Как правило, новый размер меньше текущего, то есть происходит усечение файла.

Для копирования файлов удобно применять метод

```
function CopyFrom(Source: TStream; Count: Longint):
Longint;
```

базового класса TStream. Первый параметр — поток-источник, из которого берется число байтов, заданное параметром Count, и копируется в текущий объект. При этом автоматически выполняются операции по временной буферизации данных.

Стандартные диалоговые окна Windows

Панель Dialogs

На панели Dialogs расположен ряд невизуальных компонентов, позволяющих использовать в программе стандартные диалоговые окна *Windows*, например окна выбора и сохранения файлов или изображений, окна выбора цвета и шрифта, окно настройки принтера и другие.

ВНИМАНИЕ Эти компоненты не предназначены для выполнения конкретных действий: загрузки файла, печати, изменения текущего шрифта и прочих. Они применяются только для получения от пользователя желаемых значений настроек, например ввода полного имени файла вместе с путем поиска, указания гарнитуры шрифта, задания числа печатаемых страниц.

Все эти компоненты являются наследниками класса TCommonDialog. Самый важный его метод — это функция

```
function Execute: Boolean;
```

Она выполняет открытие соответствующего окна и возвращает значение True, если пользователь щелкнул на кнопке OK. Реальные поля ввода и заголовки определяются в конкретных компонентах. Когда диалоговое окно открывается в первый раз, возникает событие OnShow, а при закрытии окна — событие OnClose.

Компонент Окно выбора файла (TOpenDialog)

Компонент предназначен для выбора файла с целью последующего открытия. Свойства и события класса TOpenDialog приведены в табл. 4.27 и 4.28.

Таблица 4.27. *Свойства класса TOpenDialog*

Свойство	Назначение
DefaultExt	Расширение имени, используемое по умолчанию. Добавляется в конец выбранного пользователем имени файла, если расширение не указано явно
FileName	Выбранное пользователем имя файла вместе с полным путем поиска
Files	Список выбранных имен файлов. В свойстве Options должен быть включен флажок ofAllowMultiSelect
Filter	Набор масок, в соответствии с которыми отбираются имена файлов для отображения в диалоговом окне. Каждая маска состоит из двух частей: названия и шаблона, — разделенных символом . Одному названию могут соответствовать несколько шаблонов. Маски отделяются друг от друга символом |
FilterIndex	Номер текущей маски. Нумерация начинается с 1
HistoryList	Список ранее выбранных файлов (тип TStrings)

Таблица 4.27. Свойство класса TOpenDialog (продолжение)

Свойство	Назначение
InitialDir	Текущий каталог, содержимое которого отображается при первом открытии диалогового окна
Options	Набор флажков, определяющих работу окна выбора файлов
Title	Заголовок диалогового окна

Среди методов этого класса следует отметить функцию

```
function GetStaticRect: TRect;
```

Она возвращает координаты прямоугольной области диалогового окна (часть клиентской области), зарезервированной для нужд разработчика (например, для отображения содержимого текущего выбранного файла).

Таблица 4.28. События класса TOpenDialog

Событие	Условие генерации
OnCanClose	Пользователь пытается закрыть диалоговое окно. Обработчик этого события позволяет проконтролировать правильность выбранного или введенного в соответствующее поле окна имени файла и разрешить или запретить закрытие
OnFolderChange	Пользователь переключился в другой каталог
OnIncludeItem	К текущему списку файлов в диалоговом окне будет добавлено новое имя. Обработчик данного события дает возможность отбирать допустимые имена по алгоритму, определяемому программистом
OnSelectionChange	Пользователь выбрал новое имя файла в диалоговом окне
OnTypeChange	Пользователь выбрал новую маску файлов (свойство Filter)

В следующем примере при щелчке на кнопке отображается диалоговое окно выбора имени файла (объект OpenDialog1), которое имеет заголовок Выбор нужного файла, а в списке отображаются все файлы, имеющие расширение .PAS. Это обеспечивается присвоением свойству FilterIndex значения 2.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
OpenDialog1.Filter := 'Все файлы (*.*)|*.*|Файлы Паскаля
(*.pas)|*.PAS';
OpenDialog1.Title := 'Выбор нужного файла';
OpenDialog1.FilterIndex := 2;
if OpenDialog1.Execute then
   begin
   AssignFile(F, OpenDialog1.FileName);
   // работа с файлом F
   end;
end;
```

Компонент Окно сохранения файла (TSaveDialog)

Этот компонент практически ничем не отличается от компонента TOpenDialog за исключением некоторых настроек, специфичных для процесса сохранения файла.

Компоненты Окно открытия рисунка (TOpenPictureDialog) и Окно сохранения рисунка (TSavePictureDialog)


Эти компоненты являются, соответственно, наследниками класса TOpenDialog и класса TSaveDialog. Диалоговые окна содержат дополнительную область для быстрого просмотра содержимого выбранного графического файла.

Компонент Окно выбора шрифта (TFontDialog)

Компонент предназначен для вызова стандартного диалогового окна выбора шрифта, доступного в системе. В соответствии с полями этого окна компонент имеет набор свойств, которые приведены ниже.

Таблица 4.29. Свойства класса TFontDialog

Свойство	Назначение
Device	Устройство, для которого отображается список доступных шрифтов. Возможные значения — fdScreen (экран), fdPrinter (принтер) и fdBoth (как экран, так и принтер)
Font	Выбранный пользователем шрифт (тип TFont)
MaxFontSize	Максимальный размер шрифта, ограничивающий содержимое показываемого списка шрифтов
MinFontSize	Минимальный размер шрифта, ограничивающий содержимое показываемого списка шрифтов
Options	Дополнительные характеристики внешнего вида диалогового окна

Если, например, на форме имеется надпись Label1, то при щелчке на кнопке Button1 следующий обработчик вызовет диалоговое окно выбора шрифта. После того как пользователь сделает выбор, шрифт, которым сделана надпись, изменится.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
if FontDialog1.Execute then
   Label1.Font.Assign(FontDialog1.Font);
end;
```

Компонент Окно выбора цвета (TColorDialog)

С помощью данного компонента вызывается стандартное диалоговое окно выбора цвета (рис. 4.1).

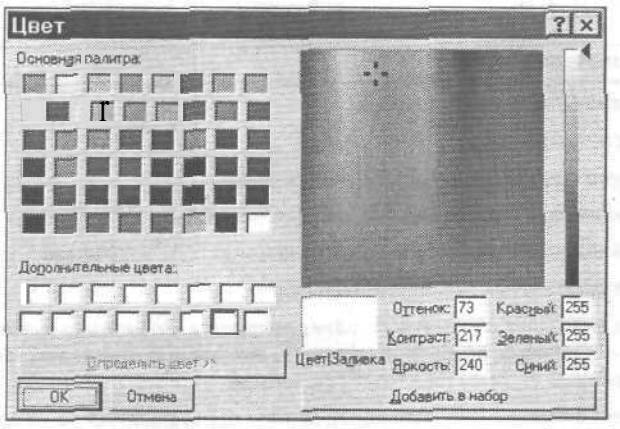


Рис. 4.1. Стандартное диалоговое окно выбора цвета в операционной системе Windows

Свойство Color (тип TColor) содержит выбранный пользователем цвет, а свойство CustomColors (тип TStrings) хранит в текстовом формате описание дополнительных пользовательских цветов. Цвет в этом формате задается шестью символами, определяющими в шестнадцатеричном виде значение цвета в соответствии с требованием цветовой системы *RGB*. Каждый байт задается двумя символами, например FFFFFF или 08EE08. Имеется также свойство Options, присутствующее у всех подобных компонентов и позволяющее выполнять тонкие специфические настройки работы окна.

В следующем примере демонстрируется, как при щелчке на кнопке Button1 происходит вызов окна выбора цвета. Выбранный цвет будет использован для изменения цвета фигуры Shape1.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
if ColorDialog1.Execute
then Shape1.Color := ColorDialog1.Color;
end;
```

Компоненты Печать и Настройка принтера и Настройка параметров страницы печати (TPrintDialog, TPrinterSetupDialog, TPageSetupDialog)

Компонент TPrinterSetup Dialog, предназначенный для настройки параметров работы принтера, не имеет оригинальных свойств, потому что эти настройки существенно различаются для разных видов принтеров. На основании этого компонента можно создавать свои собственные компоненты для конкретных принтеров.

Компонент TPrintDialog отображает стандартное окно печати *Windows*. В нем можно задать различные параметры печати, которые определяются следующими свойствами.

Таблица 4.30. Свойства класса TPageSetupDialog

Свойство	Назначение
Collate	Флажок Разобрать по копиям
Copies	Число копий
FromPage	Номер страницы, с которой начнется печать
MaxPage	Максимальное число страниц, которое может быть напечатано
MinPage	Минимальное число страниц, которое может быть напечатано
Options	Дополнительные параметры настройки
PrintRange	Вид диапазона печатаемых страниц документа. Возможные значения: prAllPages (все страницы); prSelection (страницы выбранного фрагмента); prPageNums (страницы из диапазона FromPage/ToPage)
PrintToFile	Имеет значение True, если вывод должен осуществляться не на принтер, а в файл
ToPage	Номер страницы, на которой заканчивается печать

Компонент TPageSetupDialog позволяет настроить характеристики печатаемых страниц. Они задаются в свойствах Margin Bottom, MarginLeft, Margin Right, MarginTop (нижняя, левая, правая, верхняя границы печати), PageHeight и PageWidth (высота и ширина страницы), а также в свойстве Options, описывающем дополнительные параметры. Единицы измерения размеров задаются в свойстве Units.

Компонент Поиск (TFindDialog)

Компонент используется для отображения диалогового окна поиска текстовой строки. Свойства класса TfindDialog приведены ниже.

Таблица 4.31. Свойства класса TFindDialog

Свойство	Назначение
FindText	Строка для поиска
Options	Дополнительные настройки
Position	Координата левого верхнего угла диалогового окна при его выводе на экран (в пикселах)

Из методов этого класса следует отметить процедуру

```
procedure CloseDialog;
```

Эта процедура закрывает окно, но не меняет значений установленных свойств, чтобы в дальнейшем можно было выполнить повторный поиск со старыми параметрами. При щелчке на кнопке Найти далее генерируется событие QnFind.

Компонент Поиск и замена (TReplaceDialog)

Данный компонент является наследником компонента TFindDialog. Он несколько расширяет его возможности и позволяет вводить строку для замены найден-

ного текста. Компонент имеет новое свойство ReplaceText и соответствующее ему поле в диалоговом окне. При щелчке на кнопке Заменить или Заменить все генерируется сообщение OnReplace.

Печать из программы

Технология вывода информации на принтер

Вывод произвольной информации на печать реализован в системе *Delphi 7* очень просто. В системе имеется готовый объект Printer класса TPrinter (модуль Printers), который содержит свойство Canvas типа TCanvas (холст). При отрисовке на нем произвольной информации она будет выводиться не на экран, а на принтер. Свойство Canvas объекта Printer доступно, только когда принтер подготовлен для печати данных. Для подготовки свойства Canvas используются методы

```
procedure BeginDoc;
procedure EndDoc;
```

Первый из них определяет начало печати и создает экземпляр класса TCanvas, а второй указывает на завершение печати, после чего свойство Canvas снова будет недоступно. Например, чтобы вывести на лист бумаги строку «Проверка принтера», можно воспользоваться следующим программным текстом.

```
Printer.BeginDoc;
Printer.Canvas.TextOut(50, 50, 'Проверка принтера');
Printer.EndDoc;
```

ВНИМАНИЕ При выводе графических данных на печать следует учитывать, что разрешение экрана (в пикселах) и принтера (в точках) существенно различаются. Изображение, нормально выглядящее на экране, на листе бумаги окажется совсем маленьким. Разрешение монитора обычно составляет не более 72-96 точек на дюйм, а минимальное разрешение лазерного принтера — 300 точек но дюйм.

Предварительный просмотр

Перед выводом информации на принтер желательно предварительно выяснить, что она будет собой представлять. Для этого можно создать отдельную процедуру, которая будет выполнять вывод графических данных на передаваемый в качестве параметра холст принтера или формы с учетом коэффициентов масштабирования, рассчитанных на основании текущего разрешения экрана и принтера.

```
procedure MyPaint( Canvas: TCanvas; KX, KY: real );
```

Если будет выполняться вывод на печать, то дополнительно придется выполнять соответствующие команды подготовки принтера и завершения печати.

```
if ToPrinter then Printer.BeginDoc;
if ToPrinter then MyPaint( Printer.Canvas, 1, 1 )
else MyPaint( Form1.Canvas, 0.1, 0.1 );
if ToPrinter then Printer.EndDoc;
```

Печать текста

Для вывода только текстовой информации можно использовать другой подход. Стандартная процедура AssignPrn (по аналогии с процедурой AssignFile) связывает файловую переменную типа TaxtFile с текущим принтером. При этом процедуры Write и WriteLn будут выводить строку текста сразу на принтер, соответственно начиная печать с текущей позиции строки или с новой строки.

```
procedure TForm1.Button1Click(Sender: TObject);
var MyFile: TextFile;
begin
AssignPrn(MyFile);
Rewrite(MyFile);
Writeln(MyFile, 'Проверка принтера');
System.CloseFile(MyFile);
end;
```

Свойства и методы класса TPrinter

Свойства и методы класса TPrinter приведены в табл. *4.32* и 4.33.

Таблица 4.32. *Свойства класса TPrinter*

Свойство	Назначение
Aborted	Имеет значение True, если пользователь прервал процесс печати
Canvas	Область вывода графической информации для принтера
Capabilities	Настройки режима печати (ориентация, число копий и так далее)
Copies	Число печатаемых копий
Fonts	Список шрифтов, поддерживаемых текущим принтером
Orientation	Ориентация бумаги: книжная или альбомная
PageHeight	Высота печатаемой страницы в пикселах
PageNumber	Номер текущей печатаемой страницы
PageWidth	Ширина печатаемой страницы в пикселах
PrinterIndex	Номер принтера из свойства Printers
Printers	Список названий всех принтеров, доступных в системе
Printing	Имеет значение True, когда выполняется печать
Title	Стандартный заголовок страницы

Таблица 4.33. *Методы класса TPrinter*

Метод	Назначение
procedure Abort;	Прерывание печати
procedure GetPrinter(ADevice,ADriver, APort: PChar; var ADeviceMode: THandle);	Получение информации о текущем принтере
procedure NewPage;	Начало печати новой страницы
procedure Refresh;	Обновление списка шрифтов и принтеров, установленных в системе
procedure SetPrinter(ADevice, ADriver, APort: PChar; ADeviceMode: THandle);	Указанный принтер становится текущим

Дополнительные компоненты Delphi 7 (панель Additional)

В данном разделе будут рассмотрены компоненты *Delphi 7,* расположенные на панели Additional (Дополнительные).

Компонент Быстрая кнопка (TSpeedButton)

Используется при формировании панелей управления с «быстрыми» командными кнопками.

После размещения объекта на форме изображение, помещаемое на кнопку, задается в свойстве Glyph (Значок). При этом вызывается редактор, с помощью которого выбирается нужная картинка (в формате .BMP). Большой набор готовых картинок для кнопок можно найти в каталоге Borland Shared (подкаталог Images/Buttons), который автоматически создается вместе с основным программным каталогом системы *Delphi 7* (рис. 4.2).

Как правило, «быстрые» командные кнопки используются группами. Чтобы объединить несколько таких кнопок, надо задать ненулевое значение для свойства GroupIndex. Кнопки с одинаковым значением этого свойства считаются принадлежащими к одной группе.

Если в группе нажимается одна командная кнопка, кнопка, которая уже была нажата, как правило, автоматически отпускается. Для поддержки подобного режима работы надо для всех кнопок группы установить значение True для свойства AllowAllUp. Перевести кнопку в нажатое состояние на этапе проектирования можно с помощью свойства Down, присвоив ему значение True.

Картинка для быстрой командной кнопки задается в свойстве Glyph. Верхний левый бит изображения считается цветом, который будет «прозрачным». Все точки изображения, имеющие такой цвет, на кнопке не отрисовываются. Так же устроена работа

Рис. 4.2. *Создание графического значка кнопки*

компонента TBitBtn, однако командная кнопка позволяет дополнительно указать, надолиделатькартинкупрозрачной(свойствоTransparentимеетзначениеTrue)или нет.

Свойство Flat принимает значение True, когда требуется сформировать «плоский» вид кнопки. В этом случае ее границы не видны и появляются только при наведении указателя.

Чтобы более гибко обрабатывать действия пользователя, в классе TSpeedButton имеется свойство MouseInControl, которое принимает значение True, когда указатель мыши расположен над кнопкой.

Компонент Кнопка с картинкой **(TBitBtn)**

Этот компонент предназначен для создания кнопки с картинкой. В системе имеется набор готовых шаблонов.

Картинка загружается в объект тем же способом, что и в случае компонента TSpeedButton. Каждая такая картинка может состоять из 1-4 частей, равных по ширине. Первая часть — изображение кнопки в обычном состоянии, вторая — изображение «отключенной» недоступной кнопки (свойство Enabled имеет значение False), третья — изображение на кнопке после щелчка мыши, четвертая — изображение на «нажатой» кнопке. Число составных частей изображения задается в свойстве NumGlyphs (от 1 до 4), Расстояние от картинки до границ кнопки (в пикселах) можно указать в свойстве Margin.

В свойстве Kind задается реакция кнопки на щелчок. Некоторые стандартные действия можно реализовать без дополнительного программирования. Соответствующие значения свойства Kind приведены ниже.

Таблица 4.34. Значения свойства Kind

Значение	Действие
bkCustom	Реакция кнопки определена программистом
bkOK	Закрытие формы стандартным действием. В свойство ModalResult записывается значение mrOK
bkCancel	Закрытие формы с отказом от изменений. В свойство ModalResult записывается значение mrCancel
bkYes	Подтверждение изменений. В свойство ModalResult записывается значение mrYes
bkNo	Отказ от изменений. В свойство ModalResult записывается значение mrNo
bkHelp	Вызов раздела справочной системы в соответствии со значением, записанным в свойство HelpContext
bkClose	Закрытие формы
bkAbort	Прерывание. Форма не закрывается
bkRetry	Повтор попытки
bkIgnore	Игнорирование возникшей ситуации
bkAll	Подтверждение всех изменений

Для каждого из видов кнопки, за исключением bkCustom, в системе *Delphi 7* имеется особая картинка.

ЗАМЕЧАНИЕ Способ закрытия формы, на которой расположено кнопка, определяется значением свойства ModalResult. Это свойство имеется и у обычной кнопки TButton.

С помощью свойство Layout можно указать расположение картинки по отношению к тексту (заголовку) кнопки. Возможные значения указаны ниже.

Таблица 4.35. *Значения свойства Layout*

Значение	Расположение
blGlyphLeft	Слева
blGlyphRight	Справа
blGlyphTop	Вверху
blGlyphBottom	Внизу

Расстояние между картинкой и текстом (в пикселах) задается в свойстве Spacing.

У класса TBitBtn имеется новый метод Click, который моделирует щелчок на кнопке. Он используется, когда надо закрыть форму одним из стандартных способов.

Компонент Шаблон ввода (TMaskEdit)

Этот компонент позволяет вводить данные в текстовое поле по заданному шаблону. Он полезен для приложений, где надо контролировать вводимую пользователями информацию.

После размещения компонента на форме для него надо задать структуру маски (шаблона, по которому будет осуществляться ввод) и начальное значение поля. Структура маски может быть сформирована двумя способами; с помощью редактора, вызываемого из контекстного меню объекта — пункт Input Mask Editor (Редактор маски ввода), — или программно, заданием нужного значения для свойства EditMask, которое хранит структуру маски в текстовом виде. Рассмотрим второй способ.

Маска состоит из трех частей, разделенных точками с запятой. Первая часть — собственно маска, описывающая допустимые значения символов в конкретных позициях поля. В процессе ввода в поле могут присутствовать дополнительные символы, предназначенные для более наглядного представления информации, а также для автоматического включения в итоговый текст (табл. 4.36).

Таблица 4.36. *Назначение символов первой части маски*

Символ	Значение
!	Если этот символ присутствует в маске, то необязательные символы вставляются перед маской. Если отсутствует — необязательные символы вставляются после маски
>	Все последующие символы автоматически приводятся к верхнему регистру
<	Все последующие символы автоматически приводятся к нижнему регистру
<>	Прекращение преобразования регистра
\	Последующий символ вставляется в маску. Данная возможность необходима для добавления символов, используемых в качестве элементов шаблона
L	В данную позицию разрешен только ввод буквы
I	В данную позицию разрешен только ввод буквы, но ее можно оставить пустой
A	В данную позицию разрешен только ввод буквы или цифры
a	В данную позицию разрешен только ввод буквы или цифры, но ее можно оставить пустой
C	В данную позицию разрешен ввод произвольного символа
c	В данную позицию разрешен ввод произвольного символа, но ее можно и оставить пустой
O	В данную позицию разрешен только ввод цифры
9	В данную позицию разрешен только ввод цифры, но ее можно оставить пустой
#	В данную позицию разрешен только ввод цифры или символов +/-. Позицию можно оставить пустой
:	В данную позицию вставляется символ, который в текущей версии Windows используется для разделения часов, минут и секунд при записи времени
_	В данную позицию автоматически вставляется пробел

Вторая часть маски определяет, будут ли эти дополнительные символы включены в итоговый текст. Символ 0 указывает, что они не должны включаться в результат, любой другой символ разрешает их включение. Третья часть маски —символ, запол-

няющий позиции шаблона, которые пользователь оставил пустыми (по умолчанию используется пробел).

Символы, не включенные в табл. 4.36, вставляются в результирующий текст бея изменений. В процессе ввода курсор автоматически перескакивает через них. Рассмотрим пример, в котором требуется сформировать поле для ввода московских телефонных номеров. Маска может выглядеть так:

```
MaskEdit1.EditMask := 'Тел. +7 (\0\95)000-00-00;*;*'.
```

Б процессе ввода человеку достаточно указать только 7 цифр, не нажимая дополнительных клавиш. В результате в свойстве EditText объекта MaskEdit1 может оказаться, например, такой текст: «Тел. +7(095)123-45-67». Это свойство отличается от свойства Text тем, что поля маски, не указанные пользователем, предварительно заполняются пробелами.

Отключить использование маски можно, записав в свойство IsMasked значение False. В такой ситуации можно задать значение свойства MaxLength, определяющего максимально допустимую длину вводимого текста. Для получения текущей длины содержимого текстового поля во время работы программы (которое реально отличается от содержимого свойства EditText, форматируемого автоматически), надо обратиться к методу GetTextLen.

Метод ValidateEdit выполняет проверку корректности введенной пользователем **информации** и генерирует исключительную ситуацию EDBEditError в случае несоответствия введенного текста указанной маске.

Компонент Рамка (TBevel)

Используется для создания рамок и отдельных линий оформления. Напоминает панель, но не предназначен для группировки элементов.

Свойство Shape (Образ) может принимать одно из следующих значений.

***Таблица** 4.37. Значения свойства Shape*

Значение	Форма **рамки**
bsBox	Прямоугольник. Область внутри него отображается в соответствии со значением свойства Style
bsFrame	Прямоугольник. Внутренняя область не изменяется
bsTopLine	Для выделенной области показывается только верхняя граница
bsBottomLine	Для выделенной области показывается только нижняя граница
bsLeftLine	Для выделенной области показывается только левая граница
bsRightLine	Для выделенной области показывается только правая граница
bsSpacer	Рамка не отображается. Значение используется разработчиками на этапе проектирования для выделения областей под собственные нужды

Дополнительно в свойстве Style указывается форма окаймляющих линий панели: bsLowered (вдавленные линии) или bsRaised (выпуклые линии).

Компонент Постоянный текст (TStaticText)

Основное отличие этого компонента от ранее рассмотренного компонента TLabel только в том, что он позволяет взять выводимый текст в рамку.

Форма рамки определяется свойством BorderStyle, которое может принимать одно из трех значений: sbsNone (кайма отсутствует), sbsSingle (кайма представляет собой обычную линию), sbsSunken (кайма имеет вид вдавленной области).

Компонент Фигура (TShape)

Этот компонент предназначен для отображения на форме различных геометрических фигур.

Конкретная форма геометрического объекта задается в свойстве Shape. Возможны следующие значения.

Таблица 4,38. *Значения свойства Shape*

Значение	Форма **фигуры**
stCircle	Круг
stEllipse	Эллипс
stRectangle	Прямоугольник
stRoundRect	Прямоугольник со скругленными углами
stRoundSquare	Квадрат со скругленными углами
stSquare	Квадрат

Цвет фигуры (рис. 4.3) определяется кистью объекта (свойство Brush), границы фигуры — карандашом (свойство Реп).

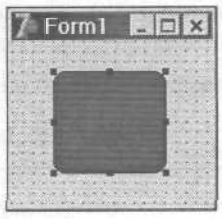


Рис. 4.3- *Фигура (квадрат со скругленными углами), размещенная на форме*

Компонент Разделитель (TSplitter)

С помощью этого компонента клиентская область формы может быть разделена на несколько панелей, размеры которых допускается изменять, просто перетаскивая границы этих панелей. Активное применение разделителя в первых 32-разрядных программах для *Windows 95* в свое время послужило хорошим стимулом развития технологии стыковки.

Создание разделителя выполняется в системе *Delphi 7* довольно оригинально. Рассмотрим пример, когда область формы надо разделить на три части, как в некото-

рых программах, предназначенных для приема электронной почты. В левой части располагается список почтовых папок, справа — список всех писем выбранной папки, а в верхней части —текущее письмо. Каждую часть представим в виде панели.

Порядок применения разделителя достаточно строгий. В первую очередь на форме помешаются объекты, которые выравниваются по границам формы, вслед за каждым объектом сразу устанавливается разделитель, а последний объект выравнивается по всей оставшейся клиентской области.

На пустой форме размешаем компонент TPanel, его свойству Align (Выравнивание) задаем значение alLeft (по левой границе формы). Затем на форму устанавливается компонент TSplitter, который сразу автоматически выровняется по правой границе объекта Panell. Следующим снова размещаем компонент TPanel, который выравниваем уже по верхней границе формы — в свойство Align записываем значение alTop. Опять добавляем компонент TSplitter, который теперь автоматически выравнивается по нижней границе объекта Panel2. В заключение, на свободную часть формы помещаем третий компонент TPanel, который выравнивается по всей оставшейся клиентской части — свойство Align принимает значение alClient (рис. 4.4),

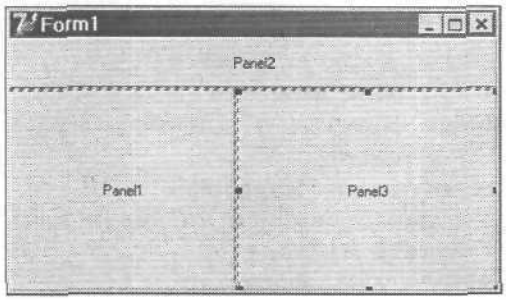


Рис. 4.4. Форма, содержащая панели, связанные разделителями

Если теперь откомпилировать и запустить программу, то, переместив курсор на область одного из разделителей и нажав левую кнопку мыши, можно произвольно менять размеры каждой из областей (панелей). Насколько близко к границам окна (в пикселах) можно приближать разделители, определяется в свойстве MinSize каждого объекта Spliter.

Свойство Beveled определяет, будет ли разделитель выглядеть объемным (для этого свойство должно иметь значение True).

Свойство ResizeStyle задает способ отображения на экране процесса перетаскивания разделителя. Оно может принимать одно из следующих значений.

Таблица 4.39. *Значения свойства ResizeStyle*

Значение	Способ отображения
rsNone	Предполагаемое новое положение разделителя не отображается, а размеры областей меняются, когда пользователь отпускает кнопку мыши
rsLine	Предполагаемое новое положение разделителя отображается в виде сплошной черной линии

продолжение

Таблица 4.39. Значения свойства ResizeStyle (продолжение)

Значение	Способ отображения
rsPattern	Предполагаемое новое положение разделителя отображается в виде толстой пунктирной линии
rsUpdate	При перемещении указателя мыши происходит немедленное изменение размеров соответствующих областей

Работать с данным полезным компонентом можно прямо из программы. В следующем примере форма делится на две части с помощью разделителя, динамически создаваемого в процессе работы. На форме предварительно размещены две панели: одна выровнена по левой границе формы, а вторая помещена в правой части формы и никак не выровнена.

В момент создания формы к ней добавится новый объект (делитель).

```
procedure TForm1.FormCreate(Sender: TObject);
var Split: TSplitter;
begin
// создаем новый объект:
Split := TSplitter.Create(Form1);
// в качестве его родителя
// задаем текущую форму:
Split.Parent := Form1;
// помещаем делитель на форму -
// он не должен накладываться на левую панель:
Split.Left := Panel1.Width + 5;
// выравниваем его так, чтобы он
// автоматически присоединился к этой панели:
Split.Align := alLeft;
// задаем минимальный допуск:
Split.MinSize := 20;
// для второй панели указываем способ выравнивания
// по всей оставшейся клиентской области:
Panel2.Align := alClient;
// теперь делитель готов к работе
end;
```

Этот пример чисто демонстрационный. Б реальной работе переменную Split надо делать не локальной, а глобальной, чтобы всегда иметь возможность освободить временно отведенную ей память. Это возможно, так как переменные типа class, как уже говорилось, на самом деле представляют собой указатели.

Компонент События приложения (TApplicationEvents)

Данный компонент может использоваться совместно с компонентом TActionList. Он позволяет принимать и обрабатывать все сообщения, адресо-

ванные приложению (объект Application), в одном месте. Этот компонент удобно применять, в частности, когда требуется обрабатывать сообщения *Windows,*

Объекты класса TApplicationEvents могут быть размещены на любых формах проекта. Все они будут получать копии сообщений, предназначенных для программы. Эти объекты способны также обрабатывать события (табл. 4.40), которые не обслуживаются компонентом TActionList, если он тоже имеется на форме.

Таблица** 4.40. События, обрабатываемые компонентам **TApplicationEvents

Событие	Действие
OnActionExecute	Действие, не обслуживаемое компонентом TActionList
OnActionUpdate	Состояние ожидания (OnUpdate), не обслуживаемое компонентом TActionList
OnActivate	Активизация приложения
OnDeactivate	Пользователь переключился на другую программу Windows
OnException	Возникновение исключительной ситуации, не обрабатываемой никаким блоком try
OnHelp	Запрос справочной системы
OnHint	Указатель находится над объектом, способным отображать всплывающую подсказку
OnIdle	Состояние ожидания (например, когда приложение ждет действия пользователя и процессорное время не тратится)
OnMessage	Получение программой сообщения от Windows
OnMinimize	Приложение должно быть свернуто
OnRestore	Приложение должно быть восстановлено из свернутого состояния к нормальному размеру
OnShortCut	Пользователь нажал клавишу. Приложение получает это сообщение до посылки сообщения о нажатии клавиши элементу управления
OnShowHint	Происходит вывод подсказки

После того как компонент TApplicationEvents размещен на форме, надо задать обработчик конкретных событий. Допустим, мы хотим показывать координаты щелчков мыши в пределах окна с помощью надписи Labell. Этот объект надо заранее установить на форму. Для формы можно создать обработчик события OnMouseUp, но это не позволит отслеживать щелчки на различных элементах управления внутри формы, так как переадресация соответствующих сообщений скрыта внутри компонентов *Da/phi 7.* Для этой цели удобно применить компонент TApplicationEvents, определив реакцию на событие OnMessage. Заголовок обработчика будет выглядеть так:

```
procedure TForm1.ApplicationEvents1Message(
    var Msg: tagMSG;
    var Handled: Boolean);
```

Здесь самый важный параметр — это Msg (его тип на самом доле называется TMsg). Он описывает сообщение, полученное от *Windows,* перед тем как оно уйдет дальше в программу. Важнейшее поле типа TMsg называется message и содержит стандартный код сообщения, полученного от *Windows* (коды всех сообщений *Windows* хранятся в модуле Messages).

В нашем случае надо «поймать» сообщение WM_LBUTTONUP (отпускание левой кнопки мыши). Экранные координаты щелчка хранятся в структуре типа TMsg, имеющей имя pt (тип TPoint). Их можно предварительно пересчитать в координаты окна с помощью стандартной функции ScreenToClient, получающей в качестве аргумента запись типа TPoint и возвращающей запись такого же типа, только с пересчитанными координатами. Целиком обработчик запишется так:

```
procedure TForm1.ApplicationEvents1Message(
   var Msg: tagMSG;
   var Handled: Boolean);
   var P: TPoint;
begin
if Msg.message = WM_LBUTTONUP then
   begin
   P := ScreenToClient(Msg.pt);
   Label1.Caption := IntToStr(P.x) + ',' + IntToStr(P.y);
   end;
end;
```

Теперь если даже на форме разместить кнопку, то при щелчке на ней надпись Label1 будет отображать координаты точки щелчка.

Компонент Таблица строк (TStringGrid)

Использование многими пользователями электронных таблиц типа Excel стало практически неотъемлемой частью применения компьютеров. В системе *Delphi 7* имеются два компонента, которые позволяют до некоторой степени сымитировать работу электронной таблицы, оставляя при этом, конечно, основную работу по реализации конкретных функций такой таблицы программистам.

Первый компонент — это таблица строк, позволяющая работать с текстовой информацией в двумерной таблице, имеющей столбцы и строки (их размеры можно менять с помощью мыши). Дополнительно, к каждой ячейке таблицы можно «привязать» свой объект, характеристики которого программист представит в виде строки, расположенной в этой ячейке.

Основное свойство таблицы строк — это двумерный массив Cells, позволяющий обращаться к содержимому ячеек и изменять их содержимое. Первое измерение - это номер строки, второе — номер столбца.

ЗАМЕЧАНИЕ Нумерация элементов в таблице строк начинается с нуля.

Число столбцов задается в свойстве ColCount, число строк — в свойстве RowCount.

Следующий код программы изменяет размер таблицы, помещенной на форму в режиме проектирования (по умолчанию принят размер 5x5 элементов), на размер 10x10 ячеек и заполняет ячейки строками, содержащими их координаты (рис, 4.5),

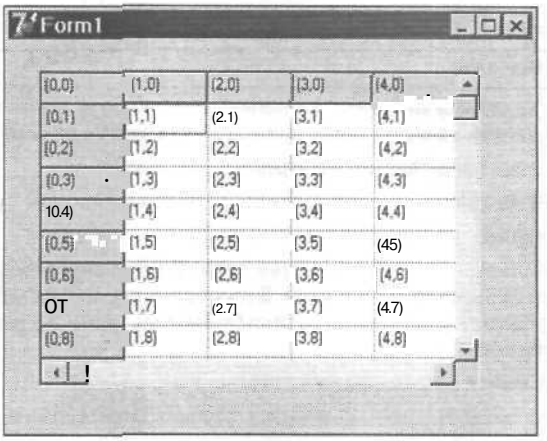


Рис. 4.5. Форма, содержащая заполненную таблицустрок

```
procedure TForm1.FormCreate(Sender: TObject);
   var c,r: integer;
begin
StringGrid1.ColCount := 10;
StringGrid1.RowCount := 10;
for c := 0 to StringGrid1.ColCount-1 do
for r := 0 to StringGrid1.RowCount-1 do
StringGrid1.Cells[c,r] :=
   '(' + IntToStr(c) + ',' + IntToStr(r) + ')';
end;
```

Обратите внимание, что левый столбец и верхняя строка, хотя и содержат текстовую информацию, фактически являются заголовочными областями. Использовать их наравне с другими ячейками не совсем правильно.

ЗАМЕЧАНИЕ Число строк и столбцов, имеющих характер заголовка, задается свойствами FixedCols и FixedRows. Если таблица не содержит заголовочной информации, эти свойства должны принимать значение 0.

Можно получить доступ ко всем элементам одного столбца или одной строки. Соответствующие свойства Col и Row имеют тип TStrings, позволяющий обращаться к нужному элементу по номеру.

Чтобы привязать к ячейке объект (наследник класса TObject), надо использовать свойство Objects, представляющее собой такой же массив, как Cells, но содержащий не строки, а объекты. Эти объекты должны создаваться, а также уничтожаться программистом вручную, иными словами, весь контроль над состоянием этого массива полностью возлагается на разработчика. Свойство Objects предоставляет только доступ к нужному объекту. Остальные свойства, предназначенные для оформления таблицы строк, приведены в табл. 4.41.

Таблица 4.41. Свойства, предназначенные для оформления таблицы строк

Свойство	Назначение
BorderStyle	Стиль отрисовки ячейки. Значение этого свойства можно комбинировать со значением свойства Ctrl3D для получения оригинального вида ячеек
Ctrl3D	Таблица представляется в «трехмерном» стиле
ColWidths	Массив, хранящий ширину каждого столбца в пикселах
DefaultColWidth	Начальная ширина столбца по умолчанию
DefaultDrawing	Если значение свойства — True, производится автоматическая отрисовка содержимого каждой ячейки. В противном случае для таблицы необходимо определить обработчик события OnDrawCell, чтобы запрограммировать процесс отрисовки ячейки
DefaultRowHeight	Начальная высота строки по умолчанию
FixedColor	Цвет области строк и столбцов, служащих заголовком таблицы
GridHeight	Высота всей таблицы (в пикселах)
GridLineWidth	Ширина (в пикселах) линий, разделяющих ячейки таблицы
GridWidth	Ширина всей таблицы (в пикселах)
Options	Множество значений (тип set of TGridOption), позволяющее задавать различные режимы работы таблицы: выделение нескольких ячеек, способ использования линий прокрутки и прочие. В частности, чтобы разрешить изменение размеров строк и столбцов, надо установить значение True для свойств goRowSizing и goColSizing, вложенных в свойство Options. Чтобы разрешить редактирование содержимого ячеек, надо записать значение True в подсвойство goEditing
RowHeights	Массив, хранящий высоту каждой строки в пикселах
ScrollBars	Наличие полос прокрутки
VisibleColCount	Число видимых в таблице столбцов (без области заголовка)

Чтобы установить для просмотра нужную область **таблицы,** надо задать номер начальной строки в свойстве TopRow, а номер начального столбца — в свойстве LeftCol. Область заголовков при этом не изменяется. Например, расположив на форме кнопку и включив указанные далее операторы в обработчик щелчка, можно быстро прокрутить таблицу к ячейке (3,5), которая будет расположена в верхнем левом углу таблицы под заголовком (рис. 4.6),

```
procedure TForm1.Button1Click(Sender: TObject);
begin
StringGrid1.LeftCol := 3;
StringGrid1.TopRow := 5;
end;
```

Чтобы выделить прямоугольную область ячеек (или одну ячейку) другим цветом, надо использовать свойство Selection. Оно доступно только во время работы программы и имеет тип TGridRect, напоминающий тип TRect.

Например, если требуется по щелчку на кнопке Button1 выделить область от ячейки (2,2) до ячейки (3,5), надо выполнить следующие операторы.

```
procedure TForm1.Button1Click(Sender: TObject);
Var GRect: TGridRect;
```

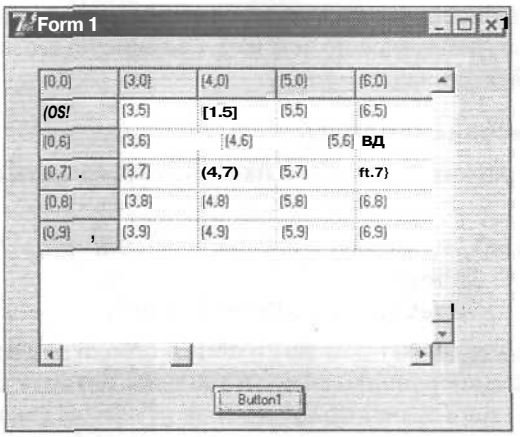


Рис. *4.6. Автоматическая прокрутка таблицы строк*

```
begin
GRect.Left  := 2;
GRect.Top  := 2;
GRect.Right  := 3;
GRect.Bottom  := 5;
StringGrid1.Selection  := GRect;
end;
```

ВНИМАНИЕ Чтобы такое выделение работало, необходимо в свойстве Options включить подсвойство goRangeSelect (установить значение True).

Если запустить описанный пример, то после выделения группы ячеек окажется, что нижняя ячейка (3,5), которая была автоматически сделана текущей, не подсвечена. Чтобы текущая ячейка таблицы выделялась отдельным цветом, надо подсвойствуgoDrawFocusSelectedсвойстваOptionsзадатьзначениеTrue. Можнотакже программно включить его в множество Options.

```
StringGrid1.Options :=
StringGrid1.Options + [goDrawFocusSelected];
```

Экранные координаты конкретной ячейки (в пикселах) определяются с помощью метода (функции) CellRect, которая, получая в качестве первого параметра помер столбца, а в качестве второго — номер строки, возвращает структуру типа TRect с координатами прямоугольника, охватывающего заданную ячейку. Если эта ячейка невидима, то все поля структуры будут иметь значения 0.

Обратный пересчет осуществляется вызовом процедуры MouseToCell.

```
procedure MouseToCell
    (X, Y: Integer; var ACol, ARow: Longint);
```

Здесь X и Y — экранные координаты точки, а значения, возвращаемые через параметры ACol и ARow (передаваемые по ссылке), соответствуют номерам столбца и строки для ячейки, содержащей эту точку.

Программная отрисовка таблицы

Чтобы выполнять отрисовку каждой ячейки, надо определить обработчик события OnDrawCell.

```
procedure TForm1.StringGrid1DrawCell(Sender: TObject;
   ACol, ARow: Integer;
   Rect: TRect; State: TGridDrawState);
```

Параметры ACol и ARow содержат номера столбца и строки рисуемой ячейки, параметр Rect — клиентские координаты области таблицы, которая должна быть отрисована, параметр State определяет статус ячейки: gdSelected (выделена), gdFocused (имеет фокус) или gdFixed (лежит в области заголовка таблицы).

Нарисовать содержимое таблицы теперь можно так (рис. 4.7):

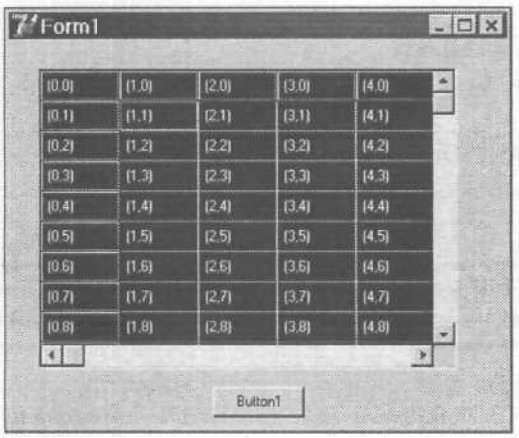


Рис. 4.7. *Ручная отрисовка таблицы с использованием белых символов на черном фоне*

```
procedure TForm1.StringGrid1DrawCell(Sender: TObject;
   ACol, ARow: Integer;
   Rect: TRect; State: TGridDrawState);
begin
with Sender aa TStringGrid do
   begin
   Canvas.Font.Color := clWhite;
   Canvas.Brush.Color := clBlack;
   Canvas.FillRect(Rect);
   Canvas.TextOut(Rect.Left+5,Rect.Top+5,
      '(' + IntToStr(ACol) + ',' + IntToStr(ARow) + ')');
   end;
end;
```

ПОДСКАЗКА Хотя с помощью данного обработчика в ячейках можно выводить и картинки, и другую графическую информацию, для подобных целей лучше применять компонент TDrawGrid, а данный компонент предназначен только для отображения строк.

Чтобы контролировать выбор пользователем конкретной ячейки, можно использовать событие OnSelectCell. Например, поместив на форму надпись, можно выводить в нее координаты текущей выбранной ячейки:

```
procedure TForm1.StringGrid1SelectCell(Sender: TObject;
   ACol, ARow: Integer;
   var CanSelect: Boolean);
begin
Label1.Caption := '(' + IntToStr(ACol) + ',' +
IntToStr(ARow) + ')'
end;
```

Параметры ACol и ARow содержат координаты выбранной ячейки. Значение параметра CanSelect (он передается по ссылке), в котором первоначально записано значение True, можно изменить на False, чтобы запретить выделение данной ячейки,

ЗАМЕЧАНИЕ Событие OnSelectCell генерируется до начала отрисовки (автоматической или явно заданной разработчиком) ячейки.

Событие OnSetEditText возникает, когда пользователь редактирует содержимое ячейки. Это событие полезно обрабатывать, если требуется отслеживать изменение строк в ячейках, чтобы обновлять содержимое объектов, связанных с ячейками.

```
procedure TForm1.StringGrid1SetEditText(Sender: TObject;
   ACol, ARow: Integer;
   const Value: String);
```

В заголовке обработчика события появился новый параметр Value, который содержит новое значение, вводимое пользователем в ячейку с координатами ACoL, ARow.

Компонент Рисуемая таблица (TDrawGrid)

Для создания таблицы строк удобнее всего использовать компонент TStringGrid. В более общем случае, когда в каждой ячейке может храниться произвольный объект, надо применять компонент TDrawGrid. При этом вся работа по визуальному представлению каждого объекта в ячейке полностью возлагается на программиста.

Большинство свойств, описывающих поведение и внешний вид ячеек таблицы, совпадают с аналогичными свойствами таблицы строк. Однако в классе TDrawGrid отсутствуют свойства Cells и Objects. Создавать массивы объектов или определять, что надо нарисовать в конкретной ячейке (обработчик события OnDrawCell), необходимо

отдельно. Класс TDrawGrid может быть использован как базовый класс для создания собственных электронных таблиц со сложным поведением.

Компонент Список с флажками (TCheckListBox)

Данный список ничем не отличается от обычного списка за исключением дополнительных флажков в начале каждой строки.

При создании такою списка дополнительно настраиваются свойства, указанные ниже.

Таблица 4.42. *Свойства класса TCheckListBox*

Свойство	Назначение
AllowGrayed	Имеет значение True, если флажки списка могут находиться в промежуточном («сером») состоянии
Checked	Массив состояний флажков. Отслеживаются только два состояния: включен (True) и выключен («серое» состояние считается выключенным)
Flat	Имеет значение True, если форма флажков плоская
ItemEnabled	Массив допустимых состояний флажков. Если для конкретного флажка задано значение True, то его состояние можно менять, в противном случае флажок недоступен для изменения
State	Массив состояний флажков, элементы которого принимают одно из трех значений: cbUnchecked (выключен), cbChecked (включен), cbGrayed («серый»)

При работе со списком флажков можно обрабатывать событие OnClickCheck, которое возникает, когда пользователь меняет состояние одного из флажков.

Компонент Прокручиваемая область (TScrollBox)

Когда по каким-то причинам в одном окне надо разместить большое число элементов управления, которые не умещаются на видимой части экрана, желательно иметь возможность прокрутки экрана для доступа к нужным элементам. Такой возможностью обладает каждая форма *Delphi 7* — если поместить компоненты за пределы видимой области, то полосы прокрутки появятся на ней автоматически.

Однако при этом в процессе прокрутки из видимой области могут уйти и панель с командными кнопками, и строка состояния, что идеологически неверно. Компонент TScrollBox позволяет организовать *к* рамках одной формы неограниченное количество областей прокрутки с оригинальным содержимым.

После размещения компонента TScrollBox на форме, внутри него можно размещать элементы управления обычным способом (рис. 4.8). После запуска программы эта область доступна для прокручивания без ограничений.

Автоматическое возникновение полос прокрутки происходит, если свойство AutoScroll имеет значение True. В противном случае разработчику надо указать наличие полос прокрутки явно, например в момент создания формы.

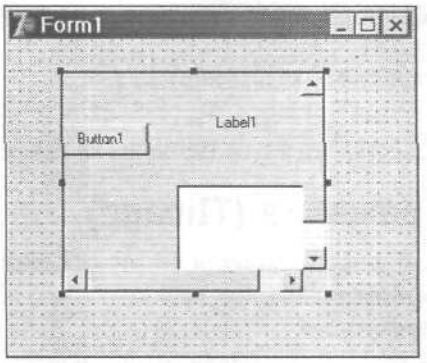


Рис. 4.8. Прокручиваемая область с элементами управления

Полосы прокрутки элемента управления

Свойствами HorzScrollBar и VertScrollBar (горизонтальная и вертикальная полосы прокрутки, класс TControlScrollBar) обладает немало компонентов Delphi 7. В том случае, когда значение свойства AutoScroll имеет значение False, настроить полосы прокрутки с помощью Инспектора объектов нельзя. Сделать это можно только непосредственно в программном коде, например так:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
with ScrollBox1.VertScrollBar do
   begin
   Range := 1000,-
   Position := 0;
   Visible := True;
   end;
end;
```

Свойство Range полос прокрутки определяет максимальную длину (в данном случае в вертикальном направлении) прокручиваемой области в пикселах. Другие свойства класса TControlScrollBar во многом аналогичны свойствам класса TScrollBar (полоса прокрутки, не привязанная к элементу управления).

Во время работы программы можно динамически включать и выключать возможность автоматической прокрутки области компонента TScrollBox. Это делается с помощью методов EnableAutoRange и DisableAutoRange. Полезен также метод:

```
procedure ScrollInView(AControl: TControl);
```

Он прокручивает область внутри объекта ScrollBox так, чтобы стал виден заданный элемент управления внутри этой области. Например, если на объект ScrollBoxl поместить список (объект ListBoxl) так, чтобы он был едва виден, то следующий код обработчика щелчка на кнопке Button1 позволит прокрутить область внутри объекта ScrollBoxl нужным образом.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
ScrollBox1.ScrollInView(ListBox1);
end;
```

Никаких нестандартных событий класс TScrollBox не имеет,

Компонент Изображение (TImage)

Данный компонент активно используется во многих программах, причем не только для отображения статических картинок, но и для создания различных анимационных эффектов.

В большинстве случаев содержимое изображения загружается из файла на этапе проектирования. Для этого служит свойство Picture (класс TPicture), описывающее точечное изображение(.BMP), значок, графический метафайл *Windows* или другой пользовательский графический ресурс. Класс TPicture (рисунок) не является компонентом *Delphi* 7, он просто входит в состав библиотеки *VCL* как вспомогательный, но на его основе могут быть созданы полноценные компоненты.

Текущее содержимое экземпляра класса хранится в одном из свойств: Bitmap (класс TBitmap), Icon (значок, класс TIcon) или Metafile (класс TMetafile, формат графического метафайла *Windows* .EMF). Обратиться к любому из этих свойств для отображения графики можно через свойство Graphic. Ширина и высота изображения (в пикселах) задаются в свойствах Width и Height.

Основные методы класса TPicture (помимо ранее рассмотренных методов класса TGraphic) приведены в табл. 4.43.

Таблица 4.43. *Основные методы класса TPicture*

Метод	Назначение
class procedure RegisterClipboardFormat(AFormat: Word; AGraphicClass: TGraphicClass);	Регистрация нового графического формата данных для работы с ним через буфер обмена Windows, в результате чего объект класса TPicture сможет обращаться к данным в буфере обмена с помощью метода LoadFromClipboardFormat. С регистрируемым форматом связывается класс AgraphicClass
class procedure RegisterFileFormat(const AExtension, ADescription: string; AGraphicClass: TGraphicCLass);	Регистрация нового графического формата данных, записанного в файлах с расширением AExtension. При обращении к этому формату в стандартных диалоговых окнах Windows (Открыть и Сохранить) он отображается с названием, указанным в параметре ADescription (например, Мои картинки)
class procedure RegisterFileFormatRes(const AExtension: String; ADescriptionResID: Integer; AGraphicClass: TGraphicClass);	Метод аналогичен предыдущему, только вместо явного указания строки с названием рисунка задается идентификатор строки ресурсов
class function SupportsClipboardFormat(AFormat: Word): Boolean;	Проверка поддержки данного формата графических данных при передаче через буфер обмена Windows
class procedure UnregisterGraphicClass(AClass: TGraphicClass);	Отмена ранее зарегистрированного графического класса AClass

Используемый далее метакласс TGraphicClass описывается так:

```
type TGraphicClass = class of TGraphic;
```

Это означает, что на его месте может присутствовать любой объект класса TGraphic или его наследник.

Событий, которые можно обрабатывать, у компонента TImage два. Это событие OnChange, возникающее, когда графическое содержимое изменилось, и событие OnProgress, возникающее при работе с некоторыми графическими форматами (в частности, с форматом JPEG), когда При обработке больших изображений надо извещать программу о текущем выполненном объеме работ.

Заголовок обработчика события OnProgress выглядит следующим образом.

```
procedure (Sender: TObject; Stage: TProgressStage;
    PercentDone: Byte; RedrawNow: Boolean;
    const R: TRect;
    const Msg: string);
```

Параметр Stage относится к перечислимому типу и может принимать одно из трех значений: psStarting (обработка начата), psRunning (обработка продолжается), psEnding (обработка завершена).

Параметр PercentDone содержит примерный объем обработанной части изображения в процентах.

Если полученную программой часть изображения (эта область описывается прямоугольником R) можно корректно нарисовать (или перерисовать, например при отображении форматов с чередованием линий типа GIF), то параметр RedrawNow будет иметь значение True.

В параметре Msg хранится некоторая строка, словесно характеризующая этап обработки изображения (например, Loading).

ВНИМАНИЕ Чтобы изображение действительно перерисовывалось, надо в свойстве IncrementalDisplay объекта Image задать значение True.

После размещения объекта Image на форме появится пунктирная рамка, которая задает (по умолчанию) размеры будущей картинки. Эти размеры желательно заранее указать в свойствах Width и Height.

Выбрав в Инспекторе объектов свойство Picture, можно вызвать специальный редактор, с помощью которого можно загрузить изображения в форматах .BMP, .ICO, .JPG или в одном из форматов графического метафайла *Windows*.

ЗАМЕЧАНИЕ Этот редактор был описан при описании компонента TSpeedButton.

Если класс TPicture предназначен только для загрузки, хранения и сохранения изображения, то с помощью класса TImage с изображением (из свойства Picture) можно осуществлять определенные манипуляции.

Прежде всего, в классе TImage имеется свойство Canvas (холст), с помощью которого можно выводить различную графическую информацию непосредственно на изображение. С помощью свойства Center картинку можно центрировать внутри заданной рамки (если она меньше рамки). Если свойство Center имеет значение True, рисунок центрируется, в противном случае его верхний левый угол совмещается с верхним левым углом рамки. Установив значение свойства Stretch равным True, можно включить режим автоматического растяжения или сжатия изображения в соответствии с положением границ рамки.

Некоторые области изображения можно сделать прозрачными, чтобы сквозь них «просвечивал» фон формы. Прозрачность определяется заданием значения True для свойства Transparent. Этот режим не применим к точечному изображению в формате .BMP.

Компонент Редактор списка строк (TValueListEditor)

Компонент TValueListEditor предназначен для создания списков строк, состоящих из пар «имя — значение». TValueListEditor напоминает компонент Таблица строк (TStringGrid), но он более простой.

Расположенный на форме объект TValueListEditor представляет собой таблицу из двух столбцов. В первом указывается имя, во втором — соответствующее ему значение. Название каждого столбца можно изменить в свойстве TitleCaptions.

На этапе проектирования исходная информация вводится в таблицу с помощью свойства Strings (для этого вызывается визуальный редактор пар значений). Строки, записываемые в это свойство во время работы программы, должны иметь формат «имя = значение» (например, 'X1 = 50').

С помощью свойства KeyOptions можно ограничить возможности данного компонента, например разрешить или запретить редактирование, добавление или удаление строк. Свойство Options дает возможность подробно настроить внешний вид объекта па форме.

Когда пользователь будет менять значение в строке данного объекта в работающей программе, ему можно предложить на выбор раскрывающийся список готовых значений. Для этого необходимо сформировать собственный обработчик события OnGetLookup. В качестве параметра KeyName передается очередное значение ключевого (первого) столбца. Программист может записать в параметр Values (тип TStrings) список соответствующих этому значению строк, которые отображаются в меню. Например:

```
procedure TForm1.ValueListEditor1GetLookup(
   const KeyName: String; Values; TStrings);
begin
   if KeyName = 'X1' then
      begin
         Values.Add('V1');
```

```
            Values.Add('V2');
        end;
    end;
```

Ячейка такого поля в правом столбце дополнительно сопровождается небольшой кнопкой-стрелкой меню. Для добавления новой строки служит метод:

```
function InsertRow(const KeyName, Value: string; Append:
Boolean): Integer;
```

для удаления — метод:

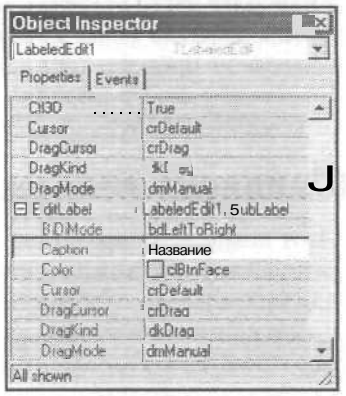
```
procedure DeleteRow(ARow: Integer);
```

а для поиска — метод:

```
function FindRow(const KeyName: string; var Row: Integer):
Boolean;
```

Компонент Текстовое поле с подписью (TLabeledEdit)

Весьма полезный компонент. В большинстве случаев поля ввода на форме сопровождаются подписями. В предыдущих версиях *Delphi* для этого требовалось создавать по отдельности поле ввода TEdit и надпись TLabel, которую надо было достаточно точно и аккуратно расположить вблизи поля ввода. Теперь, с появлением данного компонента, эта проблема решена.

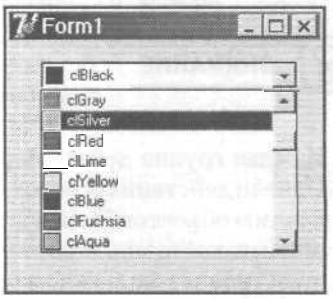


По своим свойствам Текстовое поле с подписью практически полностью совпадает с компонентом TEdit, но в нем появилось новое свойство EditLabel, представляющее собой вложенный компонент TLabel, который автоматически привязан к местоположению поля ввода.

Название (подпись) редактируется именно в этом вложенном объекте — в свойстве Caption.

Компонент Выбор цвета (TColorBox)

Компонент TColorBox представляет собой раскрывающийся список, в начале каждой строки которого вместе с названием цвета выводится небольшой окрашенный прямоугольник соответствующего цвета.



Текущий выбранный цвет хранится в свойстве Selected. С помощью подсвойства cbCustomColor (если его значение равно True) в свойстве Style пользователю разрешается задавать свой собственный цвет —

при выборе в списке строки Custom на экране возникает стандартное диалоговое окно выбора или задания нового цвета.

Компонент Панель действий меню (TActionMainMenuBar)

Данный компонент — контейнер групп действий — может использоваться только совместно с компонентом Менеджер действий (см. далее). **Он** предназначен для визуального отображения (значками) и быстрого доступа к группам наиболее часто используемых в приложении действий, как стандартных, так и определяемых программистом (порядок их расположения определяется Менеджером действий). Из диалогового окна Менеджера групп действия можно просто перетаскивать мышью на панель Меню действий.

Фон и внешний вид панелей можно задавать с помощью, соответственно, свойств Color и EdgeInner/EdgeOuter. Применение компонента рассмотрено в разделе «Менеджер действий».

Компонент Панель действий (TActionToolBar)

С помощью этого компонента можно создавать панели для быстрого доступа к наиболее часто встречающимся действиям — по аналогии с компонентом Панель действий меню. Различие состоит только во внешнем виде: данный компонент представляет каждое из действий в виде отдельной кнопки на панели. По способу своего создания и использования он практически ничем не отличается от компонента TActionMainMenuBar.

Компонент Менеджер действий (TActionManager)

Этот компонент представляет собой значительно усложненный и расширенный вариант компонента Список действий (TActionList). Он предназначен для визуального отображения и организации в группах наборов стандартных и пользовательских действий, которые предварительно определяются, например с помощью компонента Список действий.

Менеджер действий позволяет организовать логические группы действий, которые впоследствии могут быть повторно использованы в меню, навигационных панелях и панелях кнопок. Такие группы действий становятся доступными автоматически — по мере их добавления к объекту.

ВНИМАНИЕ Данный компонент работает только с панелями меню и инструментов, а также с навигационными панелями.

Каждая группа действий располагается на одной из панелей действий ActionBar. Панели действий сосредоточены в свойстве TAction Bars, представляющем собой коллекцию объектов TActionBar. В свою очередь, в классе TActionBar есть свойство Items, описывающее коллекцию (TActionClients) клиентских действий TActionClientItem. На этом уровне уже можно задать конкретное действие в свойстве Action (выбирая его

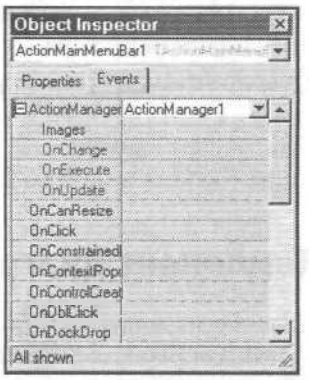
с помощью выпадающего списка) или построить более сложную древовидную иерархию групп действий путем реализации аналогичного свойства Items на вложенных уровнях.

Свойство PrioritySchedule позволяет располагать элементы меню или панелей инструментов в зависимости от частоты обращения к ним.

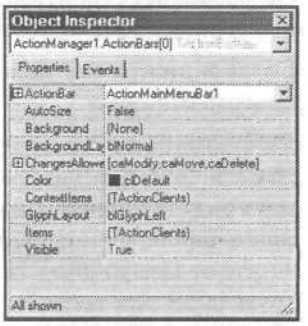
Рассмотрим следующий пример. На форме расположена кнопка Button1, при нажатии на которую в поле надписи Label1 выводится строка «Привет!». Эта функция реализована не в обработчике щелчка на кнопке, а в объекте ActionList1 (Список действий) в виде специального действия Action1 (обработчик события OnExecute), которое затем указывается в свойстве Action кнопки Button1.

```
procedure TForm1.Action1Execute(Sender: TObject);
begin
   Label1.Caption := 'Привет!'
end;
```

Следующий шаг — добавление Панели действий меню (компонент TActionMainMenuBar). В исходном состоянии на этом объекте ничего отображаться не будет, так как пока не добавлен Менеджер действий. А после того как Менеджер размещен на форме, в свойстве ActionManager Панели действий меню надо сослаться на этот новый компонент.

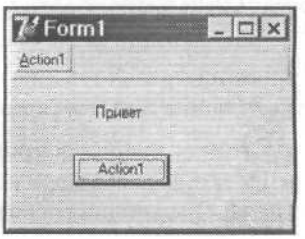


Далее в самом Менеджере с помощью свойства ActionBars (компонента панелей действий) необходимо добавить новый элемент TActionBar, в котором в свойстве ActionBar надо указать ссылку на Панель действий меню.



Добавим новый элемент в свойстве-коллекции Items этого объекта и в его подсвойстве Action укажем нужное нам действие Action1 — реакцию на нажатие кнопки. На панели меню автоматически появится название этого действия.

Двойной щелчок на Менеджере действий вызывает диалоговое окно, в котором можно создавать или удалять уже существующие панели действий, добавлять стандартные действия, настраивать внешний вид меню и задавать горячие клавиши — все это с поддержкой технологии перетаскивания.



В дополнение к организации централизованной обработки всех событий в программе Менеджер действий позволяет, в частности, сохранять (и загружать) все свои настройки на диске в файле (методы SaveToFile и LoadFromFile).

Пользовательское действие TCustomAction дополнено новым свойством GroupIndex, которое позволяет формировать группы действий с одинаковым значением данного свойства.

Компонент Диалоговое окно настройки действий (TCustomizeDlg)

Практически все настройки панелей и меню действий, выполняемые на этапе проектирования, можно сделать доступными конечному пользователю и во время работы программы. Для этого предназначен специальный компонент TCustomizeDlg.

Экземпляр класса TCustomizeDlg просто располагается на форме и связывается с Менеджером действий с помощью свойства ActionManager.

Диалоговое окно настройки действий TCustomizeDlg по своему виду и возможностям полностью совпадает с диалоговым окном, вызываемым командой контекстного меню Action List Editor Менеджера действий. Чтобы его вызвать, надо в редакторе списка действий (см. раздел, посвященный компоненту TActionList) добавить стандартное действие Tools/TCustomizeActionBars, что выполняется комбинацией клавиш CTRL+INS или командой New Standard Action (Создать стандартное действие) контекстного меню редактора.

Таким образом, данный компонент позволяет произвольно менять внешний вид и содержимое панелей и меню действий программы. Сохранять пользовательские настройки внешнего вида программы можно в файлах — для этого служит свойство FileName и методы SaveToFile и LoadFromFile.

Компонент Диаграмма (TChart)

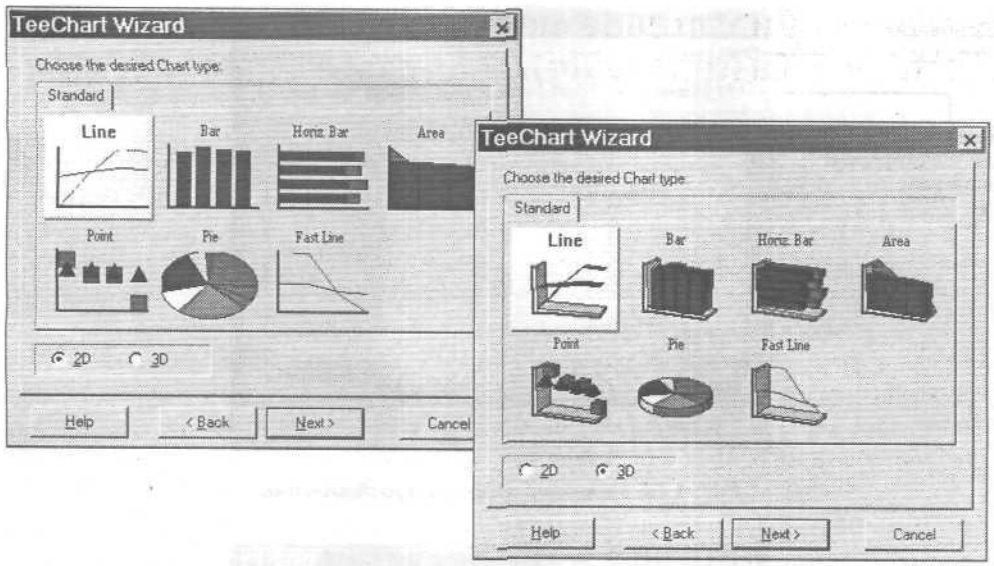
Это очень мощный и богатый возможностями компонент, разработанный Дэвидом Бернеда (версия, включенная в систему *Delphi 7,* имеет номер 4.02). Он позволяет строить красивые двух- и трехмерные диаграммы на основе различных данных, является наследником класса TPanel и наследует все свойства панели.

Создать диаграмму можно двумя способами: визуально с помощью Мастера (без программирования) и непосредственно средствами Паскаля.

Начало работы. Мастер запускается командой File ➤ New > Business ➤ TeeChartWizard (Файл ➤ Создать ➤ Деловые ➤ Мастер диаграмм), после чего разработчику надо выполнить ряд уточнений. Сначала выбирается источник данных. Пусть он не расположен в файле, а генерируется программой — переключатель Non Database Chart (Не на основе базы данных). Затем выбирается внешний вид диаграммы. Она может быть двумерной или трехмерной что определяется переключателем 2D/3D (рис. 4.9).

На следующем этапе работы Мастера флажок Show Legend (Отображать легенду) определяет наличие *легенды* — дополнительной панели, на которой указывается соответствие цветов частей диаграммы указанным значениям. Флажок Show Marks включает небольшие желтые подсказки у каждой из частей диаграммы.

На этом создание диаграммы заканчивается. После щелчка на кнопке Finish (Готово) в Проектировщике форм появится новая форма, на которой будет расположен объект Chart1. Он заполнен неким набором случайно сгенерированных значений (рис. 4.10),



РМС. *4.9. Допустимые виды двумерных и трехмерных диаграмм*

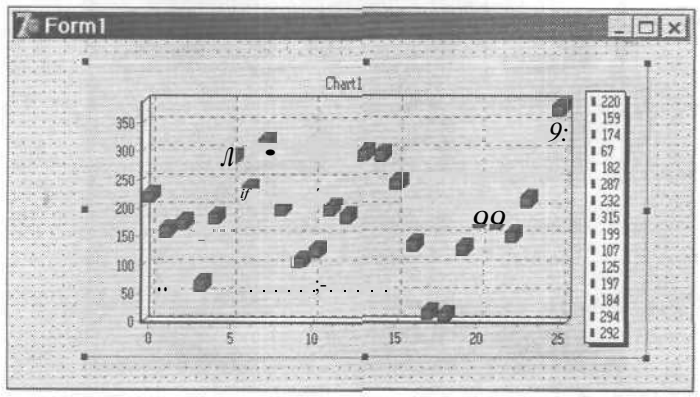


***Рис.** 4.10. Форма с автоматически сгенерированным объектом-диаграммой*

Настройка диаграммы выполняется с помощью редактора, который вызывается двойным щелчком на объекте Chart1 (рис. 4.11).

Параметры отображения диаграммы в окне определяются на вкладке Chart (Диаграмма), состоящей в свою очередь из набора дополнительных панелей.

- Панель Series (Рядданных) очень важна. Она позволяет объединять несколько диаграмм на одном графике с помощью кнопки Add (Добавить). При этом над значениями рядов данных можно выполнять различные операции, задаваемые на вкладке Functions (Функции): сложение (Add), вычитание (Subtract), умножение (Multiply), деление (Divide), взятие наибольшего (High), наименьшего (Low) или среднего (Average) значения (рис. 4.12).

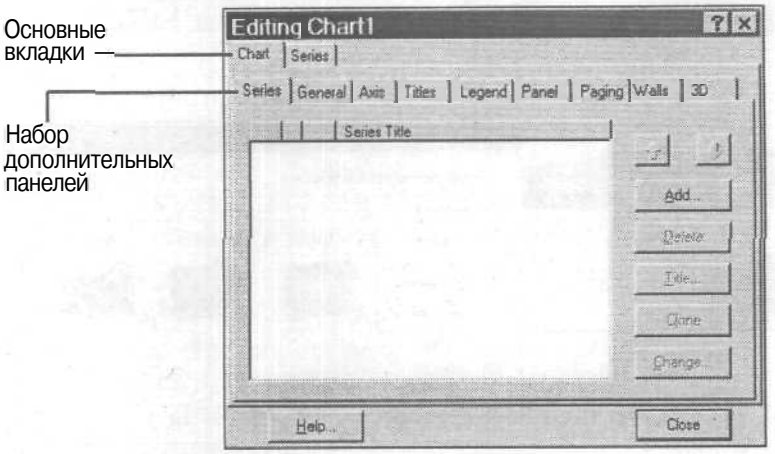


Рис. 4.11. Редактор содержимою диаграммы

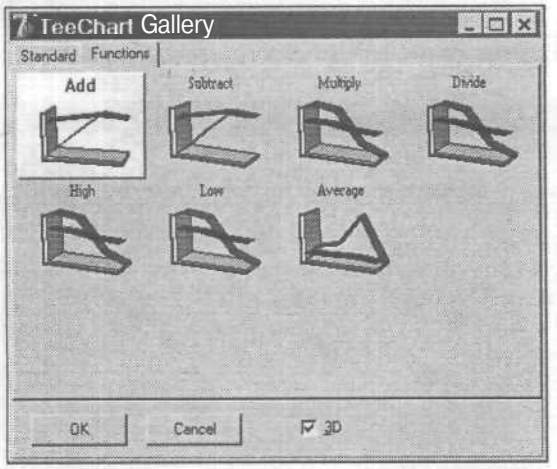
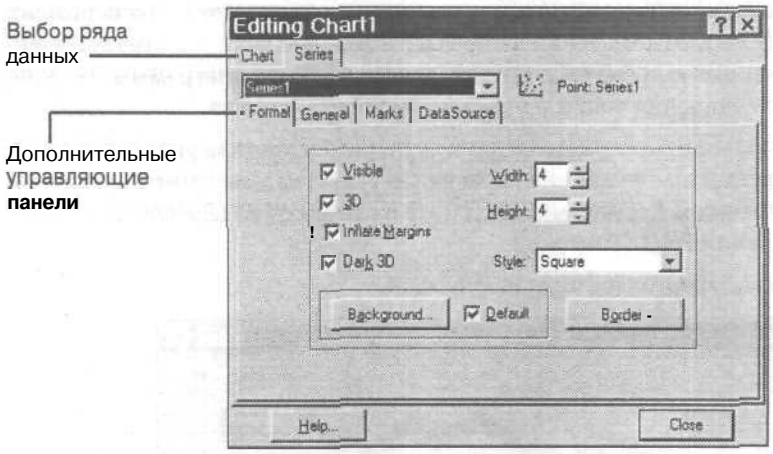


Рис. 4.12. Выбор операции, применяемой к нескольким рядам данных

- Панель General (Общие) содержит элементы управления для:
  - экспорта изображения в файл — кнопка Export (Экспортировать);
  - установки (в процентах) сдвига границ изображения по отношению к границам объекта — поля Margins (Поля);
  - масштабирования — панель Zoom (Масштаб);
  - прокрутки — панель Allow Scroll (Разрешить прокрутку).
- Средства панели Axis (Оси) отвечают за все, что касается определения координатных осей, их масштаба, заголовков, шага пунктирной сетки и так далее.
- Панель Titles (Заголовки) содержит средства для оформления заголовка.

- Панель Legend (Легенда) используется при оформлении внешнего вида и содержимого легенды.
- Средства панели Panel (Панель) описывают форму и визуальное представление панели-основы, на которой расположена диаграмма.
- Панель Pages (Страницы) служит для разделения диаграммы на страницы. Увеличивая число точек на странице с помощью поля Points per Page (Точки на страницу), можно подобрать оптимальное соотношение между наглядностью диаграмм и разумным числом страниц.
- Панель Walls (Границы) позволяет задать цвет и размеры границ диаграммы.
- Панель 3D описывает пространственное представление трехмерных диаграмм, С помощью нескольких движков проектируемую диаграмму можно вращать и масштабировать.

На вкладке Series (Ряды данных) в редакторе задаются конкретные параметры оформления каждого ряда данных (каждого графика, добавленного при помощи вкладки Chart). Выбор текущего ряда данных производится с помощью раскрывающегося списка Area (Область) (рис. **4.13).**



Рис, 4.13. Управление отображением рядов данных на диаграмме

Здесь наиболее важна панель Data Source (Источник данных). С ее помощью можно задать для ряда случайные значения (Random Values), отказаться от генерации значений (No Data) или сформировать значения текущего ряда данных как результат применения некоторой функции (раскрывающийся список Function) к значениям выбранных рядов данных. Выбор рядов данных — занесение в список Selected Series (Выбранные ряды) осуществляется с помощью кнопки >.

Программная работа с диаграммами. Рассмотрим пример создания трехмерной диаграммы и заполнения ее значениями непосредственно из программы.

На форме Form1 разместим компонент TChart и вызовем редактор. Это можно сделать также из контекстного меню объекта выбором пункта Edit Chart (Изменить диаграмму).

На панели Series (Ряд данных) вкладки Chart (Диаграмма) щелкните на кнопке Add (Добавить) и выберите подходящее трехмерное представление, например Point (Точечное). На форме появится диаграмма, заполненная случайными данными. На панели Titles (Заголовки) надо указать подходящее название диаграммы и закрыть редактор.

Диаграмма связывается с программным кодом очень просто. Большинство настроек, имеющих отношение к оформлению диаграммы, формируются в редакторе, а в программе (в разделе класса TForm1, где располагаются создаваемые в Проектировщике элементы управления) должен появиться новый объект — переменная Series1 типа TPointSeries. Она описывает последовательность значений, которые будут отображаться на диаграмме. Всю остальную работу система *Delphi 7* берет на себя — очень удобный и простой подход.

Рассмотрим основные свойства и методы класса TPointSeries (он является наследником базового класса TChartSeries, который служит основой для всех классов, описывающих содержимое конкретных типов **диаграмм).** Разработчику требуются такие возможности, как добавление и удаление точки, изменение некоторого значения, очистка всех точек, получение общего числа точек и доступ к их текущим значениям.

Пусть имеется диаграмма типа Point (Точечное представление), на которой должны располагаться условные значения результатов двух экспериментов («Эксперимент А» и «Эксперимент Б»). Эти значения вводятся с помощью двух текстовых полей, для их редактирования используется щелчок мыши на точке диаграммы. Нужны также возможности удаления точки и очистки текущего графика.

Так как требуется выводить результаты двух экспериментов (два ряда значений), надо добавить к текущей диаграмме еще один ряд. В редакторе диаграммы на панели Chart > Series (Диаграмма ➤ Ряд данных) щелкните на кнопке Add (Добавить) и выберите вид представления Point (Точечное).

Примерный вид такой формы показан на рис. 4.14.

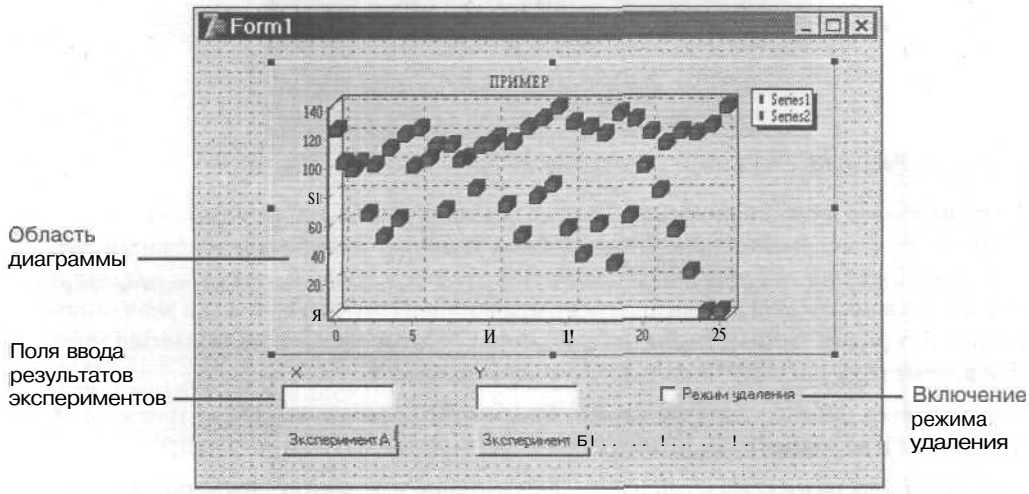
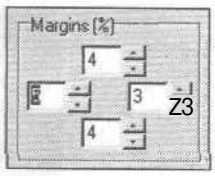


Рис. 4.14. Форма для построения диаграмм по результатам экспериментов



Чтобы подпись «Эксперимент ...» под крайней левой точкой диаграммы целиком помещалась на панели Chart1, можно немного сдвинуть левую границу области диаграммы вправо. Для этого на панели редактора Chart ➤ General (Диаграмма > Общие) можно задать значение 5% в левом поле на панели Margins (Поля).

Добавление новой точки к серии выполняется с помощью метода Add, заголовок которого выглядит следующим образом.

```
function AddXY(Const AXValue, AYValue: Double;
   Const AXLabel: String;
   AColor: TColor) : Longint;
```

Добавляемая точка задается параметрами AXValue и AYValue. Параметры AXLabel и AColor — необязательные. Первый описывает произвольное название группы, к которой будет принадлежать точка, второй — цвет этой группы. В нашем случае выберем красный цвет для точек эксперимента А, а цвет точек эксперимента Б сделаем синим. Функция возвращает позицию (номер) новой точки в свойстве XValues или YValues (массивы значений) в зависимости от того, по какому измерению добавляется на диаграмму точка.

Обработчик щелчка на кнопке ЭкспериментА (Button1) запишется следующим образом.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Series1.AddXY(
   StrToFloat(Edit1.Text),
   StrToFloat(Edit2.Text),
   'ЭкспериментА', clRed);
end;
```

Обработчик щелчка на кнопке Эксперимент Б (Button2) будет выглядеть так.

```
procedure TForm2.Button1Click(Sender: TObject);
begin
Series2.AddXY(
   StrToFloat(Edit1.Text),
   StrToFloat(Edit2.Text),
   'Эксперимент Б', clBlue);
end;
```

Теперь можно запустить программу, ввести в поля значения и добавить «результаты эксперимента» на диаграмму (рис. 4.15).

Чтобы удалить ранее введенную точку или изменить ее значение, надо предварительно определить ее номер в массиве Values. Для этого по каждому ряду данных (объекты Series1 и Series2) формируется обработчик события OnClickPointer. Его заголовок выглядит следующим образом.

```
procedure SeriesClickPointer(Sender: TCustomSeries;
   ValueIndex: LongInt; X, Y: Integer);
```

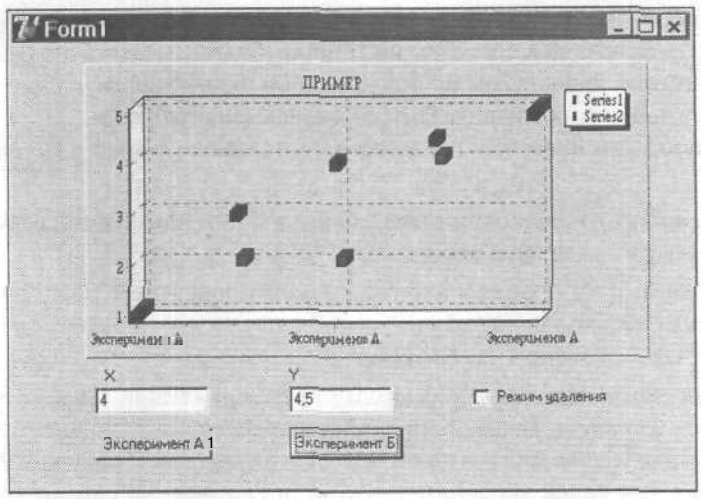


Рис. 4.15. Программа для ввода значений и динамического построения диаграммы в работе

Наиболее важный параметр ValueIndex содержит номер ближайшей точки ряда, *около* которой на диаграмме был выполнен щелчок. X и Y — это координаты точки щелчка.

Поместим на форму новый элемент — флажок Режим удаления (назовем его DeleteBox). Когда он установлен, выбираемые точки будут удаляться (после уточняющего запроса). В противном случае текущая точка будет корректироваться в соответствии со значениями, указанными в полях ввода.

ЗАМЕЧАНИЕ Можно было бы обрабатывать щелчки и с помощью события OnClick, в котором кроме номера точки передается и состояние кнопок мыши. Анализируя это состояние, можно выполнять удаление точки по *нажатию* правой кнопки мыши, а *корректировку* значения — по нажатию левой кнопки.

Удаление элемента из ряда данных осуществляется с помощью метода Delete, имеющего единственный параметр — номер элемента. Изменение текущего значения и положения на диаграмме выполняется простым изменением содержимого соответствующих элементов массивов ValueX и ValueY. Чтобы сделанные изменения отобразились на диаграмме, надо вызвать метод Repaint (Перерисовать) для соответствующего ряда данных.

Чтобы не дублировать одинаковый текст обработчиков щелчка на двух рядах точек, добавим в класс TForml метод, который будет получать в качестве параметров номер ряда данных и индекс точки, после чего выполнять все необходимые действия. Введем в часть public класса TForml заголовок такой процедуры.

```
procedure SerieClick( SNum: Integer; Index: Longint ),-
```

Теперь достаточно установить на имя данного метода указатель мыши и выбрать в контекстном меню пункт Complete class at cursor (Завершить реализацию класса).

В части реализации модуля сразу появится пустая реализация данной процедуры. В нее надо добавить проверку состояния флажка DeleteBox и в зависимости от этого состояния выполнить либо удаление точки, либо корректировку ее значения (перерисовка диаграммы осуществится автоматически).

```
procedure TForm1.SerieClick(SNum: Integer; Index: Integer);
begin
if DeleteBox.Checked then
   begin
   if SNum = 1 then Series1.Delete(Index)
   else Series2.Delete(Index)
   endelse
   begin
   if SNum = 1 then
      begin
      Series1.XValues[Index] := StrToFloat (Edit1.Text);
      Series1.YValues[Index] := StrToFloat (Edit2.Text);
      Series1.Repaint ,•
      end else
      begin
      Series2.XValues [Index] := StrToFloat (Editl .Text);
      Series2.YValues[Index] := StrToFloat (Edit2.Text);
      Series2.Repaint;
      end
   end
end;
```

Тогда обработчики щелчка на точках каждого ряда данных запишутся следующим образом.

```
procedure TForm1.Series1ClickPointer(Sender: TCustomSeries;
   ValueIndex, X, Y: Integer);
begin
SerieClick( 1,ValueIndex );
end;
procedure TForm1.Series2ClickPointer(Senderr TCustomSeries;
   ValueIndex, X, Y: Integer);
begin
SerieClick( 2,ValueIndex );
end;
```

Теперь программа позволяет с помощью щелчков удалять лишние точки и корректировать положение точек, введенных ранее.

Для удаления всех значений в ряду данных служит метод Clear.

```
Series1.Clear;
```

Вызов этого метода можно вставить, например, в обработчик щелчка па новой кнопке Очистить эксперимент А и так далее.

Компоненты Стандартная карта цветов, Карта цветов в стиле Windows XP и Черно-белая карта цветов (TStandardColorMap, TXPColorMap, TTwilightColorMap)



С помощью данных компонентов разработчик может задавать оригинальные цвета меню и панелей горячих кнопок своей программы. Конкретные цвета всех мыслимых характеристик этих элементов определяются подробными свойствами, соответствующими различным цветовым настройкам.

Карта цветов просто помещается на форме. Это невизуальный компонент. Он связывается с конкретной панелью через свойство ColorMap компонентов TActionMainMenuBar и TActionToolBar.

В процессе работы можно изменять значения различных свойств Карты цветов, на лету меняя цветовую палитру программы.

Панель Win32

Для понимания работы ряда рассматриваемых компонентов потребуется предварительно познакомиться с двумя новыми абстрактными классами TList и TCollection.

Класс Список (TList)

Мы уже рассматривали стандартный класс *Delphi 7* с именем TStrings, позволяющий работать со списком строк. В системе имеется и другой стандартный абстрактный класс TList. Он представляет собой массив указателей на объекты произвольных типов. С помощью класса TList можно добавлять в список новые объекты, удалять их, находить, сортировать и переупорядочивать.

Свойства и методы класса TList перечислены в табл. 4.44 и 4.45.

Таблица4.44. *Свойства класса TList*

Свойство	Назначение
Capacity	Размер массива указателей, определяющий отводимую для него память
Count	Текущее число элементов в списке
Items	Свойство для обращения н конкретному объекту по номеру
List	Массив указателей на объекты. Лучше использовать свойство Items

Таблица 4.45. *Методы класса TList*

Метод	Назначение
function Add(Item: Pointer); Integer;	Добавление нового элемента
procedure Clear;	Удаление всех элементов
procedure Delete(Index: Integer);	Удаление конкретного элемента

Таблица 4.45. Методы класса TList (продолжение)

Метод	Назначение
procedure Exchange(Index1, Index2: Integer);	Обмен двух элементов местами
function Expand: TList;	Расширение объема списка на 4-8-16 элементов в зависимости от его текущего размера
function IndexOf(Item: Pointer): Integer;	Номер в списке элемента, на который ссылается указатель Item. Нумерация начинается с нуля
procedure Insert(Index: Integer; Item: Pointer);	Вставка нового элемента
procedure Move(CurIndex. NewIndex: Integer);	Перемещение элемента из позиции CutIndex в позицию NewIndex
procedure Pack;	Удаление всех элементов, имеющих значение nil, то есть удаление пустых элементов, не ссылающихся на объекты
function Remove(Item: Pointer): Integer;	Удаление элемента, который определяется первым совпадением параметра Item (указатель) со значением указателя в списке

Для сортировки элементов списка применяется процедура Sort В качестве единственного параметра она получает ссылку на функцию, которая выполняет действия по сравнению значений двух объектов. Это связано с тем, что заранее не известно, объекты какой структуры будут храниться в списке.

```
type TListSortCompare =
   function (Item1, Item2: Pointer): Integer;
procedure Sort(Compare: TListSortCompare);
```

Приведем небольшой пример создания и сортировки списка. Пусть на форме расположены два объекта: список ListBox1 и кнопка Button1. При щелчке на кнопке будет создан список элементов типа TMyItem.

```
type TMyItem =
   record
   S: string;
   Ind: integer
   end;
   PMyItem = ^TMyItem;
```

Тип PMyItem — это указатель на тип TMyItem. Поле S заполняется случайными значениями из 10 символов, а поле Ind примет случайное значение от 1 до 1000. Затем этот список сортируется по возрастанию значений поля Ind, его содержимое выводится в объект ListBoxl, и, в заключение, содержимое списка вместе с его элементами будет ликвидировано. Предварительно надо описать функцию для определения, какой из объектов больше, которая будет использоваться при сортировке. Она должна возвращать значение -1, если первый объект меньше второго, 0, если объекты равны, и +1, если первый объект больше. Так как параметры функции имеют тип Pointer, в теле этой подпрограммы их надо явно приводить к типу PMyItem. Объекты будут сравниваться по значениям полей Ind.

```
function Compare(Item1, Item2: Pointer): Integer;
begin
if PMyItem(Item1)^.Ind < PMyItem(Item2)^.Ind
then Result := -1 else
if PMyItem(Item1)^.Ind > PMyItem(Item2)^.Ind
then Result := +1
else Result := 0
end;
```

Основная процедура (обработчик нажатия кнопки) будет выглядеть так.

```
procedure TForm1.Button1Click(Sender: TObject);
   var List: TList;
   P: PMyItem;
   i, j: integer;
begin
// создание списка:
List := TList.Create;
try
for i := 1 to 10 do
   begin
   // создание нового элемента:
   New(P);
   // заполнение поля S случайной строкой
   // (значение $40 соответствует пробелу)
   P^.S := '';
   for j := 1 to 10 do
      P^.S := P^.S + chr( $40 + random(20) );
   // случайное значение для поля Ind:
   P^.Ind := random(1000);
   // добавление элемента в список:
   List.Add(P);
   end;
// выполнение сортировки:
List.Sort(Compare);
// заполнение списка ListBox1:
for i := 0 to 9 do
   begin
   P := List.Items[i];
   ListBox1.Items.Add(IntToStr(P^.Ind) + ' ' + P^.S);
   end;
finally
// удалить из памяти список
// и все его элементы:
List.Free;
end;
```

Класс Коллекция (TCollection)

Класс TCollection (Коллекция) напоминает класс TList и предназначен для работы с элементами одинаковой структуры — указателями конкретного типа. В этом состоит отличие от класса TList, в котором указатели могут ссылаться на произвольные данные. В результате эффективность обработки хранимой информации повышается. Коллекция хранит объекты типа TCollectionItem. Эти объекты обладают свойствами, описанными в табл. 4.46. Свойства и методы самого класса TCollection описаны в табл. 4.47 и 4.48.

Таблица 4.46. *Свойства объектов TCollectionItem*

Свойство	Назначение
Collection	Ссылка на коллекцию, где хранится данный объект
DisplayName	Текст, отображаемый для данного объекта в редакторе коллекции. По умолчанию, это название конкретного класса объекта, наследника класса TCollectionItem
ID	Уникальный цифровой идентификатор каждого объекта в коллекции. Он может меняться, например при удалении элементов
Index	Номер элемента внутри коллекции — в массиве Items. Нумерация начинается с нуля

Таблица 4.47. *Свойства класса TCollection*

Свойство	Назначение
Count	Число элементов в коллекции
ItemClass	Тип хранимого класса. Это свойство описано следующим образом: type TCollectionItemClass = class of TCollectionItem;
Items	Массив элементов коллекции с возможностью доступа к ним с помощью индекса

Таблица 4.48. *Методы класса TCollection*

Метод	Назначение
function Add: TCollectionItem;	Создание нового элемента и добавление его в конец коллекции
procedure BeginUpdate;	Приостановка процесса перерисовки коллекции. Применяется во время длительных операций переформирования содержимого коллекции во избежание расхода времени На интенсивную перерисовку экрана
procedure EndUpdate;	Разрешение перерисовки коллекции
procedure Clear;	Удаление всех элементов из коллекции И уничтожение самих элементов
procedure Delete(Index: Integer);	Удаление конкретного элемента
function FindItemID(ID: Integer): TCollectionItem;	Возвращает элемент коллекции с идентификатором ID
function Insert(Index: Integer): TCollectionItem;	Создание нового элемента и его добавление в указанную позицию коллекции

Для работы с коллекциями в *Delphi 7* имеется специальный редактор. Он встроен в ряд компонентов системы и позволяет выполнять единообразное формирование и редактирование коллекции.

Работает этот редактор так. Создание нового элемента происходит по щелчку на кнопке Add New (Добавить новый элемент) или по нажатию клавиши INSERT. В списке появляется новый элемент с указанием его порядкового номера и имени. В Инспекторе объектов при этом показываются свойства конкретного объекта, являющегося наследником класса TCollectionItem. Они зависят от используемого компонента. Например, при работе со строкой состояния отдельные панели имеют тип TStatusPanel. Они позволяют настраивать свой вид и вместе с тем входят в состав коллекции, что дает возможность быстро менять их порядок и получать доступ к их свойствам.

Название элемента в редакторе совпадает со значением свойства Caption (Заголовок) или Text (Текст) конкретного объекта. При изменении этого свойства изменится и название в списке.

С помощью редактора может быть добавлено любое число элементов. Их разрешается выделять поодиночке и группой (при нажатой клавише CTRL).

Выделенный элемент или группа элементов удаляются с помощью кнопки Delete Selected (Удалить выделенные элементы) или клавиши DELETE. Порядок элементов в списке и, соответственно, на родительском объекте (например, порядок панелей в строке состояния) формируется кнопками Move Selected Up (Переместить выделенные элементы вверх) и Move Selected Down (Переместить выделенные элементы вниз). Эквивалентные комбинации клавиш — CTRL+ВВЕРХ и CTRL+ВНИЗ соответственно.

Поддержка Стандартных элементов управления Windows XP

Система *Delphi* автоматически поддерживает как старый набор стандартных элементов версии 5 (для *Windows95),* так и новый, версии 6, выпущенный специально для *WindowsXP.* Если *Delphi*-программа запускается в *WindowsXP,* то элементы управления могут принять вид, соответствующий нормам этой операционной системы. Для этого необходимо в каталоге, откуда запускается программа, разместить текстовый файл-манифест с названием

```
полное-имя-программы.manifest
```

Например:

```
MySoft.exe.manifest
```

Этот файл представляет собой *XML*-документ следующей структуры:

```
<?xml version='1.0' encoding-'UTF-8' standalone='yes'?>
<assembly xmlns='urn:schemas-microsoft-com:asm.v1'
manifestVersion='1.0'>
```

```
<assemblyIdentity
    version='1.0' encoding'
    processorArchitecture='X86'
    name='ИмяКомпании.НазваниеПродукта.Приложение'
    type='win32'
/>
<description>Описание приложения.</description>
<dependency>
    <dependentAssembly>
        <assemblyIdentity
            type='win32'
            name='Microsoft.Windows.Common-Controls'
            version='6.0.0.0'
            processorArchotecture='X86'
            publicKeyToken='6595b64144ccfldf'
            language='*'
        />
    </dependentAssembly>
</dependency>
</assembly>
```

Более подробную информацию о манифесте *Windows XP* можно найти в справочных руководствах *Microsoft.*

Компонент Набор страниц (TPageControl)

Компонент представляет собой набор страниц, наложенных одна на другую. Доступ к каждой странице, содержащей свой набор элементов управления, осуществляется через так называемые корешки — небольшие выступы над страницей, содержащие короткое название. Большинство настроечных диалоговых окон в различных программах для *Windows* сегодня созданы именно по такому принципу. Это И окно Параметры в редакторе *Word,* и окно Свойства обозревателя в браузере *Internet Explorer,* и всевозможные средства настройки системы *Delphi 7* и прочее. Данный элемент управления хорош в первую очередь тем, что позволяет эффективно экономить экранное пространство, фактически неограниченно увеличивая его «глубину».

Первоначально компонент, помещенный на форму, будет пустым — не содержащим ни одной страницы. Новая страница добавляется командой New Page (Создать стрэ-

ницу) из контекстного меню. При этом в списке объектов в Инспекторе объектов (и в описании класса TForm1) появляется описывающий ее новый объект TabSheet1.

Добавим таким образом две страницы. На этапе проектирования между ними можно переключаться простым щелчком мыши на корешке. В клиентской области каждой страницы можно размещать любые компоненты *Delphi 7*.

Названия, указанные на закладках, меняются так. Сначала выбирается нужная страница (но не объект PageControl1!) при помощи щелчка на ее клиентской части или выбором в списке Инспектора объектов. Затем нужное название вводится в свойство Caption (Заголовок), например Лист 1 и Лист 2 (рис. 4.16).

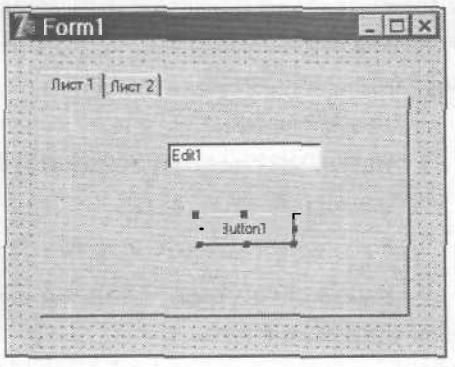


Рис. 4.16. *Страницы вкладок в процессе формирования формы*

Рассмотрим такой пример. На первой странице располагается кнопка, на второй — текстовое поле. При щелчке на кнопке в этом поле отображается строка «Привет!»

После размещения объектов на страницах вкладок (обратите внимание, как при переходе от вкладки к вкладке пропадают одни элементы управления и появляются другие) надо определить обработчик щелчка на кнопке. В нем будет только один оператор присваивания, который обратится к объекту TabSheet2 (второй странице) и запишет в находящееся на ней поле строку «Привет!»

ВНИМАНИЕ Реально все объекты, размещаемые на страницах компонента TPageControl, считаются принадлежащими непосредственно родительской форме — классу TForm1, поэтому явно указывать страницы при обращении к этим объектам не обязательно.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Edit1.Text := 'Привет!';
end;
```

При работе данной программы после щелчка на кнопке переходить на страницу Лист 2 приходится вручную. Это неудобно, поэтому добавим команду автоматического переключения на другую страницу. Для этого применяется свойство

ActivePageIndex родительского объекта PageControl1, которое содержит номер открытой страницы (нумерация начинается с нуля).

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Edit1.Text := 'Привет!';
PageControl1.ActivePageIndex := 1;
end;
```

Имеется также свойство ActivePage, которое содержит не номер активной (видимой) страницы, а непосредственно ссылку на нее. Для переключения страниц можно использовать и это свойство, если заранее известно, какую страницу надо показать:

```
PageControl1.ActivePage := TabSheet2;
```

Текущее число страниц хранится в свойстве PageCount, а массив самих страниц — в свойстве Pages, которое не предназначено для изменения. Например, чтобы поменять название первой страницы в процессе работы программы, можно использовать следующий оператор.

```
PageControl1.Pages[0].Caption := 'Новый заголовок';
```

Кроме того, имеется еще ряд свойств, методов и событий, унаследованных от родительского класса TCustomTabControl (наследником этого класса является и компонент TTabControl). Они перечислены в табл. 4.49–4.51.

Таблица 4.49. *Свойства класса TCustomTabControl, наследуемые классом TPageControl*

Свойство	Назначение
HotTrack	Имеет значение True, если текст корешка при наведении указателя выделяется ярким цветом
Images	Список картинок, которые отображаются на корешках вместе с текстом. Номер картинки в списке соответствует порядковому номеру страницы в объекте
MultiLine	Имеет значение True, если корешки разрешено отображать в несколько строи
OwnerDraw	Имеет значение True, если объект выполняет отрисовку самостоятельно
RaggedRight	Имеет значение True, если корешки разрешено сжимать в зависимости от ширины объекта
ScrollOpposite	Определяет, как будет перемещаться строка корешков, когда корешки отображаются в несколько строк и выбран корешок из другой строки. Принимает значение True, если строка перемещается в зависимости от значения свойства TabPosition, или False, если строка перемещается в нижний ряд
Style	Стиль объекта. Возможные значения — tsTabs (стандартный вид), tsButtons (корешки имеют вид кнопок), tsFlatButtons (корешки имеют вид плоских кнопок)
TabHeight	Высота корешка в пикселах
TabPosition	Определяет, где отображаются корешки- Возможные значения — tpTop (в верхней части объекта), tpBottom (в нижней части объекта), tpLeft (в левой части объекта), tpRight (в правой части объекта)
TabWidth	Ширина корешка в пикселах

Таблица 4.50. Методы класса TCustomTabControl, наследуемые классом TPageControl

Метод	Назначение
function GetHitTestInfoAt(X, Y: Integer): THitTests;	Определяет, к какой части элемента управления относится указанная точка (в координатах клиентской области)
function IndexOfTabAt(X, Y: Integer): Integer;	Номер «орешка, которому принадлежит указанная точка (или –1)
function RowCount: Integer;	Число строк корешков
procedure ScrollTabs(Delta: Integer);	Прокрутка корешков, если многострочный режим не поддерживается. Величина (число корешков) прокрутки указывается в параметре Delta. Прокрутка ведется в правую сторону, если значение параметра положительно, и в левую — в противном случае
function TabRect(Index: Integer): TRect;	Прямоугольник, содержащий координаты (границы) корешка с номером Index

Таблица 4.51. События класса TCustomTabControl, наследуемые классом TPageControl

Событие	Условие генерации
OnChange	Переход к новой вкладке
OnChanging	Пользователь выбрал новую вкладку, но процесс переключения страниц еще не произошел. Обычно обрабатывается, если надо сохранить (запомнить) состояние элементов управления старой страницы или установить начальные состояния элементов управления новой страницы в зависимости от некоторых условий
OnDrawTab	Программная отрисовка корешка
OnGetImageIndex	Генерируется перед отрисовкой корешка с картинкой. Обработка этого события позволяет динамически задавать картинку во время работы программы

При работе с компонентом TPageControl имеется возможность добавления к нему новых страниц «на лету», во время работы программы. Для этого надо создать новую страницу (класс TTabSheet) и в ее свойстве PageControl указать имя переменной — родительского объекта. Вся остальная привязка к родителю произойдет автоматически.

Например:

```
procedure TForm1.Button1Click(Sender: TObject);
var T: TTabSheet;
begin
T := TTabSheet.Create(Self);
T.Caption := 'New';
T.PageControl := PageControl1;
end;
```

После щелчка на кнопке Button1 на экране появится новая страница с корешком New.

Компонент Переключаемые страницы (TTabControl)

Данный компонент, как и компонент TPageControl, является наследником класса TCustomTabControl. Он имеет общие черты с набором страниц, однако главное его отличие в том, что, несмотря на наличие нескольких корешков, реально доступна только одна страница и только одна клиентская область. То есть, пользователь может переключать только корешки, а не страницы. При переключении корешков содержимое единственной страницы, естественно, не изменяется автоматически.

Данный компонент используется, когда надо реализовать особо сложную логику работы такого элемента управления. Контролировать содержимое каждой «виртуальной» страницы приходится программно, то есть динамически создавать и уничтожать или делать невидимыми различные группы элементов и графические изображения в моменты переключения между корешками.

Оригинальных свойств, методов и событий данный компонент не имеет.

Компонент Список изображений (TImageList)

Этот компонент незаменим при создании приложений, в которых производится работа *с* графическими изображениями. Он позволяет хранить наборы картинок фиксированного размера, обращаться к ним по номерам и осуществлять вывод изображений на экран различными способами. В частности, при реализации различных анимационных эффектов может использоваться «прозрачный» цвет.

Перед загрузкой изображений в данный компонент (он невизуальный) надо настроить ряд его основных свойств (табл. 4.52). При работе со списком изображений прежде всего требуется определить, как его элементы отображаются на холсте: просто копируются или рисуются с учетом *маски,* определяемой цветом маскируемой области. Все точки, попавшие в эту область, либо вообще не выводятся, либо отображаются одним и тем же цветом.

Таблица 4.52. *Свойства компонента TImageList*

Свойство	Назначение
AllocBy	Число новых элементов, добавляемых, если при добавлении очередного изображения требуется расширение списка. Например, если значение свойства AllocBy равно 4 и список содержит четыре картинки, то при добавлении пятой длина списка увеличится до 8 элементов (на 4). С помощью данного свойства можно регулировать распределение памяти, если планируется активная работа программы с большим числом изображений
BkColor	Фоновый цвет рисунка. При выводе этим цветом заполняется маскируемая область
BlendColor	Цвет переднего плана. Используется, когда изображение рисуется как выделенное (например, имеющее фокус)
Count	Число картинок в списке

продолжение

Таблица 4.52. Свойства компонента TImageList (продолжение)

Свойство	Назначение
DrawingStyle	Способ вывода изображения на экран. Возможные значения — dsFocused (цвет картинки смешивается с цветом BlendColor на 25%), dsSelected (цвет картинки смешивается с цветом BlendColor на 50%), dsNormal (в качестве фонового цвета используется цвет BkColor). Если значение свойства равно clNone (цвет не задан), то изображение рисуется с использованием прозрачного цвета и маски. Значение dsTransparent указывает, что изображение рисуется прозрачным независимо от значения свойства BkColor
ImageType	Тип изображения. Возможные значения — itImage (картинка), itMask (маска)
Masked	Имеет значение True, если при выводе изображения используется маска. Маскируемая часть изображения либо выводится как прозрачная, либо заполняется цветом, указанным в свойстве BkColor
ShareImages	Имеет значение False, если область памяти, выделенная для списка картинок, освобождается после завершения работы программы. Значение True имеет смысл устанавливать, только если доступ к списку картинок выполняется нестандартным путем, например обращением к ресурсам другой программы

В свойствах Width и Height указываются ширина и высота одного изображения в пикселах. Все изображения в списке должны иметь одинаковые размеры.

Изображения удобнее всего загружать в список с помощью встроенного редактора. Он вызывается двойным щелчком на объекте ImageList (рис. 4.17).

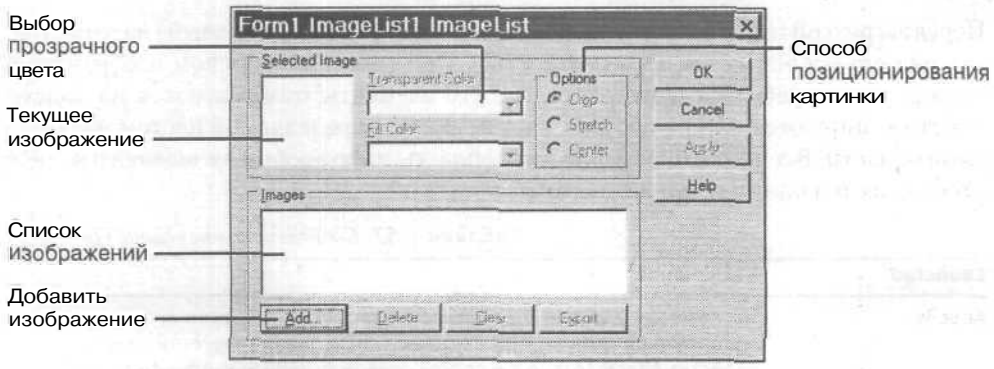


Рис. 4.17. Редактор списка изображений

Новые картинки добавляются с помощью кнопки Add (Добавить). Пока редактор не закрыт, добавленные изображения можно редактировать. Раскрывающийся список Transparent Color (Прозрачный цвет) позволяет выбрать цвет, который будет считаться прозрачным. Пикселы этого цвета не выводятся на холст. Если прозрачность не нужна, следует выбрать значение clNone.

На панели Options (Параметры) задается способ позиционирования картинки.

- Вариант Crop (Обрезка) означает, что изображение вкладывается в элемент списка, начиная от его верхнего левого угла.
- Вариант Stretch (Растяжение) указывает, что изображение сжимается или растягивается, чтобы соответствовать параметрам ширины и высоты элемента списка.
- Вариант Center (По центру) позиционирует изображение по центру допустимого прямоугольника. Если оно в нем не помещается, то выступающие части обрезаются.

Основные методы класса TImageList перечислены в табл. 4.53.

Таблица 4.53. Основные методы класса TImageList

Метод	Назначение
function Add(Image, Mask: TBitmap): Integer;	Добавление в список изображения и его маски (черно-белого точечного изображения, где белый цвет обозначает прозрачную область). Возвращаемое значение — номер изображения в списке. Нумерация начинается с нуля
function AddIcon(Image: TIcon): Integer;	Добавление значка. Значки всегда рисуются прозрачными
procedure AddImages(Value: TCustomImageList);	Копирование всех изображений из другого объекта ImageList
function AddMasked(Image: TBitmap; MaskColor: TColor): Integer;	Аналогично методу Add, но вместо дополнительной картинки-маски задается цвет изображения, который считается прозрачным, подобно работе с редакторомсписка
procedure Clear;	Удаление всех изображений из списка
procedure Draw(Canvas: TCanvas; X,Y, Index: Integer; Enabled: Boolean=True);	Основная процедура вывода изображения с номером Index на холст Canvas в позицию (X Y). Способ отображения определяется значением свойства DrawingStyle. Если значение свойства Enabled равно False, то изображение будет рисоваться в сером «выключенном» стиле
function FileLoad(ResType: TResType; Name: string; MaskColor: TCotor): Boolean;	Загрузка изображения из файла ресурсов с именем Name. В параметре ResType указывается тип ресурса, в параметре MaskColor задается прозрачный цвет
procedure Get8itmap(Index: Integer; Image: TBitmap);	Получение элемента изображения с номером Index как объекта класса TBitmap. Результат возвращается во втором параметре, который должен быть переменной
procedure GetIcon(Index: Integer; Image: TIcon);	Получение элемента изображения (или его части) с номером Index как объекта класса TIcon
procedure GetImages(Index: Integer; Image, Mask: TBitmap);	Разделение изображения с номером Index на две части: цветное точечное изображение и монохромную маску. Эти изображения возвращаются во втором и третьем параметрах

продолжение

Таблица 4.53. Основные методы класса TImageList(продолжение)

Метод	Назначение
procedure Insert(Index:Integer; Image, Mask: TBitmap);	Добавление изображения и маски в конкретную позицию списка
procedure InsertIcon(Index: Integer; Image: TIcon);	Добавление значка в конкретную позицию списка
procedure InsertMasked(Index: Integer; Image: TBitmap; MaskColor: TColor);	Добавление изображения с маскируемым цветом в конкретную позицию списка
procedure Move(CurIndex, NewIndex: Integer);	Перестановка изображения в списке из позиции CurIndex в позицию NewIndex
procedure Replace(Index: Integer; Image, Mask: TBitmap);	Замена элемента списка на новое изображение (с маской)
procedure ReplaceIcon(Index: Integer; Image: TIcon);	Замена элемента списка (значка) на новое изображение
procedure ReplaceMasked(Index: Integer; NewImage: TBitmap; MaskColor: TColor);	Замена элемента списка на новое изображение (с маскируемым цветом)
function ResourceLoad(ResType: TResType; Name: string; MaskColor: TColor): Boolean;	Загрузка изображения из ресурса программы под названием Name. В параметре ResType указывается тип ресурса, в параметре MaskColor — прозрачный цвет

Из-за ошибки в библиотеке *Windows*Comctl32.dll на компьютерах, где установлены системы *Windows* с разными версиями этой библиотеки, в случае формирования списка на этапе проектирования данный компонент может работать некорректно. Поэтому лучше загружать изображения в список динамически. Это можно сделать следующим образом.

```
var B: TBitmap;
...
B := TBitmap.Create;
B.LoadFromFile('tmp.bmp'),-
Imagelist1.AddMasked(B, clWhite);
```

Данный компонент удобен, во-первых, тем, что позволяет с помощью метода Draw выводить на экран изображения с прозрачными областями, которые легко формируются или задаются, а во-вторых, элементы списка изображений активно применяются во многих других компонентах *Delphi 7*, например при создании списков элементов, деревьев и т. п.

Компонент Текстовый редактор (TRichEdit)

Текстовый редактор представляет собой стандартный элемент управления *Windows* Rich text edit control (Область форматирования текста), выполненный в виде компонента *Delphi 7* (как и большинство других элементов управления *Windows*). По сравнению с компонентом *TMemo* он обладает расширенными воз-

можностями, такими как форматирование отдельных абзацев текста, поддержка формата *RTF* и другими.

После размещения на форме данный компонент готов к работе. В свойстве Lines (тип TStrings) можно указать начальное содержимое панели редактора.

Основная особенность данного компонента — это возможность форматирования отдельных абзацев. Такое форматирование выполняется на основе двух свойств компонента: Paragraph, определяющего характеристики текущего абзаца текста, и SelAttributes, определяющего характеристики выделенного текста.

Свойство Paragraph имеет тип TParaAttributes. Основные свойства этого типа приведены в табл. 4.54.

Таблица 4.54. Основные свойства класса TParaAttributes

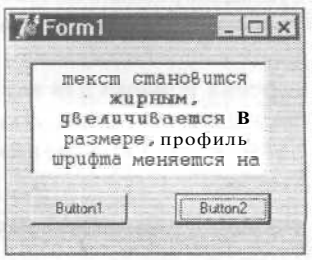
Свойство	Назначение
Alignment	Выравнивание. Возможные значения — taLeftJustify (по левому краю), taCenter (по центру) и taRightJustify (по правому краю)
FirstIndent, LeftIndent, RighUndent	Отступ (в пикселах) первой строки абзаца, а также поля между его левой и правой границами и границами панели редактора
Numbering	Превращение текста в список путем добавления с левой стороны маркеров. Такое форматирование происходит, если значение данного свойства устанавливается в nsBullet
Tab	Массив сдвигов (абсолютных позиций по отношению к левой границе), которые определяют, где будет останавливаться курсор при нажатии клавиши TAB
TabCount	Число элементов в массиве Tab

Свойство SelAttributes имеет тин TTextAttributes, состоящий, в свою очередь, из свойств, характерных для класса TFont: CharSet (набор символов), Color (цвет), Height (высота), Name (имя), Pitch (фиксированная/переменная ширина), Size (размер), Style (стиль).

Дополнительное подсвойство Protected, имеющее тин Boolean, позволяет сделать выделенный текст недоступным для редактирования.

Получить позицию курсора поможет метод GetCaretPos, скопировать выделенный текст в переменную — метод GetSelText. Компонент TRichEdit наследует также большинство методов, связанных с выделением текста и работой с буфером данных, от класса TCustomEdit, на основе которого создан и ранее рассмотренный компонент TMemo.

Возможности данного компонента продемонстрируем на следующем примере. На форме имеется панель редактора и две кнопки. По щелчку на первой выделенный текст выравнивается по центру и оформляется как список (с маркером в начале строки). По щелчку на второй кнопке выделенный текст становится полужирным, увеличивается в размере, оформляется шрифтом Courier и красным цветом.



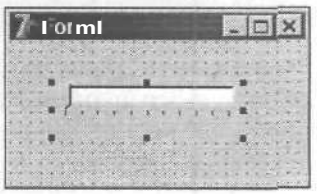
```
procedure TForm1.Button1Click(Sender: TObject);
begin
with RichEdit1 do
   begin
   Paragraph.Numbering := nsBullet;
   Paragraph.Alignment := taCenter;
   end
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
with RichEdit1.SelAttributes do
   begin
   Color r= clRed;
   Style := [fsBold];
   Size := 14;
   Name := 'Courier';
   end;
end;
```

Отличительная особенность компонента TRichEdit — возможность сохранения и загрузки форматированного текста в формате *RTF,* понимаемом всеми коммерческими редакторами. Чтобы указать объекту RichEdit1 на необходимость сохранения и загрузки содержимого в этом формате, надо значение свойства PlainText (Простой текст) сделать равным False.

Например, следующий обработчик щелчка на кнопке сохранит отформатированное содержимое объекта RichEdit1 в виде файла test.rtf.

```
procedure TForm1.Button3Click(Sender: TObject);
begin
RichEdit1.PlainText ;= false;
RichEdit1.Lines.SaveToFile('test.rtf');
end;
```

Компонент Движок (TTrackBar)



Движок обычно применяется там, где надо в визуальном режиме выставить с помощью мыши какое-либо приближенное значение, что выполняется перетаскиванием движка по шкале. В старых версиях *Windows* для этого использовались компоненты типа полос прокрутки. Движок стал стандартным элементом управления в 32-разрядных версиях *Windows.*

Внешний вид движка настраивается с помощью следующих свойств (табл. 4.55).

Таблица 4.55. Свойства компонента TTrackBar

Свойство	Назначение
Frequency	Частота засечек
Min Max	Минимальная и максимальная допустимые границы
Orientation	Ориентация: горизонтальная (значение trHorizontal) или вертикальная (значение trVertical)
SelStart SelEnd	Начало и конец «оптимального» диапазона в рамках границ Min/Max по аналогии с приборами управления. Область оптимального диапазона выделяется дополнительными засечками и другим цветом
SliderVisible	Видимость движка
ThumbLength	Толщина полосы движка в пикселах
TickMarks	Положение засечек. Возможные значения: tmBottomRight (снизу); tmTopLeft (сверху); tmBoth (с обеих сторон)
TickStyle	Способ отображения засечек на движке. Возможные значения: tsAuto (автоматически); tsManual (программно); tsNone (вообще не отображаются)

Основное свойство, определяющее положение движка, — это свойство Position. Его значение можно считывать и менять в процессе работы программы. Единственный полезный метод — процедура SetTick программной установки засечек.

```
procedure SetTick(Value: Integer);
```

Засечка ставится в точке шкалы движка, соответствующей значению Value.

При изменении значения свойства Position генерируется сообщение OnChange.

Компонент Индикатор (TProgressBar)

Индикаторы активно применяются во многих программах для отображения сведений о ходе длительного процесса (например, процесса инсталляции).

Свойства Min, Max, Position, Orientation аналогичны свойствам компонента TTrackBar. Необходимо также отметить свойство Smooth. Если оно имеет значение True, то в полосе заполнения (отсчет в ней ведется слева направо) отображается сплошная линия, в противном случае — сегментированная.

ПОДСКАЗКА Если требуются более гибкие возможности оформления индикатора, можно воспользоваться компонентом TGauge с панели Samples (Образцы), который позволяет дополнительно определять цвет индикатора и выводить процент заполнения.

Компонент Счетчик (TUpDown)

Данный элемент управления используется, как правило, в связке с другими элементами. Он дает возможность изменять числовые величины (например,

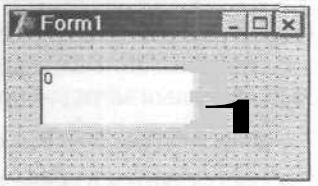
в текстовом поле) с помощью кнопок со стрелками и курсорных клавиш. При этом можно задать допустимые границы этих величин.

Текущее значение счетчика хранится в свойстве Position, а допустимые границы — в свойствах Min и Max. Свойство ArrowKeys разрешает или запрещает трактовать нажатия курсорных клавиш как команды изменения значения. Свойство Increment определяет шаг изменения значения свойства Position. Если свойство Thousands имеет значение True, то каждые три цифры в элементе управления, связанном с данным объектом, отделяются запятой.

Допускается циклическое изменение значения свойства Position. Если значение свойсгва Wrap равно true, то после превышения свойством Position максимального предела Max оно получает значение Min и наоборот.

Чаще всего данный компонент используют совместно с компонентом TEdit. Для этого на форме размещают компонент TEdit и в раскрывающемся списке его свойства Associate (Ассоциировать) выбирают объект UpDown1, когорый в результате автоматически присоединяется к полю.

После этого программу можно откомпилировать и запустить. Теперь при щелчках на стрелках объекта UpDown1 или при нажатии курсорных клавиш, когда фокус ввода установлен на текстовом поле Edit1, содержимое последнего будет меняться, отображая значение, хранящееся в свойстве Position.



ЗАМЕЧАНИЕ Подобным образом компонент TUpDown может присоединяться и к другим элементам управления, например кнопкам. В этом случае изменяемое значение свойства Position будет отображаться в качестве заголовка кнопки.

Сразу после щелчка на стрелке объект получает сообщение OnChanging, в котором в случае его обработки можно подтвердить или запретить изменение значения Position, записав, соответственно, значение True или False в параметр AllowChange. Еще одно сообщение OnChangingEx с помощью параметра обработчика Direction дополнительно уточняет, какая клавиша (ВВЕРХ или ВНИЗ) нажата. После завершения изменения значения Position генерируется сообщение OnClick.

Компонент Горячая клавиша (THotKey)

Данный весьма полезный компонент позволяет запросить у пользователя определенную «горячую» комбинацию клавиш, которая в дальнейшем будет использована для вызова часто выполняемого действия.

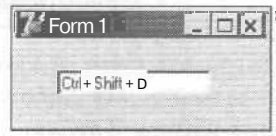
При создании экземпляра данного компонента предварительно надо определить «запрещенные» клавиши, нажатие которых одновременно с «горячей» комбинацией не учитывается. Допустим, что разработчика интересуют только комбинации двух

клавиш: обычной алфавитно-цифровой и управляющей CTRL. Тогда с помощью Инспектора объектов надо в свойстве InvalidKeys установить значения всех подсвойств, за исключением hcCtrl, равным True. Значения подсвойств, указанных ниже, используются и в других свойствах данного компонента.

Таблица 4.56. *Значения* ***подсвойств*** *свойства InvalidKeys*

Значение	Назначение
hcNone	Нажатие обычных клавиш без использования управляющих не допускается
hcShift	Клавиша SHIFT
hcCtrl	Клавиша CTRL
hcAlt	Клавиша ALT
hcShiftCtrl	Клавиши SHIFT+CTRL
hcShiftAlt	Клавиши SHIFT+ALT
hcCtrlAlt	Клавиши CTRL+ALT
hcShiftCtrlAlt	Клавиши SHIFT+CTRL+ALT
hkExt	Клавиша EXTRA

В свойстве Modifiers задаются допустимые управляющие клавиши.



Когда программа с настроенным компонентом THotKey запущена, при установке курсора в поле этого компонента и нажатии допустимой комбинации клавиш выводится описание этой комбинации. Во время работы программы доступ к введенной комбинации можно получить через свойство HotKey (тип TShortCut). Для получения текстового представления комбинации клавиш используется стандартная функция ShortCutToText().

Процедура ShortCutToKey позволяет выделить из комбинации виртуальный код клавиши (параметр Key) и состояние управляющих клавиш (параметр Shift).

```
procedure ShortCutToKey(ShortCut: TShortCut;
   var Key: Word; var Shift: TShiftState);
```

Чаще всего данный компонент применяется для быстрого выполнения действий, подготовленных с помощью компонента TActionList, или для доступа к действиям, привязанным к элементам меню. У соответствующих компонентов (TMenuItem, TAction) имеется свойство Shortcut, которому надо присвоить значение свойства HotKey.

Разместим на форме меню с единственным пунктом (дадим ему название F1), при выборе которого в объекте Label1 (надпись) будет выводиться строка, описывающая нажатые клавиши.

```
procedure TForm1.F1Click(Sender: TObject);
begin
Label1.Caption := ShortCutToText(HotKey1.HotKey);
end;
```

При щелчке на кнопке Button1 в свойство пункта меню Shortcut будет занесена текущая введенная комбинация клавиш объекта HotKey1.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
F1.ShortCut := HotKey1.HotKey;
end;
```

После компиляции и запуска программы, ввода комбинации клавиш в поле HotKeyl (например, CTRL+P) и щелчка на кнопке Button1 в надписи Label1 появится строка Ctrl+P.

Компонент Анимация AVI (TAnimate)

Данный компонент позволяет организовать на форме небольшую анимацию — воспроизвести клип в формате *A VI (Audio Video Interleaved,* расширение файла .AVI) без воспроизведения звука.

Б зависимости от цели применения компонент TAnimate можно настроить заранее, на этапе проектирования, или динамически, во время работы программы.

ВНИМАНИЕ Единственное обязательное условие: клип AV] должен быть подготовлен заранее. Способы создания таких клипов в книге не рассматриваются.

Загрузить клип можно, обратившись к свойству FileName, в котором указывается имя файла .AVI, либо указав в свойстве CommonAVI один из следующих стандартных клипов *Windows.*

Таблица 4.57. *Значения свойства CommonAVI*

Имя клипа	Содержание
aviNone	Клип указан в свойстве FileName
aviFindFolder	Поиск папки
aviFindFile	Поиск файла
aviFindComputer	Поиск компьютера
aviCopyFiles	Копирование файлов
aviCopyFile	Копирование файла
avi RecycleFile	Перемещение файла в корзину
avi EmptyRecycle	Очистка корзины
aviDeleteFile	Удаление файла

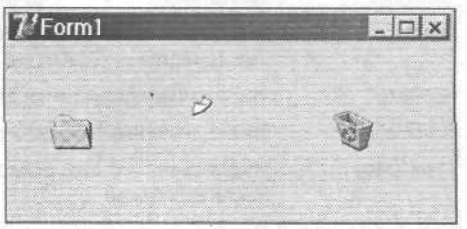
Стандартные клипы *Windows* имеют различный размер, поэтому для свойства AutoSize надо задать значение True, Свойство Transparent определяет, будет ли клип прозрачным или используется фоновый цвет.

Перед запуском клипа его надо открыть, для чего в свойство Open записывается значение True. При этом выполняется загрузка клипа в память и подготовка его к

воспроизведению. При открытии клипа генерируется сообщение OnOpen, в обработчике которого удобно задать начальные значения различных свойств.

Свойство FrameCount определяет число кадров в клипе, свойства FrameWidth и FrameHeight — ширину и высоту кадра в пикселах (эти размеры одинаковы для всех кадров), свойство Repetitions — число повторов клипа.

Синхронизировать демонстрацию клипа с сообщениями таймера поможет свойство Timers, если его значение сделать равным True. В противном случае влиять на процесс воспроизведения клипа, пока он не закончится, невозможно. В свойствах StartFrame и StopFrame задается диапазон реально отображаемых кадров клипа.



Чтобы начать демонстрацию клипа, надо для свойства Active установить значение True (при этом генерируется событие OnStart). Прервать показ можно с помощью метода Stop (событие OnStop), сделать текущим первый кадр — с помощью метода Reset, показать конкретный кадр — с помощью метода Seek. Нумерация кадров начинается с 1.

Вот пример воспроизведения клипа но щелчку на кнопке Button1.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Animate1.Open := true;
Animate1.Active := true;
end;
```

Компонент Календарь (TMonthCalendar)

С помощью календаря можно быстро и легко выбрать нужную дату с помощью мыши.

Сразу после размещения на форме календарь готов к работе (рис. 4.18).

Рис. 4.18. Календарь, размещенный на форме

В красном кружке выделена текущая дата (она же более развернуто указана в нижней части календаря), синим подсвечена дата, выбранная при помощи мыши. Слева показаны номера недель в году (свойство WeekNumbers). С помощью кнопок в верхней части календаря можно перемещаться по месяцам. Чтобы вернуться к месяцу с текущей датой, достаточно щелкнуть на ней в нижней части календаря.

Календарь имеет следующие свойства.

Таблица 4.58. *Свойства класса TMonthCalendar*

Свойства	Назначение
CalColors	Цвета элементов календаря
MaxDate	Максимальная дата, до которой календарь может быть прокручен помесячно. Тип свойства — TDate (си. ниже)
MinDate	Минимально допустимая для просмотра дата
MultiSelect	Имеет значение True, если разрешается выбирать диапазон дат
ShowToday	Имеет значение True, если текущая дата дополнительно отображается в нижней части календаря
ShowTodayCircle	Имеет значение True, если текущая дата выделяется красным кружком

Форматы даты и времени

Работа с календарем происходит на основе типа TDate (или его аналога TDateTime). Текущая выбранная дата записывается в свойство Date этого типа. Само значение хранится в запакованном формате в поле типа Double. Кроме типа TDateTime можно применять также системный формат TSystemTime, который содержит полное описание текущих даты и времени:

```
type TSystemTime = record
   wYear: Word;
   wMonth: Word;
   wDayOfWeek: Word;
   wDay: Word;
   wHour: Word;
   wMinute: Word;
   wSecond: Word;
   wMilliseconds: Word;
   end;
```

Можно использовать и формат TTimeStamp.

```
type TTimeStamp = record
   Time: Integer;
   Date: Integer;
   end;
```

В тюле Time хранится число миллисекунд, прошедшее с полуночи, в поле Date — число дней от 1 января 0001 года плюс 1.

Для работы с этими форматами в системе Delphi 7 имеется набор стандартных подпрограмм (табл. 4.59). Кроме того, в приложениях, где требуется выполнять задачи, жестко привязанные к временным промежуткам, можно воспользоваться функцией GetTickCount, возвращающей число миллисекунд, прошедших с начала суток (с полуночи) до данного момента, хотя лучше в подобных ситуациях применять компонент TTimer (см. далее).

Таблица 4.59. *Стандартные подпрограммы для работы с форматами даты и времени*

Подпрограмма	Назначение
function Date: TDateTime;	Возвращает текущую дату
function DateTimeToStr(DateTime: TDateTime): string;	Преобразование даты и времени в строку
procedure DateTimeToSystemTime(DateTime: TDateTime; var SystemTime: TSystemTime);	Преобразование даты/времени из формата TdateTime в формат TSystemTime
function DateToStr(Date: TDateTime): string;	Преобразование даты в строку
function DayOfWeek(Date: TDateTime): Integer;	Номер дня недели, число от 1 до 7 (1 — воскресенье, 7 — суббота)
procedure DecodeDate(Date: TDateTime; var Year, Month, Day: Word);	Выделение из даты года, месяца и дня
procedure DecodeTime(Time: TDateTime; var Hour, Min, Sec, MSec: Word);	Выделение из даты часов, минут, секунд и миллисекунд
function EncodeDate(Year, Month, Day: Word): TDateTime;	Преобразование явно заданных года, месяца и дня в дату формата TDateTime
function EncodeTime(Hour, Min, Sec, MSec: Word): TDateTime;	Преобразование явно заданных часов, минут, секунд и миллисекунд в дату формата TDateTime
function FormatDateTime(const Format: string; DateTime: TDateTime): string;	Преобразование даты/времени в строку в соответствии с собственным форматом (подробнее описание задания формата см. в справочной системе Delphi 7)
function IsLeapYear(Year: Word): Boolean;	Имеет значение True, если год високосный
function MSecsToTimeStamp(MSecs: Comp): TTimeStamp;	Перевод числа миллисекунд в формат TTimeStamp
function Now: TDateTime;	Возвращает текущие дату и время
function StrToDate(const S: string): TDateTime;	Преобразование строки, хранящей описание даты в формате, соответствующем локальным требованиям Windows, в дату формата TDateTime
function StrToDateTime(const S: string): TDateTime;	Преобразование строки, хранящей описание даты в формате, соответствующем локальным требованиям Windows, в дату и время формата TDateTime
function StrToTime(const S: string): TDateTime;	Преобразование строки, хранящей описание даты в формате, соответствующем локальным требованиям Windows, во время в формате TDateTime

продолжение

Таблица 4.59. Стандартные подпрограммы для работы с форматами даты и времени (продолжение)

Подпрограмма	Назначение
function SystemTimeToDateTime(const SystemTime: TSystemTime): TDateTime;	Преобразование времени из формата TSystemTime в формат TDateTime
function Time: TDateTime;	Возвращает текущее время
function TimeStampToDateTime(const TimeStamp: TTimeStamp): TDateTime;	Преобразование из формата TTimeStamp в формат TDateTime
function TimeStampToMSecs(const TimeStamp: TTimeStamp): Comp;	Преобразование значения времени в формате TTimeStamp в число миллисекунд
function TimeToStr(Time: TDateTime): string;	Преобразование времени в строку

Если выбран диапазон дат, то завершающая дата запишется в свойство EndDate. Для программной установки даты можно использовать метод MsgSetDateTime.

```
function MsgSetDateTime(Value: TSystemTime): Boolean;
```

Компонент Поле ввода даты/времени (TDateTimePicker)

Компонент представляет собой раскрывающийся список и используется для ввода даты и времени с клавиатуры (по формату в соответствии с локальными настройками *Windows*).При раскрытии списка открывается календарь (компонент TMonthCalendar).

На этапе проектирования настраиваются следующие свойства компонента.

Таблица 4.60. Свойства класса TDateTimePicker

Свойство	Назначение
DateFormat	Представление даты в коротком (dfShort) или длинном (dfLong) формате
DateMode	Выбор способа работы компонента. Он может использоваться как раскрывающийся список (dmComboBox) или иметь счетчик для изменения даты (dmUpDown)
Kind	Если значение свойства равно dtkDate, компонент используется для ввода даты, в противном случае (значение dtkTime) — для ввода времени
ShowCheckbox	Имеет значение True, если рядом с полем отображается флажок (его состояние можно проверить, обратившись к свойству Checked)

Когда дата или время в поле изменяются, генерируется сообщение OnChange. При раскрытии списка, содержащего календарь, объект получит сообщение OnDropDown, при закрытии — сообщение OnCloseUp.

Компонент Панель заголовков (THeaderControl)

Панель заголовков позволяет разместить на форме заголовки произвольных элементов. Порядок и размеры этих заголовков можно менять. Простой алго-

ритм работы данного компонента позволяет подстраивать размеры других объектов под размеры разделов заголовка.

В процессе проектирования компонента настраиваются следующие свойства.

Таблица 4.61. *Свойства класса THeaderControl*

Свойство	Назначение
DragReorder	Имеет значение True, если разрешено менять порядок разделов заголовка с помощью метода перетаскивания
FullDrag	Имеет значение True, если в ходе перетаскивания рисуется не граница раздела, а весь раздел целиком
HotTrack	Имеет значение True, если при наведении указателя на заголовок он выделяется другим цветом
Images	Ссылка на объект ImageList, содержащий картинки для разделов заголовка
Style	Вид разделов заголовка. Возможные значения — hsButtons (объемный), hsFlat (плоский)

Сами разделы задаются в свойстве Sections (класс THeaderSections, коллекция объектов класса THeaderSection) с помощью редактора, работа которого описывалась в разделе, посвященном классу TCollection.

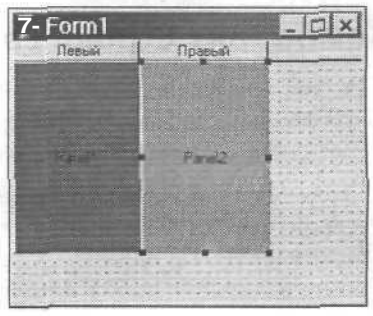
Каждый из разделов заголовка (THeaderSection) может работать как кнопка, реагирующая на щелчки, если свойство AllowClick имеет значение True. Номер картинки (если она требуется) указывается в свойстве ImageIndex, ширина раздела — в свойстве Width. Надпись (заголовок раздела) хранится в свойстве Text. Раздел можно рисовать программно — тогда в свойстве Style указывается значение hsOwnerDraw.

Использовать панель заголовков можно, только обрабатывая предназначенные для нее сообщения. Таких сообщений пять.

Таблица 4.62. *Сообщения класса THeaderSection*

Сообщение	Условие генерации
OnDrawSection	Перерисовка раздела, если значение его свойства Style равно hsOwnerDraw
OnSectionClick	Щелчок мыши на разделе
OnSectionDrag	Перетаскивание раздела в новую позицию
OnSectionResize	Произошло изменение размеров раздела
OnSectionTrack	Идет процесс изменения размера раздела

Создадим два раздела с заголовками Левый и Правый. Под каждым из заголовков разместим панель (компонент TPanel) так, чтобы ее размер совпадал с размером раздела заголовков. Укажем для левой панели цвет clRed, а для правой — clLime.



При изменении размеров раздела заголовка требуется, чтобы пропорционально менялся размер присоединенной панели, а при щелчке на заголовке должен меняться цвет панели. Для этого надо обработать события OnSectionTrack и OnSectionClick.

ЗАМЕЧАНИЕ Вместо события OnSectionTrack можно обрабатывать событие OnSectionResize, но это не позволит выполнять перерисовку панелей динамически, в процессе измерения размера.

```
procedure TForm1.HeaderControl1SectionTrack(
   HeaderControl: THeaderControl;
   Section: THeaderSection; Width: Integer;
   State: TSectionTrackState);
begin
// устанавливается новое значение ширины раздела:
Section.Width := Width;
// размеры панелей берутся из массива разделов
// HeaderControl1.Sections.Items:
Panel1.Width := HeaderControl1.Sections.Items[0].Width;
Panel2.Width := HeaderControl1.Sections.Items[1].Width;
// правая панель смещается:
Panel2.Left := HeaderControl1.Sections.Items[1].Left;
end;
procedure TForm1.HeaderControl1SectionClick(
   HeaderControl: THeaderControl;
Section: THeaderSection);
begin
// проверка, на каком разделе выполнен щелчок
// и какого он был цвета
// в зависимости от этого
// цвет меняется на новое значение:
if Section = HeaderControl1.Sections.Items[0] then
if Panel1.Color <> clBlue
   then Panel1.Color := clBlue
   else Panel1.Color := clRed
else if Panel2.Color <> clLime
   then Panel2.Color := clLime
   else Panel2.Color := clYellow
end;
```

Компонент Строка состояния (TStatusBar)

Редкая программа обходится без строки состояния, в которой указывается «горячая» подсказка и выводится дополнительная информация. Строка состояния обычно делится на несколько панелей.

Сразу после размещения компонента на форме строка состояния автоматически прикрепляется к нижней части формы. В дальнейшем она сама подстраивается под изменяемую ширину формы. Высоту строки подсказки можно менять либо с помощью мыши, либо с помощью свойства Height.

В самом простом варианте строка состояния работает, как одна большая панель. При этом свойство SimplePanel получает значение True, а выводимый текст записывается в свойство SimpleText. Характеристики шрифта, которым будут делаться надписи, задаются, как обычно, в свойстве Font, если значение свойства UseSystemFont (Использовать системный шрифт) равно False.

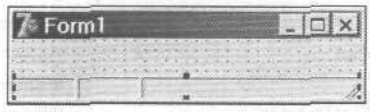
Текст в строке состояния можно привязать к текущей подсказке любого элемента управления на форме, тогда в строке состояния будет автоматически отображаться подсказка для текущего элемента управления. Для этого свойство AutoHint должно иметь значение True. С помощью свойства SizeGrip можно задать отображение в правом нижнем углу строки состояния небольшого треугольника, позволяющего изменять размер формы.

Часто используется и свойство Panels (тип TStatusPanels, наследник класса TCollection), представляющее собой коллекцию объектов типа TStatusPanel, панелей, из которых составляется строка состояния. Для их создания и обработки применяется стандартный редактор коллекций.

Главные свойства класса TStatusPanel перечислены ниже.

Таблица 4.63. *Свойства класса TStatusPanel*

Свойство	Назначение
Alignment	Способ выравнивания текста на данной панели
Bevel	Вид панели (выпуклая, вдавленная или обычная)
Style	Определяет вывод текста или программное формирование содержимого
Text	Текст панели
Width	Ширина панели в пикселах



Доступ к панелям осуществляется через стандартное для класса TCollection свойство Items. Для примера, поделим строку состояния на две панели. На правой панели будем показывать позицию указателя мыши, на левой — состояние кнопок мыши (нажата или отпущена одна из ее кнопок).

Создадим обработчик события главной формы OnMouseMove (При перемещении мыши):

```
procedure TForm1.FormMouseMove(Sender: TObject;
   Shift: TShiftState; X, Y: Integer);
begin
StatusBar1.Panels.Items[0].Text :=
   ' ('+IntToStr(X)+', '+IntToStr(Y)+')' ,-
if ssLeft in Shift
   then StatusBar1.Panels.Items[1].Text := 'Нажата'
   else StatusBar1.Panels.Items[1].Text := 'Отпущена'
end;
```

Чтобы зафиксировать конец последней (в нашем случае, второй) панели с помощью вертикального разделителя, явно задав ее размер, можно добавить в редакторе панелей третью панель, которая использоваться не будет. Ее наличие приведет к появлению на форме нужного разделителя.

Компонент Панель инструментов (TToolBar)

Эта панель используется для управления быстрыми командными кнопками и другими «горячими» элементами управления. Она позволяет формировать многострочные наборы инструментов.

Сразу после создания панель инструментов автоматически (как и строка состояния) размещается у края формы, только у верхнего, а не у нижнего.

Таблица 4.64. *Свойства компонента TToolBar*

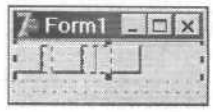
Свойство	**Назначение**
EdgeBorders	Сложное свойство. Определяет наличие или отсутствие видимых краев (полосок) с каждой из четырех сторон панели
EdgeInner, EdgeOuter	Вид окаймляющих полос (вдавленный, выпуклый) с внутренней и внешней стороны
ButtonWidth, ButtonHeight	Ширина и высота (в пикселах) элементов управления, располагаемых на панели
DisabledImages	Указывает на объект типа TImageList (список картинок), который содержит изображения кнопок, отображающих их «отключенное» состояние. Если значение для данного свойства не задано, используются обычные картинки кнопок, преобразованные к палитре, содержащей только тона серого
Flat	Имеет значение True, если используется стиль «прозрачных» кнопок, когда фон панели и кнопок не рисуется
HotImages	Указывает на объект типа TImageList (список картинок), который содержит изображения кнопок, появляющихся при наведении на кнопку указателя мыши. Подобный стиль характерен для кнопок броузера Microsoft Internet Explorer версии 4 и выше
Images	Указывает на объект типа TImageList (список картинок), который содержит рабочие изображения кнопок
Indent	Сдвиг левой границы панели в пикселах
List	Имеет значение True, если необходимо показывать заголовки кнопок справа от изображений, а не под ними
ShowCaptions	Имеет значение True, если текстовые заголовки кнопок должны отображаться на панели
Transparent	Имеет значение True, если панель — прозрачная. Это свойство не влияет на прозрачность объектов на панели
Wrapable	Имеет значение True, если элементы управления на панели должны автоматически образовывать новые строки, при условии что им не хватает пространства формы

Панель инструментов может работать с кнопками класса TToolButton, которые не присутствуют на палитре компонентов *Delphi 7*. Они специально сконфигурированы для использования только совместно с компонентом TToolBar и добавляются на панель выбором пункта New Button (Создать кнопку) в ее контекстном меню.

Массив таких кнопок хранится в свойстве Buttons. Их текущее число определяется значением свойства ButtonCount.

Допустим, требуется подготовить панель с тремя кнопками. Вторая кнопка должна быть отделена от первой простым разделителем (свободным пространством), а третья кнопка от второй — так называемым *кнопочным разделителем,* который используется для разделения групп кнопок.

Порядок последовательного выбора пунктов контекстного меню панели должен быть следующим: New Button (Создать кнопку), New Separator (Создать разделитель), New Button (Создать кнопку), New Button (Создать кнопку), New Button (Создать кнопку).



Третья кнопка будет использоваться как кнопочный разделитель. Для этого свойство Style получает значение tbsDivider. Теперь необходимо подготовить объект ImageList и загрузить в него изображения трех кнопок (рис. 4.19).

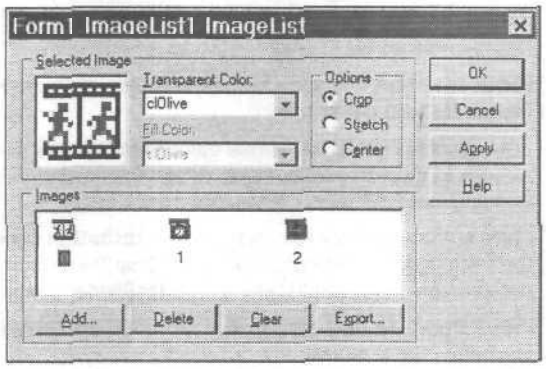
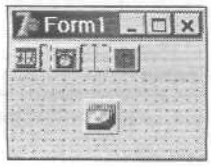


Рис. 4.19. Выбор изображений для кнопок панели управления

Далее значение свойства ButtonWidth для панели надо сделать равным ширине картинок, выбранных для кнопок (значению свойства Width объекта ImageList1), а в свойстве Images панели надо выбрать в раскрывающемся списке значение ImageList1, в результате чего на кнопках сразу отобразятся подготовленные картинки.



Чтобы третье изображение из списка ImageList (с номером 2, так как отсчет ведется с нуля) соответствовало четвертой кнопке, надо в ее свойство ImageIndex внести значение 2.

Таблица 4.65- Основные свойства класса TToolButton

Свойства	Назначение
AllowAllUp	Имеет значение True, если все кнопки группы, к которой относится данная кнопка, могут одновременно находиться в отжатом состоянии
Down	Состояние кнопки (значение True, если кнопка нажата)
DropdownMenu	Ссылка на меню кнопки (компонент TPopupMenu). Дополнительно, для свойства кнопки Style надо задать значение tbsDropDown. При этом к правой части кнопки добавляется небольшая панель со стрелкой, при щелчке на которой и открывается меню
Grouped	Имеет значение True, если данная кнопка входит в состав группы. Режим работы в группе задается значением tbsCheck для свойства кнопки Style. В группе может быть нажата только одна кнопка, при щелчке на отпущенной кнопке все остальные отжимаются, а она остается в нажатом состоянии. Такой режим действует только в диапазоне смежных кнопок, которые не отделены друг от друга разделителями и все имеют значение True для свойства Grouped и значение tbsCheck для свойства Style
ImageIndex	Номер картинки из списка картинок, на который ссылается родительский объект (TToolBar)
Indeterminate	Имеет значение True, если кнопка находится в «промежуточном» состоянии. Это позволяет показывать «частичность» выполняемых функций. Свойство Down при этом автоматически принимает значение False
Index	Порядковый номер кнопки
Marked	Имеет значение True, если кнопка изображается как отключенная (покрытая мелкой сеткой)
MenuItem	Если в данном свойстве указан один из доступных на форме пунктов любого из существующих меню, то щелчок на данной кнопке приведет к выполнению действия, связанного с этим пунктом
Style	Стиль кнопки. Возможные значения — tbsButton (командная кнопка), tbsCheck (кнопка-переключатель), tbsDropDown (кнопка-меню), tbsSeparator (кнопка-разделитель) и tbsDivider (широкий разделитель)
Wrap	Имеет значение True, если требуется, чтобы следующая кнопка размещалась на новой строке панели

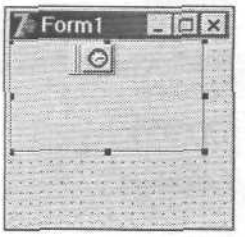
Некоторые методы класса TToolBar позволяют выполнять ряд действий, имитирующих работу с кнопками TToolButton.

Таблица 4.66. Методы класса TToollBar

Метод	Назначение
procedure Click Button (Button: TToolButton);	Имитация щелчка на указанной кнопке
procedure InitMenu(Button:TToolButton);	Вызов меню, привязанного к указанной кнопке

События OnCustomDraw и OnAdvancedCustomDraw можно обрабатывать, если надо перерисовывать внешний вид (фон) панели (полностью или частично). Обработка события OnAdvancedCustomDrawButton позволяет программно выполнять отрисовку кнопок класса TToolButton. Перерисовка конкретной кнопки осуществляется в обработчике события OnCustomDrawButton.

Компонент Панель управления (TControlBar)



Данный компонент расположен на панели Additional (Дополнительные). Но он умышленно перенесен в данный раздел, потому что используется обычно вместе с объектами типа TToolBar. При размещении такого объекта на панели управления, к нему добавляется специальный корешок, полоса перетаскивания расположенная слева. С помощью корешка этот компонент можно перетаскивать в пределах панели управления.

После размещения компонента на форме ему придаются нужные размеры, после чего на его поверхности размещаются другие элементы управления (как правило, это панели инструментов и горячие кнопки TSpeedButton).

Все эти элементы автоматически получают возможность перетаскивания в рамках панели управления. Для этого свойства AutoDock и AutoDrag должны иметь значение True.

Прочие свойства класса TControlBar приведены ниже.

***Таблица** 4.67. Основные свойства класса TControlBar*

Свойство	Назначение
BevelEdges, BevelInner, BevelOuter, BevelKind, BevelWidth	Оформление внешнего вида панели: наличие окаймляющих линий, их форма и толщина
Picture	Картинка, используемая в качестве фона панели
RowSize	Высота одной строки панели. Это свойство используется, чтобы панель могла определить, когда нужно выполнять автоматическое переупорядочивание расположенных на ней элементов управления

Компонент Расширенная панель управления (TCoolBar)

Данный компонент сочетает в себе возможности компонентов TToolBar и TControlBar. Он имеет свойство Bands, представляющее собой коллекцию (наследник класса TCollection) элементов класса TCoolBand («плавающих панелей»). С помощью этого свойства можно вручную формировать содержимое и внешний вид каждой плавающей панели.

Работа с конкретными плавающими панелями осуществляется с помощью свойства Bands (класс TCoolBands), представляющего собой коллекцию объектов-наследников класса TCoolBand. Как и любую коллекцию, это свойство можно менять с помощью редактора коллекций.

Основные свойства классов TCoolBar и TCoolBand (плавающей панели с полоской перетаскивания) приведены в табл. 4.68 и 4.69.

Таблица 4.68. Основные свойства класса TCoolBar

Свойство	Назначение
Bitmap	Фоновый рисунок
EdgeBorders, EdgeInner, EdgeOuter	Наличие и оформление окаймляющих линий панели
FixedOrder	Имеет значение True, если пользователю разрешено перемещать объекты на панели, но не разрешено менять их порядок
FixedHeight	Имеет значение True, если размер полос перетаскивания фиксирован для всех объектов на панели и не подстраивается под их высоту
Images	Список картинок, которые будут использоваться в качестве полос перетаскивания вместо стандартного изображения
ShowText	Имеет значение True, если рядом с каждой плавающей панелью отображается текстовая строка
Vertical	Имеет значение True, если порядок плавающих панелей определяется сверху вниз. В противном случае порядок определяется слева направо

Таблица 4.69. Основные свойства класса TCoolBand

Свойство	Назначение
Bitmap	Фоновый рисунок
Break	Имеет значение True, если данная панель будет располагаться с новой строки
Control	Элемент управления, расположенный на панели
FixedBackground	Имеет значение True, если фоновое изображение выравнивается по всему размеру панели. В противном случае производится выравнивание по ее верхнему левому углу
Height	Высота панели в пикселах
HorizontalOnly	Имеет значение True, если данную панель необходимо прятать, когда свойство Vertical родительской панели принимает значение True
ImageIndex	Номер картинки, которая отображается в качестве полосы перетаскивания
MinWidth, MinHeight	Минимально допустимые ширина и высота панели при изменении ее размера
Text	Текст, отображаемый в левой части панели (название панели), когда у родительского объекта свойство ShowText имеет значение True

Компонент Прокрутка страниц (TPageScroller)

Этот компонент позволяет задавать видимую область для различных элементов управления (например, панелей). Доступ к невидимой части осуществляется с помощью стрелок, автоматически появляющихся по краям области прокрутки. С использованием этого компонента выполнены все панели компонентов *Delphi 7*. Данный компонент отличается от компонента TScrollBox наличием кнопок со стрелками и возможностью задавать направление прокрутки только в горизонтальном или только в вертикальном направлении.

После того как компонент размещен на форме и его размеры заданы, готовится объект, который будет помещен в область прокрутки. Затем в свойстве Control указывается ссылка на него. Соответствующий элемент управления автоматически перемешается в область прокрутки, в которой возникают стрелки для прокрутки.

Основные свойства класса TPageScroller приведены ниже.

***Таблица** 4.70. Свойства класса TPageScroller*

Свойство	Назначение
AutoScroll	Имеет значение True, если прокрутка автоматически выполняется при наведении указателя на кнопку со стрелкой (без щелчка на ней)
ButtonSize	Размер кнопки со стрелкой в пикселах
Margin	Расстояние (в пикселах) между прокручиваемой областью и границами объекта PageScroller
Orientation	Направление прокрутки: горизонтальное (soHorizontal) или вертикальное (soVertical)
Position	Величина сдвига (в пикселах) прокручиваемой области

Единственное событие, которое имеет смысл обрабатывать — это событие OnScroll. Оно формируется, когда выполняется прокрутка содержимого.

```
procedure PageScrollerScroll(Sender: TObject;
   Shift: TShiftState; X, Y: Integer;
   Orientation: TPageScrollerOrientation;
   var Delta: Integer);
```

Параметр Shift определяет состояние управляющих клавиш и кнопок мыши, параметры X и Y — текущее положение прокручиваемой области, параметр Delta определяет размер шага прокрутки. Если прокрутка ведется от конца к началу, значение этого параметра отрицательно.

В следующем примере во время работы программы надпись Label1 динамически отображает текущую горизонтальную координату прокручиваемой области. Вначале надо разместить на форме компонент TPageScroller и произвольный объект (например, панель). Затем этот объект надо выбрать в свойстве Control объекта PageScrollerl, и он автоматически переместится в область прокрутки. После этого можно создать обработчик события OnScrolL

```
procedure TForm1.PageScroller1Scroll(Sender: TObject;
   Shift: TShiftState; X, Y: Integer;
   Orientation: TPageScrollerOrientation;
   var Delta: Integer);

begin

Label1.Caption := IntToStr(X);

end;
```

Компонент Список элементов (TListView)

В отличие от ранее рассмотренного компонента TListBox, представляющего собой стандартный список строк, существующий еще с первых версий *Windows,* список элементов содержит значительно больше возможностей представления информации.

Данный компонент ориентирован на представление данных в виде структуры «объект — набор свойств», например файлов вместе со своими характеристиками: размером, датой создания, атрибутами. Использовать его как список однородной информации некорректно.

Первоначально, сразу после создания, в списке не содержится ни одного элемента. Режим его будущей работы определяется значением свойства ViewStyle. Возможны следующие значения.

Таблица 4.71. *Значения свойства ViewStyle*

Значение	Режим работы
vsIcon	Каждый элемент представлен полноразмерным значком с подписью, который можно перетаскивать. Так работают папки Windows 9x в режиме Крупные значки
vsSmallIcon	Каждый элемент представлен маленьким значком с подписью справа от него. Эти значки можно перетаскивать. Так работают папки Windows 9x в режиме Мелкие значки
vsList	Каждый элемент представлен маленьким значком с подписью справа от него. Эти значки, расположенные по столбцам, перетаскивать нельзя. Там работают папки Windows 9x в режиме Список
vsReport	Объект работает как обычный список с несколькими столбцами

Число и свойства столбцов задаются в свойстве Columns, которое представляет собой коллекцию объектов типа TListColumn. Основное их свойство — Caption. Оно содержит заголовок столбца. Пользователь может редактировать его во время работы программы, если значение свойства списка Readonly не равно True и если заголовки не работают в режиме кнопок.

Выбор элементов может выполняться путем ввода первых букв имени, при этом фокус перескакивает по элементам списка в соответствии с набранной пользователем строкой.

Данные для списка формируются в свойстве Items (тип TListItems), представляющим собой список объектов типа TListItem. Эти объекты можно создавать на этапе проектирования с помощью специального редактора (рис. 4.20).

После щелчка на кнопке New Item (Новый элемент) в поле Caption (Заголовок) задается его имя, в поле Image Index (Номер рисунка) — номер рисунка (значка) из списка рисунков LargeImages или SmallImages (см. ниже), в поле State Index (Номер состояния) — номер рисунка из списка рисунков StateImages. Каждый элемент, в свою очередь, может состоять из нескольких вложенных элементов (больший уровень вложен-

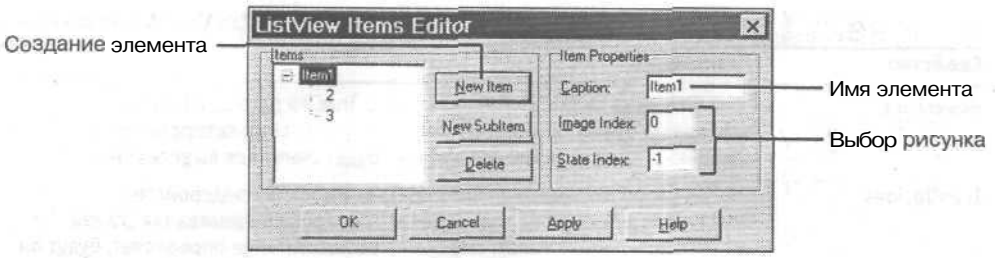


***Рис. 4.20.** Редактор списка объектов*

ности не допускается), представляющих собой, по замыслу создателей, свойства этого элемента.

Названия элементов и номера картинок хранятся в свойстве Items, а обратиться к вложенным объектам (свойствам) можно через подсвойство SubItems (см, далее).

Настройка структуры и внешнего вида списка осуществляется с помощью свойств, приведенных в табл. 4.72.

***Таблица4.72.** Свойства класса TListView*

Свойство	Назначение
AllocBy	Число элементов списка, хранимое в памяти. Управляя этим значением, можно существенно повысить быстродействие некоторых операций по работе со списком, например сортировки
Checkboxes	Имеет значение True, если 8 начале каждой строки списка отображается флажок
ColumnClick	Имеет значение True, если заголовкам столбцов разрешено работать в режиме кнопок: допустимы щелчки на заголовках и обработка этих щелчков. Такая возможность полезна для сортировки содержимого списка щелчком на его заголовке
FlatScrollBars	Имеет значение True, если полосы прокрутки должны выглядеть плоскими
FullDrag	Имеет значение True, если разрешается полностью перерисовывать заголовки столбцов во время перетаскивания, а не только отображать границы
GridLines	Имеет значение True, если между элементами списка рисуются разделительные линии
HideSelection	Имеет значение True, если выделение текущего элемента списка автоматически сбрасывается при переключении фокуса на другой элемент формы. Такая возможность используется, когда ведется одновременная работа с несколькими списками, что позволяет быстро определить, какой список имеет фокус
HotTrack	Имеет значение True, если выбор элемента осуществляется наведением на него указателя (без щелчка)
HotTrackStyles	Если значение свойства HotTrack равно True, то способ выделения элемента определяется комбинацией значений данного свойства (множества). Возможные значения — htHandPoint (указатель мыши принимает вид руки), htUnderlineCold (невыделенные элементы подчеркиваются), htUnderlineHot (выделенный элемент подчеркивается)

продолжение

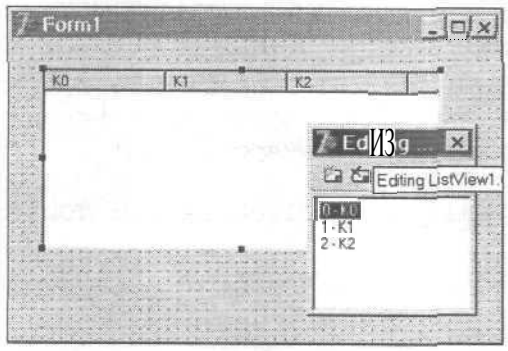
Таблица 4.72. Свойства класса TListView (продолжение)

Свойство	Назначение
HoverTime	Если значение свойства HotTrack равно True, то данное свойство определяет время (в миллисекундах), по истечении которого после наведения на него указателя элемент будет считаться выделенным
IconOptions	Способ упорядочения значков в списке. Имеет три подсвойства. Подсвойство Arrangement определяет порядок выравнивания (слева направо или сверху вниз), Подсвойство AutoArrange определяет, будут ли значки переупорядочиваться автоматически. Подсвойство WrapText определяет, будет ли заголовок выравниваться по ширине значка или располагаться слева от него
LargeImages	Список картинок-значков. Стиль отображения определен значением vsIcon
MultiSelect	Имеет значение True, если разрешается выбирать несколько элементов списка
OwnerData	Имеет значение True, если обработка содержимого списка выполняется в тексте программы. Подобный список называется виртуальным, И разработчик берет на себя программирование его основных функций, связанных с динамическим формированием значений элементов. Это требуется при обработке больших наборов данных
OwnerDraw	Имеет значение True, если рисование списка и его элементов явно выполняется в тексте программы по алгоритму разработчика
RowSelect	Имеет значение True, если разрешается выделять целую строку списка
ShowColumnHeaders	Имеет значение True, если отображаются заголовки столбцов
SmallImages	Список картинок-значков. Стиль отображения определен значением, отличающимся от vsIcon
SortType	Способ автоматической сортировки списка. Возможные значения — stNone (сортировка не выполняется), stData (сортировка выполняется на основе значений свойства Data каждого элемента списка), stText (сортировка выполняется на основе значений свойства Caption каждого элемента списка), stBoth (сортировка выполняется на основе значений как свойства Data, так и свойства Caption)
StateImages	Список картинок, отражающих промежуточное состояние объекта

Рассмотрим работу списка элементов в различных режимах на примере. Заранее надо подготовить три набора картинок (значков) по три элемента в каждом. Они будут указаны в свойствах Largelmages, SmallImages и Statelmages.

ВНИМАНИЕ Общая идеология работы данного компонента не меняется при переходе к различным формам его внешнего представления и отражает работу папок Windows 9x. В крайнем левом столбце отображается значок, затем идут его имя и свойства. Над каждым столбцом выводится заголовок. Б режиме vsReport самый левый столбец (с номером 0) содержит имя элемента, а все последующие — имена его свойств, что соответствует классическому представлению файлов папки.

Установим для свойства ViewStyle значение vsReport. Редактируя свойство Columns, создадим три столбца: K0, K1 и K2 (рис. 4.21).



Рыс. 4.21. *Создание списка элементов с тремя столбцами*

С помощью редактора свойства Items добавим три элемента Э1, Э2 и Э3, каждый из которых будет иметь по два подсвойства СЭ*1 и СЭ*2. Звездочкой здесь обозначен номер элемента (рис. 4.22).

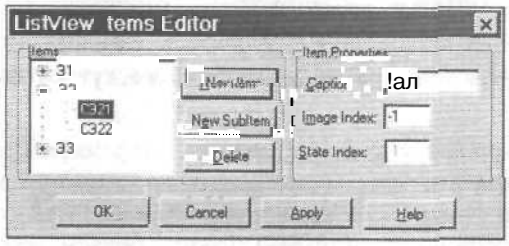


Рис. 4.22. *Формирование элементов, входящих в список*

Укажем для каждого элемента номер его картинки. Список примет вид, показанный на рис. 4.23.

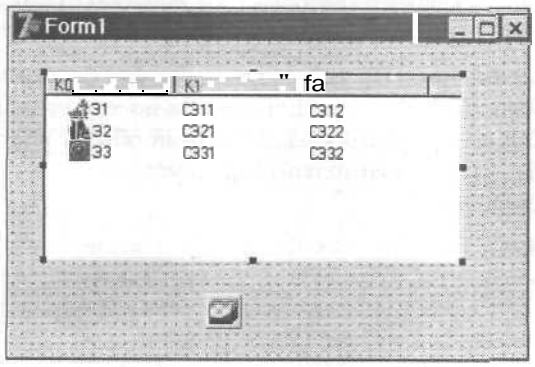


Рис. 4.23. *Форма, содержащая подготовленный список элементов*

Теперь можно откомпилировать и запустить программу. Размеры столбцов регулируются с помощью мыши стандартным способом.

Добавим на форму четыре кнопки. Каждая из них будет отвечать за представление списка в ином режиме.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
ListView1.ViewStyle := vsReport;
end;
procedure TForm1.Button2Click(Sender: TObject);
begin
ListView1.ViewStyle := vsIcon;
end;
procedure TForm1.Button3Click(Sender: TObject);
begin
ListView1.ViewStyle := vsSmallIcon;
end;
procedure TForm1.Button4Click(Sender:TObject),-
begin
ListView1.ViewStyle := vsList;
end;
```

Как только изменяется значение свойства ViewStyle, тут же изменяется и внешний вид списка.

Теперь научимся заполнять список динамически, во время работы программы. Допустим, требуется показывать содержимое заданного каталога: список имен всех вложенных каталогов и файлов с их характеристиками (датой создания, размером и признаками «скрытый» и «системный»). Таким образом, всего столбцов в списке будет пять, а сам список будет работать в режиме vsReport,

Создадим новое приложение и разместим на нем пустой список, текстовое поле и кнопку. Для визуального разделения файлов и папок подготовим список (ImageList) из двух маленьких значков, первый из которых (под номером 0) соответствует файлу, а второй (под номером 1) — папке.

В обработчике создания формы сформируем структуру списка и определим его внешний вил. Для добавления и настройки свойств нового столбца используется метод Add свойства Columns, который создает новый объект класса TListColumn и возвращает ссылку на него. Заключительный цикл устанавливает ширину каждого столбца равной 150 пикселам.

```
procedure TForm1.FormCreate(Sender: TObject);
var NewColumn: TListColumn;
    i: integer;
begin
with ListViewl do
    begin
    ViewStyle := vsReport;
```

```
    NewColumn := Columns.Add;
    NewColumn.Caption := 'Название';
    NewColumn := Columns.Add;
    NewColumn.Caption := 'Дата создания';
    NewColumn := Columns.Add;
    NewColumn.Caption := 'Размер, байтов';
    NewColumn := Columns.Add;
    NewColumn.Caption := 'Скрытый';
    NewColumn ;= Columns.Add;
    NewColumn.Caption := 'Системный';
    for i := 0 to 4 do
       Columns[i].Width := 100;
    end
end;
```

Перебор файлов будем выполнять с помощью функций FindFirst/FindNext. Для этого подготовим процедуру (метод формы TForm1), которая будет записывать содержимое структуры TSearchRec в конец списка.

```
procedure TForm1.AddNewFile(F: TSearchRec);
begin
with ListView1.Items.Add, F do
   begin
   Caption := Name;
   if (Attr and faDirectory) o O
      then ImageIndex := 0
      else ImageIndex := 1;
   SubItems.Add(DateTimeToStr((FileDateToDateTime(Time))));
   SubItems.Add(IntToStr(Size)];
   if (Attr and faHidden) o O
      then SubItems.Add('да')
      else SubItems.Addf'нет');
   if (Attr and faSysFile) o O
      then SubItems.Add('да')
      else SubItems.Add('нет');
   end;
end;
```

В операторе with используются два параметра. Первый, представляющий собой экземпляр класса TListItem, создается динамически с помощью метода Add, а второй описывает текущую структуру TSearchRec, из которой необходимо извлечь нужную информацию для списка.

Первоначально задается название самому элементу списка, а также указывается номер картинки в зависимости от типа файла (каталог или обычный файл). Далее новые строки добавляются с помощью метода Add свойства Subltems, принадлежащего свойству Items, У этого метода один параметр — текстовая строка, записываемая в

свойство Caption. Ее содержимое зависит от наличия или отсутствия различных характеристик у анализируемого файла.

В поле Edit1 будет вводиться полный путь поиска для каталога и маска файла, например C:*.*. Содержимое этого каталога будет отображаться после щелчка на кнопке Button1. Вот обработчик этого события.

```
procedure TForm1.Button1Click(Sender: TObject);
var F: TSearchRec;
    fa: integer;
begin
fa := faAnyFile;
if FindFirst(Edit1.Text, fa, F) <> 0 then Exit;
AddNewFile(F);
while FindNext(F) = 0 do
   AddNewFile(F);
FindClose(F);
end;
```

После запуска программы и щелчка на кнопке Button1 будет получен результат, подобный изображенному на рис. 4.24.

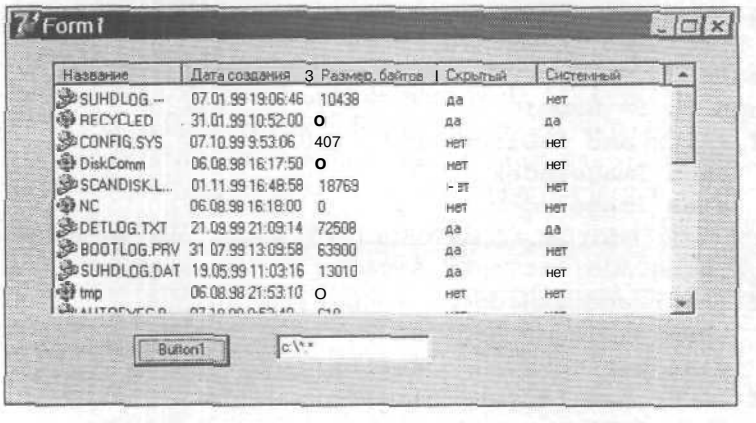


Рис. 4.24. *Отображение содержимого каталога и атрибутов файлов в виде списка*

Папки и файлы здесь расположены вперемешку. Это неправильно. Их надо отсортировать в соответствии с номерами картинок: первыми должны идти папки (значение свойства ImageIndex равно 0), за ними — обычные файлы. Сортировка в классе TListView выполняется путем вызова метода AlphaSort, который обычно сортирует все элементы в списке в порядке «возрастания» их имен, если для объекта ListView1 не определен обработчик события OnCompare (Сравнить два элемента). Если же он определен, то сортировка выполняется на основе результатов его работы.

Сформируем такой обработчик. Два объекта, Item1 и Item2, будут сравниваться по значениям свойства ImageIndex. В параметр Compare записывается отрицательное

число, если первый объект «меньше» второго; 0, если они равны; положительное число, если первый объект «больше» второго.

```
procedure TForm1.ListView1Compare(Sender: TObject;
    Item1, Item2: TListItem;
    Data: Integer; var Compare: Integer);
begin
if Item1.ImageIndex = Item2.ImageIndex
then Compare := 0 else
if Item1.ImageIndex < Item2.ImageIndex
then Compare := -1
else Compare := +1
end;
```

Осталось только добавить в конец метода Button1Click вызов сортировки.

```
ListView1.AlphaSort;
```

Теперь каталоги будут отображаться в списке первыми.

Класс TListView содержит немало дополнительных возможностей. В частности, каждый элемент списка (класс TListItem) имеет свойство Data (тип Pointer), в котором может храниться произвольная информация, связанная с конкретным элементом.

В заключение приведем перечень наиболее важных методов и событий для класса TListView (табл. 4.73 и 4.74). Некоторые методы (например, получение элемента, ближайшего к указанной точке клиентской области) могут показаться на первый взгляд странными. Не надо забывать, что основное назначение данного компонента — работа в стиле папок *Windows 9x,* когда пользователь может щелкать на значке или рядом с ним.

Таблица 4.73. Основные методы класса TListView

Метод	Назначение
procedure Arrange(Code: TListArrangement);	Задает способ выравнивания значков, когда значение свойства ViewStyle равно vsIcon или vsSmallIcon
function FindCaption(StartIndex: Integer; Value: string; Partial, Inclusive, Wrap: Boolean): TListItem;	Поиск элемента списка, заголовок которого совладает со значением параметра Value. Если параметр Inclusive имеет значение True, то поиск выполняется начинал с элемента с номером, хранящимся в параметре StartIndex. При этом, если параметр Wrap также имеет значение True, то поиск по достижении конца списка продолжается С его начала. Параметр Partial разрешает не искать полное совпадение заголовков, а использовать значение Value как подстроку
function FindData(StartIndex: Integer; Value: Pointer; Inclusive, Wrap: Boolean): TListItem;	Аналогично предыдущему, только производится сравнение не заголовков, а связанных с объектом данных (свойство Data)

продолжение

Таблица 4.73. Основные методы класса TListView (продолжение)

Метод	Назначение
function GetHitTestInfoAt(X, Y: Integer): THitTests;	Возвращает подробную информацию об указанной точке клиенткой области. Тип THitTests описывает различные местоположения точки
function GetItemAt(X, Y: Integer): TListItem;	Возвращает элемент списка, области которого принадлежит указанная точка
function GetNearestItem(Point: TPoint; Direction: TSearchDirection): TListItem;	Возвращает элемент списка, ближайший к указанной точке по заданному направлению (параметр Direction)
function GetNextItem(StartItem: TListItem; Direction: TSearchDirection; States: TItemStates): TListItem;	Возвращает элемент списка, следующий за указанным в параметре StartItem в направлении Direction. Если используется список StateImages, то можно учитывать и наличие свойства State
function GetSearchString: string;	Возвращает текущую строку, которую пользователь ввел для поиска нужного элемента
procedure Scroll(DX, DY: Integer);	Прокрутка содержимого списка на DX пикселов по горизонтали и DY пикселов по вертикали
function StringWidth(S: string): Integer;	Возвращает ширину строки S в пикселах с учетом текущего шрифта списка
procedure UpdateItems(FirstIndex, LastIndex: Integer);	Перерисовка диапазона элементов списка в диапазоне от FirstIndex до LastIndex

Таблица 4.74. Основные события класса TListView

Событие	Условие генерации
OnCustomDraw, OnAdvancedCustomDraw	Программная отрисовка внешнего вида списка
OnCustomDrawItem, OnAdvancedCustomDrawItem	Программная отрисовка элемента списка
OnCustomDrawSubItem, OnAdvancedCustomDrawSubItem	Программная отрисовка вложенного элемента (свойства) списка
OnChange	Элемент списка был изменен
OnChanging	Происходит изменение элемента списка
OnColumnClick	Щелчок мышкой на заголовке столбца
OnColumnDragged	Заголовок столбца был перемещен (перетащен мышью) в новое место
OnColumnRightClick	Щелчок на заголовке столбца правой кнопкой мыши
OnData	Генерируется перед тем, как элемент списка должен быть нарисован. Данное сообщение обрабатывается, когда содержимое каждого элемента формируется программно (режим «виртуального списка»)
OnDataFind	Запрос на поиск данных от метода FindData
OnDataHint	Изменен диапазон видимых на экране элементов (например, при прокрутке)

Таблица 4.74. Основные события класса TListView (продолжение)

Событие	Условие генерации
OnDataStateChange	Изменение состояния элемента (событие возникает, только если значение свойства OwnerData равно True)
OnDeletion	Пользователь отдал команду на удаление элемента
OnDrawItem	Программная отрисовка содержимого элемента (событие возникает, только если значение свойства OwnerData равно True)
OnEdited	Завершено редактирование элемента
On Editing	Происходит редактирование элемента
OnGetImageIndex	Генерируется перед отображением элемента на экране. Его можно обрабатывать, чтобы динамически задавать номер картинки-значка из списка картинок
OnGetSubItemImage	То же для вложенного элемента (SubItem)
OnInfoTip	Пользователь навел указатель мыши на элемент и задержал его
OnInsert	В список добавлен новый элемент
OnSelectItem	В списке выбран элемент

Для использования всех возможностей компонента TListView необходимо также познакомиться с методами класса TListItems (свойство Items в списке). Эти методы представлены в табл. 4.75. Важны также свойства и методы класса TListItem, который характеризует конкретный элемент списка (табл. 4.76 и 4.77).

Таблица 4.75. Методы класса TListItems

Метод	Назначение
function Add: TListItem;	Создание нового элемента и его добавление в конец списка. Функция возвращает ссылку на этот элемент
procedure BeginUpdate; procedure EndUpdate;	Процедура BeginUpdate блокирует перерисовку списка, а процедура EndUpdate снимает блокировку. Эти методы обычно используют во время выполнения большого числа изменений, чтобы не замедлять работу перерисовкой ненужных временных деталей
procedure Gear;	Удаление всех элементов списка и освобождение занимаемой ими памяти
procedure Delete(Index: Integer);	Удаление указанного элемента
function IndexOf(Value: TListItem): Integer;	Возвращает номер элемента, указанного в качестве параметра
function Insert(Index: Integer): TListItem;	Создание нового элемента и его добавление в указанную позицию списка. Функция возвращает ссылку на этот элемент
procedure SetCount(Value: Integer);	Задание числа элементов в списке

Таблица 4.76. *Свойства класса TListItem*

Свойство	Назначение
Caption	Заголовок элемента
Checked	Имеет значение True, если флажок элемента включен (свойство Checkboxes должно иметь значение True)
Cut	Элемент рисуется в виде, показывающем, что он вырезан командой Cut (Вырезать). Все действия по реализации процедуры *такого* рисования разработчик должен программировать самостоятельно
Data	Свойство, имеющее тип Pointer и указывающее на связанный с элементом объект
Focused	Имеет значение True, если элемент имеет фокус
ImageIndex	Номер значка в списке картинок
Index	Положение элемента в коллекции TlistItems
Left	Горизонтальный сдвиг от левой границы списка
Position	Свойство типа TPoint, определяющее координаты (в пикселах) элемента внутри списка
Selected	Имеет значение True, если элемент выделен
StateIndex	Номер значка из дополнительного списка картинок
SubItemImages	Список картинок для свойств данного элемента
SubItems	Список названий свойств элемента (тип TStrings)

Таблица 4.77. *Методы класса TListItem*

Метод	Назначение
procedure Delete;	Удаление элемента из списка. Для освобождения занимаемой им памяти надо использовать метод Free
function DisplayRect(Code: TDisplayCode): TRect;	Определяет прямоугольные координаты элемента с учетом параметра Code (границы всего элемента, только значка, только текста, значка и текста)
function GetPosition: TPoint;	Определение положения элемента в списке: смещение верхнего левого угла относительно начала списка
procedure SetPosition(const Value: TPoint);	Установка нового положения элемента
procedure MakeVisible(PartialOK: Boolean);	Прокрутка списка так, чтобы элемент стал видимым. Если значение параметра PartialOK равно True и элемент уже частично виден, то прокрутка не выполняется
procedure Update;	Перерисовка элемента

Компонент Дерево (TTreeView)

Сложные структуры данных в *Windows* обычно представляются двумя способами: в виде только что рассмотренного списка и в виде дерева. Так, например, отображается структура каталогов в Проводнике: слева иерархическая структура диска в виде раскрывающихся значков с обозначениями «–» (развернут) и «+» (свернут), а справа — содержимое выбранного каталога в виде списка.

Для создания подобных деревьев, отображающих иерархические структуры данных, в системе *Delphi 7* реализован компонент TTreeView.

Процесс создания дерева достаточно прост. Его начальную структуру можно сформировать в редакторе, аналогичном редактору компонента TListView, только уровень вложенности элементов в таком списке не ограничен. У компонента TListView поддерживался только один уровень вложенности по схеме «объект — набор свойств».

Каждому узлу дерева может соответствовать своя картинка (рис. 4.25). Ее номер указывается в поле редактора Image Index (Номер картинки), а сам список картинок задается в свойстве Images. Дополнительно, для каждого узла можно указать номер картинки, отражающей его выделенное состояние (свойство Selected Index), и номер картинки, отражающей его дополнительное состояние (свойство State Index).

Имена узлов можно редактировать, как обычные названия объектов *Windows*.

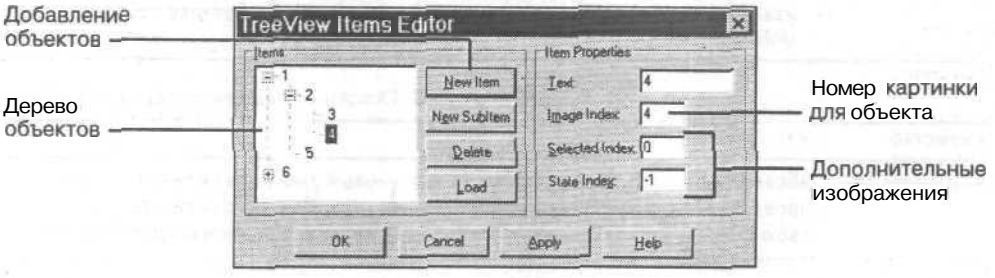


Рис. 4.25. *Редактирование структуры объектов для представления в виде дерева*

Многие свойства дерева совпадают со свойствами объекта TListView, но есть и небольшие отличия, вызванные необходимостью отображать неограниченные иерархии объектов и только одним режимом работы.

Основные свойства компонента TTreeView приведены в табл. 4.78. Сами узлы хранятся в свойстве Items (класс TTreeNodes) и имеют тип TTreeNode. Класс TTreeNodes содержит свойство Item — массив объектов типа TTreeNode. Основные свойства класса TTreeNode приведены в табл. 4.79.

ПОДСКАЗКА

Доступ к узлам по номеру и, особенно, формирование новых элементов дерева во время работы программы — процесс, требующий значительных вычислительных ресурсов, поэтому желательно выполнять максимально возможную часть работы по формированию структуры дерева на этапе проектирования.

Таблица 4.78. *Основные свойства компонента TTreeView*

Свойство	Назначение
AutoExpand	Имеет значение True, если узлы дерева будут автоматически разворачиваться и сворачиваться при перемещении фокуса
ChangeDelay	Пауза в миллисекундах между выделением узла дерева и генерацией сообщения OnChange. Обработка этого сообщения позволяет, например, отобразить содержимое данного узла в другой части формы. Для Проводника Windows эта пауза равна 50 мс
HideSelection	Имеет значение True, если с элемента, теряющего фокус, снимается выделение
Indent	Расстояние в пикселах между узлами дерева
RightClickSelect	Имеет значение True, если разрешается выделять узлы дерева с помощью правой кнопки мыши
ShowButtons	Имеет значение True, если слева от узлов отображаются кнопки с символами + и -
ShowLines	Имеет значение True, если отображаются линии, соединяющие узлы
ShowRoot	Имеет значение True, если отображаются линии, соединяющие узлы верхнего уровня
StateImages	Список картинок для отображения дополнительного состояния узлов
ToolTips	Имеет значение True, если разрешена всплывающая подсказка для каждого узла дерева. Показывать такую подсказку надо в обработчике события OnHint

Таблица 4.79. *Основные свойства класса TTreeNode*

Свойство	Назначение
AbsoluteIndex	Абсолютный номер узла в дереве. Самый первый узел имеет номер 0, далее происходит нумерация всех потомков этого узла. При этом, если у потомка в свою очередь есть подчиненные узлы, то нумерация продолжается с первого потомка и так далее
Count	Число потомков узла
Cut	Имеет значение True, если объект рисуется как «вырезанный». Действия по поддержке этой операции программист должен реализовать самостоятельно
Data	Свойство имеет тип Pointer и указывает на связанный с узлом объект
Deleting	Имеет значение True, если данный узел находится в состоянии удаления. Этот процесс может быть длительным, если удаляется узел с большим числом потомков
Expanded	Имеет значение True, если узел развернут, то есть кнопка находится в состоянии «–»
Focused	Имеет значение True, если узел имеет фокус
HasChildren	Имеет значение True, если узел имеет потомков
ImageIndex	Номер картинки в списке картинок
Index	Номер узла в списке потомков вышестоящего родителя. Первый узел-потомок имеет номер D, второй — 1 и так далее
Item	Массив узлов, являющихся потомками данного
IsVisible	Имеет значение True, если узел виден

Таблица 4.79. Основные свойства класса TTreeNode (продолжение)

Свойство	Назначение
Level	Уровень глубины узла. Верхний уровень имеет номер 0, следующий уровень — номер 1 и так далее
Selected	Имеет значение True, если узел выделен
SelectedIndex	Номер картинки, которая показывается, если узел выделен
Text	Текст, выводимый в узле
TreeView	Ссылка на родительский объект TTreeView

Работу дерева проще всего понять па следующем примере. Пусть на форме имеется пустой объект TreeViewl, текстовое поле и две кнопки: Узел и Потомок. После ввода имени и щелчка на кнопке Узел в дерево добавляется новый узел на текущем уровне. При щелчке на кнопке Потомок новый узел добавляется в число потомков текущего узла.

Главное, что требуется в этом примере, — оперативно отслеживать в программе перемещение пользователем фокуса по дереву, чтобы не просматривать в поисках выделенного узла (значение свойства Selected которого равно True) весь массив узлов каждый раз заново. Для этого можно **обрабатывать** событие OnGetSelectedIndex, которое формируется, когда возникает потребность в отрисовке конкретного выделенного узла. Запомним этот узел в переменной MyNode, которая будет принадлежать классу формы TForm1 и иметь тип TTreeNode.

```
MyNode: TTreeNode;
```

Б момент создания формы эта переменная должна получить начальное значение, указывающее, что ни один узел не выбран.

```
procedure TForm1.FormCreate(Sender: TObject);
begin
MyNode := nil
end;
```

Обработчик события OnGetSelectedIndex запишется следующим образом.

```
procedure TForm1.TreeView1GetSelectedIndex(
   Sender: TObject;
   Node: TTreeNode);
begin
MyNode := Node;
end;
```

При щелчке на кнопке Узел новый узел будет добавляться к дереву с помощью метода

```
function Add(Node: TTreeNode; const S: string): TTreeNode;
```

Этот метод добавляет новый узел на один уровень с узлом Node (или на самый верхний уровень, если вместо узла указано значение **nil)** и возвращает ссылку на этот узел. Новому узлу в свойство Text записывается значение строки S.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
If TreeView1.Items.Count = 0
then TreeView1.Items.Add(nil, Edit1.Text)
else TreeView1.Items.Add(MyNode,Edit1.Text)
end;
```

Предварительно проверяется, есть ли в дереве хотя бы один узел.

Для добавления нового потомка узла используется метод

```
function AddChild(Node: TTreeNode; const S: string)
: TTreeNode;
```

Он аналогичен предыдущему, за исключением того, что новый узел добавляется не на один уровень с узлом Node, а становится его потомком, последним в списке всех потомков.

```
procedure TForm1.Button2Click(Sender: TObject);
begin
if TreeView1.Items.Count o 0
then TreeView1.Items.AddChild(MyNode,Edit1.Text)
end;
```

При добавлении узла-потомка требуется, чтобы родительский узел уже существовал, поэтому достаточно проверить, что в дереве есть хотя бы один элемент.

Для удаления текущего элемента можно использовать следующий оператор.

```
TreeView1.Items.Delete(MyNode);
```

При этом фокус переместится на родительский узел удаляемого объекта. Узел удаляется вместе со всеми его потомками.

Теперь можно формировать дерево произвольной сложности (рис. 4.26).

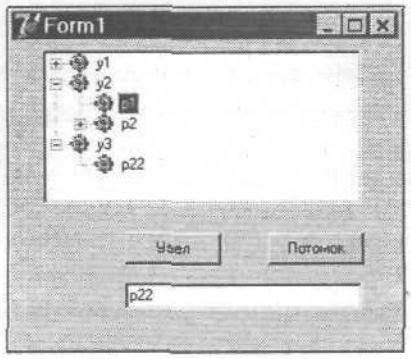


Рис. 4.26. *Формирование дерева, реализованного программно*

Когда дерево создано во время работы программы, его структуру желательно сохранить на жестком диске до следующего сеанса. Это можно сделать с помощью метода SaveToFile.

```
TreeView1.SaveToFile('TREE.TXT');
```

Дерево сохраняется в текстовом формате в наглядном виде — с отступами. Впоследствии загрузить дерево из файла можно с помощью метода

```
procedure LoadFromFile(const FileName: string);
```

Узлы в дереве можно автоматически сортировать. Момент пересортировки задается в свойстве SortType одним из трех следующих значений.

***Таблица** 4.80. Значения свойств SortType*

Значение	Основание для сортировки
stData	Узлы пересортировываются, когда изменяется их свойство Data
stText	Узлы пересортировываются, когда изменяется их свойство Text
stBoth	Узлы пересортировываются, когда изменяются оба эти свойства

Значение stNone означает, что сортировка не выполняется.

Способ сортировки определяется обработчиком события OnCompare, о котором рассказывалось при описании компонента TListView. Например, чтобы отсортировать все элементы дерева в убывающем порядке их названий, надо установить значение свойства SortType равным stText и написать следующий текст обработчика.

```
procedure TForm1.TreeView1Compare(Sender: TObject;
   Node1, Node2: TTreeNode;
   Data: Integer; var Compare: Integer);
begin
if Node1.Text > Node2.Text then Compare := -1 else
if Node1.Text < Node2.Text then Compare := +1
else Compare := 0
end;
```

ВНИМАНИЕ Реально процедура сортировки выполняется, когда происходит редактирование названия узла или связанных с ним данных, а также при изменении значения свойства SortType.

Основные методы класса TTreeView и события класса TTreeView, не совпадающие с событиями класса TListView, приведены в табл. 4.81 и 4.82. Основные методы класса TTreeNodes (массив узлов) приведены в табл. 4.83.

***Таблица 4.81.** Основные методы класса TTreeView*

Метод	Назначение
function AlphaSort: Boolean;	Сортировка всех узлов дерева в алфавитном порядке
procedure FullCollapse;	Сжатие всех раскрытых узлов дерева
procedure FullExpand;	Раскрытие всех узлов дерева
function GetHitTestInfoAt(X, Y: Integer): THitTests;	Подробная информация и том, какой части дерева (тип THitTests) принадлежит указанная точка клиентской области (координаты в пикселах)

продолжение

Таблица 4.81. Основные методы класса TTreeView (продолжение)

Метод	Назначение
function GetNodeAt(X, Y: Integer): TTreeNode;	Получение узла дерева, которому принадлежит указанная точка клиентской области. Возвращает значение nil, если такого узла нет
function IsEditing: Boolean;	Возвращает значение True, если выполняется редактирование одного из узлов дерева

Таблица 4.82. Основные события класса TTreeView, не совпадающие с событиями класса TListView

Событие	Условие генерации
OnCollapsed	Узел был свернут
OnCollapsing	Идет процесс сворачивания узла
OnExpanded	Узел был развернут
OnExpanding	Идет процесс разворачивания узла

Выше уже говорилось, что обращение к узлам дерева напрямую по номеру — операция очень неэффективная. Следующий пример показывает, как быстро перебрать все узлы дерева.

```
var CurItem: TTreeNode;
begin
CurItem := TreeView1.Items.GetFirstNode;
while Curltem <> nil do
   begin
   // выполнить нужные действия над узлом Curltem
   Curltem := CurItem.GetNext;
   end;
end;
```

Таблица 4.83. Основные методы класса TTreeNodes

Метод	Назначение
function AddChildFirst(Node:TTreeNode; const S: string): TTreeNode;	Добавление узла первым потомком узла Node (метод AddChild добавляет узел последним потомком)
function AddChildObjectFirst(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode; function AddChildObject(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;	То же, но с новым узлом через его свойство Data связывается объект, передаваемый через указатель Ptr. Метод AddChildObjectFirst добавляет узел в начало, а метод AddChildObject — в конец списка узлов-потомков
function AddFirst(Node: TTreeNode; const S: string): TTreeNode;	Добавить узел первым на уровне узла Node (метод Add добавляет узел последним на этом уровне)
function AddObject(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode; function AddObjectFirst(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;	То же, но с новым узлом через его свойство Data связывается объект, передаваемый через указатель Ptr. Метод AddObjectFirst добавляет узел в начало, а метод AddObject — в конец списка узлов на уровне узла Node

Таблица 4.83. Основные методы класса TTreeNodes (продолжение)

Метод	Назначение
procedure BeginUpdate; procedure EndUpdate;	Приостановка и возобновление перерисовки дерева. Применяется для ускорения продолжительных операций над деревом
function GetFirstNode: TTreeNode;	Получить первый узел дерева (с номером 0}
function Insert(Node: TTreeNode; const S: string): TTreeNode;	Добавить узел перед узлом Node
function InsertObject(Node: TTreeNode; const S: string; Ptr: Pointer): TTreeNode;	То же, но с добавляемым узлом через его свойство Data связывается объект, передаваемый через указатель Ptr

В табл. 4.84 приведены методы класса TTreeView (узел дерева).

Примечания.

- Узел считается видимым, если все его родительские узлы развернуты.
- Если найти подходящий узел не удалось, соответствующие методы возвращают значение nil .

Таблица 4.84. Методы класса TTreeView

Метод	Назначение
function AlphaSort: Boolean;	Сортировка всех потомков узла
procedure Collapse(Recurse: Boolean);	Сжатие узла
procedure Delete;	Удаление узла и всех его потомков
procedure DeleteChildren;	Удаление всех потомков узла
function DisplayRect(TextOnty: Boolean): TRect;	Возвращает прямоугольник, которым узел ограничивается на экране. Если значение параметра TextOnly равно True, то а прямоугольник записывается только область текстового имени узла
function EditText: Boolean;	Начинает редактирование имени узла
procedure EndEdit(Cancel: Boolean);	Завершает редактирование узла. Если значение параметра Cancel равно True, то восстанавливается прежнее значение свойства Text
procedure Expand(Recurse: Boolean);	Разворачивает узел. Если значение параметра Recurse равно True, то разворачиваются и все узлы-потомки
function GetFirstChild: TTreeNode;	Возвращает первый узел из списка потомков
function GetLastChild: TTreeNode;	Возвращает последний узел из списка потомков
function GetNext: TTreeNode; function GetPrev: TTreeNode;	Возвращает следующий (GetNext) или предыдущий (GetPrev) узел по отношению к текущему с учетом невидимых узлов и узлов-потомков
function GetNextChild(Value: TTreeNode): TTreeNode; function GetPrevChild(Value: TTreeNode): TTreeNode;	Возвращает следующий (GetNextChild) или предыдущий (GetPrevChild) узел-потомок по отношению к узлу-потомку Value

продолжение

Таблица 4.84. Методы класса TTreeView (продолжение)

Метод	Назначение
function GetNextSibling: TTreeNode; function GetPrevSibling: TTreeNode;	Возвращает следующий (GetNextSibling) или предыдущий (GetPrevSibling) узел на уровень текущего узла, независимо от того, виден ли он
function GetNextVisible: TTreeNode; function GetPrevVisible: TTreeNode;	Возвращает следующий (GetNextVisible) или предыдущий (GetPrevVisible) видимый узел
function HasAsParent(Value: TTreeNode): Boolean;	Возвращает значение True, если узел Value является родительским для текущего узла
Function IndexOf(Value: TTreeNode): Integer;	Возвращает позицию узла в списке узлов-потомков узла Value. Если узел Value не прямой родитель текущего узла, то функция возвращает значение -1
Procedure MakeVisible;	Разворачивает подходящие вышестоящие узлы таким образом, чтобы текущий узел стал видимым
Procedure MoveTo(Destination: TTreeNode; Mode: TNodeAttachMode);	Перемещает текущий узел в область узла Destination. Конкретное положение определяется значением параметра Mode

Компонент Расширенное поле со списком (TComboBoxEx)

Этот компонент является наследником класса TComboBox (Поле со списком). При этом ряд второстепенных свойств стандартного раскрывающегося списка скрыт и недоступен программисту. Наиболее важное отличие расширенного списка — новое свойство ItemsEx. Оно представляет собой коллекцию объектов TComboBoxExItems и позволяет задавать для каждой строки списка ее заголовок, вложенный рисунок (ссылку на компонент Список изображений) и выполнять различные действия над этой коллекцией: выбирать элемент, добавлять новые или удалять существующие строки и сортировать их.

Панель System (Системные компоненты)

Компонент Таймер (TTimer)

В программах, выполняющих действия, связанные с моделированием или обработкой графики, предназначенных для общения с пользователем в реальном режиме времени или выполняющих продолжительные вычисления, необходимо особое отношение к загрузке процессора. Во-первых, требуется постоянно следить за текущим временем, а во-вторых, нельзя монопольно захватывать ресурсы процессора, то есть выполнять многочасовые вычисления, не позволяя работать другим приложениям.

Простой периодический контроль текущего времени (например, с помощью подпрограмм получения текущих даты/времени или стандартной функции GetTickCount, возвращающей число миллисекунд, прошедших с полуночи) непосредственно из программы — решение неправильное. Оно не позволяет системе *Windows* управлять работой других программ.

Корпорация *Microsoft* рекомендует отслеживать сообщения, поступающие от системного таймера *Windows* (такое системное сообщение называется WM_TIMER). С помощью компонента TTimer можно включить генерацию подобных сообщений с заданной периодичностью (в миллисекундах) и выполнять определенную часть вычислений именно в обработчике этого события. Конечно, приходится отказываться от простой модели непрерывного цикла вычислений и разбивать решаемую задачу на более мелкие части, что требует определенных навыков правильной постановки задач с ориентацией на особенности операционной системы. Благодаря тому, что система *Windows* может контролировать ресурсы системы в момент получения программой системного сообщения, пользователь сможет, например, спокойно работать в текстовом редакторе, в то время как в фоновом режиме выполняется длительный процесс.

Компонент TTimer невизуальный. Помимо названия он имеет всего два свойства.

- Свойство Enabled принимает значение True, если требуется, чтобы включился системный генератор сообщений WM_TIMER.
- Свойство Interval задает фиксированный промежуток времени между сообщениями в миллисекундах.

Событие, обрабатываемое данным компонентом, только одно — OnTimer. Оно возникает, когда истекает указанный в свойстве Interval промежуток времени с момента последней генерации этого события.

В качестве примера рассмотрим программу, которая имитирует работу электронных часов. На форму надо поместить надпись и таймер. Паузу можно задать равной одной секунде (значение 1000 в свойстве Interval). Обработчик события OnTimer запишется так.

```
procedure TForm1.Timer1Timer(Sender: TObject);
var DateTime : TDateTime;
begin
DateTime :=- Time;
Label1.Caption := TimeToStr(DateTime);
end;
```

Текущее время возвращается стандартной функцией Time и затем преобразуется в текстовое представление с помощью функции TimeToStr().

Компонент Мультимедийный проигрыватель (TMediaPlayer)

Данный компонент включает в себя ряд возможностей Универсального проигрывателя *Windows* и предназначен для воспроизведения в рамках программы му-

зыкальных и видеоклипов в различных форматах, поддерживаемых драйвером *MCI (Media Control Interface)*. Управление воспроизведением осуществляется при помощи набора кнопок, напоминающих кнопки музыкальных центров (рис. 4.27).

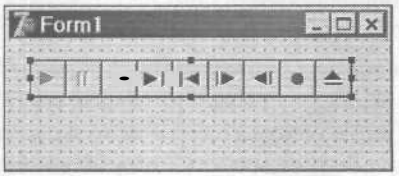


Рис. 4.27. *Размещение мультимедийною проигрывателя (управляющих кнопок) на форме*

Настройка компонента сводится к настройке видимости различных кнопок управления с помощью свойства ColoredButtons. Можно сделать невидимым и весь компонент, установив его свойство Visible равным False. Это нужно, если требуется, например, проигрывать фоновую музыку или прокручивать небольшой анимационный клип. Кроме того, с помощью свойства EnabledButtons некоторые кнопки можно оставить видимыми, одновременно запретив их использование.

Тип мультимедийного файла, который будет воспроизводиться, задается одним из двух возможных способов: явно в свойстве DeviceType (Тип устройства) или загрузкой файла на этапе проектирования (с использованием свойства fileName). В последнем случае в качестве типа устройства указывается значение dtAutoSelect (автоматическое определение типа в зависимости от расширения файла).

Если проигрыватель должен воспроизводить видеоинформацию, ему надо указать область формы, в которой будет демонстрироваться видео. Как правило, для этого используется компонент TPanel. Соответствующий объект задается в свойстве Display. В поставку *Delphi 7* входит пример Coolstuf, содержащий два видеоклипа. Этот пример хранится в подкаталоге Demos\Coolstuf той папки, куда установлена система *Delphi 7*. Можно выбрать клип cool.avi, а затем откомпилировать программу, указав в свойстве Display предварительно подготовленную панель Panell. Если теперь щелкнуть на кнопке проигрывателя Play, то в области панели будет воспроизведен небольшой клип (рис. 4.28).

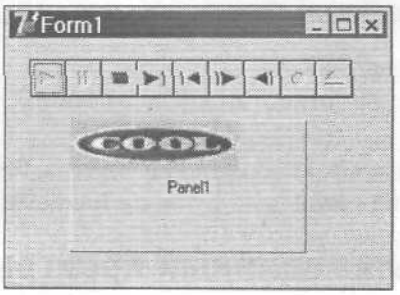


Рис. 4.28. *Воспроизведение видеоклипа в пределах формы*

Основные свойства и методы класса TMediaPlayer приведены в табл. 4.85 и 4.86.

ПОДСКАЗКА Если управление проигрывателем осуществляется непосредственно из программы, то желательно сделать этот объект невидимым, чтобы действия пользователя не оказывали влияния на его работу.

Таблица 4.85. *Основные свойства класса TMediaPlayer*

Свойство	Назначение
Auto Enable	Имеет значение True, если проигрыватель автоматически управляет состоянием своих кнопок. Например, когда клип закончен, кнопка Play становится недоступной, пока не будет выполнена команда Перемотка в начало
AutoOpen	Имеет значение True, если при запуске выполняется автоматическое открытие устройства MCI, предназначенного для воспроизведения информации заданного типа
AutoRewind	Имеет значение True, если по достижении конца клипа будет выполнена автоматическая перемотка в начало
StartPos EndPos	Точки клипа, с которых начинается (StartPos) и прекращается (EndPos) его выполнение. Задаются в условных единицах, принятых для конкретного устройства, например в кадрах или интервалах времени
Error ErrorMessage	Код и описание ошибки, возникшей в результате выполнения последней операции воспроизведения/записи
Frames	Число условных кадров, прокручиваемых при исполнении методов Step или Back
Length	Длина клипа в условных единицах
Mode	Текущее состояние устройства
TimeFormat	Описание реального значения условной единицы. Например, значение tfMilliseconds означает миллисекунды, tfFrames — кадры и так далее
Position	Текущая позиция в воспроизводимом файле в условных единицах
Shareable	Имеет значение True, если к устройству MCI во время его использования могут обращаться и другие программы
Tracks	Число доступных дорожек в открытом устройстве MCI
Track Length	Массив длин дорожек (например, TrackLength[1])
Track Position	Массив начальных позиций для дорожек

Таблица 4.86. *Основные методы класса TMediaPlayer*

Метод	Назначение
procedure Back;	Вернуться назад на указанное в свойстве Frames число условных единиц
procedure Close;	Закрыть устройство MCI
procedure Eject;	Отсоединить текущий клип от устройства. Для проигрывателя CD-ROM вызывает извлечение диска из дисковода
procedure Next;	Перейти к началу следующей дорожки
procedure Open;	Открыть устройство MCI
procedure Pause;	Приостановка воспроизведения

продолжение

Таблица 4.86. Основные методы класса TMediaPlayer(продолжение)

Метод	Назначение
procedure Play;	Начало воспроизведения
procedure Previous;	Перемотка к началу текущей дорожки
procedureResume;	Продолжить запись/воспроизведение, если устройство находится в режиме паузы
procedure Rewind;	Перемотка в начало
procedure Save;	Сохранить текущий клип в файле, указанном в свойстве FileName. Метод применяется, когда проигрыватель используется для записи
procedure StartRecording;	Начало записи
procedure Step;	Продвинуться вперед на указанное в свойстве Frames число условных единиц
procedure Stop;	Прекратить запись/воспроизведение

Перед использованием проигрывателя надо открыть устройство *MCI*, а по окончании использования — закрыть. Следующий пример демонстрирует, как можно программно извлечь диск из дисковода *CD-ROM,* например по щелчку на форме (событие OnCLick).

```
procedure TForm1.FormClick(Sender: TObject);
begin
MediaPlayer1 .DeviceType := dtCDAudio;
MediaPlayer1.Open;
MediaPlayer1.Eject;
MediaPlayer1.Close;
end;
```

Что нового мы узнали?

В этом уроке мы научились

- 0 работать с буфером обмена из программы;
- 0 выполнять рисование;
- 0 производить чтение и запись файлов;
- ☑ использовать стандартные диалоговые окна;
- 0 применять расширенный набор элементов управления;
- 0 автоматически генерировать диаграммы.

Основы работы с базами данных

- ❒ Понятие о базах данных
- ❒ Реализация работы с СУБД в системе **Delphi**
- О Утилиты для работы с СУБД
- О Работа с автономными СУБД на ПК
- ❒ Основные методы работы с набором данных
- ❒ Описание компонентов панели **BDE**
- ❒ Описание компонентов панели **DataControl**

Понятие о базах данных и СУБД

База данных и система управления базой данных

Общие требования к системам обработки данных

В главе, посвященной работе с *файлами,* рассматривались функции обработки файлов, содержащих наборы записей одного типа. При решении реальных задач организовывать обработку структурированной информации, хранящейся на жестком диске, приходится очень часто. Например, это требуется при составлении всевозможных каталогов, списков сотрудников, прейскурантов, описей товаров на складе, наборов документов и во множестве других случаев. Практически все современные крупные программные системы активно работают с данными, хранящимися на периферийных устройствах.

Однако, если при такой работе используются обычные файлы, мы сталкиваемся с рядом очень серьезных проблем. Поясним это на примере, который в дальнейшем будет реализован с помощью системы *Delphi 7.*

Пусть требуется создать программу, представляющую собой персональный справочник по компьютерным играм. Сейчас информацию о вышедших или разрабатываемых играх можно почерпнуть из самых разных источников: как с многочисленных узлов Интернета, так и из толстых ежемесячных журналов. При этом новые игры появляются почти каждый день, и заядлому игроману нелегко ориентироваться в большом потоке информации. Даже анонсированные игры любимого жанра быстро забываются, и потом бывает очень трудно вспомнить, что же представляет собой та или иная игра: то ли это долгожданный хит, о котором рассказывал журнал А, то ли скучный клон, о котором предупреждал журнал Б.

Какую информацию должен хранить справочник по играм? Желательно, например, чтобы каждая его запись имела следующие поля:

- название игры;
- жанр;
- удалось ли сыграть;
- фирма-издатель;
- фирма-разработчик;
- год выхода;
- время выхода (сезон);
- возможность сетевой игры;
- ссылка на странички в Интернете;
- ссылка на статьи в журналах;
- оценка в баллах (по 10-балльной шкале);

- оценка графики в баллах;
- оценка звука в баллах;
- комментарии.

Очевидно, некоторые поля можно заполнить задолго до выхода игры. Можно установить, например, высокий балл «интересности» ожидаемой игре, и создать систему, которая при запуске напоминала бы, на какие игры следует обратить внимание в тот или иной период времени.

Подобную структуру довольно просто описать в виде записи Паскаля (record). Еще лучше использовать для этого класс, включающий методы обработки этих данных. Этот класс можно записать в файл и потом просматривать его содержимое в поисках нужной информации.

На данном примере рассмотрим недостатки использования простых файлов с последовательностью записей (такие файлы нередко называют *плоскими)* и обычных программных способов их обработки.

1. У разработчиков возникает потребность в сложных системах, способных быстро выполнять поиск по заданному условию (она обычно называется *запросом).* Например, требуется просмотреть все игры конкретного жанра, которые были запланированы к выпуску в течение 2002 года и еще не побывали в вашем дисководе *CD-ROM.* Если в справочнике насчитываются тысячи игр, то просматривать файл от начала до конца не очень эффективно, особенно если структура запроса сложная. В крупных программных комплексах размеры файлов нередко достигают гигабайтов (в известных промышленных и коммерческих корпорациях базы данных уже занимают терабайты!). На выполнение простого запроса при этом могут потребоваться часы и даже дни работы.

   Отсюда первое требование к программе; способность быстро ориентироваться в записанной на диске информации и универсальным методом выделять нужные наборы записей.

2. У пользователя наверняка будет желание просмотреть информацию своего справочника с помощью самых разных запросов, например найти все игры одной серии, одной фирмы и так далее. Каждый раз программировать условие отбора утомительно, а если захочется предоставить такой справочник и своим друзьям, то вносить изменения в готовую программу окажется невозможно.

   Второе требование к справочнику: он должен позволять формировать запросы к данным с помощью простых визуальных средств настройки без использования программирования.

3. Когда информация отобрана в соответствии с запросом, она представляет собой набор записей, удовлетворяющих условию запроса. Этот набор надо как-то хранить. Если он большой, то его потребуется записать на диск. Необходимо также показывать его в виде отчета с возможностью перемещения от записи к записи, к началу набора, к концу и так далее.

   Третье требование к справочнику: возможность хранить временные наборы записей и формировать на их основе отчеты.

4. Данные в систему надо вводить, а в дальнейшем иметь возможность корректировать их и удалять. Соответственно, четвертое требование к справочнику — возможность просмотра, редактирования и удаления записей.

5. Система *Windows,* как всем хорошо известно, не является абсолютно надежной. Кроме того, часто приходится учитывать возможность сбоев в электросети, которые могут приводить к внезапному прекращению работы программы. Если это происходит в момент редактирования файла, то не исключено, что все данные, хранящиеся в нем, будут утеряны. Можно самостоятельно попытаться написать программный код, повышающий надежность операций с файлами (дублирующий действия и данные, выполняющий контрольное считывание после записи и так далее), но такая работа весьма трудоемка. Поэтому пятое требование к справочнику — надежность работы.

6. Желательно иметь возможность быстро находить нужную запись, если известно только содержание некоторых полей. Сделать это, используя «плоский» файл, довольно трудно. Если запоминать физический номер записи в файле, то при удалении записей придется корректировать все остающиеся номера, что существенно замедляет работу программы.

   Шестое требование к справочнику; возможность создания *ссылок* на другие записи аналогичной структуры.

7. Многие игры выпускаются одной и той же компанией. Это означает, что в поле «Фирма-издатель» будут регулярно встречаться одинаковые названия. Ввод их не столь уж трудоемок, но если компания надумает сменить название (а такое случается), то придется вручную исправить множество записей, что очень неудобно.

   Отсюда напрашивается седьмое требование к нашей системе: она должна уметь хранить названия фирм (и другие специфические элементы записей) отдельно от основных записей справочника и поддерживать *перекрестные ссылки* между различными группами записей. Так, хотя общепризнанных жанров компьютерных игр известно не много, процесс их появления никогда не прекращается. С развитием новых технологий старые жанры делятся на дополнительные категории, а на их стыках зарождаются другие.

   Поэтому, хотя и допустимо хранить фиксированный набор названий жанров, правильнее поступить с ними по аналогии с названиями фирм: выделить в отдельную группу, а в основных записях помещать только ссылки.

Понятие базы данных

На основе данного примера сформулируем понятие базы данных. Оно будет похоже на одно из множества классических определений баз данных, которое предложено международной Ассоциацией по языкам систем обработки данных КОДАСИЛ.

База данных — это совокупность записей различного типа, содержащая перекрестные ссылки.

В частности, структура записей описания игр будет отличаться от структуры записей описания фирм и записей описания жанров. Записи одного типа внутри базы данных хранятся в *таблицах*. Для фирм используется одна таблица, для игр — другая. Между записями каждой из таблиц устанавливаются ссылки: в записи таблицы игр имеется ссылка на записи таблицы с названиями фирм, таблицы с названиями жанров и так далее.

ВНИМАНИЕ Обратите внимание: в определении базы данных умышленно не упоминается понятие файла.

Файл — это совокупность записей одного типа, в котором перекрестные ссылки отсутствуют.

Более того, в определении вообще нет упоминания о компьютерной архитектуре. Дело в том, что, хотя в большинстве случаев база данных действительно представляет собой один или (чаще) несколько файлов, физическая их организация существенно отличается от логической. Таблицы могут храниться как в отдельных файлах, так и все вместе. И наоборот, для хранения одной таблицы иногда используются несколько файлов. Для поддержки перекрестных ссылок и быстрого поиска обычно выделяются дополнительные специальные файлы.

В ряде случаев базы данных реализуются вообще без привлечения файловой системы. Это связано как с ее ограничениями (в ряде операционных систем размер файла не может превышать 2 Гбайта), так и с необходимостью обеспечить максимально быструю работу с жестким диском. Например, некоторые системы работают с носителями, подвергнутыми простейшему форматированию, несовместимому со стандартами *DOS* или *Windows* и предназначенному исключительно для упрощения процесса доступа к данным. Есть программы, обрабатывающие данные на специализированных дисковых носителях *RAID* и множестве других периферийных устройств, назначение которых только одно: повысить скорость доступа к большим объемам сведений.

Поэтому при работе с базами данных обычно применяются понятия более высокого логического уровня: *запись* и *таблица,* без углубления в подробности их физической структуры.

Таким образом, сама по себе база данных — это только набор таблиц с перекрестными ссылками. Чтобы универсальным способом извлекать из нее группы записей, обрабатывать их, изменять и удалять, требуются специальные программы, которые называются *системами управления базами данных* или сокращенно СУБД.

Среди приведенных нами требований к будущему справочнику можно выделить как требования к обрабатываемой базе данных, так и требования к программе как системе управления этими данными.

Два последних требования (наличие перекрестных ссылок внутри одной таблицы и между таблицами) относятся к базе данных, остальные (формирование и выполнение запросов, создание отчетов, обеспечение надежности хранения и целостности данных) — к СУБД и различным вспомогательным программам.

Функциональные возможности разных СУБД лежат в весьма широком диапазоне. В некоторые комплексы встроены специализированные языки программирования и даже целые системы визуальной разработки, генераторы сложных отчетов, аналитические модули. Они способны поддерживать одновременную работу с данными десятков тысяч пользователей. Другие СУБД обладают только базовым набором возможностей (хранение данных в таблицах и выполнение запросов). При их использовании разработчикам приходится самостоятельно программировать работу по созданию отчетов, удобному просмотру и редактированию содержимого базы данных и так далее. В зависимости от предоставляемых ими средств СУБД обычно сильно различаются по цене.

Модели баз данных

Таблицы, в которых хранятся данные, состоят из наборов записей одинаковой структуры. Можно сказать, что таблица — это двумерный массив, где строки образованы отдельными записями, а столбцы — полями этой записи. Более точно таблица представляется как одномерный массив переменной длины из записей конкретной структуры (тип **record** Паскаля).

Модель базы данных, состоящей из подобных таблиц, называется *реляционной*. Практически все ведущие производители СУБД поддерживают именно эту модель баз данных, и в книге в дальнейшем будет рассказываться именно о ней. Реляционная модель хороша тем, что проста в работе и реализации и позволяет создавать быстро работающие системы.

Имеется еще несколько моделей баз данных. Некоторые из них значительно эффективнее реляционной, но не получили широкого распространения из-за сложности создания подходящих СУБД.

- В *иерархической* модели данные организованы в виде деревьев.
- В *сетевой* модели каждый узел (набор) базы данных взаимодействует с другими узлами посредством сложной системы связей.
- В последнее время признание завоевывает *объектная* модель данных, когда в базе хранятся не только данные, но и методы их обработки в виде программного кода. Это перспективное направление, пока также не получившее активного распространения из-за сложности создания и применения подобных СУБД.

Архитектура СУБД

Приложения, использующие базы данных, обычно принято относить к одной из программных архитектур, имеющих свои плюсы и минусы.

Локальная архитектура

И программа, и база данных расположены на одном компьютере. В такой архитектуре работает большинство настольных приложений.

Файл-серверная архитектура

База данных расположена на мощном выделенном компьютере (сервере), а персональные компьютеры подключены к нему по локальной сети. На этих компьютерах установлены клиентские программы, обращающиеся к базе данных по сети. Преимущество такой архитектуры заключается в возможности одновременной работы нескольких пользователей с одной базой данных.

Недостаток такого подхода — большие объемы информации, передаваемой по сети. Вся обработка выполняется на клиентских местах, где фактически формируется копия базы данных. Это приводит к ограничению максимально возможного числа пользователей и большим задержкам при работе с базой. Эти задержки вызываются тем, что на уровне конкретной таблицы одновременный доступ невозможен. Пока программа на одном из клиентских мест не закончит работу с таблицей (например, не выполнит модификацию записей), другие программы не могут обращаться к этой таблице. Это называется *блокировкой на уровне таблицы* и исключает возникновение путаницы в ее содержимом.

Клиент-серверная архитектура

В такой архитектуре на сервере не только хранится база данных, но и работает программа СУБД, обрабатывающая запросы пользователей и возвращающая им наборы записей. При этом программы пользователей уже не работают напрямую с базой данных как набором физических файлов, а обращаются к СУБД, которая выполняет операции. Нагрузка с клиентских мест при этом снимается, так как большая часть работы происходит на сервере. СУБД автоматически следит за целостностью и сохранностью базы данных, а также контролирует доступ к информации с помощью службы паролей. Клиент-серверные СУБД допускают блокировку на уровне *записи* и даже отдельного *поля.* Это означает, что с таблицей одновременно может работать любое число пользователей, но доступ к функции изменения конкретной записи или одного из ее полей обеспечен только одному из них.

Основной недостаток этой архитектуры — не очень высокая надежность. Если сервер выходит из строя, вся работа останавливается.

Распределенная архитектура

В сети работает несколько серверов, и таблицы баз данных распределены между ними для достижения повышенной эффективности. На каждом сервере функционирует своя копия СУБД. Кроме того, в подобной архитектуре обычно используются специальные программы, так называемые *серверы приложений.* Они позволяют оптимизировать обработку запросов большого числа пользователей и равномерно распределить нагрузку между компьютерами в сети. Если, помимо работы с данными, требуется выполнить интенсивные вычисления (например, анализ сложной информации), программы для выполнения этих задач (они обычно называются *компонентами)* могут автоматически запускаться на более мощных сетевых компьютерах. Это практически полностью снимает нагрузку с клиентских мест. Такая архитектура также называется *компонентной.*

Недостаток распределенной архитектуры заключается в довольно сложном и дорогостоящем процессе ее создания и сопровождения (администрирования), а также в высоких требованиях к серверным компьютерам.

Интернет-архитектура

Доступ к базе данных и СУБД (расположенным на одном компьютере или в сети) осуществляется из браузера по стандартному протоколу. Это предъявляет минимальные требования к клиентскому оборудованию. Такие программы называют «тонкими клиентами», потому что они способны работать даже на ПК с процессором 80386.

Благородя стандартизации всех протоколов и интерфейсов взаимодействия в Интернете такие системы легко создавать и внедрять. Например, можно не организовывать локальную сеть, а обращаться к серверу через Интернет или использовать протоколы Интернета в локальной сети (в таком случае говорят о технологиях интранет). В этом случае не требуется разрабатывать специальные клиентские программы или придумывать собственные спецификации обмена данными между сервером и клиентскими местами. Достаточно использовать готовые браузеры и Программные решения.

В данной книге рассматриваются способы создания программ для всех описанных архитектур.

Реализация работы с СУБД в системе Delphi

Технология BDE для доступа к данным

При создании программ, работающих с базами данных, в системе *Delphi* традиционно используется механизм *Borland Database Engine (BDE).* В состав *Delphi 7* входит версия *BDE 5.2,* которую, впрочем, можно бесплатно обновлять разными способами (например, обратившись к *Web-узлу* http://www.borland.com/).

Этот механизм реализован в виде набора библиотек, которые обеспечивают для программы, написанной на Паскале, простой и удобный доступ к базам данных независимо от их архитектуры. При использовании механизма *BDE* разработчик может не задумываться о том, как его программа будет работать с базой данных на физическом уровне: локально, в файл-серверной, либо в клиент-серверной архитектуре. Вдобавок при переходе к использованию СУБД разных производителей программисту не потребуется менять исходный код своей программы. Достаточно внести изменения только в настройки *BDE.*

Драйверы баз данных

Такое удобство достигается благодаря тому, что механизм *BDE* представляет собой программную прослойку (*middleware*) между клиентской программой и базой данных (или СУБД). Запрос из приложения передается внутрь механизма *BDE,* который использует специализированные системные программы (драйверы) для непосредственной работы с СУБД.

ВНИМАНИЕ В дальнейшем для простоты под СУБД будем понимать как специальную серверную программу управления базой данных, так и драйвер для обращения к наборам файлов в формате файл-серверных СУБД.

Такие драйверы выпускаются для каждой СУБД, и механизм *BDE* настраивается на их использование с помощью специального редактора, вызываемого из утилиты SQL Explorer (Проводник SQL), которая открывается командой Database ➤ Explore (База данных > Проводник). Драйверы могут работать с базами данных в стандарте таких СУБД, как *Paradox, dBase, FoxPro.* Эти СУБД ранее были построены как файл-серверные, поэтому драйвер реально представляет собой весьма сложную программу, выполняющую множество функций СУБД.

Существуют и драйверы для работы с клиент-серверными СУБД *(MS SQL Server, InterBase, Oracle).* Такие драйверы устроены проще. Они только передают запросы и команды из *BDE* в СУБД и получают обратно результаты их выполнения. Всю работу по обработке данных выполняет СУБД.

В поставку *BDE* входит два набора драйверов.

- Первый набор предназначен для файл-серверных СУБД *dBASE, Paradox, FoxPro, Access* и данных в текстовом формате.
- Второй набор ориентирован на клиент-серверные СУБД *InterBase, IBM DB2, Informix, ORACLE, Sybase* и *Microsoft SQL Server.* Этот набор называется *SQL Links.*

Конечно, кроме системы *Delphi* в мире существует немало пакетов создания программ, которые позволяют обращаться к любым СУБД. Поэтому давно разработан и существует стандартный протокол *ODBC (Open Database Connectivity Interface,* открытый интерфейс взаимодействия с базами данных), напоминающий независимую работу *BDE.* Драйверы *ODBC* выпущены для всех без исключения СУБД, и разработчик может использовать в *BDE* драйверы *ODBC.*

Реализация в системе *Delphi* прослойки *BDE* позволяет не привязывать программу к конкретной СУБД. Если потребуется расширить число пользователей программы и перейти, например, с файл-серверной СУБД *dBase* на более мощную СУБД *InterBase,* достаточно изменить несколько настроек *BDE,* не исправляя исходные тексты.

ЗАМЕЧАНИЕ Начиная с седьмой версии Delphi, корпорация Borland отказалась от поддержки набора SQL Links для клиент-серверных приложений, работающих с базами данных. При создании клиент-серверных программ предлагается использовать набор компонентов dbExpress.

Утилиты для работы с СУБД

Создание базы данных

Структура базы данных

Теперь, когда мы познакомились с понятием СУБД и принципами работы механизма *BDE,* перейдем к практической реализации справочника игр.

Используемая база данных будет состоять из четырех таблиц. В рамках каждой из них хранятся наборы записей одинаковой структуры.

В первой таблице (назовем ее Firms) будут храниться названия фирм, во второй - жанры (Genres), в третьей — ссылки на *Web*-страницы Интернета и статьи в журналах (Articles). Одной игре может быть посвящено несколько Web-страниц и статей. Четвертая таблица (Games) будет основной.

Следующий шаг — выбор архитектуры базы данных и конкретной СУБД. Так как справочник рассчитан на одного пользователя, достаточно использовать локальную СУБД. Допустим, это будет СУБД *Paradox,* поддерживаемая механизмом *BDE.*

ЗАМЕЧАНИЕ Все локальные СУБД, реализация доступа к которым имеется в механизме 6DE, обладают малосущественными отличиями. Их выбор определяется, скорее, личными пристрастиями разработчика.

Если бы программа создавалась для клиент-серверной архитектуры, то предварительно потребовалось бы средствами самой СУБД сформировать на жестком диске базу данных и входящие в нее таблицы. В нашем случае для автономного приложения целостного понятия «база данных» не существует, это просто набор из четырех таблиц, хранимых в отдельных файлах.

Создание таблиц

Для создания таблиц в системе *Delphi* 7 имеется приложение Database Desktop (Работа с автономной СУБД). Оно вызывается командой Tools ➤ Database Desktop (Сервис > Работа с автономной СУБД) (рис. 5.1).

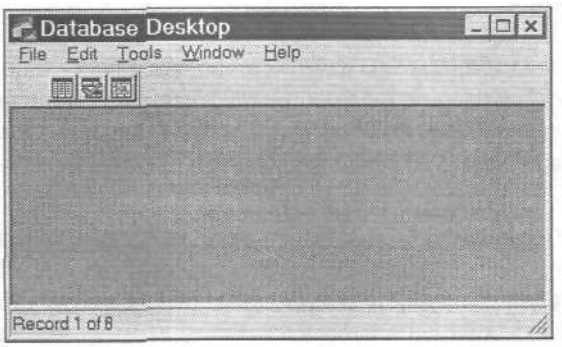


Рис. 5.1. Окно конструирования автономной базы данных

Новая таблица создается командой File > New ➤ Table (Файл ➤ Создать ➤ Таблица). В открывшемся диалоговом окне надо выбрать формат таблицы (каждая СУБД хранит данные в собственном формате). Укажите в раскрывающемся списке Table type (Формат таблицы) пункт Paradox 7 (формат СУБД *Paradox* версии 7) и щелкните на кнопке OK. Следующее окно предназначено для формирования структуры создаваемой таблицы (рис. 5.2).

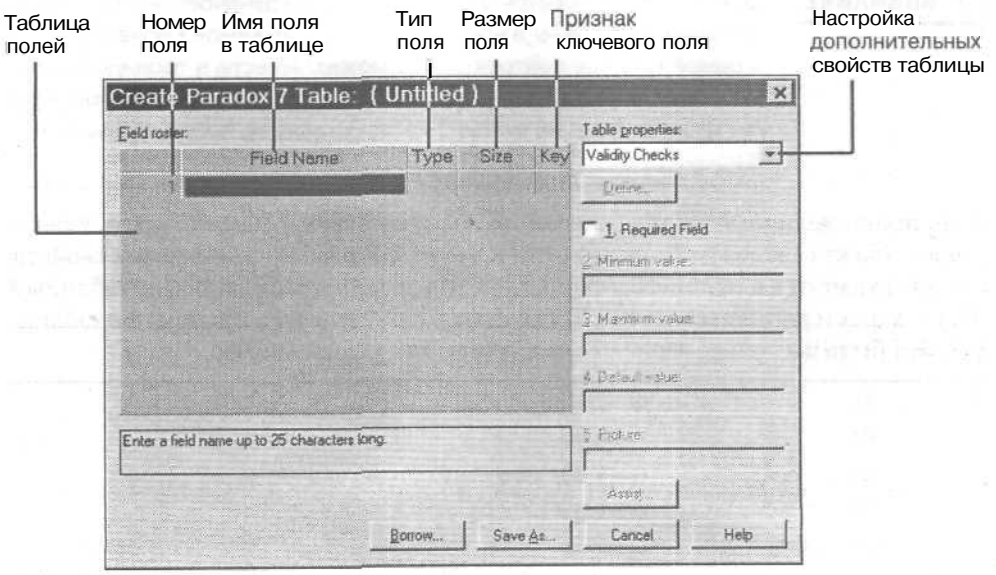


Рис. 5.2. Формирование структуры таблицы базы данных

В первом столбце автоматически указывается порядковый номер поля, во втором задается название поля внутри таблицы, в третьем — тип поля, в четвертом - размер поля (для текстовых полей). В пятом столбце указывается, является ли данное поле *ключевым*.

Ключевое поле

Как правило, каждая таблица базы данных включает *ключевое* поле. Оно содержит уникальный идентификатор, позволяющий отличить одну запись от другой. Например, каждая фирма и каждая игра могут (и должны) иметь такой неповторимый идентификатор (или *ключ).* В качестве ключей обычно используется цифровое значение.

Во-первых, ключи нужны для повышения эффективности доступа к данным внутри таблицы (это связано с технологией работы реляционных СУБД), а во-вторых, они используются для создания перекрестных ссылок между таблицами. В нашем случае, чтобы сослаться из записи, описывающей игру, на запись таблицы, хранящей название фирмы-производителя, достаточно указать ключ этой записи.

В качестве ключа может выступать, конечно, и название фирмы, но оно будет занимать много места (для ключа-числа достаточно нескольких байтов), что понизит эффективность работы СУБД. Кроме того, если название изменится, придется корректировать все записи таблицы Games, где упоминается эта фирма, что совсем неудобно. При использовании цифрового ключадостаточно внести изменение только в одну запись таблицы Firms,

ВНИМАНИЕ Если в таблице имеется ключевое поле, это означает, что все значения в нем уникальны и неповторимы. Содержимое ключевого поля в одной таблице повторяться не может. То есть, в таблице Games не может быть двух описаний (записей), посвященных одной игре, а в таблице Firms не может быть двух записей, посвященных одной фирме.

В принципе, наличие в таблице ключевого поля не является обязательным. У некоторого объекта с ключом К1 из таблицы А может быть набор однотипных свойств, который хранится в таблице Б в формате «ключ родительской записи из таблицы А (К1) — характеристики свойств». Тогда для каждой записи из таблицы А в таблице Б может быть несколько записей с ее ключом, как указано ниже.

К1	свойства
К1	свойства
К1	свойства
К2	свойства
КГ	свойства
...	
Кп	свойства
Кп	свойства

Таким образом, таблица Б не содержит ключевого поля. Составить набор нужных записей для объекта из таблицы А с ключом К5 можно с помощью следующего условного запроса.

```
выбрать из таблицы Б все записи,
значение первого поля которых равно К5.
```

Именно по такому принципу мы организуем, в частности, таблицу Articles, в которой каждая запись будет состоять из двух частей. В первой хранится ключ записи из таблицы Games, а во второй — текстовый адрес *Web*-страницы или сведения о номере журнала. Так как одной игре может быть посвящено несколько **ffeA-страниц** или статей, возникает только что описанная структура, в которой уникальность значений в каждом столбце отсутствует. Могут повторяться как ключи, так и имена *Web*-узловдля разных игр.

Конечно, создать ключ можно и для таблицы Б. Для этого потребуется дополнительное поле.

```
ключ / ключ родительской записи / свойства
```

ПОДСКАЗКА Ключевые поля рекомендуется создавать для всех таблиц. Использование ключей позволяет существенно повысить скорость работы с таблицами,

Индексация

С понятием ключа тесно связано понятие *индекса.* Если таблица предназначена для выдачи наборов данных на основании всевозможных запросов («отобрать все игры, вышедшие в указанный период», «посмотреть все ролевые игры, в которые я еще не играл»), то такую таблицу желательно проиндексировать по каждому полю, для которого ожидается активное использование в запросах.

Индексы содержат краткую информацию о каждой записи и организованы таким образом, что позволяют очень быстро получать доступ к нужной информации без сканирования всей таблицы.

В СУБД *Paradox 7* ключевое поле всегда индексировано. Такой индекс называется *первичным* и всегда один. Для увеличения скорости сортировки и поиска данных в других полях можно добавить неограниченное число *вторичных* индексов.

Создание таблицы Genres

Вернемся к открытому окну приложения Database Desktop. Первый столбец содержит номер поля и пока недоступен для изменения. Второй столбец хранит название поля.

Какой должна быть структура таблицы, хранящей названия жанров игр? В ней можно создать единственное текстовое поле, но в этом случае перекрестные ссылки организовываются неэффективно, В них войдут длинные текстовые строки, что при больших объемах информации приведет к существенному замедлению работы всей системы.

Поэтому правильно сделать так: завести для каждого названия жанра свой ключ (два или четыре байта), а в таблице при ссылке на жанр указывать именно его.

С помощью запросов в программе формируются комбинации этих таблиц, благодаря чему для пользователя отображаются не ключи, а названия жанров. Выполняется это примерно таким условным запросом:

```
показать список пар значений 'название игры - жанр', для
которых значение поля Жанр из таблицы Games равно значению
ключевого поля из таблицы Genres
```

При работе с таблицами разработчику приходится мыслить категориями множеств, отличными от категорий обычного, линейного программирования. Если при написании программ на Паскале используются переменные, то при работе с базами данных работа выполняется над *группами* (или *наборами)* записей, поля которых выбираются из нескольких таблиц подчас но довольно сложным правилам комбинирования. Нередко из-за этого профессиональные разработчики и проектировщики баз данных и профессиональные разработчики программ плохо понимают друг друга, так как общаются на разных уровнях абстракции.

Проектирование баз данных — это искусство не менее сложное, чем искусство проектирования программ. В современных коммерческих приложениях базы данных состоят из тысяч таблиц, каждая из которых насчитывает сотни полей. Спроектировать все так, чтобы обеспечивалась эффективная работа при выполнении разных запросов, очень сложно.

Ключевое поле, как правило, указывается первым. Некоторые СУБД требуют этого в обязательном порядке. Например, таков используемый сейчас формат СУБД *Paradox 7.* Отступать от этого порядка не рекомендуется. Название ключевого поля обычно тоже стандартно — ID (от английского *Identifier — идентификатор).*

Каким выбрать тип ключевого поля (столбец Type)? Значение для каждой новой записи должно быть оригинальным, поэтому проще всего начать условную нумерацию записей с 1 и каждой следующей записи давать номер (значение ключа) на единицу больше предыдущего.

Этот процесс для таблиц с ключами настолько типичен, что он был автоматизирован. Щелкните правой кнопкой мыши над областью поля Type (Тип поля) и выберите в контекстном меню значение Autoincrement (Автоприращение). Теперь при добавлении в таблицу новой записи значение данного поля будет автоматически увеличиваться и разработчику не придется отслеживать уникальность первичных ключей. Пункт столбца Size (размер поля записи) заполняется автоматически, а вот в столбце Key надо указать, что данное поле записи ключевое. Нажатие клавиши ПРОБЕЛ приведет к появлению в этом столбце звездочки — признака ключевого поля. Убрать этот признак можно повторным нажатием той же клавиши.

Для каждого поля можно дополнительно указать, требуется ли его обязательное заполнение при добавлении новой записи. И для ключевого поля, и для названия жанра дело обстоит именно так, поэтому надо установить флажок Required Field (Обязательное поле). В текстовом поле Minimum value (Минимальное значение) окна можно указать начальное (минимальное) значение данного поля, то есть число, с которого начнется отсчет индексов. Укажите в этом поле значение 0 (рис. 5.3).

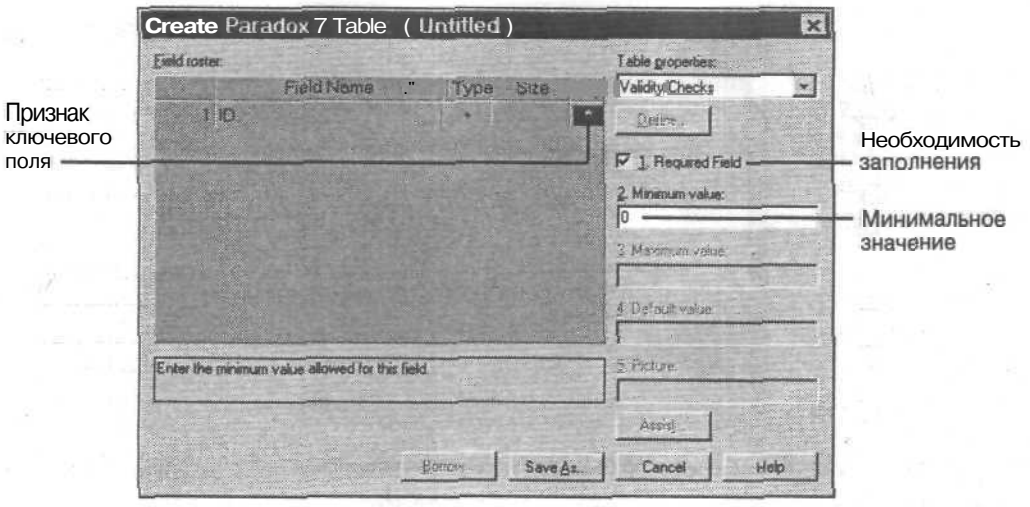


Рис. 5.3. *Задание поля и его свойств*

Переход к следующему полю происходит с помощью клавиши ВНИЗ. Теперь надо ввести название поля с текстом, описывающим жанр. Пусть это название имеет вид GenreName. Далее указывается тип поля. В СУБД *Paradox 7* имеется специальный тип Alpha для хранения текстовых значений. Максимальная длина такого поля 255 символов. Укажите в столбце Size число 50, что для названия жанра более чем достаточно. Затем установите флажок Required Field (Обязательное поле).

Таблицу можно довольно гибко настроить для определенного режима работы с использованием раскрывающегося списка Table properties (Свойства таблицы).

Определим для таблицы вторичный ключ. Для этого в раскрывающемся списке выберем строку Secondary Indexes (Вторичные ключи) и щелкнем на кнопке Define (Определить). В появившемся диалоговом окне переместим с помощью кнопки со стрелкой поле GenreName в список Indexed fields (Индексированные поля). Поле ID уже используется в качестве первичного индекса. Укажем также, что все значения в поле GenreName уникальные, установив флажок Unique (Уникальное).

По щелчку на кнопке OK приложение попросит ввести имя для вторичного индекса (рис. 5.4). Необходимо указать оригинальное имя, не совпадающее с именами полей, например GenreID.

ЗАМЕЧАНИЕ

Запрос появляется, если вторичный индекс создается на основе поля, в котором индексируемые значения не различаются по регистру букв (то есть, строки «Blizzard» и «BUZZARD» считаются одинаковыми). Это определяется флажком Case sensitive (Чувствительность к регистру). В противном случае имя вторичного индекса генерируется автоматически.

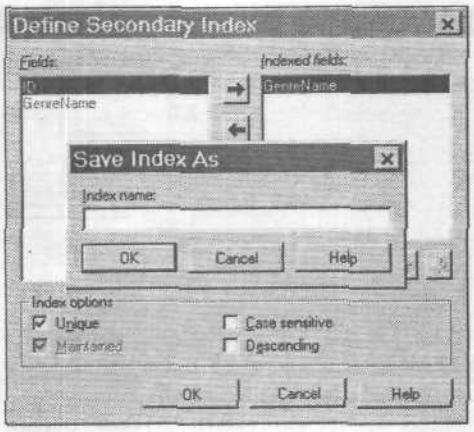


Рис. 5.4. Выбор имени для вторичного индекса

На этом создание таблицы закончено. В будущем можно будет внести дополнительные изменения в ее структуру. Осталось щелкнуть на кнопке Save As (Сохранить таблицу) и выбрать подходящий каталог с помощью открывшегося диалогового окна.

Программа предложит сохранить созданную таблицу в файле Genres.db. Расширение .DB является стандартным для файлов *Paradox*. В поле Имя файла можно указать как полный путь поиска, так и «псевдоним» *(alias)* каталога, в который будет записана таблица.

Удобство использования псевдонимов состоит в том, что они позволяют назначить длинному пути поиска сокращение (например, :GAME:), а затем, при переносе программы на другой компьютер, быстро переопределить значение псевдонима, не привязываясь заранее к конкретному каталогу. В данной ситуации лучше всего сохранить таблицу в каталог :WORK:, предлагаемый по умолчанию в качестве рабочего каталога.

Таким же способом надо создать таблицу Firms. В ней тоже будет два поля, ключевое ID и текстовое (50 символов) FirmName (название фирмы). Для поля FirmName надо создать вторичный индекс FirmNameID.

Таблица для хранения ссылок на *Web-узлы* и статьи журналов оказывается несколько сложнее. Для нее надо определить три поля: ключевое поле ID, поле типа Long Integer (Длинное целое) с именем GameID и текстовое поле Info длиной 100 символов. В поле GamelD будет храниться ключ записи из таблицы Games, содержащий информацию об игре, для которой предназначен данный *Web*-узел. Все поля надо пометить как требующие наличия значения.

Проиндексировать таблицу можно по полю Info. Требовать уникальности значений в этом поле не следует, потому что один и тот же сайт может быть посвящен нескольким играм. Можно также провести индексацию по полю GamelD. Укажите в качестве имени вторичного индекса GameIND. Этот индекс нам еще понадобится.

ЗАМЕЧАНИЕ После создания таблиц их структура, содержимое и индексы будут сохранены в нескольких файлах. Каждая таблица Paradox, например, хранится в четырех файлах. Это надо учитывать при создании архивных копий базы данных.

Главная таблица

Теперь осталось создать основную таблицу Games. В ней содержатся следующие поля.

Таблица5.1. Поля таблицы Games

Имя поля	Тип поля
ID	Ключевое юле. Начнем индексацию с 1 (поле Minimum Value)
Name	Лоле типа Alpha длиной 30 символов. Название игры
GenreID	Поле типа Long Integer. Ключ записи из таблицы Genres, определяющий жанр игры
Played	Поле типа Logical (аналог типа Boolean в Паскале). Признак того, удалось пи сыграть в данную игру
Publishier	Поле типа Long Integer. Ключ записи из таблицы Firms, определяющий фирму-издателя (можно указывать значение 0)
Developer	Поле типа Long Integer. Ключ записи из таблицы Firms, определяющий фирму-разработчика (можно указывать значение 0}
GameYear	Поле типа Date. Год выхода игры
Season	Поле типа Alpha длиной 20 символов- Время выхода (текстовое описание, например: осень)
Net	Поле типа Logical. Возможность сетевой игры
Playability	Поле типа Number. Оценка сюжета игры в баллах
Graphics	Поле типа Number. Оценка графики в баллах
Sound	Поле типа Number. Оценка звука в баллах
Comment	Поле типа Memo. Комментарий теоретически произвольной длины. Комментарии хранятся отдельно от значений других полей за исключением некоторой части, длину которой указывать обязательно. Пусть это будет 100 символов (поле Size). В эту длину, скорее всего, уложится большая часть комментариев

Для всех полей, за исключением поля Comment, надо указать обязательное наличие значения. Для таблицы можно создать вторичный индекс на основе поля Name.

В дальнейшем некоторые свойства таблиц и полей можно будет менять, выполнив команду приложения File ➤ Open ➤ Table (Файл ➤ Открыть ➤ Таблица). После появления таблицы на экране (отображается ее содержимое, первоначально пустое) надо перейти в режим ее реструктуризации щелчком на кнопке Restructure (Изменить структуру). После внесения изменений щелкните на кнопке Save (Сохранить), для их фиксации.

ВНИМАНИЕ Перестраивать таблицы после внесения данных очень опасно. Это может привести к неработоспособности приложения.

Перекрестные ссылки

Важный заключительный этап формирования базы данных состоит в создании перекрестных ссылок между таблицами. На этом этапе надо явно указать, например, что ключевое поле таблицы Genres связано с полем GenreID таблицы Games. Это выполняется так: в раскрывающемся списке Table Properties выбирается пункт Referential Integrity (Целостность ссылок). С помощью кнопки Define (Определить) определяются все связи данной таблицы с ключевыми полями других таблиц.

По щелчку на этой кнопке открывается диалоговое окно, в левой части которого указан список всех полей таблицы, а в правой — другие таблицы в рабочем каталоге. Зададим связь поля GenreID с ключевым полем таблицы Genres, которая считается родительской. Выберите поле в левом списке и щелкните на кнопке со стрелкой вправо. В центральной области окна в колонке Chield field (Подчиненное поле) появляется название GenreID [I]. Обозначение [I] указывает на тип этого поля.

В правом списке надо выбрать таблицу Genres.db и щелкнуть на кнопке со стрелкой влево. В столбце Parent's key (Ключевое поле родительской таблицы) появится надпись ID [+]. Знак «плюс» означает признак ключевого поля (рис. 5.5).

Рис. 5.5. Формирование перекрестных ссылок между таблицами

После этого можно щелкнуть на кнопке OK и в небольшом диалоговом окне ввести произвольное название созданной связи, предположим GenreInt. После этого снова щелкните на кнопке OK. В списке ниже кнопки Define (Определить) возникнет новая связь GenreInt. Таким же способом надо создать связи DevInt (поле Developer и ключ таблицы Firms) и PubInt (поле Publisher и ключ таблицы Firms).

Мы не организовали явную связь с таблицей Articles, потому что это возможно только на основе ключевого поля, которое в таблице Articles не несет смысловой нагрузки. Поле GameID не подходит для создания связи, потому что там могут храниться одинаковые (не уникальные) значения индексов игр.

Если сохранить текущую таблицу под именем Games и перейти к структуре таблицы Firms, то при выборе в раскрывающемся списке Table properties (Свойства таблицы) свойства Dependent Tables (Подчиненные таблицы) в списке появятся два упоминания таблицы Games.db. Это связано с тем, что на ключевое поле текущей таблицы имеются две ссылки из таблицы Games. На этом создание структуры базы данных и таблиц можно считать законченным.

Отношения между таблицами

Среди проектировщиков баз данных принята определенная терминология, которую полезно знать и прикладным программистам. Отношения между записями в таблицах делятся на три типа.

Таблица 5.2. Отношения между записями

Тип отношения	Способ связи
Один и одному	Каждой записи соответствует роено одна запись другой таблицы. В принципе, можно вообще обойтись одной таблицей
Один ко многим	Примером может служить связь таблиц Games и Genres. В таблице Genres хранится только одна запись, описывающая конкретный жанр, а в таблице Games записей, ссылающихся на этот жанр, может быть много
Многие ко многим	Это отношение не всегда поддерживается в реляционных СУБД, и для его реализации часто вводят промежуточные таблицы. Например, таблица Articles нужна именно для таких целей. Она способна хранить ключи, позволяющие описать такие отношения, когда одна игра обсуждается на нескольких узлах (в нескольких статьях) и, наоборот, один узел посвящен нескольким играм

Добавление базы данных в BDE

База данных создана в виде набора четырех таблиц, но система *BDE* пока не знает об их существовании. Хотя работать с этими таблицами в принципе можно, указывая полные пути поиска в свойствах соответствующих компонентов, такой подход неправилен. Он не позволяет работать с базой данных как целостным понятием и напрямую обращаться к ней по имени (псевдониму).

При регистрации в системе *BDE* созданной группы таблиц как целостной базы данных поможет приложение SQL Explorer (Проводник SQL), запускаемое командой Database ➤ Explore (База данных ➤ Проводник). В левой части окна приводится список всех зарегистрированных в системе *BDE* баз данных, в правой — свойства текущей базы, выбранной в этом списке.

Создадим новую базу данных. Для этого выполняется команда Object ➤ New (Объект > Создать) и в диалоговом окне выбора драйвера указывается значение STANDARD (рис. 5.6).

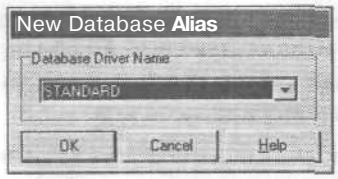


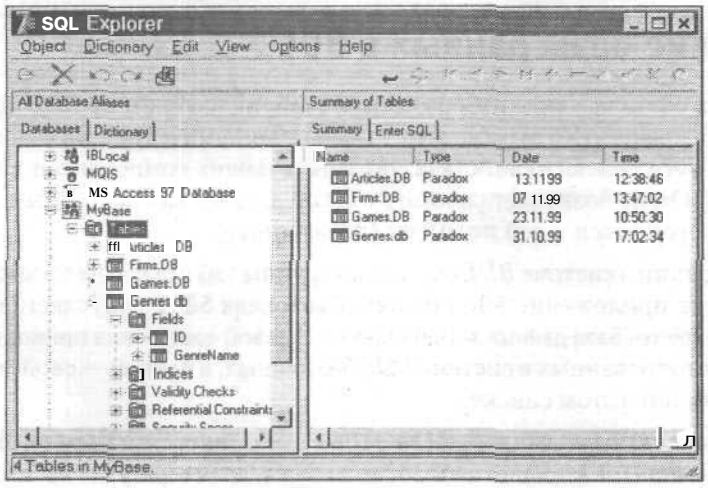
Рис. 5.6. *Создание базы данных и выбор драйвера*

После щелчка на кнопке OK в списке появится новый элемент, помеченный зеленым треугольником. Это означает, что регистрация базы данных не завершена. По умолчанию формируется имя базы STANDARD1, изменим его на MyBase. Убедимся, что в свойстве DEFAULT DRIVER (Драйвер по умолчанию) стоит значение PARADOX. В свойстве PATH (Путь поиска для каталога, в котором хранятся таблицы) укажем рабочий каталог (для которого создан псевдоним :WORK:). Этот каталог, как правило, отображается первым при щелчке на кнопке выбора.

Теперь зарегистрированную в системе *BDE* базу надо сохранить. Для этого в контекстном меню объекта MyBase выберем пункт Apply (Применить настройки). На вопрос о необходимости сохранения изменений ответим Yes (Да). Теперь наши таблицы доступны из среды *BDE* под именем базы данных MyBase. Если раскрыть объект MyBase, щелкнув на значке «+» перед его именем, па правой панели SQL Explorer будут показаны все четыре таблицы, а значок базы помечается зеленой рамкой, указывающей, что база данных MyBase открыта.

ЗАМЕЧАНИЕ Работа с базами данных напоминает работу с файлами. Их сначала надо открыть, а после выполнения всех операций — обязательно закрыть.

Таким же образом можно раскрыть каждую таблицу и посмотреть список ее полей (Fields) с описанием типов и других свойств (рис. 5.7).



ft/c. *5.7. Просмотр структуры базы данных при помощи SQL Explorer*

После изучения структуры базу MyBase надо закрыть с помощью команды Close контекстного меню. Выделение объекта будет снято. Теперь можно закрыть окно SQL Explorer и вернуться в систему *Delphi 7*.

Работа с автономными СУБД на ПК

Создание модуля данных

Доступ к базе данных в системе *Delphi 7* выполняется достаточно просто, с использованием богатого набора невизуальных компонентов работы с СУБД. Как правило, эти компоненты группируются в создаваемой программе в специальном модуле данных (TDataModule). Модуль данных представляет собой хранилище объектов, которое позволяет централизованно управлять их работой и отделяет программную логику, связанную с базами данных, от программного кода, выполняющего вычислительные действия и отображение данных на форме.

Создадим новый проект и добавим в него модуль данных командой File > New ➤ Data Module (Файл > Создать ➤ Модуль данных) (рис. 5.8). Чтобы из главной формы можно было получить доступ к содержимому модуля данных, в списке доступных модулей в файле Unit1.pas нужно указать его имя (Unit2).

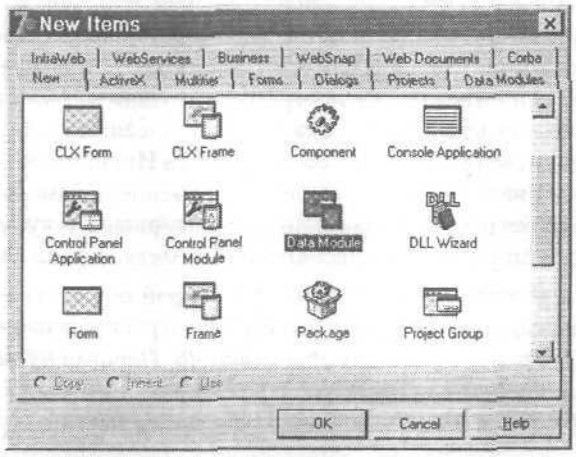


***Рис. 5.8.** Создание нового модуля данных*

```
unit Unit1;
interface
uses
Windows, Messages, SysUtils,
Classes, Graphics, Controls,
Forms, Dialogs,
Unit2;
```

Доступ к таблицам базы данных

В модуле данных необходимо разместить компоненты, которые обеспечат доступ к нужным нам четырем таблицам базы данных (рис. 5.9).

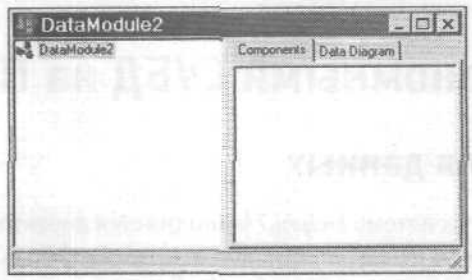


Рис. 5.9. *Управление доступом к таблицам базы данных из модуля данных*

Для доступа к таблице любой базы данных используется компонент TTable с панели компонентов BDE (Доступ к данным через механизм BDE). Дерево, описывающее логические связи между компонентами, можно просмотреть, выполнив команду View ➤ Object TreeView (Просмотр ➤ Дерево объектов). Можно изменять положение объектов в этом дереве методом перетаскивания, а также добавлять к ним новые объекты. С помощью правой кнопки мыши можно быстро получить доступ к основным свойствам каждого из элементов дерева. Все они доступны также из контекстного меню объектов, расположенных в правой части окна.

Первоначально узлы Alias (псевдоним базы данных) и Table1 (объект класса TTaIe) выделяются красным цветом. Это означает, что объект определен не полностью и пока что не может использоваться в программе. Это действительно так: ведь доступ к таблице с помощью объекта Tablel пока не организован. Поэтому в модуле данных надо выделить единственный объект Table1. Теперь в Инспекторе объектов для свойства DatabaseName (Имя базы данных) выберите значение MyBase. В раскрывающемся списке перечисляются и другие базы данных, доступные в текущей системе. После этого на левой панели модуля данных красная пометка с узла Alias будет снята.

Теперь требуется указать таблицу, связь с которой осуществляет объект Tablel. Нужная таблица базы данных MyBase выбирается в раскрывающемся списке свойства TableName. Пусть это будет таблица Games.db. Переименуем таблицу Tablel в Games, чтобы в дальнейшем ориентироваться в названиях было проще. Таким же образом надо добавить в модуль данных и остальные таблицы (рис. 5.10).

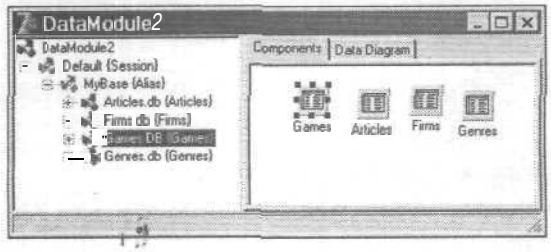


Рис. ***5.10.*** *Добавление таблиц в модуль данных*

Чтобы к этим таблицам можно было обращаться из программы, для каждой из них свойству Active надо присвоить значение Erue.

Динамические и постоянные поля

В дальнейшем поля таблиц можно использовать в программе двумя способами: либо как динамические (*dynamic*), либо как постоянные *(persistent)*.

Предположим, что используются исключительно поля, исходно присутствующие в таблицах, и над ними планируется выполнять только простые операции (просмотр, редактирование). С такими полями можно работать как с динамическими (такой режим принят по умолчанию).

Однако чаще всего возникает потребность в более гибкой обработке данных с возможностью создания временных дополнительных «псевдополей», например, вычисляемых на основе других значений. В то же время другие компоненты будут расценивать такие поля как полноценные поля конкретной таблицы, что позволит, манипулируя их свойствами, быстро настраивать нужные режимы отображения информации.

Для этого на основе таблицы создают набор постоянных полей. Когда таблица выбрана в модуле данных, надо выбрать в контекстном меню пункт Fields Editor (Редактор полей). В появившемся списке постоянных полей, первоначально пустом, откройте контекстное меню. Все поля таблицы можно добавить в список с помощью команды Add all fields (Добавить все поля).

В дальнейшем поля можно удалять из списка (физически они остаются в таблице), возвращать обратно, менять порядок следования, настраивать свойства (например, свойство видимости Visible, которое недоступно для динамических полей) и так далее.

С такими псевдополями очень удобно работать еще и потому, что они добавляются в модуль данных как отдельные поля класса TDataModule. К ним можно обращаться напрямую как к переменным по их названиям на основе описания класса:

```
type
TDataModule2 = class(TDataModule)
Games: TTable;
Articles: TTable;
Firms: TTable;
SourceArticles: TDataSource;
SourceGames: TDataSource;
SourceFirms: TDataSource;
ArticlesID: TAutoIncField;
ArticlesGameID: TIntegerField;
ArticlesInfo: TStringField;
```

продолжение

```
  FirmsID: TAutoIncField;
  FirmsFirmName: TStringField;
  Genres: TTable;
  SourceGenres: TDataSource;
  GenresID: TAutoIncField;
  GenresGenreName: TStringField;
  GamesID: TAutoIncField;
  GamesName: TStringField;
  GamesGenreID: TIntegerField;
  GamesPlayed: TBooleanField;
  GamesLastGameID: TIntegerField;
  GamesPublishier: TIntegerField;
  GamesDeveloper: TIntegerField;
  GamesGameYear: TDateField;
  GamesSeason: TStringField;
  GamesNet: TBooleanField;
  GamesPlayability: TFloatField;
  GamesGraphics: TFloatField;
  GamesSound: TFloatField;
  GamesComment: TMemoField;
  private
  { Private declarations }
  public
  { Public declarations }
  end;
```

Обратите внимание, что к таким объектам перед названием соответствующего им поля приписывается впереди название таблицы. Например, поле Net таблицы Games будет представлено в классе TDataModule2 как переменная GamesNet.

ПОДСКАЗКА Следует учитывать, что после создания постоянных полей программа будет считать, что у таблицы имеются только поля, указанные в этом списке (их может быть меньше или больше, чем реальных полей). Из этих соображений лучше всегда использовать постоянные поля как «промежуточный уровень» между физическими таблицами и их представлением в программе, создавая базовый набор таких полей сразу же после добавления таблицы в модуль данных.

Источники данных

Экранные элементы для работы с СУБД во многом напоминают обычные элементы управления *Windows*. Они отличаются тем, что предназначены для редактирования полей таблиц, а не переменных программы. В принципе, их можно напрямую подключать к компонентам TTable, однако в системе *Delphi 7* реализо-

ван более гибкий подход — создан компонент промежуточного уровня TDataSource (Источник данных) с панели Data Access (Доступ к данным).

Этот компонент служит посредником между таблицами СУБД и экранными элементами управления. Зачем так сделано? Компонент TDataSource позволяет, во-первых, абстрагироваться от конкретной СУБД. Например, если потребуется сделать многопользовательский справочник по играм, достаточно настроить таблицы на другую, более мощную СУБД с аналогичной схемой базы данных, способную работать в сети. Элементы управления этого не заметят, потому что будут обращаться к источнику данных, а не к конкретным таблицам. Во-вторых, в системе *Delphi 7* понятие источника данных значительно шире, чем таблица. Таким источником теоретически может служить любое виртуальное устройство, способное предоставлять данные в виде наборов записей. Например, записи могут генерироваться программными компонентами, исполняющимися где-то в Интернете.

В нашем случае каждый источник данных после размещения в модуле данных связывается с конкретной таблицей с помощью свойства DataSet. Соответственно, в модуле данных надо подготовить четыре таких объекта, связать каждый из них со своей таблицей и дать им подходящие названия (рис. 5.11).

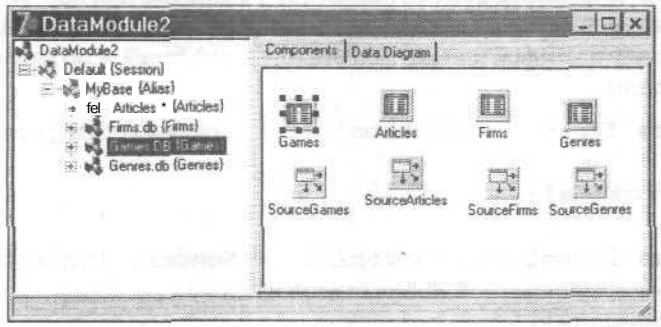


***Рис. 5-11.** Описание таблиц базы данных в качестве источников данных*

В процессе связывания источники данных будут перемещаться «вглубь» дерева, описывающего эти таблицы (рис. 5.12).

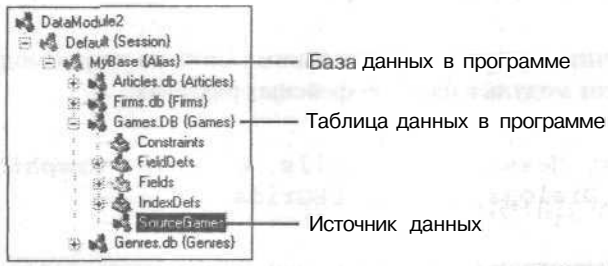


*Рис. **5.12.** Представление структуры базы данных в виде дерева*

Теперь все готово к последнему шагу — проектированию пользовательского интерфейса.

Компоненты для отображения и редактирования данных

Подготовка приложения

Перед тем как закладывать в базу информацию о конкретных играх, надо, очевидно, предварительно ввести названия компаний и жанры игр. Для этого создадим на форме панель инструментов (компонент TToolBar) и разместим на ней четыре кнопки. Пусть первая открывает диалоговое окно, в котором можно вводить, редактировать и удалять записи таблицы Firms. Вторая кнопка предназначена для открытия окна редактирования таблицы Genres. Третью кнопку будем использовать для добавления информации об игре. Пригодится и четвертая кнопка, которая будет закрывать таблицы базы данных.

ПОДСКАЗКА Подберите картинки для кнопок самостоятельно.

Создайте в рамках текущего проекта три новых формы и настройте их так, чтобы они работали в модальном режиме: свойство BorderStyle должно иметь значение bsDialog. Форма Form3 открывается по щелчку на первой кнопке ToolButton1, форма Form4 — по щелчку на второй кнопке ToolButton2, форма Form5 — по щелчку на третьей кнопке ToolButton3.

```
procedure TForm1.ToolButton1Click(Sender: TObject);
begin
Form3.ShowModal;
end;
procedure TForm1.ToolButton2Click(Sender: TObject);
begin
Form4.ShowModal;
end;
procedure TForm1.ToolButton3Click(Sender: TObject);
begin
Form5.ShowModal;
end;
```

Чтобы получить доступ к модулю данных Unit2 из новых модулей, надо указать ссылку на этот модуль в их интерфейсных разделах.

```
uses
Windows, Messages, SysUtils, Classes, Graphics, Controls,
Forms, Dialogs, Grids, DBGrids,
Unit2;
```

Отдельно рассмотрим процесс закрытия формы. Чтобы все внесенные в таблицы изменения были сохранены в базе данных, эти таблицы по окончании работы необходимо закрыть. У каждого класса, описывающего набор данных в *Delphi 7,* имеется для этого метод Close.

```
procedure TForm1.ToolButton4Click(Sender: TObject);
begin
DataModule2.Articles.Close;
DataModule2.Games.Close;
DataModule2.Genres.Close;
DataModule2.Firms.Close;
Close;
end;
```

Компонент Таблица данных (TDBGrid)

Чтобы получить возможность редактировать, добавлять или удалять записи таблицы, писать программный код не требуется. Достаточно разместить на форме Form3 компонент TDBGrid с панели Data Controls (Элементы управления данными). Он во многом напоминает ранее рассмотренный компонент TStringGrid. В его свойстве DataSource следует указать нужный источник данных, в данном случае DataModule2.Source Firms.

Аналогичные действия надо выполнить и с формой Form4, только источником данных лослужит DataModule2.SourceGenres.

Пока что в таблицах показываются все поля, в том числе и ключевое поле ID. Так как его значение будет увеличиваться автоматически, отображать это поле и позволять его редактировать на данном этапе не следует.

Столбцы таблицы описываются свойством Columns, представляющим собой коллекцию элементов типа TDBGridColumn. По умолчанию отображаются все столбцы, а с помощью редактора формируется определенный набор. В нашем случае требуется показывать единственное неключевое поле. После добавления в редакторе коллекции элемента TColumn в списке его свойств надо найти свойство FieldName (Имя поля таблицы) и выбрать в раскрывающемся списке для таблицы фирм значение FirmName, а для таблицы жанров — GenreName.

Откомпилируйте и запустите программу. По щелчку на первой кнопке появится форма, в поля которой вводят названия фирм, а по щелчку на второй кнопке — форма для ввода названий жанров (рис. 5.13).

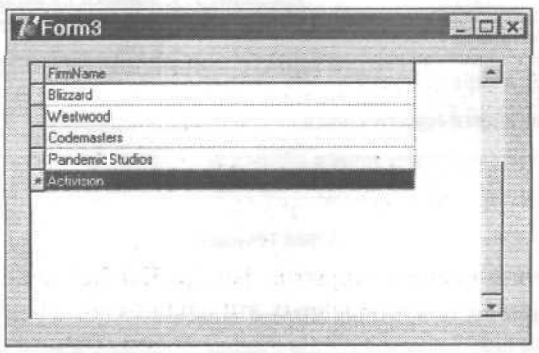


Рис. 5.13. *Ввод данных в таблицу с помощью формы, определенной программно*

Названия вводят набором с клавиатуры, установив курсор в нужном поле, добавление записи — нажатием клавиши ВНИЗ. В последнем случае в таблице автоматически добавляется новое пустое поле.

Компонент Навигатор (TDBNavigator)

Для упрощения навигации по таблице (что немаловажно, если в ней очень много записей) в системе *Delphi 7* имеется компонент TDBNavigator.

Он обычно размещается на форме под компонентом TDBGrid и привязывается к нему через свойство DataSource. Значение этого свойства должно совпадать со значением такого же свойства связанной таблицы (рис. 5.14).

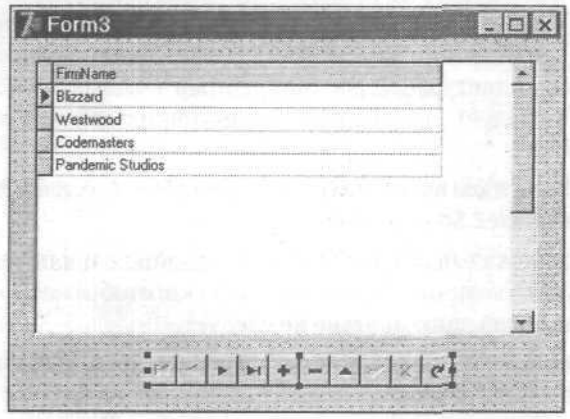


Рис. 5.14. *Размещение на форме таблицы данных и присоединенного средства навигации*

Навигатор позволяет перемещаться по набору записей вперед и назад, переходить к первой и последней записи и выполнять ряд других функций. Функции Навигатора доступны как при щелчках на его кнопках во время работы приложения, так и из Программного кода.

Компонент содержит десять кнопок.

Таблица 5.3. *Кнопки компонента TDBNavigator*

Кнопка	Назначение
First	Переход к первой записи
Prior	Переход к предыдущей записи
Next	Переход к следующей записи
Last	Переход к последней записи
Insert	Добавление новой записи перед текущей
Delete	Удаление текущей записи
Edit	Разрешить редактирование текущей записи
Post	Записать изменения, внесенные в текущую запись, в таблицу базы данных

Таблица 5.3. Кнопки компонента TDBNavigator(продолжение)

Кнопка	Назначение
Cancel	Отменить режим редактирования и восстановить старые значения полей текущей записи
Refresh	Обновить содержимое текущего набора данных (требуется, если с набором одновременно работают несколько пользователей)

Свойства, методы и события класса TDBNavigator приведены в табл. 5.4-5.6.

Таблица 5.4. Свойства класса TDBNavigator

Свойство	Назначение
ConfirmDelete	Имеет значение True, если необходимо отображать диалоговое окно с запросом на подтверждение удаления записи
DataSource	Источник данных
Flat	Имеет значение True, если кнопки на объекте будут выглядеть плоскими
Hints	Массив строк (тип TStrings), хранящий всплывающие подсказки для каждой из кнопок
VisibleButtons	Указание на видимость кнопок

Таблица 5.5. Методы класса TDBNavigator

Метод	Назначение
procedure BtnClick(Index: TNavigateBtn);	Имитация щелчка на кнопке Навигатора
procedure SetBounds(ALeft, ATop, AWidth, AHeight: Integer);	Изменение размера объекта на форме во время работы программы

Таблица 5.6. События класса TDBNavigator

Событие	Условия генерации
BeforeAction	Пользователь щелкнул на кнопке, но соответствующее действие еще не выполнено
OnClick	Выполнено одно из действий навигатора

Таким образом, подготовительная часть справочника готова, при этом ручного программирования практически не требовалось.

Добавление записей

Организовать добавление новых записей в таблицу Games с помощью элемента TDBGrid довольно сложно. Это связано с тем, что в этой таблице хранятся значения ключей из других таблиц (например, в поле GenreID). Редактировать такие значения вручную, не зная, какому названию соответствует определенное значение, бессмысленно.

В системе *Delphi 7* предусмотрена возможность связывания разных таблиц по ключевым полям, но она обычно используется на этапе просмотра и редактирования

данных. Добавление новых записей в такие сложные таблицы, как Games, проще осуществлять в отдельных формах программы.

Попробуем представить себе, как может выглядеть один из вариантов пользовательского интерфейса.

- Название игры вводится в текстовое поле.
- Жанр выбирается из списка уже имеющихся в таблице Genres.
- Признак того, удалось ли сыграть в данную игру, задается флажком.
- В отдельном списке перечисляются названия фирм. Фирму-издателя и фирму-разработчика можно выбрать из этого списка с помощью кнопок Издатель и Разработчик.
- Для задания года и времени выхода предназначены дополнительные текстовые поля.
- Поддерживает ли игра многопользовательский режим, определяет еще один флажок.
- Указать адрес в Интернете и названия статей, посвященных игре, позволяют специальные текстовые поля.
- Характеристики игры по десятибалльной шкале определяют еще три поля.
- Комментарий вводится в текстовую область TMemo.

Некоторые элементы управления создаются на основе рассмотренных ранее визуальных компонентов *Delphi 7* (рис. 5.15).

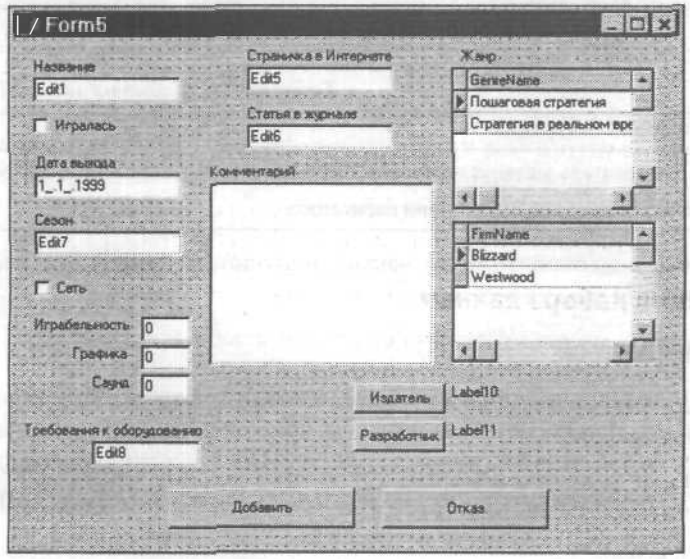


Рис. 5.15. Примерный вид формы для ввода сведений об игре

Опишем работу полей, которые непосредственно привязаны к источникам данных. Прежде всего, это жанр игры. Для него надо создать отдельную таблицу (компонент TDBGrid), связанные с источником данных DataModule2.SourceGenres. С помощью редактора столбцов в таблицу надо добавить все поля. Их два — ключевое поле ID и поле названия фирмы. Теперь мы имеем возможность определить ключ по выбранному названию. Однако доступным для просмотра должно быть только поле GenreName. Для столбца ID свойство Visible имеет значение False.

Показывать содержимое поля GenreName требуется в режиме «только чтение*. Свойство ReadOnly для этого столбца должно иметь значение True. Аналогично на форме создается еще одна таблица, в которой будут указываться все значения ноля FirmName источника данных DataModule2.SourceFirms. Поле ID здесь также необходимо сделать невидимым.

Как будет выполняться обновление данных? Новая запись включается в таблицу по щелчку на кнопке Добавить. В случае щелчка на кнопке Отказ изменений не происходит.

Набор данных

Информация таблицы базы данных описывается в Паскале классом TDataSet, на основе которого, в частности, создан компонент TTable. С его помощью выполнить Добавление новой записи проще всего.

При работе с наборами данных TDataSet (и их наследниками) используется понятие *указателя набора данных.* Он определяет, какая запись таблицы базы данных в настоящий момент является текущей. Когда мы работаем с таблицами TDBGrid, менять, удалять или добавлять в любой момент времени можно только одну запись. Она выделяется в таблице звездочкой и считается текущей. У класса TDataSet имеются свойства и методы для перемещения указателя по набору данных. Считается, что любые операции по изменению или удалению информации выполняются над текущей записью — той, на которой установлен указатель.

Запись изменений в базе данных выполняется только после вызова метола Post. Подобный подход удобен тем, что позволяет отменить изменения, внесенные в текущую запись, если вдруг обнаруживается, что значение некоторого поля указано некорректно. Когда запись состоит из большого числа полей, такое случается нередко. Для отказа от модификации следует вызвать метод Cancel.

Модификация набора данных

Чтобы выполнить модификацию набора данных, надо прежде всего убедиться, что он открыт. Состояние набора проверяется значением свойства Active, которое в этом случае должно быть равно True. Открытие набора выполняется с помощью метода Open (без параметров).

Следующий шаг — определение типа вносимых изменений. Данные могут добавляться (метод Insert без параметров) или модифицироваться (метод Edit беи параметров). Модификации подвергаются поля текущей записи. После выполнения всех изменений их *надо* зафиксировать в базе данных (метод Post) или *отказаться* от сохранения (метод Cancel).

Поля записи

Процесс внесения изменений в поля текущей записи выполняется с помощью обычных операторов присваивания. Поля текущей записи хранятся в свойстве набора данных Fields, имеюшем тип TFields. Это список элементов типа TField, описывающих конкретные поля. Нумерация полей начинается с нуля. Например, для таблицы Genres нулевым элементом будет поле ID, а первым — поле GenreName.

Класс TField имеет набор очень удобных свойств, позволяющих обращаться к каждому полю в зависимости от его реального типа в базе данных. Например, свойство AsString позволяет получить доступ к значению поля в текстовом формате.

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

Другие аналогичные свойства приведены ниже.

Таблица 5.7.Свойства класса TField

Свойство	Назначение
As Boolean	Формат Boolean
AsCurrency	Формат Currency
AsDateTime	Формат TDateTime
AsFloat	Формат Double
AsInteger	Формат Integer
AsString	Формат String
AsVariant	Формат Variant

При этом следует учитывать реальный тип поля в базе данных, чтобы не возникало ошибочных ситуаций типа попытки записать строку в числовое поле.

Другое важнейшее свойство класса TField — Value позволяет обращаться к текущему содержимому поля напрямую. Остальные свойства класса TField будут рассмотрены позже.

ЗАМЕЧАНИЕ На самом деле, каждое поле отражается в наборе данных не как тип TField, а как один из его наследников в зависимости от типа этого поля. Например, строковое поле отображается в виде типа TStringField, числовое поле с автоприращением — в виде типа TAutoIncField, многострочное поле — в виде типа TBlobField и так далее.

Добавление новой записи

Теперь можно определить, как программа будет работать при вызове окна добавления новой записи. Для простоты условимся, что пользователь всегда вводит правильные значения, чтобы сосредоточиться на логике работы с базой данных.

ПОДСКАЗКА На практике при вводе данных в конкретные поля совершают самые разные ошибки. Если для обычной программы это чревато ее аварийным завершением, то для базы данных такая ситуация может закончиться еще более плачевно, вплоть до потери большого обьема данных. Поэтому необходимо всегда контролировать возникновение конфликтов с помощью блока try/except и осуществлять всевозможные проверки корректности значений, введенных пользователем.

Далее требуется сформировать значения всех полей текущей записи (точнее, новой записи, которая будет добавлена с помощью метода Insert и станет текущей) на основе значений, введенных пользователем в поля формы. Большинство из этих значений не вызывает никаких проблем, а дополнительный программный код потребуется для определения полей, ссылающихся на ключевые поля других таблиц. Это, прежде всего, идентификаторы фирмы-издателя и фирмы-разработчика.

Предварительно определим в классе формы TForm5 две переменные.

```
PubID, DevID: Integer;
```

По щелчку на кнопке Издатель сохраним в переменной PubID значение ключа (поле ID) текущей записи из таблицы DBGrid2 с названиями компаний. Само текущее название отобразим в виде надписи Label10.

```
procedure TForm5.Button1Click(Sender: TObject);
begin
PubID := DBGrid2.Fields[0].AsInteger;
Label10.Caption := DBGrid2.Fields[1].AsString;
end;
```

Сходные действия будут выполнены при щелчке на кнопке Разработчик.

```
procedure TForm5.Button2Click(Sender: TObject);
begin
DevID := DBGrid2.Fields[0].AsInteger;
Label11.Caption := DBGrid2.Fields[1].AsString;
end;
```

Теперь можно запрограммировать процесс записи данных из полей формы в поля текущей записи набора Games.

```
procedure TForm5.Button4Click(Sender: TObject);
begin
// добавляем пустую запись в набор Games:
DataModule2.Games.Insert;
// некоторые поля берем из полей формы:
DataModule2.Games.Fields[1].AsString := Edit1.Text;
DataModule2.Games.Fields[2].AsInteger :=
   DBGrid1.Fields[0].AsInteger;
```

продолжение

```
DataModule2.Games.Fields[3].AsBoolean :=
   CheckBox1.Checked;
// коды фирм - из промежуточных переменных:
DataModule2.Games.Fields[4].AsInteger := PubID;
DataModule2.Games.Fields[5].AsInteger := DevID;
DataModule2.Games.Fields[6].AsDateTime :=
   StrToDateTime(MaskEdit1.Text);
DataModule2.Games.Fields[7].AsString := Edit7.Text;
DataModule2.Games.Fields[8].AsBoolean :=
   CheckBox2.Checked;
DataModule2.Games.Fields[9].AsInteger :=
   StrToInt(Edit2.Text),-
DataModule2.Games.Fields[10].AsInteger :=
   StrToInt(Edit3.Text);
DataModule2.Games.Fields[11].AsInteger :=
   StrToInt(Edit4Text ] ,-
// если в многострочное поле что-то введено,
// то используется приведение типа TField
// к типу TBlobField, предназначенному
// для хранения больших двоичных
// (произвольного типа) объектов
// и вызов метода Assign для
// присваивания значения
if Memo1.Lines.Count > 0 then
   begin
   TBlobField(DataModule2.Games.Fields[12]).BlobType := ftMemo;
   TBlobField(DataModule2.Games.Fields[12]).Assign(Memo1.Lines);
   end;
// изменения заносятся в таблицу базы данных
DataModule2.Games.Post;
// если указан узел или статья,
// занести ключ текущей сформированной записи
// и название узла/статьи в таблицу Articles
if (Edit5.Text o '') or (Edit6.Text <> '') then
   begin
   if Edit5.Text <> '' then
      begin
      DataModule2.Articles.Insert;
      DataModule2.Articles.Fields[1].AsInteger :=
      DataModule2.Games.Fields[0].AsInteger;
      DataModule2.Articles.Fields[2] .AsString 1= Edit5.Text;
      DataModule2.Articles.Post;
      end;
   if Edit6.Text o '' then
      begin
```

```
        DataModule2.Articles.Insert;
        DataModule2.Articles.Fields[1].AsInteger :=
        DataModule2.Games.Fields[0].AsInteger;
        DataModule2.Articles.Fields[2].AsString := Edit6.Text;
        DataModule2.Articles.Post;
        end;
    end;
end;
```

Просмотр и редактирование данных

Последний этап создания справочника — реализация режима просмотра главной таблицы Games. Использовать для этого компонент TDBGrid пока не удастся. Ведь в этом случае будут отображаться столбцы GenreID и ArticleID с числовыми значениями ключей, а пользователь должен видеть соответствующие значения из таблиц Genres, Articles и других.

Реализовать подобные требования можно с помощью так называемых полей соответствия (*Lookup fields*) которые создаются в редакторе полей (Fields editor) для физической таблицы Games. Они имитируют реальные поля, выполняя подстановку значений полей из других таблиц, соответствующих имеющимся ключевым значениям.

Для таблиц определен еще один способ доступа к полю: по его имени с помощью метода FieldByName:

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

Например, вместо оператора

```
DataModule2.Games.Fields[1].AsString := Edit1.Text;
```

записывающего данные в поле таблицы Games с индексом 1, можно использовать метод доступа к полю по его названию Name.

```
DataModule2.Games.FieldByName('Name').AsString :=
Edit1.Text;
```

Оба способа имеют преимущества и недостатки. В первом случае надо постоянно следить за порядком полей в таблице, во втором — приходится вручную изменять названия полей в программе, если они были изменены в базе данных.

Рассмотрим использование полей соответствия на примере связи с таблицей Genres. Добавим с помощью редактора полей в таблицу Games новое поле GenreName. Оно должно размещаться в конце списка, чтобы не влиять на порядок ранее определенных полей, доступ к которым происходит по индексам.

Новое поле добавляется в редакторе командой New Field (Создать поле) из контекстного меню. В появившемся диалоговом окне в текстовых полях указываются следующие значения.

- В поле Name (Имя) - имя нового поля (GenreName).
- В поле Type (Тип) — тип подставляемого поля (String, текстовое).

- На панели Field type (Тип поля) включите переключатель Lookup (Поле соответствия).

Когда этот переключатель включен, активизируются раскрывающиеся списки на панели Lookup definition (Определение поля соответствия).

- В раскрывающемся списке KeyFields (Ключевые поля) указывается поле GenrelD, связывающее таблицы между собой.
- В раскрывающемся списке Dataset (Связываемый набор данных) выбирается таблица Genres.
- В раскрывающемся списке Lookup Keys (Поля соответствия) указывается поле ID таблицы Genres, связанное с полем GenrelD текущей таблицы Games.
- В раскрывающемся списке Result Field (Поле результата) указывается поле GenreName таблицы Genres, подставляемое вместо поля GenrelD таблицы Games (рис. 5.16).

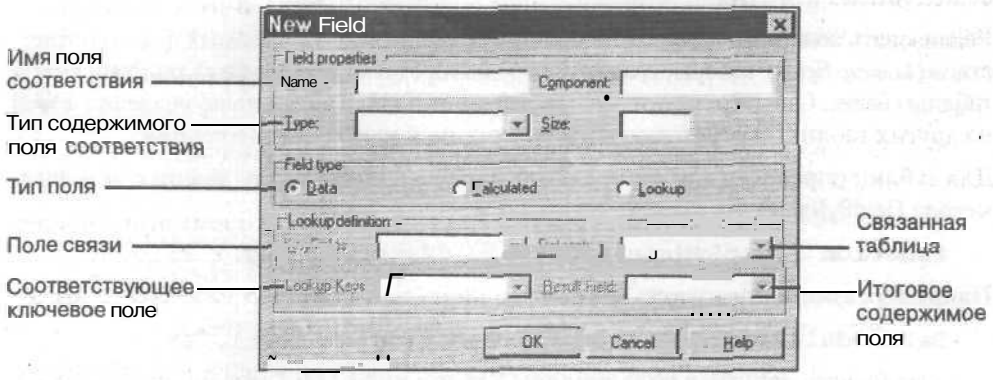
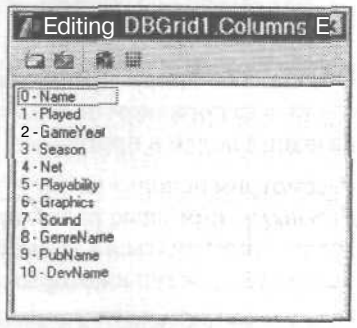


Рис. 5,16. *Задание параметров поля соответствия*

Аналогичным образом надо добавить поля PubName и DevName, которые связывают текущую таблицу Games по ключевым полям Publisher и Developer с таблицей Firms по ее полю соответствия ID. Результирующим будет поле FirmName.



Теперь на основе всех полей таблицы Games можно создавать объект просмотра. На главной форме Form1 разместим компонент TDBGrid. В качестве источника данных укажем набор DataModule2.SourceGames, после чего с помощью редактора столбцов определим список столбцов.

Из этого списка удалены поля, хранящие числовые значения ключей других таблиц, но оставлены поля соответствия. Добавив Навигатор (компонент TDBNavigator), мы получим возможность быстрого перемещения по таблице, редактирования ранее введенных значений и их удаления.

Обратите внимание на работу полей соответствия. Если выделить такое поле, то в его правой части появится кнопка со стрелкой. Щелкнув на ней, мы получим список всех значений, связанных с данным полем. Новые значения можно выбирать из этого списка. Например, таким способом легко указать другую фирму-издателя или фирму-производителя. При этом, что очень важно, автоматически произойдет изменение связанного ключевого поля Publisher или Developer, которое получает новое значение, соответствующее значению ключа для выбранного названия фирмы.

Отображение полей соответствия

Пока что осталась нерешенной еще одна проблема — способ отображения названий *Web-узлов* и статей, связанных с данной игрой. Если реализовать в форме Form5 ввод нескольких значений для текущей записи, то число таких записей, приходящееся на одну игру, может быть неограниченным. Соответствие между игрой и дополнительными материалами устанавливает поле GameID в таблице Articles.

Для отображения подобной информации в системе *Delphi 7* имеется компонент TDBLookupListBox (Список полей соответствия).

Разместив его на форме рядом с таблицей, надо определить следующие свойства.

- Свойство DataSource должно указывать на источник исходных данных DataModule2.SourceGames.
- Свойство DataField должно указывать на поле ID, по которому выполняется связь с таблицей Articles.
- Свойство ListSource должно указывать на источник данных DataModule2.SourceArticles, откуда извлекаются значения для заполнения списка.
- Свойство KeyField определяет ключевое поле этого источника, привязываемое к полю, указанному в свойстве DataField.
- Свойство ListField указывает поле, из которого будут браться значения для списка.

Если теперь задать для некоторой игры статью и *Web-узел,* то при выборе записи в компоненте TDBGrid значения, соответствующие игре, будут отображены в списке как выделенные. При переключении на другую запись выделение переместится на другую группу элементов списка.

ЗАМЕЧАНИЕ Вместо списка полей соответствия можно использовать компонент Раскрывающийся список полей соответствия (TBDLooupComboBox) с аналогичными возможностями.

Связывание двух таблиц

Отображение записей таблицы Articles, соответствующих текущей записи таблицы Games, можно реализовать иначе. С помощью свойств MasterSource/MasterFields две таблицы связываются друг с другом по ключевому полю.

Зададим значение свойства MasterSource для таблицы Articles равным SourceGames (главная таблица, на записи которой ссылается подчиненная таблица). Вызовем специальный редактор отношений, обратившись с помощью Инспектора объектов к свойству MasterFields. В раскрывающемся списке Available Indexes (Доступные индексы) надо выбрать значение GameIND. Мы выбрали созданный на этапе проектирования таблицы вторичный индекс по полю GameID и теперь можем связать его с полем ID таблицы Games с помощью кнопки Add (Добавить), как показано на рис. 5.17.

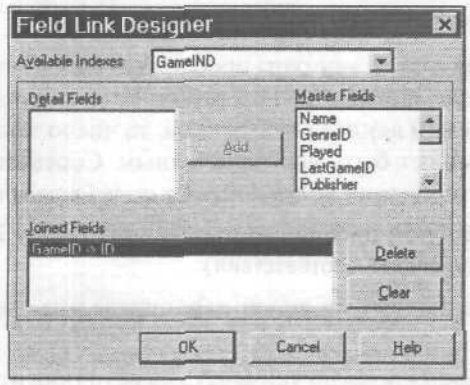


РИС. 5.17. *Связывание таблиц с использованием созданного ранее вторичного индекса*

После закрытия диалогового окна разместим на форме Form1 новый компонент TDBGrid, в котором в качестве источника данных (свойство DataSource) укажем DataModule2.SourceArticles. Теперь откомпилируем и запустим справочник. Оказывается, что при перемещении по записям таблицы Games в добавленном списке DBGrid2 происходит автоматическое перемещение к записи, в которой значение поля GameID совпадает со значением поля ID текущей записи таблицы Games.

Заключение

В данной главе были рассмотрены два способа работы с таблицами: программный, когда значения конкретным полям задаются с помощью элементов управления *Windows*, и визуальный, основанный на использовании готового компонента TDBGrid.

В большинстве автономных приложений достаточно использовать второй подход, который значительно проще и не предъявляет повышенных требований к правильности ввода. Кроме того, дополнительные проверки вводимой информации можно настроить как с помощью свойств псевдополей таблицы (которые создаются с помощью редактора полей), так и с помощью свойств отдельных столбцов.

Однако в серьезных приложениях приходится применять первый подход (предварительный ввод значений в поля формы). При этом совершенно необходимо проверять вводимые пользователем значения и тщательно выверять корректность их взаимосвязей, потому что небольшая ошибка в системе ввода финансовой программы способна привести к существенному искажению информации.

Оба эти подхода широко распространены, и надо уметь пользоваться каждым из них. Дополнительно можно отметить такие направления совершенствования схемы справочника, как более тщательный контроль вводимой информации, расширение структуры базы данных (например, для хранения системных требований к игре) и так далее. Создание на основе такой заготовки полноценного продукта поможет понять классические принципы проектирования и создания приложений, работающих с базами данных.

Основные методы работы с набором данных

Сортировка набора данных

При создании таблиц базы данных рассказывалось о возможности формирования индексов для полей таблицы, значительно повышающих скорость доступа к данным. В некоторых СУБД правильно организованная индексация — единственный способ увеличить эффективность работы. При этом приходится учитывать множество нюансов, специфичных для конкретной СУБД (например, большое число индексов может повысить скорость выборки записей из базы, но существенно замедлить операции добавления в нее новой информации).

Подключение к таблице вторичного индекса (исходно считается, что она упорядочена по первичному индексу) осуществляется выбором в свойстве IndexName имени этого индекса, созданного, например, с помощью приложения Database Desktop. Если проиндексировать поле Playability в таблице Games и назвать индекс PlayInd, то после его выбора в свойстве IndexName выполняется сортировка полей таблицы по значению этого поля.

Присвоить новое значение свойству IndexName можно и во время работы программы, что позволяет показывать записи таблицы в разном порядке в зависимости от действий пользователя.

Для клиент-серверных СУБД вместо свойства IndexName обычно используется свойство IndexFieldNames, позволяющее указать список названий полей, описывающих индекс, через точку с запятой. В нашем случае этот список будет содержать только название поля Playability.

ВНИМАНИЕ Свойства IndexName и IndexFieldNames не допускают совместного использования. При внесении значения в одно из этих свойств другое свойство автоматически очищается.

Создать индекс для таблицы можно не только на этапе проектирования, но и во время работы программы. Для этого предназначен метод

```
procedure AddIndex(const Name, Fields: String;
   Options: TIndexOptions; const DescFields: String='');
```

Название нового индекса указывается в параметре Name. К этому названию предъявляются требования конкретной СУБД. Список полей, на которых базируется индекс, перечисляется в параметре Fields через точку с запятой.

Свойство индекса определяется параметром Options, имеющим следующий тип.

```
type TIndexOption = (ixPrimary, ixUnique, ixDescending,
   ixCaseInsensitive, ixExpression, ixNonMaintained);
```

Важнейшие значения параметра Options перечислены ниже.

Таблица 5.8. Значения параметра Options

Значение	Свойства индекса
ixPrimary	Первичный индекс
ixUnique	Все значения индексных полей уникальны
ixDescending	Сортировка по индексу выполняется в порядке убывания
ixCaseInsensitive	Процесс сортировки нечувствителен к регистру содержимого текстовых полей

Описание индексов таблицы можно просмотреть в Проводнике соответствующего модуля данных в разделе IndexDefs.

Для удаления вторичного индекса можно воспользоваться методом

```
procedure DeleteIndex(const Name: String);
```

Получить список доступных индексов позволяет метод

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

Вычисляемые поля

Когда в программе используются не физические поля таблицы, а псевдополя, сформированные в редакторе полей, это придает приложению дополнительную гибкость и позволяет создавать несуществующие поля с оригинальными характеристиками. Один из таких типов — поля соответствия — мы уже рассмотрели. Теперь перейдем к описанию вычисляемых полей, значения которых берутся не из таблиц базы данных, а рассчитываются на основе значений уже существующих полей текущей записи.

Пусть в главную таблицу просмотра требуется добавить поле, содержащее звездочку, если названия фирмы-производителя и фирмы-издателя совпадают. Для этого добавим в список полей таблицы Games новое поле под названием CalcField, указав при этом его тип String и выбрав с помощью переключателя Field type (Тип поля) вариант Calculated (Вычисляемое).

После этого создадим обработчик события OnCalc Fields для таблицы Games, в котором проанализируем значения полей PubName и DevName и выполним присваивание вычисляемому полю подходящего значения. Нужные данные доступны через переменные GamesPubName и GamesDevName.

```
procedure TDataModule2.GamesCalcFields(DataSet: TDataSet);
begin
if GamesPubName.Value = GamesDevName.Value
   then GamesCalcField.Value := '*'
   else GamesCalcField.Value := '';
end;
```

Осталось только добавить новое поле в таблицу DBGrid1 с помощью редактора столбцов, после чего она примет вид, показанный на рис. 5.18.

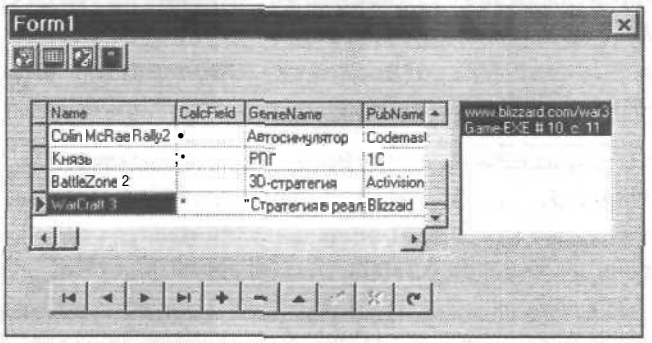


Рис. 5.18. Просмотр таблицы, при выводе которой используется вычисляемое поле

ПОДСКАЗКА При работе с вычисляемыми полями надо учитывать значение свойства AutoCalcFields. Если оно имеет значение True, то событие OnCalcFields генерируется, когда открывается таблица базы данных, происходит переключение режима работы набора данных в состояние dsEdit (редактирование), фокус перемещается между объектами, привязанными к таблице, а также при удалении записи. Когда пользователь ведет с таблицей интенсивную работу, а объем вычислений велик, обработчик события OnCalcFields может замедлить работу приложения, поэтому значение свойства AutoCalcFields в подобном случае лучше установить равным False.

Закладки (Bookmarks)

Когда в таблице хранится много записей, часто возникает потребность в быстром переходе от одной записи к другой, например, для сравнения значений, посмотра аналогичных данных и так далее. Классы, созданные на основе класса TDataSet, имеют набор свойств и методов, с помощью которых программист может формировать действия по созданию неограниченного числа закладок, указывающих на конкретные записи в наборе, и потом использовать их для быстрого перехода к нужной записи.

Свойство Bookmark класса TTable хранит значение текущей закладки. Для ее создания используется метод

```
function GetBookmark: TBookmark;
```

Он формирует закладку для текущей записи. Для перемещения к конкретной закладке применяется метод

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

Для удаления указанной закладки служит метод

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

Проверить, корректно ли значение закладки (указывает ли она на существующую запись), можно с помощью метода

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

Закладки разрешается сравнивать с помощью метода

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

Этот метод возвращает значение 0, если закладки идентичны; -1, если закладка Bookmark1 меньше закладки Bookmark2; +1, если верно обратное.

На основе этих методов несложно написать программный код для обработки списка закладок текущего набора данных. Например можно подготовить динамический массив, который будет хранить примерно такую структуру.

```
type TMyBookmark =
  record
  Bookmark: TMyBookmark;
  BookmarkName: string[50];
  end;
```

Закладка добавляется в массив в момент ее создания, одновременно из текстового поля считывается текстовое описание закладки. По щелчку на определенной кнопке можно отображать список всех имен закладок, а при выборе пользователем конкретного имени — осуществлять перемещение к нужной записи с помощью метода GotoBookmark.

Фильтры

Просматривать таблицы, содержащие большое число записей, неудобно. Часто требуется выделить только группу записей, связанных некоторым условием. Пусть нам требуется отобрать все игры, выпущенные до 2001 года, или получить список несыгранных игр или игр, у которых оценка Playability не ниже 8. Для этого набор данных надо отфильтровать по заданному условию.

Такое условие (оно называется *запросом)* записывается в свойство Filter набора данных в виде текстовой строки и содержит логическое выражение, в котором используются поля текущего набора данных, константы, логические операции и операции отношения.

ЗАМЕЧАНИЕ Запросы при работе с СУБД обычно записываются с помощью специального языка работы с базами данных SQL. Этот язык поддерживается практическими всеми современными СУБД, и о нем будет более подробно рассказано ниже. Значение, записанное в свойство Filter, автоматически преобразуется системой Delphi 7 в запрос SQL, который и передается на выполнение конкретной системе управления базами данных.

Например, чтобы выявить все несыгранные игры, у которых каждая из оценок не ниже 5, надо записать в свойство filter следующую строку

```
Playability >= 5 and Graphics >= 5 and
   Sound >= 5 and Played = False
```

При запуске программы в списке *DBGrid1* отображаются только записи, удовлетворяющие данному условию.

ВНИМАНИЕ Чтобы разрешить фильтрацию значений, надо свойство набора данных Filtered установить равным True.

Если строка заносится в свойство Filter не на этапе проектирования, а во время работы программы (например, по щелчку на кнопке), то отбор нужных записей выполняется динамически.

На структуру условия накладываются определенные ограничения. Для многих автономных и файл-серверных СУБД (в частности, используемой в нашем примере СУБД *Paradox)* не допускается сравнивать значения полей друг с другом.

```
Sound < Graphics  (недопустимо!)
```

В то же время, большинство клиент-серверных СУБД такую возможность поддерживают. Это связано с особенностями реализации языка *SQL*.

В процессе фильтрации можно использовать дополнительные настройки свойства FilterOptions. Настройка foCaseInsensitive определяет, будет ли учитываться регистр при сравнении текстовой информации, а настройка foNoPartialCompare позволяет выполнить фильтрацию по полному совпадению значений текстовых полей или считать строки одинаковыми, если одна из них заканчивается звездочкой.

```
'123*' = '12345'
```

Выделение диапазонов

Использование фильтров позволяет формировать очень сложные запросы, однако на больших наборах данных фильтрация происходит медленно. Поэтому для компонента TTable реализована группа методов, позволяющих оперативно задавать допустимые границы значений для конкретных полей таблицы базы данных.

ЗАМЕЧАНИЕ При использовании большинства файл-серверных СУБД данный метод работает только с индексированными полями.

Первым вызывается метод

```
procedure SetRangeStart;
```

Он приводит таблицу в состояние приема начального значения диапазона. За этим методом должны следовать операторы присваивания начальных значений индексированным полям.

Далее вызывается метод

```
procedure SetRangeEnd;
```

С его помощью аналогичным способом задается конечная граница диапазона.

Отбор записей, попавших в сформированный диапазон, осуществляется обращением к методу

```
procedure ApplyRange;
```

Например, если требуется по щелчку на кнопке Button1 отобрать все записи, в которых значение поля Playability не меньше 5 и не больше 10, можно использовать следующий код:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
DataModule2.Games.SetRangeStart;
DataModule2.Games.FieldByName('Playability').AsInteger := 5;
DataModule2.Games.SetRangeEnd;
DataModule2.Games.FieldByName('Playability').AsInteger := 10;
DataModule2.Games.ApplyRange;
end;
```

Для отмена режима работы таблицы с применением диапазона служит метод

```
procedure CancelRange;
```

Поиск в таблице

В больших базах данных немыслимо выполнять корректировку и редактирование информации без средств поиска нужной записи. Проще всего выполнять подобный поиск с помощью метода

```
function FindKey(const KeyValues: array of const): Boolean;
```

Параметр, называемый *ключом поиска,* описывает список значений для каждого из полей, указанных в свойстве IndexName или IndexFieldNames. Например, чтобы переместить указатель базы данных на запись таблицы Games, в которой значение поля Name равно «Blizzard», надо выполнить следующую команду:

```
DataModule2.Games.FindKey(['Blizzard']);
```

При этом требуется, чтобы таблица Games была проиндексирована по полю Name и в качестве индекса в свойстве IndexName стояло название соответствующего вторичного индекса. Метод FindKey возвращает значение False, если найти подходящую запись не удалось. Для удобства пользователя неплохо добавить в приложение метод

```
procedure FindNearest(const KeyValues: array of const);
```

Этот метод пытается найти запись, значение соответствующего поля которой если и не совпадает точно с параметром KeyValues, то, как минимум, ближе всего к нему. Например, к целому числу 5 ближе всего числа 4 и 6, к строке 'AA' — строка 'AB' и так далее. Данный метод практически всегда выполняется успешно, потому что любое значение будет *относительно* близким указанному.

ЗАМЕЧАНИЕ Не рекомендуется злоупотреблять данным методом, потому что его выполнение может потребовать больших ресурсов.

Ключ поиска можно также формировать с помощью метода

```
procedure SetKey;
```

Соответствующий способ напоминает формирование диапазона. После того как ключ поиска сформирован, можно перейти к нужной записи с помощью метода

```
function GotoKey: Boolean;
```

Переход к записи, похожей по значению, осуществляется с помощью метода

```
procedure GotoNearest;
```

Например:

```
DataModule2.Games.SetKey;
DataModule2.Games.FieldByName('Playability').AsInteger := 4;
DataModule2.Games.GotoNearest;
```

Для поиска можно также использовать универсальные методы

```
function Locate(const KeyFields: string; const KeyValues:
    Variant; Options: TLocateOptions): Boolean;
function Lookup(const KeyFields: string; const KeyValues:
    Variant; const ResultFields: string): Variant;
```

Первый метод получает в качестве параметров список полей, по которым будет выполняться поиск, список значений для каждого из полей и набор вспомогательных настроек поиска (тип TLocateOptions).

Например, поиск значения 5 в индексированном поле Playability может выполняться следующим образом.

```
DataModule2.Games.Locate('Playability', VarArrayOf([5]), []);
```

Обратите внимание на то, как множество [5] преобразуется к типу Variant.

Второй метод в дополнение к поиску возвращает массив значений полей (типа Variant), названия которых указаны в параметре ResultFields.

```
DataModule2.Games.Lookup('Playability', VarArrayOf([5]),
  'Sound');
```

Навигация по таблице

Пользователь может перемещаться к началу и концу набора записей, к следующей или предыдущей записи по отношению к текущей, с помощью компонента TDBNavigator и встроенных возможностей компонента TDBGrid.

В классе TTable содержится набор методов, позволяющих выполнять подобное перемещение указателя внутри набора данных программно.

Таблица 5.9. Методы класса TTable

Метод	Назначение
procedure First;	Указатель устанавливается на первую запись набора данных
procedure Last;	Указатель устанавливается на последнюю запись набора данных
procedure Next;	Указатель перемещается к следующей записи набора данных
procedure Prior;	Указатель перемещается к предыдущей записи набора данных
function MoveBy (Distance: Integer): Integer;	Происходит перемещение указателя на число записей, указанное в параметре, по отношению к текущей записи. Если значение параметра отрицательно, то перемещение осуществляется к началу набора. Функция возвращает число записей, на которое указатель был смещен реально

Следующий пример показывает, как выполнить просмотр всего набора данных и подсчитать число записей, у которых значение поля Played равно True.

```
var i, N, Max: Integer;
begin
N := 0;
Max := DataModule2.Games.RecordCount;
DataModule2.Games.First;
for i := 1 to Max do
  begin
  if DataModule2.Games.FieldByName('Played').AsBoolean
    then inc(N);
  DataModule2.Games.Next;
  end;
Label1.Caption := IntToStr(N)
end;
```

Кроме того, несколько методов предназначено для перемещения по отфильтрованному списку. Это функции FindFirst, FindLast, FindNext и FindPrior, отличающиеся от рассмотренных выше только приставкой Find в названии и возвращаемым значением типа Boolean (True, если перемещение указателя успешно прошло).

Описание компонентов панели BDE

В данной главе детально рассматриваются компоненты панели компонентов BDE (Доступ к данным через механизм BDE) с точки зрения их свойств, методов и обрабатываемых событий.

Класс TTable (Таблица)

На базовом классе TDataSet (абстрактный набор данных) основан класс TBDEDataSet, который реализует функциональность *BDE* при работе с наборами данных. Его наследник — класс TDBDataSet — отвечает за связь с базой данных. Именно на его основе созданы компоненты, способные работать с реляционной информацией в виде таблиц, организованных в столбцы и строки. Такая работа выполняется с уже знакомым по практическому примеру классом TTaBle, характеристики которого и будут рассмотрены (табл. 5.10-5.17).

Таблица 5.10. *Наиболее важные свойства класса TTaBle, унаследованные от класса TDataSet*

Свойство	Назначение
Active	Имеет значение True, если набор данных открыт
Bof	Имеет значение True, если указатель расположен на первой записи в наборе данных
Eof	Имеет значение True, если указатель расположен на последней записи в наборе данных
FieldCount	Число полей в наборе данных
FieldDefs	Список определений полей для набора данных (тип TFieldDefs)
FieldList	Список названий полей набора данных
Fields	Основное свойство для доступа к полям набора данных
FieldValues	Массив полей набора данных, доступ к которым осуществляется по названию поля. Например: DataModule2.Games.FieldValues ['Name'] := 'NEW_FIELD';
Found	Имеет значение True, если выполнение одного из методов поиска в наборе закончилось успешно
Modified	Имеет значение True, если текущая запись изменена
State	Состояние набора данных. Определяет, открыт ли набор, выполняется ли редактирование или добавление записи и так далее

Таблица 5.11. *Наиболее важные свойства, унаследованные от класса TBDEDataSet*

Свойство	Назначение
RecNo	Номер текущей записи в наборе данных
RecordCount	Число записей в наборе данных. Для большинства СУБД получение значения этого свойства может потребовать значительных расходов ресурсов
RecordSize	Размер записи в байтах

Таблица 5.12. Важнейшие свойства класса TTable

Свойство	Назначение
CanModify	И меет значение True, если разрешено редактировать, изменять или удалять записи в наборе данных
DefaultIndex	Имеет значение True, если при открытии таблицы записи сортируются в соответствии со значением первичного ключа
Exclusive	Имеет значение True, если таблица на время работы с ней программы блокируется для доступа к ней других программ. Такой режим поддерживается не всеми СУБД
Exists	Имеет значение True, если таблица существует в базе данных. Свойство проверяется во время работы программы, чтобы выяснить, существует ли физически таблица, указанная в программе
IndexDefs	Коллекция определений индексов (каждый из которых относится к классу TIndexDef), сформированных для данной таблицы
IndexFieldCount	Число полей для текущего индекса
KeyExclusive	Имеет значение True, если граничные записи заданного диапазона записей таблицы не входят в этот диапазон
KeyFieldCount	Число полей ключа, используемое при поиске на его основе
ReadOnly	Таблица доступна только для просмотра
StoreDefs	Имеет значение True, если определения полей и индексов хранятся вместе с модулем данных или формой, на которой создан объект класса TTable
TableLevel	Требование к наличию драйвера BDE определенного уровня
TableType	Тип таблицы (Paradox, dBase, текстовый и так далее)

***Таблица 5.13.** Наиболее важные методы класса TTable, унаследованные от класса TDataSet*

Метод	Назначение
Procedure Append;	Добавление новой пустой записи в конец набора данных
Procedure AppendRecord(const Values: array of const);	Добавление новой записи в конец набора данных, заполнение ее полей значениями из параметра-массива и сохранение этих значений в базе данных
Procedure ClearFields;	Удаление содержимого всех полей текущей записи
Procedure Close;	Закрытие набора данных
Procedure Delete;	Удаление текущей записи и перемещение указателя к следующей записи набора
Procedure Edit;	Начало режима редактирования текущей записи (по окончании редактирования надо вызывать метод Post для сохранения значений в базе данных)
Function FindField(const FieldName: string): TField;	Поиск поля с именем, указанным в параметре FieldName, в текущем наборе данных. Если поле не найдено, функция возвращает значение nil. Данный метод полезно использовать для гарантированного безопасного изменения значений полей: `// безопасно: DataModule2.Games.FindField('Playability').AsInteger := 9;`

Таблица 5.13. Наиболее важные методы класса TTable, унаследованные от класса TDataSet (продолжение)

Метод	Назначение
	// небезопасно: DataModule2.Games.Fields[10].AsInteger := 9; Если поле отсутствует (например, по каким-то причинам удалено администратором СУБД), то в последнем случае в программе возникнет ошибка
Procedure GetFieldNames(List: Tstrings);	Список названий полей возвращается в массиве строк List. Его можно занести в обычный элемент управления типа Список: DataModule2.Games.GetFieldNames(ListBox1.Items);
Procedure Insert;	Добавление новой пустой записи в набор данных. Ее позиция обычно определяется позицией указателя, но может зависеть и от конкретной СУБД
Procedure InsertRecord(const Values: array of const);	Добавление новой записи в набор данных. Ее поля заполняются значениями из параметра-массива, а позиция определяется этими значениями в зависимости от наличия индексированных полей
Function IsEmpty: Boolean;	Возвращает значение True, если в наборе данных нет ни одной записи
Function IsLinkedTo(DataSource: TdataSource): Boolean;	Возвращает значение True, если набор данных связан с указанным источником данных
Procedure Open;	Открытие набора данных
Procedure Refresh;	Обновление набора данных и загрузка обновленных записей. Полезная операция, если с базой данных работает несколько пользователей
Procedure SetFields(const Values: array of const);	Занесение во все поля текущей записи значений из параметра-массива

Таблица 5.14. Наиболее важные методы, унаследованные ат класса TBDEDataSet

Метод	Назначение
Procedure Cancel;	Отмена изменений, внесенных в текущую запись, если еще не был вызван метод Post
Procedure FlushBuffers;	Запись всех сделанных в наборе изменений в базу данных
Function GetBlobFieldData(FieldNo: Integer; var Buffer: TBlobByteData): Integer;	Копирование двоичных данных (Blob) из записи с указанным в параметре FieldNo номером в динамический массив данных Buffer
procedure GetIndexInfo;	Обновление информации о текущем индексе набора
function IsSequenced: Boolean;	Возвращает значение True, если разрешено перемещение от записи к записи присваиванием номера записи свойству RecNo. В противном случае для перехода к нужной записи надо применять методы Next/Prior.
procedure Post;	Запись внесенных изменений в базу данных

продолжение

Таблица 5.14. *Наиболее важные методы, унаследованные от класса TBDEDataSet (продолжение)*

Метод	Назначение
procedure Translate(Src, Dest: PChar; ToOem: Boolean);	По умолчанию считается, что наборы данных BDE работают со строками в формате OEM, а система Delphi 7 использует формат ANSI, принятый в Windows. Если значение параметра ToOem раано True, то строка Src преобразовывается в строку Dest из формата ANSI в формат OEM. Если значение ToOem равно False, преобразование происходит в противоположном направлении

Таблица5.15. *Наиболее важные методы, унаследованные от класса TBDDataSet*

Метод	Назначение
function CheckOpen(Status: DBIResult): Boolean;	Возвращает значение True, если результат обращения к механизму BDE успешен. В противном случае в параметр Status будет записан код ошибки

Таблица 5.16. *Некоторые методы класса TTaЬIe*

Метод	Назначение
functionBatchMove(ASource: TBDEDataSet; AMode: TbatchMode): Longint;	Перемещение записей из набора данных TBDEDataSet в таблицу
procedure DeleteTable;	Удаление таблицы (стирание всех ее записей и удаление структуры из базы данных)
procedure EmptyTable;	Удаление всех записей из таблицы
procedure GotoCurrent(Table: TTable);	Синхронизация указателя для двух наборов данных. Указатель устанавливается на запись, являющуюся текущей для таблицы-параметра. Эти таблицы (объекты класса TTaЬle) должны быть созданы на основе одной физической таблицы базы данных, то есть значения их свойств DatabaseName и TableName должны совпадать
procedure RenameTable(const NewTableName: String);	Переименование текущей таблицы и всех вспомогательных файлов. Метод работает только с базами данных Paradox и dBASE

Таблица5.17. *Важнейшие события класса TTaЬIe*

Событие	Условие генерации
AfterCancel BeforeCancel	После (After) или до (Before) завершения метода Cancel (отмена внесенных изменений)
AfterClose BeforeClose	После (After) или до (Before) завершения закрытия набора данных
AfterDelete BeforeDelete	После (After) или до (Before) выполнения удаления записи
AfterEdit BeforeEdit	После (After) или до (Before) редактирования записи

Таблица 5.17. Важнейшие события класса TTable (продолжение)

Событие	Условие генерации
AfterInsert Befo reinsert	После (After) или до (Before) вставки новой записи
AfterOpen BeforeOpen	После (After) или до (Before) открытия набора данных
AfterPost BeforePost	После (After) или до (Before) сохранения записи в базе данных
AfterRefresh BeforeRefresh	После (After) или до (Before) обновления данных в наборе данных
AfterScroll Before Scroll	После (After) или до (Before) перемещения указателя к новой записи
OnCalcFields	Происходит определение значения вычисляемого поля
OnDeleteError	Была выполнена попытка удаления записи и возникла исключительная ситуация
OnEditError	Была выполнена попытка изменения или добавления записи и возникла исключительная ситуация
OnFilterRecord	Данное событие может обрабатываться для задания оригинального метода фильтрации записей (когда значение свойства Filtered установлено в True)
OnNewRecord	Выполняется вставка или добавление новой записи
OnPostError	Была выполнена попытка сохранения новой или измененной записи и возникла исключительная ситуация

Таблицы можно создавать динамически, во время работы программы. Для этого предназначен метод CreateTable.

В следующем примере в базе данных MyBase создается новая таблица Demo, состоящая из двух полей: ID и Name. Формируются два индекса: первичный, по полю ID, и вторичный, по полю Name. Объект Table1 должен быть подготовлен заранее, например добавлением к модулю данных компонента TTable без каких-либо изменений его свойств, заданных по умолчанию.

```
with DataModule2.Table1 do
begin
// убедиться, что таблица закрыта:
Active := False;
// имя базы ранных:
DatabaseName := 'MyBase';
TableType := ttParadox;
TableName := 'Demo';
with FieldDefs do
  begin
  // добавляем определение первого поля:
  with AddFieldDef do
    begin
```

продолжение

```
      // название
      Name := 'ID';
      // тип - автоматическое увеличение
      DataType := ftAutoInc;
      // требуется наличие данных в поле
      Required := True;
      end;
    // второе поле:
    with AddFieldDef do
     begin
      // название
      Name := 'Name';
      // тип - строка
      DataType := ftString;
      // длина - 50 символов
      Size := 50;
      end;
    end;
  // описываем индексы
  with IndexDefs do
    begin
    with AddIndexDef do
      begin
      // название индекса для первичного ключа не требуется
      Name := '';
      // индексируемое поле
      Fields := 'ID';
      // признак первичного ключа
      Options := [ixPrimary];
      end;
    // второй индекс
    with AddlndexDef do
      begin
      // название индекса
      Name := 'DemoInd';
      // индексирование по полю Name
      Fields := 'Name',•
      // свойства - не различать заглавные и строчные символы
      Options := [ixCaseInsensitive];
      end;
    end;
  // вызов метода создания таблицы
  CreateTable;
  end;
  end;
```

После выполнения этого кода в рабочем каталоге :WORK: окажется новая таблица Demo, структуру которой можно просмотреть с помощью приложения Database Desktop.

ЗАМЕЧАНИЕ Описание типа TTableType, задающего допустимые значения для свойства TableType, хранится в модуле DBTables, который надо подключить в директиве uses.

Класс **Поле записи** (TField)

Данный класс описывает конкретное поле записи, которое представлено в программе виртуально. То есть класс TField служит как бы оболочкой для физического поля записи, дополняя его набором свойств и методов, требуемых разработчику. В отличие от класса TFieldDef, описывающего физическое (реально существующее) поле, на основе класса TField создаются псевдополя: вычисляемые поля, поля соответствия и другие.

Как уже говорилось, тип TField реально в программе не присутствует. Вместо него используется множество его наследников, соответствующих конкретным типам полей записи (например, TDateField для поля, хранящего дату, TGraphicField для поля, хранящего графическое изображение, и так далее). Вместе с тем большинство свойств описано в этом родительском классе.

Основные свойства, методы и события класса TField приведены в табл. 5.18-5.20.

Таблица 5.18. Основные свойства класса TField

Свойство	Назначение
Alignment	Способ выравнивания полей в элементах управления, ответственных за отображение наборов данных
AutoGenerateValue	Свойство (тип TAutoRefreshFlag) определяет, берется ли значение юля из физического набора данных или вычисляется по определенному алгоритму. Таковы ключевые поля с автоматически увеличивающимся значением, поля, значения в которые подставляются программно, и т. д.
Calculated	Имеет значение True, если значение поля вычисляется в обработчике события OnCalcFields
CanModify	Имеет значение True, если значение поля можно редактировать
ConstraintErrorMessage	Строка, которая отображается в диалоговом окне сообщения об ошибке при вводе в поле неверного значения
CustomConstraint	Выражение SOU накладывающее дополнительные требования на значение в поле. Например, можно указать диапазон допустимых значений
DataSet	Набор данных, к которому принадлежит поле
DataSize	Объем памяти в байтах, необходимый для хранения значения поля
DataType	Тип данных поля

продолжение

Таблица 5.18. Основные свойства класса TField (продолжение)

Свойство	Назначение
DefaultExpression	Выражение SQL, которое определяет значение в поле по умолчанию, если значение не вносится в это поле явно
DisplayLabel	Заголовок столбца в табличном элементе управления, соответствующем данному полю
DisplayName	Дополнительное название поля, которое используется для вспомогательных целей, например при выводе сообщения об ошибке
DisplayText	Текстовое представление значения поля, которое отображается в элементах управления
DisplayWidth	Максимально допустимое число символов при отображении значения поля
EditMask	Маска редактирования, контролирующая значение, вводимое пользователем в данное поле
FieldKind	Вид поля. Возможные значения — fkData (содержит данные), fkCalculated (вычисляемое), fkLookup (поле соответствия), fkInternalCalc (вычисляемое поле, значение которого заносится в базу данных), fkAggregate (агрегированное)
FieldName	Название поля физической таблицы базы данных
FieldNo	Порядковый номер поля физической базы данных
FullName	Полное имя поля, если оно является вложенным в другое поле
HasConstraints	Имеет значение True, если поле включает ограничения на допустимые значения
ImportedConstraint	Выражение SQL, накладывающее ограничения на значение поля и выполняемое на сервере
Index	Номер индекса для текущего поля
IsIndexField	Имеет значение True, если поле индексировано
IsNull	Имеет значение True, если поле не содержит значения. Большинство СУБД допускают возможность хранения полей, значение в которых отсутствует
Offset	Объем памяти в байтах, которая дополнительно выделена для хранения записи (в частности, для хранения вычисляемых полей и полей, содержащих Blob-объекты)
Origin	Название поля в исходном наборе данных. Требуется, когда поле используется при формировании запроса TQuery, так как реальные поля разных таблиц могут иметь одинаковые имена
ParentField	Родительское поле. Возможность существования вложенных полей имеется в некоторых СУБД, например Oracle
ProviderFlags	Описывает (тип TProviderFlags), как будет использоваться поле в процессе внесения изменений в серверный набор данных
ReadOnly	Имеет значение True, если поле нельзя редактировать
Required	Имеет значение True, если поле обязано содержать значение
Size	Размер физического поля в условных единицах. Для типа данных ftString это максимально допустимое число символов, для типа ftGraphic — число байтов, отведенных для хранения рисунка в буфере
Text	Текстовое представление текущего содержимого поля

Таблица 5.18. Основные свойства класса TField (продолжение)

Свойство	Назначение
ValidChars	Множество допустимых символов, на основе которого будет формироваться текстовое представление поля
Value	Свойство (тип Variant), позволяющее обращаться к значению поля напрямую
Visible	Имеет значение True, если поле видимо при его использовании в элементах управления панели Data Controls (Элементы управления данными)

Таблица 5.19. Методы класса TFieid

Метод	Назначение
procedure Clear;	Сделать значение поля пустым. В терминологии языка SQL — записать константу NULL
function FocusControl;	Перемещение фокуса на текущей форме к элементу, который отображает значение данного поля
function GetData(Buffer: Pointer; NativeFormat:Boolean=True): Boolean; procedure SetData(Buffer: Pointer, NativeFormat: Boolean=True);	Содержимое поля записывается в буфер (параметр Buffer). Запись производится не в соответствии с типом поля, а непосредственно в том виде, а каком поле хранится в базе данных
class function IsBlob: Boolean;	Возвращает значение True, если в поле хранится Blob-объект
function IsValidChar(InputChar: Char): Boolean;	Возвращает значение True, если символ, указанный а качестве параметра, разрешено хранить в данном поле
procedure SetFieldType(Value: TFieldType);	Метод предназначен только для указания конкретных типов, которые могут быть основаны на данном классе. Например, тип TBlobField может поддерживать типы данных ftBlob, ftFmtMemo и т. д.

Таблица 5.20. События класса TFieid

Событие	Условие генерации
OnChange	Данные в поле были изменены
OnGetText	Произошел запрос значений свойств DisplayText и Text
OnSetText	Свойство Text получает новое значение
OnValidate	Дачные в поле будут изменены. Событие можно обрабатывать, если требуется выполнить дополнительные проверки корректности информации

Класс Описание поля записи (TFieldDef)

Класс TFieldDef используется для описания физического поля таблицы базы данных. Как только новая таблица добавляется в модуль данных, для нее формируется описание всех полей (рис. 5.19).

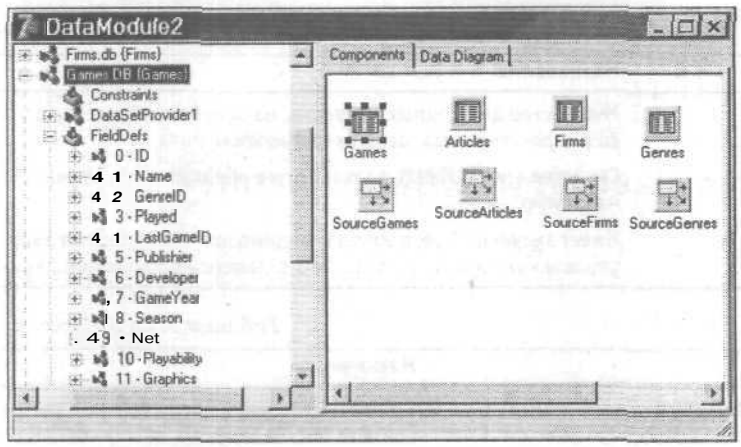


Рис. 5.19. Описание полей таблицы, автоматически сформированное в модуле данных

В дальнейшем можно сформировать описание виртуальных полей, например выбрав в контекстном меню строки Fields в окне просмотра модуля данных пункт New Field (Создать поле). После этого работать в программе с классом TFieldDef станет невозможно из-за того, что в блоке связи с источником данных произойдет автоматическая замена полей класса TFieldDef на поля класса TField.

Класс наследует свойства и методы классов TNamedItem (базовый класс определений элементов базы данных) и TCollectionItem (элемент коллекции).

Таблица 5.21. *Свойства класса TFieldDef*

Свойство	Назначение
Attributes	Атрибуты поля (скрытое, только для чтения, должно иметь значение и так далее)
ChildDefs	Массив определений полей-потомков
DataType	Тип физического поля (TFieldType)
FieldClass	Класс (class of TField), связанный с конкретным типом поля. Например, если значение свойства DataType равно ftString, то данное свойство будет иметь значение TStringField
FieldNo	Номер юля в физической таблице базы данных
InternalCalcField	Имеет значение True, если поле является вычисляемым, а не присутствует в таблице реально
ParentDef	Ссылка на определение родительского поля, если такое поле существует
Precision	Число цифр, отведенное для хранения значения поля
Required	Имеет значение True, если требуется наличие значения поля в физической таблице
Size	Размер поля в физической таблице (в условных единицах)
DisplayName	Название поля, которое отображается в редакторе коллекций в процессе проектирования
Name	Название, которое связывает физическое поле базы данных и определение этого поля

Описание компонентов панели Data Control

Компонент Навигатор (TDBNavigator)

Компонент позволяет управлять указателем связанного с ним набора данных.

Данный компонент рассматривался при создании справочника по играм. Достаточно указать в его свойстве DataSource подчиненный источник данных. Если на форме имеются компоненты, связанные с этим же источником, то их содержимое (значение полей текущей записи) изменяется при использовании кнопок навигатора. Компонент содержит следующие кнопки.

Таблица 5-22. Кнопки компонента TDBNavigator

Кнопка	Название	Назначение
	First	Перемещение к первой записи в наборе
	Prior	Перемещение к предыдущей записи
	Next	Перемещение к следующей записи
	Last	Перемещение к последней записи в наборе
	Insert	Вставка новой записи в месте текущего расположения указателя
	Delete	Удаление текущей записи. Если значение свойства Confirm Delete равно True, запрашивается подтверждение
	Edit	Редактирование текущей записи
	Post	Сохранение изменений, внесенных в таблицу базы данных
	Cancel	Отмена внесенных изменений
	Refresh	Обновление таблицы путем нового считывания данных из базы данных

Придать Навигатору «плоский» вид можно с помощью свойства Flat, перечень видимых кнопок задается в свойстве Visible Buttons.

Метод SetBounds позволяет задать нестандартные размеры панели Навигатора, метод BtnClick программно имитирует щелчок на одной из его кнопок.

Компонент Надпись данных (TDBText)

По аналогии с компонентом TLabel компонент TDBText позволяет отображать содержимое отдельного поля записи.

Источник данных указывается в свойстве DataSource, нужное поле — в свойстве DataField.

Компонент Поле редактирования (TDBEdit)

Компонент позволяет редактировать значение отдельного поля текущей записи.

Используемые свойства не отличаются от свойств компонента TDBText. Дополнительное свойство Readonly позволяет запретить редактирование данных.

Компонент является наследником компонента THaskEdit и позволяет проверять соответствие введенной информации заданной маске с помощью метода ValidateEdit, который в случае несоответствия маски и текущих данных генерирует исключительную ситуацию. Маска доступна через свойство EditMask поля записи, связанного с компонентом (но не через закрытое свойство EditMask своего родителя), а значение, созданное на основе маски, хранится в свойстве EditText.

Компонент Многострочное поле (TDBMemo)

С помощью этого компонента отображается содержимое двоичного поля записи (например, Blob).

Если требуется, чтобы в поле показывалось его полное содержимое (графика, текст), значение свойства AutoDisplay надо сделать равным True.

Метод LoadMemo позволяет переносить содержимое поля Blob в данный компонент, если известно, что оно хранит текстовые данные. Значение свойства AutoDisplay при этом не проверяется.

В качестве примера разместим компонент TDBMemo на форме справочника TForm1 и свяжем его с полем Comment таблицы Games через свойства DataField и DataSource. Если текущая запись содержит многострочный комментарий, то он будет корректно отображаться в объекте DBMemo1.

Компонент Изображение (TDBImage)

С помощью компонента TDBImage можно воспроизводить на экране рисунки, хранящиеся в базе данных. Например, несложно расширить базу данных справочника так, чтобы она хранила в поле Blob картинки из игр.

Рисунки отображаются в данном поле автоматически, если значение свойства AutoDisplay равно True. В противном случае рисунок можно загрузить вызовом метода LoadMemo. Предполагается, что в поле текущей записи хранится изображение в формате, который данный компонент распознает корректно (например .BMP). Это изображение можно, например, скопировать, используя свойство Picture.

```
Image1.Picture.Assign(DBImage1.Picture);
```

Задав свойству QuickDraw значение True, ускоряют вывод рисунка на экран, что часто полезно при активном просмотре больших наборов записей, но при этом ухудшается качество изображения.

Свойство Stretch позволяет подстраивать (сжимать или растягивать) изображение под текущие размеры компонента TDBImage на форме.

Компонент Список данных (TDBListBox)

Этот компонент предназначен для выбора нового значения поля из списка.

Список допустимых значений готовится заранее и заносится в свойство Items (тип TStrings, массив строк).

Компонент Поле данных со списком (TDBComboBox)

Как и компонент TDBList, данный компонент позволяет показывать и выбирать новое значение поля в раскрывающемся списке значений. Набор текста в области ввода позволяет быстро переместиться к нужной записи в списке или ввести значение, отсутствующее в нем.

В дополнение к свойствам, унаследованным от компонента TComboBox, в классе TDBComboBox имеется свойство Style типа TComboBoxStyle, позволяющее определить способ отображения элементов в списке. В остальном данный компонент ничем не отличается от TComboBox.

Компонент Флажок данных (TDBCheckBox)

С помощью элемента-флажка, привязанного к полю таблицы, имеющему логический тип (Boolean), можно отображать состояние этого поля и менять его значение. Не возбраняется применять этот компонент и для полей, которые могут принимать одно из двух произвольных значений.

Список значений, при которых компонент будет считаться «включенным», задается в текстовом свойстве ValueChecked. Элементы перечисляются через точку с запятой.

```
DBCheckBox1.ValueChecked := 'True;Yes';
```

Соответственно, список значений, при которых компонент будет считаться выключенным, задается в свойстве ValueUnchecked.

```
DBCheckBox1.ValueUnchecked := 'False;No';
```

Регистр, в котором записаны эти значения, не учитывается.

ВНИМАНИЕ Если поле таблицы имеет логический тип (он называется по-своему в разных СУБД, но присутствует всегда), строки, занесенные в свойства ValueChecked/ValueUnchecked, не анализируются, а состояние определяется только наличием в поле значения True (включено) или False (выключено).

Компонент Группа переключателей данных (TDBRadioGroup)

Этот компонент позволяет выводить ограниченный набор значений полей в наглядном виде. Каждому значению можно поставить в соответствие один из переключателей группы.

Список названий переключателей заносится в свойство Items (тип TStrings). Список соответствующих им значений поля хранится в свойстве Values (тип TStrings).

Например, значениям 5, 7 и 10 для поля Playability можно поставить в соответствие три переключателя группы: средне, хорошо и отлично.

```
with DBRadioGroup1 do
begin
Items.Add('средне');
Items.Add('хорошо');
Items.Add('отлично');
Values.Add('5'),-
Values.Add('7'),-
Values.Add('10');
end;
```

Доступ к текущему значению поля, связанного с компонентом, можно получить через свойство Value.

Компонент Поле форматирования (TDB Rich Edit)

Компонент наследует все свойства компонента TRichEdit и позволяет представлять содержимое двоичных полей Blob из базы данных как форматированный текст.

Компонент используется так же, как и компонент TDBMemo.

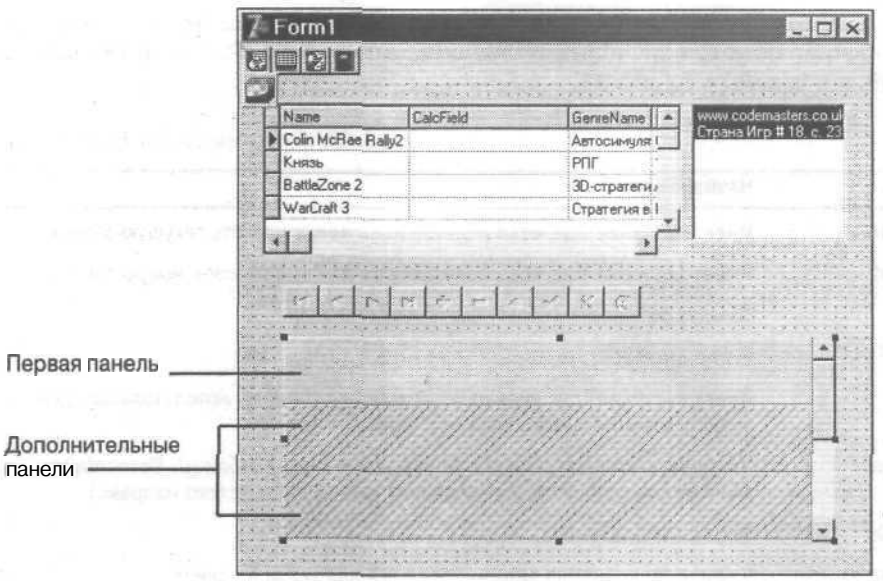


Рис. 5.20. *Размещение на форме компонента для представления записей в свободной форме*

Компонент Свободная форма (TDBCtrlGrid)

Компонент напоминает таблицу записей (TDBGrid), но позволяет представлять данные не в жестко заданном табличном виде, а с помощью выбранного набора компонентов с панели Data Controls (Элементы управления данными) и в произвольном виде. Например, можно задать число столбцов в свободной форме.

Когда компонент помещается на форму, он представляется в виде набора больших панелей, первая из которых пуста, а остальные заштрихованы и приведены только для наглядности (рис. 5.20).

На первой панели надо разместить компоненты, предназначенные для отображения конкретных полей записи, и связать их через свойство DataField с соответствующим полем набора данных. Само имя набора заносится в свойство DataSource автоматически. Оно совпадает со значением аналогичного свойства родительского компонента TDBCtrlGrid.

ВНИМАНИЕ В число размещаемых компонентов не могут входить такие компоненты, как TDBGrid, TDBNavigator, TDBListBox, TDBRadioGroup, TDBLookupList, TDBRichEdit, то есть все, представляющие несколько значений поля.

Во время работы программы на каждой из панелей с помощью соответствующих объектов отображаются значения полей отдельных записей. Свойства класса TDBCtrlGrid приведены в табл. 5.23.

Таблица 5.23. Свойства класса TDBCtrlGrid

Свойство	Назначение
AllowDelete	Имеет значение True, если пользователь может удалять текущую запись
AllowInsert	Имеет значение True, если пользователь может вставлять новую запись
Canvas	Область рисования (холст) для текущей записи
ColCount	Число столбцов
EditMode	Имеет значение True, если данный компонент может использоваться для редактирования, добавления или удаления записей
Orientation	Порядок, в котором записи отображаются внутри таблицы. Возможные значения — goVertical (сверху вниз), goHorizontal (слева направо)
PanelBorder	Форма границ панели
PanelCount	Число одновременно видимых записей для каждой панели
PanelHeight	Высота каждой панели
PanelIndex	Порядковый номер панели
PanelWidth	Ширина каждой панели
RowCount	Число одновременно видимых панелей внутри свободной формы
SelectedColor	Фоновый цвет для панели, которая отображает содержимое текущей записи в наборе данных
ShowFocus	Имеет значение True, если вокруг текущей записи изображается прямоугольник, обозначающий фокус

Для управления перемещением от записи к записи можно применять компонент TDBNavigator, а можно выполнять нужные действия программно с помощью метода

```
procedure DoKey(Key: TDBCtrlGridKey);
```

Параметр Key определяет одно из конкретных действий, например: переход к следующей панели (gkNextTab), переход к панели, расположенной правее, если таблица представлена в несколько столбцов (gkRight), и так далее.

Так как компонент TDBCtrlGrid имеет свойство Canvas, он может обрабатывать событие OnPaintPanel, чтобы самостоятельно отрисовывать образ панели.

ПОДСКАЗКА Данным компонентом можно воспользоваться для упрощения структуры формы TForm5, которая создавалась для программного ввода значений в поля таблицы на основе обычных элементов управления Windows.

Компонент Диаграмма данных (TDBChart)

Ранее мы рассматривали компонент TChart, позволяющий строить всевозможные графики и диаграммы на основе данных, вводимых пользователем или генерируемых программно.

Компонент TDBChart, являющийся наследником компонента TChart, предназначен для отображения в подобном виде информации, получаемой из базы данных. Он настраивается так же, как компонент TChart, но в качестве источника данных для него указывается таблица или запрос.

Допустим, на форме имеется запрос Query1, структура записи в котором состоит из трех полей: двух числовых P1, P2 и текстового P3.

Третье значение может быть привязано к первой паре и через ключевое поле. Оно может описывать характеристики текущей записи, например указывать, к какому опыту относится текущая пара значений. Добавим на форму компонент TDBChart (рис. 5.21) и определим ряд данных.

Далее на вкладке Data Source (Источник данных) выбираются:

- в раскрывающемся списке — значение DataSet (Набор данных);
- в списке Dataset (Набор данных) — имя запроса Query1;
- в поле Labels (Подписи) — имя поля P3;
- в поле X — имя поля P1, значения из которого будут откладываться по оси *X;*
- *в* поле Y — имя ноля P2, значения из которого будут откладываться по оси 7.

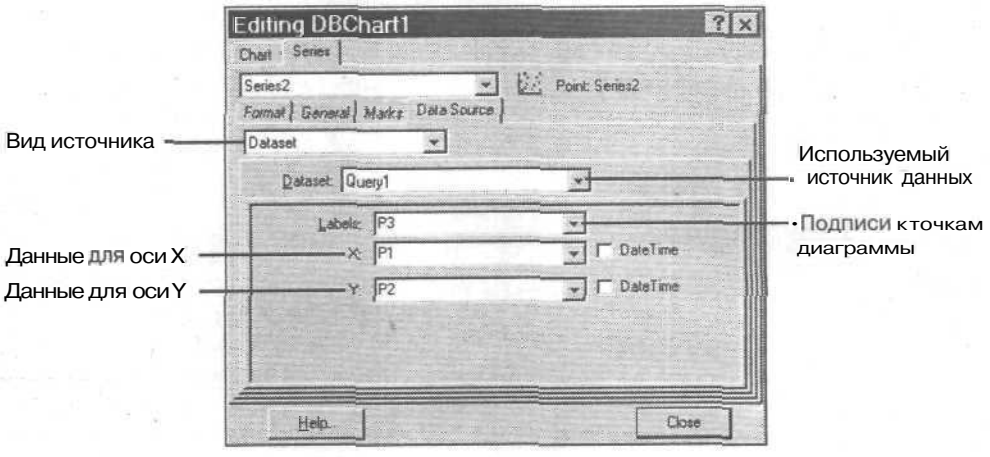


Рис. 5.21. *Построение диаграммы на основе информации из базы данных*

Теперь осталось только перевести набор Query1 в активное состояние (установить значение его свойства Active равным True), и на диаграмме отобразится график в выбранном формате.

Что нового мы узнали?

В этом уроке мы научились

- формировать структуру базы данных;
- создавать модули данных;
- выполнять отображение и редактирование данных;
- осуществлять фильтрацию и поиск данных;
- использовать компоненты для работы с данными.

УРОК 6 Дополнительные средства работы с базами данных

- Проектирование СУБД
- Создание запросов
- Компоненты панели **BDE**
- Основы языка построения запросов SQL
- Создание отчетов
- Средства анализа данных и принятия решений

Проектирование СУБД

При работе с большим числом таблиц очень трудно отслеживать все взаимосвязи между ними: по ключевым полям, полям соответствия и т. д. Поэтому наиболее известные СУБД имеют специальные утилиты, позволяющие в визуальном редакторе выполнять проектирование и анализ структуры базы данных, а именно представлять взаимосвязи между таблицами в виде прямоугольников и стрелок. Такой способ представления структуры обычно выполняется на основе одной из стандартных *нотаций* в системах моделирования, называемых *CASE-системами*.

В системе *Delphi 7* имеется возможность представления структуры базы данных, используемой в проекте, в графическом виде. Реализуется она следующим образом. Прежде всего надо перейти к окну редактора и в правой нижней части выбрать вкладку Diagram (Схема данных). Исходно панель пуста. Ее можно заполнять, перетаскивая объекты из Просмотрщика дерева объектов.

Связь через свойства

Кнопка Property (Свойства) позволяет устанавливать связь между объектами через их свойства. Переместите объект Games.DB из Просмотрщика дерева на панель Diagram (Схема данных).

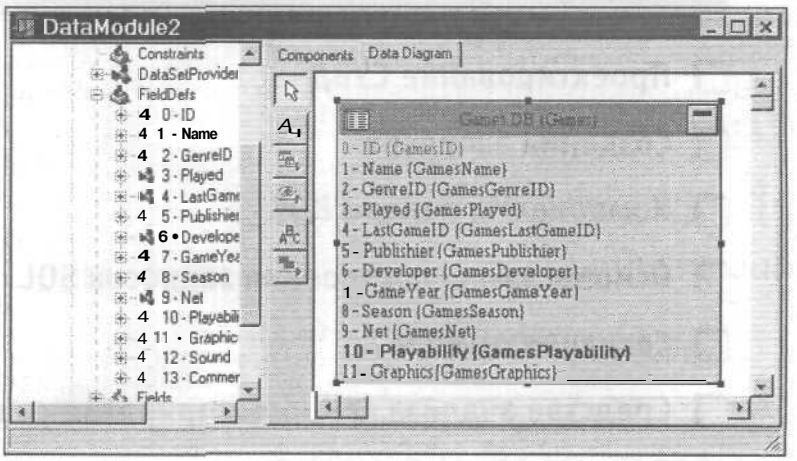


РИС. ff.f. *Наглядное представление таблицы базы данных в виде схемы данных*

Этот объект наглядно представляется в виде таблицы со списком полей. Ключевое поле выделено серым цветом; индексированное поле — полужирным шрифтом; виртуальные поля, не присутствующие в физической таблице, — курсивом. Следующим перемещается объект SourceItems (Источник данных), и между таблицей и источником сразу же возникает связь (рис. 6.2).

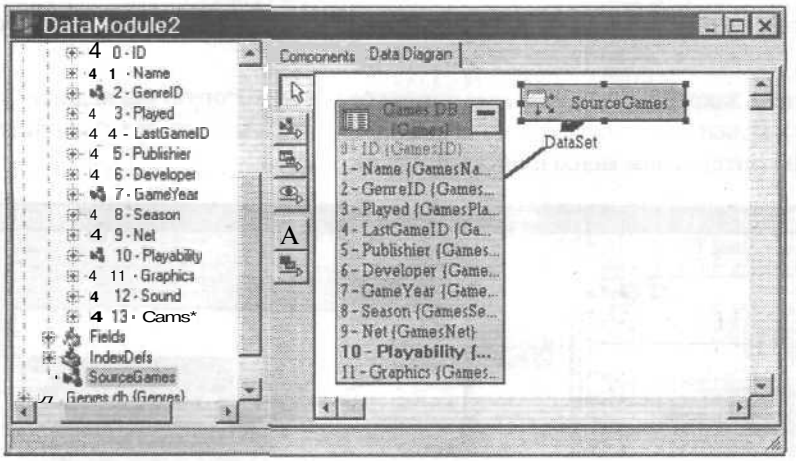


Рис. 6.2. Связь между таблицей и источником данных

Если на основе существующей таблицы требуется создать новый источник данных, не обращаясь к Инспектору объектов, это можно сделать следующим образом.

1. Разместите в модуле данных новый объект класса TDataSource и вернитесь на панель Data Diagram (Схема данных).
2. Теперь перетащите созданный объект с панели Просмотрщика на правую панель. Он пока что не связан ни с какой таблицей.
3. Щелкните на кнопке Property (Свойства), после чего проведите методом протягивания линию от таблицы Items до нового источника данных DataSource2. Нужная связь создастся при этом автоматически.
4. Чтобы удалить связь, надо выделить ее при помощи мыши, после чего в контекстном меню выбрать пункт Remove (Удалить).

Связанные таблицы

Ранее рассказывалось, как связать две таблицы (в частности, Items и Parts) на основе свойств MasterSource/MasterFields. Если к текущей схеме данных добавить таблицу Parts при сохранении описанной связи, то на схеме у этой таблицы появятся две новые связи. Вторая связь, начинающаяся и заканчивающаяся значками в виде прямоугольников, описывает главную таблицу Items для подчиненной таблицы Parts. Главная таблица выделена более крупным прямоугольником.

Удалим эту связь, выделив ее и выбрав в контекстном меню пункт Remove (Удалить), а затем создадим снова. Для этого, щелкнув на кнопке Master Detail (Подчинение таблицы), проведем линию при нажатой левой кнопке мыши от таблицы Items (главной) к таблице Parts (подчиненной). На экране появится диалоговое окно, в котором в списке Available Indexes (Доступные индексы) надо выбрать первичный индекс Primary. Затем следует указать связь поля PartNo таблицы Parts, выбираемого в списке Detail Fields (Поля подчиненной таблицы), с полем PartNo из списка Master

Fields (Поля главной таблицы). Связь устанавливается щелчком на кнопке Add (Добавить).

Если теперь закрыть окно, на схеме появится связь, которую мы недавно удалили (рис. 6.3). В дальнейшем ее можно отредактировать, выделив при помощи мыши и выбрав в контекстном меню пункт Edit (Изменить).

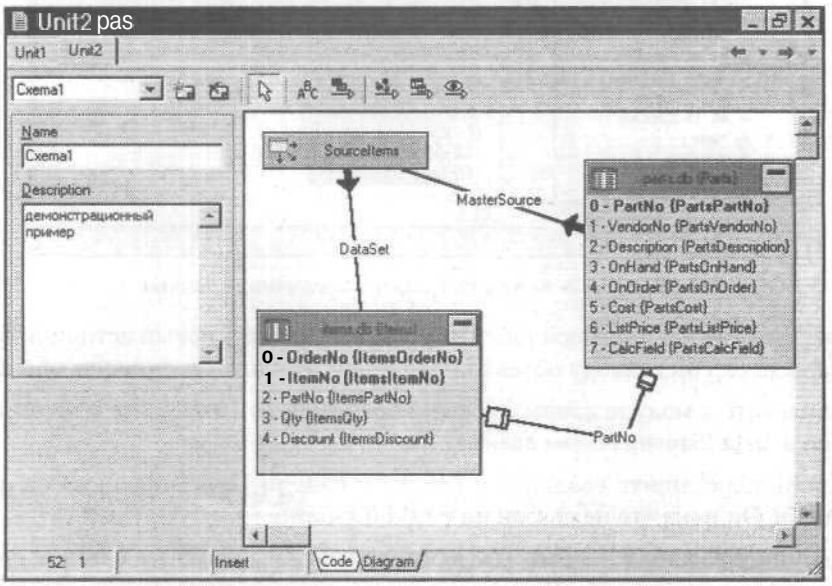


РИС. *6.3.* *Создание связи между таблицами при помощи схемы ванных*

Поля синхронизации

Кнопка Lookup (Соответствие) позволяет показать в проектируемой базе данных связи по полям соответствия. Разместив на схеме две оставшиеся таблицы: Firms и Genres, — можно сразу увидеть две новые связи со значком в виде «глаза», указывающим, из какой таблицы подставляются значения вместо ключевых полей таблицы Games. Эти связи нельзя редактировать, их можно лишь удалять и создавать заново.

Для примера удалим связь с таблицей Genres и, щелкнув на кнопке Lookup (Соответствие), протянем линию от таблицы Games, в которой имеются поля соответствия, к таблице Genres. Возникнет уже виденное нами диалоговое окно, в котором определяется новое поле соответствия.

Родительская связь

Предположим, что мы переместили на схему из левой части, описывающей иерархию объектов в базе данных, два объекта, которые не связаны ни одним из вышеупомянутых отношений. Если один из них является родителем другого, то от наследника к родителю протянется линия связи с крупной стрелкой в начале линии. Например,

поместив рядом с таблицей Firms поле FirmName, можно сразу увидеть связь между ними (рис. 6.4):

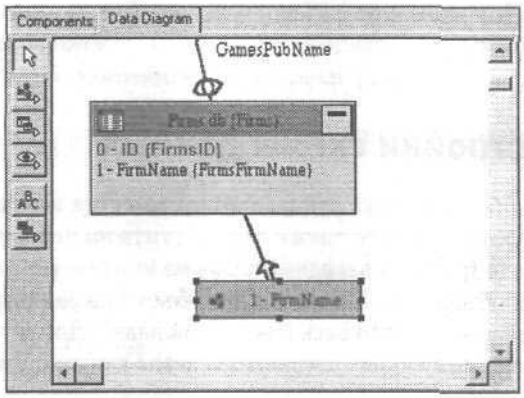
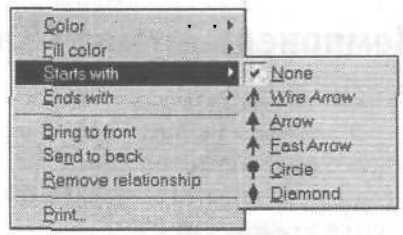
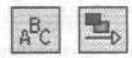


Рис. 6.4. *Связь между родителем и наследником на схеме данных*

Комментарии

Как и при обычном программировании, при проектировании отношений между элементами базы данных, особенно если число таблиц велико, не обойтись без примечаний к различным объектам схемы. Блок примечаний добавляется щелчком на кнопке Comment Block (Блок комментария) с последующим щелчком на схеме. В точке щелчка возникнет прямоугольник, внутри которого можно вводить произвольный текст. Затем надо щелкнуть на кнопке Comment Allude (Привязка комментария) и протянуть линию от примечания к нужному объекту (рис. 6.5).





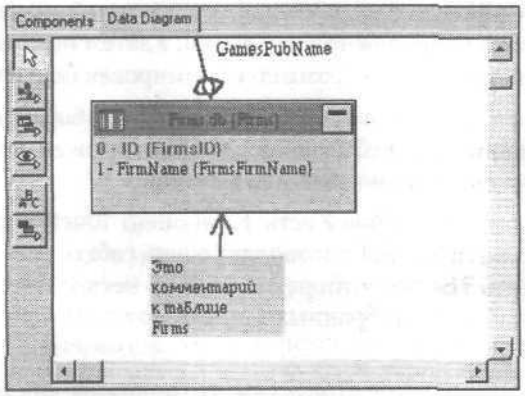


Рис. 6.5. *Таблица* на *схеме данных с присоединенным примечанием*

С помощью кнопки Comment Allude (Привязка комментария) можно также представлять в виде стрелок дополнительные взаимосвязи между объектами таблицы без всяких ограничений. При этом с помощью пунктов контекстного меню Start with (Начало) и End with (Конец) для каждой такой связи можно формировать оригинальный вид, задавая разную форму начальной и конечной областей линии связи.

Визуальные настройки схемы данных

Для каждой связи и любого объекта схемы данных имеется возможность выбрать оригинальные цвета. Объект можно также переместить на передний план схемы с помощью пункта Bring to front (На передний план) из контекстного меню, если таблица закрыта другими объектами. Переместить объект на задний план позволяет команда контекстного меню Send to back (На задний план). Допустимо также изменение размеров и положения любых элементов схемы данных, в том числе подписей у связей.

Создание **запросов**

Компонент Запрос (TQuery)

Выше уже были рассмотрены основные возможности системы *Delphi 7* по просмотру содержимого таблиц. Однако в реальной работе пользователя обычно интересует более подробная дополнительная информация о содержимом базы данных. При этом он не должен задумываться над тем, в каких физических таблицах базы хранятся нужные ему сведения.

Например, нас может интересовать список игр, относящихся к определенному жанру и выпущенных определенной фирмой в течение заданного периода времени, в которые мы еще не играли. В принципе, отобрать и показать на экране подходящий набор записей можно, написав программный код, который сначала просканирует таблицу Firms в поисках ключа нужной компании, а затем выполнит фильтрацию записей из набора Games, также программно сформировав подходящий запрос.

Однако такой код может быть не очень эффективным, а работать с итоговым набором записей будет неудобно. Было бы удобнее, чтобы итоговый набор записей представлялся в программе аналогично обычной таблице.

Такая возможность в системе *Delphi 7* есть. Компонент TQuery с панели BDE (Доступ к данным через механизм BDE) позволяет очень гибко, визуально или программно, определить условие отбора записей из нескольких таблиц, а работать с итоговым набором отобранных записей можно обычным способом, как с компонентом TTable.

Проиллюстрируем это на примере. Допустим, что пользователь хочет иметь в программе возможность просмотреть игры фирмы-разработчика «*Blizzard*», вышед-

шие не ранее 1998 года, в которые он еще не играл. Для этого разместим на панели Components (Компоненты) модуля данных компоненты TQuery и TDataSource. В свойстве DataSet источника данных (назовем его QuerySource) укажем значение Query1 (имя объекта-запроса), а в самом объекте Query1 зададим название базы данных (свойство DatabaseName должно получить значение MyBase).

Формирование структуры запроса

Набор данных, на основе которого будет строиться запрос, не задан, но пока что это и не требуется. Выделим объект Query1 и в его контекстном меню выберем пункт SQL Builder (Построитель запросов).

ЗАМЕЧАНИЕ

SQL — это специальный язык программирования, предназначенный для формирования запросов к базам данных. Нужный нам запрос можно было бы описать на этом языке и затем занести соответствующий текст в свойство SQL (тип TSfrings) объекта Query1. Но если разработчик не знает этого языка, он может воспользоваться визуальным построителем запросов SQL Builder, который на основании действий пользователя сгенерирует текст на языке SQL автоматически.

Первоначально экран построителя пуст (рис. 6.6). Название текущей базы данных (MyBase) указывается в раскрывающемся списке Database (База данных) в верхнем правом углу окна. Запрос строится на основе двух таблиц: Games и Firms. Добавление таблиц к запросу выполняется выбором пунктов Games.DB и Firms.DB в раскрывающемся списке Table (Таблица). При этом на экране возникнут образы соответ-

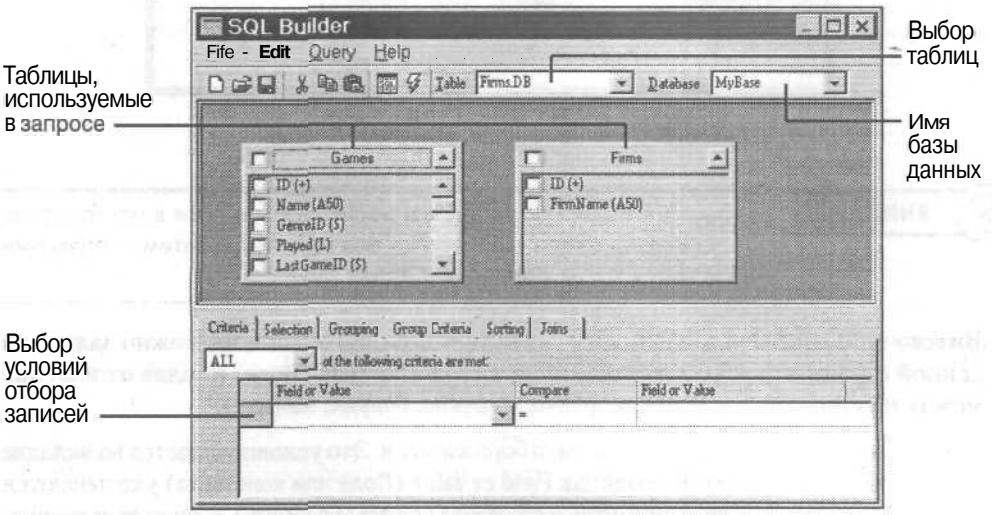


Рис. 6.6. *Формирование запроса к базе данных с использованием визуального построителя*

ствующих таблиц. Ключевые поля помечены знаком «+». В нижней части окна показан набор страниц, с помощью которых в визуальном режиме можно задавать всевозможные условия отбора записей.

Но прежде необходимо установить связь между таблицами. Для этого найдем в таблице Games поле Developer и протянем от него линию к таблице Firms. Между таблицами сразу же сформируется связь, отображаемая в виде линии, а поле Publisher будет помещено на самый верх таблицы и выделено полужирным шрифтом. Точно так же выделяется и поле ID таблицы Firms. Это означает, что между таблицами установлена связь по ключевому полю (рис. 6.7).

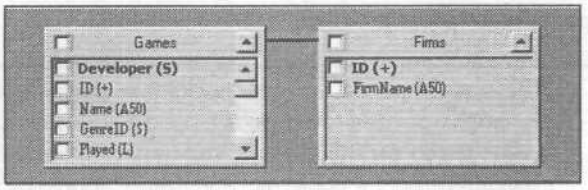


Рис. 6.7. *Установление связи между таблицами по ключевому полю в ходе формирования запроса*

Раскрывающийся список на вкладке Joins (Соединения) содержит список связей, установленных между таблицами. Сейчас в нем выбрана только что созданная связь (Games<-> Firms), но таких связей может быть сколько угодно. В нижней части страницы вкладки показано соответствие полей для выбранной связи (рис. 6.8).

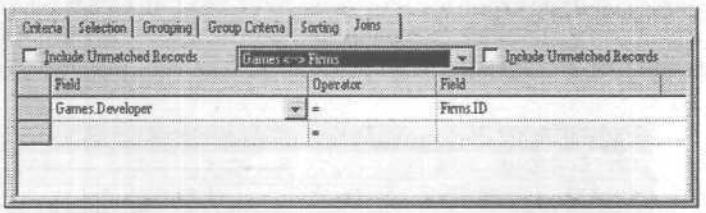


Рис. 6.8. Сведения о связи между таблицами

ВНИМАНИЕ Обратите внимание на то, как записываются поля в построителе запросов. Сначала указывается имя таблицы, а затем — название поля через точку, например Games.Developer.

Вместо того чтобы устанавливать связи при помощи мыши, их можно задать на данной странице, выбрав нужные поля в столбцах Field (Поле) и задав отношение между ними в столбце Operator (равно, не равно, больше, меньше).

Следующий шаг — создание условия отбора записей. Это условие задается на вкладке Criteria (Условия отбора). В столбцах Field or Value (Поле или константа) указываются сравниваемые поля или константы, в столбце Compare (Сравнить) — оператор сравнения. Число строк с условиями может быть неограниченным.

Сначала укажем, что надо отобрать только те игры, в которые мы еще не играли (то есть, значение поля Played должно быть равно False). Для этого щелкнем на ячейке левого столбца, откроем раскрывающийся список и выберем значение Games.Played. В правом столбце введем значение False, а содержимое центрального столбца (символ =, условие равенства) оставим без изменений (рис. 6.9).

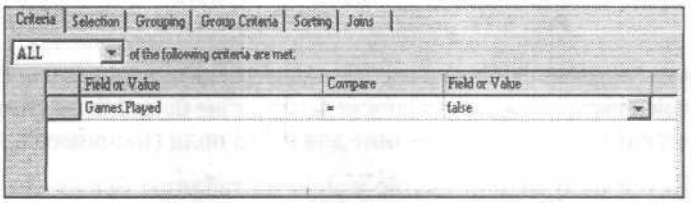


Рис. 6.9- Формирование условия на отбор записей

В следующей строке выберем поле Firms.FirmName. В списке доступны поля обеих таблиц, так как мы связали их на вкладке Join (Соединение). Зададим условие равенства содержимого поля строке 'Blizzard'. Строки в языке *SQL* берутся в одинарные кавычки, как и в Паскале.

ВНИМАНИЕ После того кок в список условий отбора добавляется новая строка, слева от предыдущих строк появляется слово AND, означающее логическую связку «И». Таким образом, отбираются записи, для которых выполняются все заданные условия (рис. 6.10).

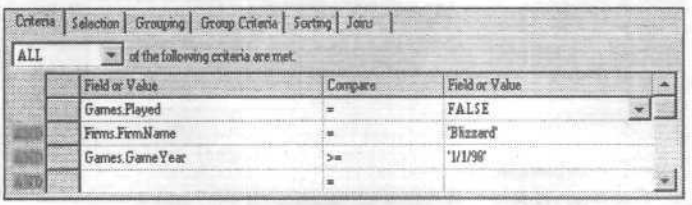


Рис. 6.10. Полный список сформированных условий

Последнее условие заключается в проверке даты. Дата вводится в виде строки '1/1/1998', взятой в одинарные кавычки.

Заключительный шаг — отбор тех полей, которые надо показать пользователю. Допустим, нас интересуют названия игр и жанры. Для отбора соответствующих полей есть два способа.

1. В таблице в верхней части построителя запросов выбирается поле, которое требуется показать — это поле Name, По щелчку оно помечается синей галочкой (рис. 6.11), а сама таблица — серой. Серый цвет говорит о том, что из данной таблицы будут взяты не все поля. Если щелкнуть на флажке не записи, а таблицы, выделяются все поля.

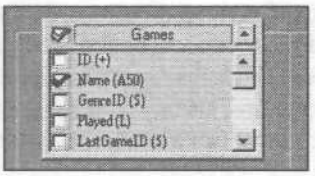


Рис. *6.11. Выбор отображаемых полей*

2. На вкладке Selection (Выбор) в столбце Field (Поле) нужное поле (Games.Name) выбирается в раскрывающемся списке, а в столбце Output Name (Имя при выводе) указывается произвольное название для этого поля (например Game Name).

Теперь требуется получить название жанра из таблицы Genres. Для этого надо повторить вышеописанную процедуру, расширив текущий запрос. Кратко опишем основные шаги.

1. Добавляем новую таблицу Genres.DB из списка Table.
2. Устанавливаем связь между таблицами.
3. Помечаем поле Name и называем его Genre Name с помощью вкладки Selection (рис. 6.12).

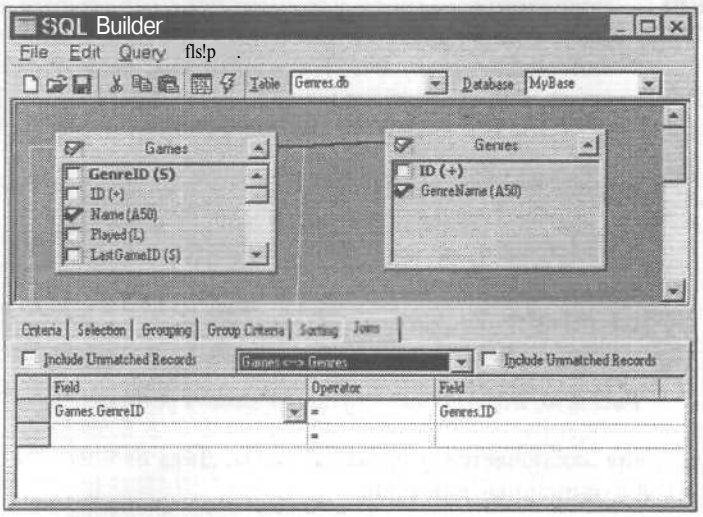


Рис. 6.12. *Включение в поля запрос для вывода информации из дополнительной таблицы*

На этом формирование запроса закончено. Теперь его надо сохранить. Щелкните на кнопке Save the current query (Сохранить текущий запрос) и укажите в диалоговом окне имя файла, в который записывается автоматически сгенерированный текст на языке *SQL,* соответствующий спроектированному запросу. Этот файл можно просмотреть. Содержимое его примерно следующее.

```
/* ALIAS: MyBase */
SELECT Games.Name Games.'Game Name', Genres.GenreName
FROM 'Games.DB' Games
   INNER JOIN 'Firms.DB' Firms
   ON  (Games.Developer = Firms.ID)
   INNER JOIN 'Genres.db' Genres
   ON  (Games.GenreID = Genres.ID)
WHERE   (Games.Played = FALSE)
   AND  (Firms.FirmName =  'Blizzard')
   AND  (Games.GameYear =  '01/01/1998')
```

Более подробно язык *SQL* будет рассмотрен позже.

Непосредственно из Проектировщика запросов можно посмотреть, правильно ли сформирован запрос и что он будет отображать. Для этого надо щелкнуть на кнопке ExecuteQuery (Выполнить запрос). В этом случае на экране отображается текстовая таблица, соответствующая результатам запроса.

Отображение содержимого запроса

Теперь осталось только отобразить набор нужных закисей на экране. Для этого в редакторе полей для объекта Query1 добавим два поля, подготовленных в запросе. После этого разместим на форме еще один элемент TDBGrid и укажем в качестве источника данных значение DataModule2.QuerySource. После этого остается добавить в этот элемент два столбца, и таблица на форме отобразит набор записей, удовлетворяющих заданному условию. Иными словами, это список игр, у которых значение поля Name равно строке Blizzard, дата выпуска не ранее 1 января 1998 года, а значение поля Played равно False. Конечно, надо, чтобы такие записи имелись в базе данных, иначе таблица DBGrid останется пустой.

ВНИМАНИЕ Не забудьте установить значение свойства Active запроса Query1 равным True.

Другие возможности Проектировщика запросов

Логические **связки** условий

Сейчас запрос выполняется, если одновременно истинны все условия, заданные на вкладке Criteria (Условия отбора) — они связаны логической операцией AND (И). В раскрывающемся списке на этой вкладке такая операция обозначается пунктом ALL (Все). Если его изменить на ANY (Хотя бы одно), то слово AND слева от каждого условия заменится на слово OR (ИЛИ). Это означает, что отбираются записи, для которых истинно *любое* из указанных условий (например, игра не сыграна *или* дата выхода не ранее 1998 года и так далее).

Другое возможное значение — NONE (Ни одно) — представляет собой операцию отрицания, аналог операции not в Паскале. Если оно задано, отбираются те записи, для которых указанное условие не истинно, а ложно.

Наконец, значение NOTALL (Не все) требует, чтобы хотя бы одно из списка условий было ложным. Запись отбирается только в таком случае.

Итоговые вычисления

В прикладных программах, работающих с базами данных, постоянно возникает потребность в быстром получении информации, характеризующей текущее состояние определенного объекта (в нашем случае — конкретной игры) не только по набору ее признаков (других полей выбранной записи), но и по дополнительным критериям, например по числу ссылок на игру в Интернете и числу статей, ей посвященных.

В других программах возникает потребность определить для конкретного покупателя, информация о котором хранится в одной таблице Customers, самый большой заказ, который он когда-либо сделал. Заказы хранятся в другой таблице Orders, одно из полей (ID) которой содержит идентификатор заказчика — код ключевого поля первой таблицы.

Для отображения подобной информации надо предварительно просмотреть таблицу заказов, и для каждой записи, описывающей заказ конкретного клиента, проверить, не является ли он максимальным. Такой цикл необходимо выполнить для каждого из заказчиков (каждой записи таблицы Customers). Запрограммировать подобный процесс довольно сложно, и работать он будет не очень эффективно.

При работе с Построителем запросов имеется возможность использовать готовые *итоговые функции* (нахождения максимального или среднего значения конкретного поля и т. п.).

Добавим к таблицам в Проектировщике новую таблицу Articles и установим связь между ее полем GameID и ключевым полем ID таблицы Games.

Требуется добавить новое итоговое поле, которое будет выводить число записей в таблице Articles, значение поля GameID в которых совпадает с текущим значением ключа ID таблицы Games. Выполняется это следующим образом.

1. Сначала на вкладке Selection (Выбор) в столбце Field (Поле) выберите поле Articles.GameID. В столбце Output Name (Имя при выводе) его можно назвать TMP.
2. Открыв контекстное меню этой строки щелчком правой кнопкой мыши, выберите пункт Summary (Итог). Этот пункт определяет, что данное поле будет не простым, а итоговым. При этом в список добавляется новый столбец Summary (Итог), содержащий поле только в текущей записи.
3. В этом столбце выберите в раскрывающемся списке нужную итоговую функцию.

Таблица 6.1. Итоговые функции SQL

Итоговая функция	Назначение
SUM	Сумма всех значений в поле
SUM DISTINCT	Сумма всех несовпадающих значений в поле. Если в указанном поле в нескольких записях хранится одно и то же значение, оно будет учтено только один раз
COUNT	Число всех значений в поле
COUNT DISTINCT	Число всех несовпадающих значений в поле
AVG	Средняя величина всех значений в поле
AVG DISTINCT	Средняя величина всех несовпадающих значений в поле
MIN	Минимальное значение в поле
MAX	Максимальное значение в поле

Нужная нам функция — COUNT.

4. На вкладке Grouping (группировка) надо выбрать оба доступных поля: Games.Game Name и Games.Genre Name — и переместить их на панель Grouped On (Включены в группу). Так определяется подмножество полей, к которому будет применена итоговая функция (рис. 6.13).

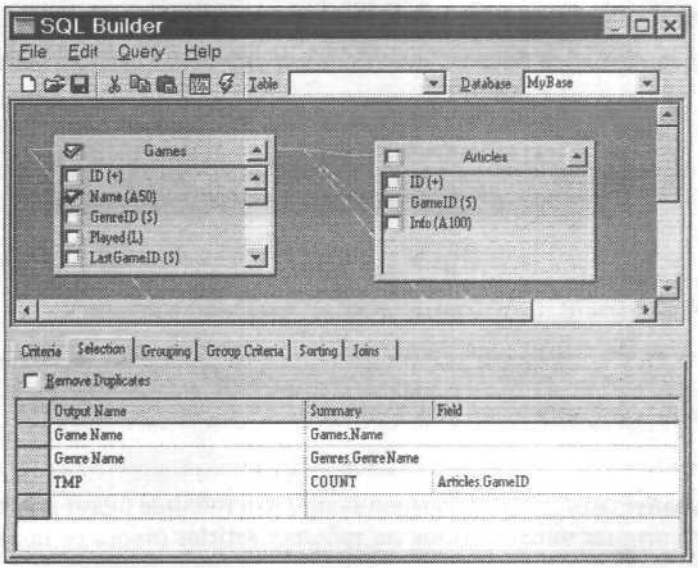


РИС. *6.13. Запрос, включающий итоговые поля*

Если теперь запустить запрос, то в таблице появится новое поле TMP, в котором для каждой записи отображается число «привязанных» к ней записей из таблицы Articles (рис. 6.14).

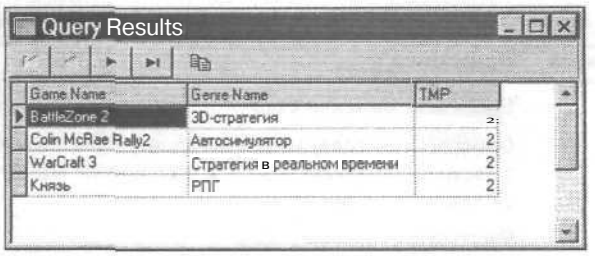


Рис. 6.14. Результаты обращения к запросу, содержащему итоговые поля

Сравнение **итоговых полей**

В ряде случаев итоговые функции желательно использовать в выражениях. Например, мы хотим отобрать только те записи, у которых число ссылок на статьи и страницы в Интернете не менее трех.

Для этого на вкладке Group Criteria (Групповой критерий отбора) надо выполнить следующие действия (рис. 6.15).

1. В столбце Summary (Итог) указать итоговую функцию COUNT.
2. В столбце Field or Value (Поле или значение) выбрать поле Articles.GameID.
3. В столбце Operator (Оператор) выбрать логическую операцию >=.
4. В последнем столбце Field or Value (Поле или значение) указать значение 3,

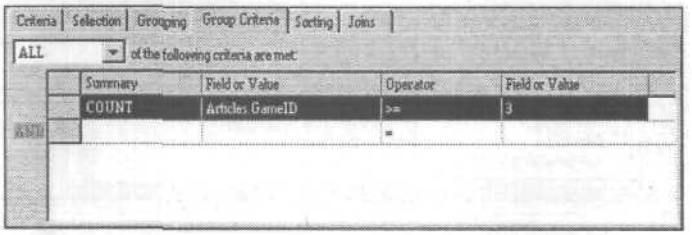


Рис. 6.15. Формирование условия для отбора по групповому критерию

Таким образом, сформировано условие

```
COUNT(Articles.GameID) >= 3
```

Если теперь запустить запрос на выполнение, то в таблице будут показаны только те записи, у которых число ссылок на таблицу Articles (через ее поле GameID) не меньше трех.

Подобных условий можно подготовить сколько угодно. Как и в случае с условиями, создаваемыми на вкладке Criteria (Условия отбора), где нельзя использовать итоговые поля, произвольные логические связки между условиями (И, ИЛИ, НЕ и другие) можно задавать с помощью раскрывающегося списка слева в верхней части вкладки.

Сортировка

Последняя из еще не рассмотренных возможностей Построителя запросов — задание порядка сортировки в формируемом запросе (наборе записей). Доступные для отображения поля, перечисленные на левой панели Output Fields (Поля вывода) можно добавлять в список сортируемых полей Sorted By (Сортировать по полю) с помощью кнопки Add (Добавить). Если в списке несколько таких полей, то сначала выполняется упорядочивание по первому полю списка, для одинаковых значений этого поля выполняется сортировка по второму полю в списке и так далее.

Порядок полей в списке меняется с помощью кнопок со стрелками в верхней части списка. Порядок сортировки (по возрастанию или по убыванию) задается с помощью кнопок A...Z (в возрастающем порядке) и Z...A (в убывающем порядке).

Компоненты панели BDE

Компонент Сеанс связи с СУБД (TSession)

По умолчанию система *Delphi 7* создает стандартный сеанс для любого приложения, которое использует компоненты доступа к СУБД, поэтому явно применять компонент TSession имеет смысл только для выполнения специфических действий по настройке драйверов и псевдонимов баз данных, системному администрированию и управлению сложным программным комплексом.

В следующем примере показано, как можно вывести в список ListBox1 все названия таблиц текущей базы данных MyBase:

```
procedure TForm1.Button1Click(Sender: TObject);
var StringList: TStringList;
begin
   StringList := TStringList.Create;
   Session.GetTableNames('MyBase', '*.db', False, False,
StringList);
   ListBox1.Items := StringList;
   StringList .Free;
end;
```

Переменная Session — это стандартная переменная системы *Delphi 7*, описывающая сеанс связи по умолчанию. Вместо нее можно использовать собственный объект, разместив в модуле данных компонент TSession.

Компонент Базаданных (TDatabase)

Создавать объект класса TSession не обязательно, но явно описать каждую базу данных в программе необходимо. Для этого применяется компонент

TDatabase. Он полезен не только для сложных приложений. Его можно использовать и в нашем примере для повышения надежности некоторых операций.

Транзакции

Одна из важнейших областей применения компонента TDatabase — поддержка *транзакций*. Когда в базу данных вносятся определенные изменения: редактируются или удаляются записи, происходит ввод новой информации, — вернуться к ее прежнему состоянию и *отменить* все сделанные изменения невозможно. В нашем примере это не страшно, но если в таблицу базы данных, расположенной на другом компьютере, заносится, например, большой блок записей и неожиданно происходит обрыв связи, то какая-то часть информации безвозвратно теряется. Самое плохое, что обнаружить факт утраты данных в используемой нами модели невозможно. Средства, позволяющие узнать, было ли успешно завершено изменение таблицы, отсутствуют.

Поэтому все серьезные программы объединяют каждую группу действий над базой данных в *транзакцию* — логически неделимую элементарную операцию, которая может быть либо выполнена полностью, либо не выполнена совсем. То есть, если организовать процесс внесения тысячи записей в таблицу удаленной базы данных в виде транзакции, то она заканчивается успешно, только когда в таблицу внесена вся тысяча записей и от СУБД поступило подтверждение корректности операции. В противном случае транзакция считается неудачной н в результирующую таблицу не заносится ни одной записи. Даже если передача прервется па 999-й записи, *SQL*-сервер, принимающий информацию, не получит подтверждения от клиентской программы о завершении транзакции и уничтожит все полученные данные.

В состав транзакции может входить любое количество операций над таблицами; например, когда пользователь меняет одну запись, удаляет другую и добавляет третью, то все эти действия можно объединить в транзакцию. Если она будет отменена, не произойдет ни изменения, ни удаления, ни добавления данных.

ЗАМЕЧАНИЕ При создании любых приложений обработки и передачи информации, которыми пользуется не один человек, о несколько, обычно реализуются различные виды поддержки транзакций. Речь не обязательно идет о работе с базами данных. Например, при обмене сообщениями в Интернете (особенно в приложениях электронной коммерции) почти всегда требуется получить подтверждение о приеме законченного блока информации. Подобные процессы обеспечиваются так называемыми серверами, или мониторами, транзакций — программами, похожими на SQL-серверы. Отличие этих программ состоит в том, что они не обрабатывают данные в таблицах, а обеспечивают гарантированный прием и передачу блоков информации по сети.

Настраивается объект класса TDatabase, размещенный на форме (или в модуле данных), просто. В свойстве DatabaseName указывается название базы данных вместе с расширением и полным путем поиска.

В свойство AiiasName заносится псевдоним базы данных в системе *BDE* (например, MyBase). Вместо псевдонима можно задать имя драйвера *BDE* (свойство DriverName), но разрешается указать только одно из свойств: AiiasName или DriverName.

При необходимости указывается имя сеанса (если в программе есть компонент TSession, отличный от стандартного сеанса Default). Затем значение свойства Connected переводится в значение True. В этот момент СУБД может запросить имя пользователя и пароль для доступа к базе, после чего объект класса TDatabase будет готов к работе. Чаще всего он применяется в программе для поддержки транзакций. Основная техника транзакционной работы такова:

1. Отдается команда *SQL*-серверу о том, что стартует новая транзакция:

```
Database1.StartTransaction;
```

2. Выполняются действия по внесению изменений в базу данных. Они помещаются в блок try для контроля корректности их выполнения. В конце блока указывается команда Commit, подтверждающая успешное завершение транзакции:

```
try
  // работа с базой данных
  Database1.Commit;
```

3. В случае возникновения исключительной ситуации (невозможности работы с СУБД) надо отменить транзакцию и сгенерировать программную ошибку искусственно, чтобы не дать выполниться завершающим операторам текущей подпрограммы:

```
except
  Database1.Rollback;
  raise;
end;
```

Главное событие, которое имеет смысл обрабатывать, — это событие OnLogin, возникающее в момент подключения приложения к СУБД. Заголовок его обработчика выглядит следующим образом:

```
procedure TForm1.Database1Login(Database: TDatabase;
LoginParams: TStrings] ;
```

В параметре LoginParams разработчик должен указать специфическую информацию, требуемую для подключения к СУБД. Как правило, это имя пользователя и пароль. Например, для соединения с СУБД *InterBase* можно использовать следующий способ стандартного подключения:

```
LoginParams[0] := 'USERNAME=SYSDBA';
LoginParams[1] := 'PASSWORD=masterkey';
```

Компонент TDatabase является наследником класса TCustomConnection — нового класса *Delphi 7,* базового для всех компонентов, связывающих источники и наборы

данных. С помощью класса TCustomConnection можно разрабатывать свои собственные компоненты для эффективного доступа к СУБД разных производителей.

Компонент Хранимая процедура (TStoredProc)

Наиболее мощные СУБД (хотя и не все) способны не только выполнять запросы *SQL* от клиентских программ, но и запускать на сервере, где они работают, так называемые *хранимые процедуры.*

Эти процедуры хранятся на сервере, а не в клиентском приложении. Они могут быть написаны на языке *SQL,* или на другом языке программирования (например, *Java).* Эго позволяет более эффективно выполнять операции над наборами данных (например, сортировку или фильтрацию).

Обычно хранимые процедуры, как и большинство подпрограмм Паскаля, требуют задания параметров для своей работы. Эти параметры указываются в свойстве Params (тип TParams, массив элементов типа TParam с большим количеством возможностей настройки). В этом же свойстве возвращается результат работы хранимой процедуры (набор значений).

Класс TStoredProc является прямым наследником класса TDBDataSet. В следующем примере хранимой процедуре StoredProc1 передается один параметр из текстового поля Edit1. Процедура выполняется на сервере, а результат ее работы, сохраненный в параметре с именем Output, записывается обратно в это же поле:

```
StoredProc1.Params[0].AsString := Edit1.Text;
StoredProc1.Prepare;
StoredProc1.ExecProc;
StoredProc1.GetResults;
Edit1.Text := StoredProc1.ParamByName('Output').AsString;
```

Компонент Групповая обработка (TBatchMove)

Данный компонент позволяет выполнять операции над набором записей или целой таблицей. Эти операции включают добавление или удаление группы записей и копирование всей таблицы.

Полезный метод у класса TBatchMove один. Он осуществляет выполнение подготовленного дейсгвия:

```
procedure Execute;
```

Например, чтобы перенести в таблицу Table1 набор записей, получаемых в результате исполнения запроса Query1, можно использовать следующий код. Структура таблицы должна соответствовать структуре запроса.

```
with BatchMove1 do
  begin
    // указываем источник записей - запрос Query1:
    Source := Query1;
```

```
        // указываем приемник записей - таблицу Table1:
        Destination := Table1;
        // режим работы - копирование:
        Mode := batCopy;
        // выполнить копирование;
        Execute;
        // вывести число скопированных записей
        // в виде надписи
        Label1.Caption := IntToStr(MovedCount) + ' записей
скопировано';
end;
```

Компонент Обновление базы данных (TUpdateSQL)

В некоторых случаях возникает необходимость внести обновления в набор данных, который непосредственно не предназначен для изменения. Такие наборы формируются, как правило, с помощью запроса или хранимой процедуры и обычно доступны в режиме «только чтение».

Компонент TUpdateSQL после размещения на форме связывается с объектом класса TQuery или TStoredProc через свойство UpdateObject этих объектов. Затем задается оператор *SQL,* который автоматически выполняется после формирования соответствующего набора данных.

Компонент Вложенная таблица (TNestedTable)

Некоторые СУБД допускают вложение целой таблицы базы данных в одно поле записи. Для доступа к такой вложенной таблице используется компонент TNestedTable (наследник класса TBDEDataSet). Никаких собственных свойств и методов у него нет, а в свойстве DataSetField (тип TDataSetField) указывается название поля набора данных, содержащего вложенные таблицы. Вэтом наборе данных подчиненный компонент TNestedTable задается с помощью свойства NestedDataSetClass.

Кэшированные обновления (Cached Updates)

При интенсивной работе с СУБД в сети с невысокой пропускной способностью или в системе с большим числом одновременно работающих пользователей порой остро встают вопросы оптимизации работы такой системы. Один из возможных выходов — использование технологии кэшированных обновлений, когда набор данных, с которым работает пользователь, не отправляется сразу после редактирования серверу базы данных, а хранится в локальном буфере на клиентском компьютере. Затем все накопленные обновления отсылаются СУБД за один раз (например, при вызове соответствующего метода). Такой подход помимо всех преимуществ (прежде всего, выигрыша в быстродействии) имеет и ряд существенных недостатков:

- данные могут существенно измениться, пока идет работа с их локальной копией;
- пользователь не знает об изменении данных на сервере;
- процесс согласования локальной и серверной копий набора данных — весьма сложная задача.

Чтобы включить режим кэширования обновлений (свойство CachedUpdates компонентов TTable, TQuery, TStoredProc), надо в это свойство записать значение True. По умолчанию данный режим выключен. Отсылка обновлений из кэша серверу базы данных выполняется, когда вызывается метод ApplyUpdates. Этот процесс можно контролировать, обрабатывая события OnUpdateRecord и OnUpdateError.

ЗАМЕЧАНИЕ Если для доступа к данным используется стандартный, поставляемый с Delphi сервер BDE, корпорация Borland рекомендует применять вместо компонента TClientDataSet компонент TBDEClientDataSet, специально предназначенный для работы с кэшированными обновлениями BDE. Он содержит ряд специфических возможностей, характерных именно для этого сервера.

Основы языка построения запросов SQL

Зачем надо знать SQL

Хотя с помощью Построителя запросов (SQL Builder) можно сформировать практически сколь угодно сложный запрос, в ряде случаев над набором данных необходимо выполнить операции, специфические для конкретной СУБД. В таком случае текст запроса, сгенерированный автоматически, требуется подправить вручную.

Кроме того, практически в любой СУБД имеется возможность выполнения хранимых процедур, написанных на языке *SQL*, поэтому обходиться без знания этого языка при создании приложений, работающих с базами данных, весьма неудобно. Да и вообще, если разработчик планирует сосредоточиться на создании именно таких программ, без знания языка *SQL* ему не обойтись, потому что целый ряд операций гораздо проще и эффективнее выполнять с помощью простых операторов *SQL*, чем посредством обращения к классам системы *Delphi 7*.

В данной главе рассматриваются только наиболее важные команды языка *SQL*.

Выполнение выражений SQL

Операторы *SQL* могут выполняться непосредственно из программного кода. В этом случае текст оператора указывается в свойстве *SQL* компонента TQuery и затем выпол-

няется с помощью метода ExecSQL. Эти операторы можно также выполнять из утилиты SQL Explorer (Проводник SQL).

Рассмотрим последний вариант. Загрузим эту утилиту командой Database > Explore (База данных ➤ Проводник), выберем демонстрационную базу данных DBDEHOS, откроем ее и выберем таблицу customer.db (Заказчики) (рис. 6.16).

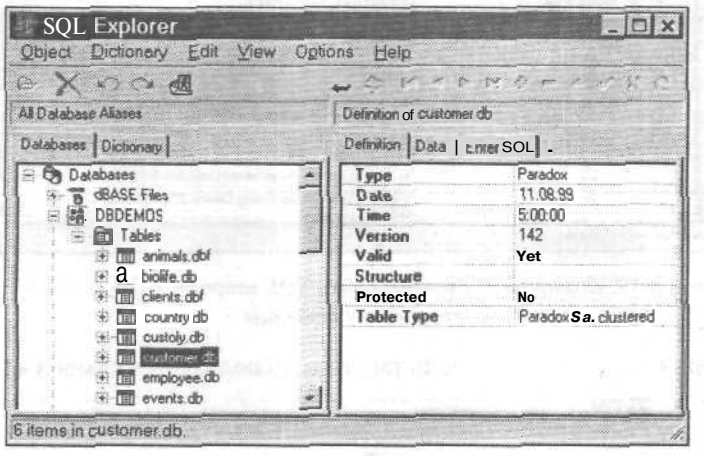


РИС. *6.16. Демонстрационная база данных, открытая с помощью утилиты Проводник SQL*

На вкладке Data (Данные) можно посмотреть содержимое этой таблицы, а на вкладке EnterSQL (Ввод операторов SQL) — ввести и выполнить оператор *SQL*.

Оператор SELECT

Данный оператор является самым распространенным оператором языка *SQL*. Он формирует набор данных, отбирая записи из одной или нескольких таблиц на основании некоторого условия.

Базовая форма записи

Этот оператор записывается следующим образом.

```
SELECT список-полей FROM название-таблицы;
```

Например:

```
SELECT Company, Phone FROM customer;
```

Как и в Паскале, слова SELECT и FROM являются зарезервированными ключевыми словами языка *SQL*. Регистр значения не имеет, но операторы *SQL* принято записывать в верхнем регистре.

Введя такую строку на вкладке Enter SQL (Ввод операторов SQL) и щелкнув на кнопке Execute Query (Выполнить запрос), получим таблицу (временный набор данных), состоящую из двух указанных нами столбцов (рис. 6.17).

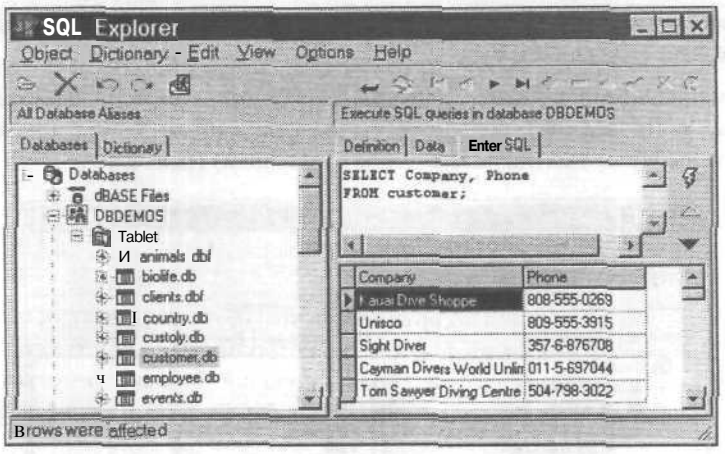


Рис. 6.17. *Создание с помощью языка SQL запроса, включающего заданные поля таблицы*

Если требуется выбрать все поля из таблицы, используется символ «*».

```
SELECT * FROM customer;
```

Удаление повторяющихся записей

Если после ключевого слова SELECT указать ключевое слово DISTINCT, то в результирующий список не включаются записи, содержащие повторяющиеся значения.

Сравните следующие запросы (рис. 6.18).

```
SELECT DISTINCT Country FROM customer;
SELECT Country FROM customer;
```

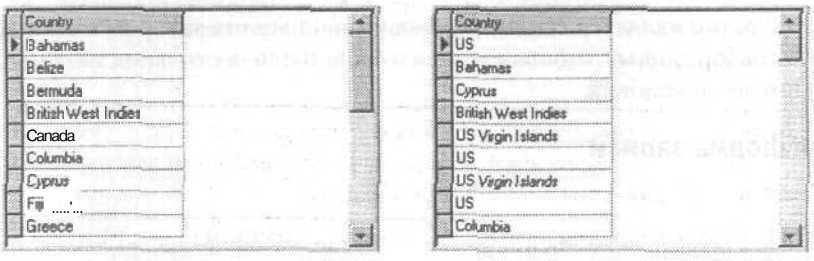


Рис. 6.18. *Отбор с учетом и без учета повторения значений*

Условный отбор значений

Когда требуется отобрать записи по определенному условию, можно использовать ключевое слово WHERE, за которым следует условие, напоминающее условное выражение Паскаля. В нем используются операции сравнения <, >, =, о, >=, <=, логические операции OR, AND, NOT и круглые скобки. В результирующий набор данных попадают только те записи, значения полей которых удовлетворяют указанному условию.

```
SELECT список FROM таблица WHERE условие;
```

Например, чтобы отобрать американские компании, заказавшие товар не ранее 1993 года, надо написать следующий запрос.

```
SELECT Company FROM customer
   WHERE (Country = 'US') AND
         (LastInvoiceDate >= '1/1/1993');
```

Упорядочение записей

Если итоговый набор требуется отсортировать, то надо использовать ключевые слова ORDER BY, после которых указывается поле, по которому выполняется сортировка. По умолчанию она выполняется в возрастающем порядке. Если же требуется выполнить ее в убывающем порядке, то после имени поля добавляется ключевое слово DESC:

```
SELECT Company, Addr1 FROM customer
   WHERE (Country = 'US') AND
         (LastInvoiceDate >= '1/1/1993')
   ORDER BY Addr1 DESC;
```

Использование нескольких таблиц

Если в операторе SELECT надо использовать поля нескольких таблиц, то перед этими полями через точку указываются имена соответствующих таблиц. Например:

```
SELECT customer.Company, orders.SaleDate
FROM customer, orders
WHERE customer.CustNo = orders.CustNo;
```

Произойдет формирование набора данных, содержащего два столбца: названия компаний из таблицы customer и даты продажи из таблицы orders для тех записей каждой из таблиц, у которых совпадают значения одинаково называющихся полей CustNo (идентификатор заказчика).

ЗАМЕЧАНИЕ У оператора SELECT имеется еще немало различных дополнений, позволяющих формировать наборы данных весьма сложными способами.

Оператор INSERT

Этот оператор используется для добавления новой записи. Записывается он следующим образом.

```
INSERT INTO таблица
VALUES (список-значений);
```

Типы значений в списке должны соответствовать типам значений полей таблицы. Например:

```
INSERT INTO industry
VALUES (4400, 'Video', 'VideoCameras');
```

Происходит добавление в таблицу industry.db новой записи с указанными значениями.

Оператор UPDATE

С помощью оператора UPDATE выполняют редактирование (изменение) значений поля по всей таблице.

```
UPDATE таблица
SET поле = значение, ...;
```

Если надо изменить конкретную запись или группу записей, надо дополнительно использовать ключевое слово WHERE, уточняющее диапазон редактируемых записей.

Например, если требуется изменить код записи из предыдущего примера с 4400 на 4000, надо записать и выполнить такой оператор:

```
UPDATE industry
SET IND_CODE = 4000
WHERE IND_CODE = 4400;
```

Оператор DELETE

Этот оператор удаляет записи из таблицы. Он может использоваться совместно с ключевым словом WHERE.

```
DELETE FROM таблица;
```

Такой оператор уничтожает все записи в таблице.

```
DELETE FROM industry
WHERE IND_CODE = 4000;
```

Этот оператор удалит записи, у которых значение поля IND_CODE равно 4000.

Создание таблицы

Оператор CREATE TABLE

Таблица в современных СУБД нередко создается с помощью визуальных средств, однако некоторые эффективные, но максимально упрощенные СУБД поставляются без вспомогательного инструментария, и создавать структуру базы данных приходится вручную. При этом прежде всего требуется создать таблицу. Делается это с помощью следующего оператора

```
CREATE TABLE название
 (поле тип дополнительные-описания,
...);
```

Например, если требуется создать таблицу TEST, состоящую из двух полей: ID и Stroka, — причем первое из них является первичным ключом (целое число), а второе — набором символов (до 50), то соответствующий оператор запишется следующим образом.

```
CREATE TABLE TEST
( ID integer NOT NULL PRIMARY KEY,
  Stroka char(50) );
```

Чтобы создать индекс для одного из полей, применяется следующий оператор.

```
CREATE INDEX название ON таблица (список-полей);
```

Например:

```
CREATE INDEX MyInd ON TEST (Stroka);
```

Удаляется индекс следующей командой.

```
DROP INDEX название-индекса;
```

Таблица целиком удаляется схожим способом.

```
DROP TABLE название-таблицы;
```

ЗАМЕЧАНИЕ Язык SQL содержит еще очень много разнообразных возможностей, но даже с помощью этого небольшого набора операторов можно эффективно работать с базами данных. Более подробную информацию можно посмотреть в учебниках по языку SQL и документации по конкретным СУБД.

Создание отчетов

Принципы создания отчетов в Delphi 7

Отчеты используются в самых разных программах. Особенно активно они формируются во всевозможных приложениях, связанных с автоматизацией делопроизводства, когда документы хранятся в электроном виде 8 базе данных, но их требуется регулярно выводить на печать. Отчеты в системе *Delphi 7* ориентированы в первую очередь на печать информации из таблиц баз данных.

В системе *Delphi 7* отчет — это виртуальный образ бумажного листа, который в дальнейшем без изменений воспроизводится на принтере. В седьмой версии *Delphi* появился набор компонентов Rave Reports (панель Rave), который заменил морально устаревший набор QReports. Эти компоненты позволяют подготовить произвольное число виртуальных страниц отчетов в простом визуальном редакторе. Компоненты набора обладают множеством характеристик, что позволяет детально настроиться на возможности конкретного принтера.

Работа с отчетом

Рассмотрим пример создания простого отчета, основанного на таблице Clients из базы *DBDEMOS*. Разместим на форме компонент TTable, настроим его соответствующим образом и сделаем активным — установим для свойства Active значение True. Добавим на форму компонент TRvDataSetConnection для организации связи будущего отчета с базой данных. В его свойстве DataSet укажем название объекта-таблицы Table1.

Теперь надо спроектировать внешний вид отчета. Для этого вызовем визуальный *Rave-проектировщик* Rave Visual Designer командой Tools ➤ Rave Designer. Окно проектировшика состоит из четырех основных частей. В верхней части расположены кнопки управления и панели компонентов. В центре можно видеть проектируемый отчет. В левой части находится редактор свойств текущего объекга, в правой разработчику доступен Просмотрщик объектов, схожий с аналогичным инструментом *Delphi*.

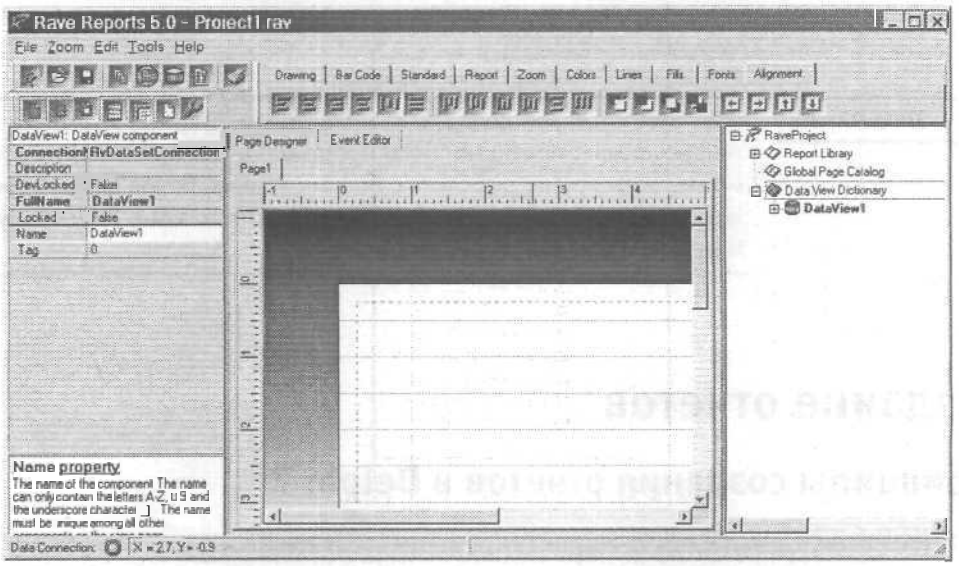


Рис. *6.19. Главный экран проектировщика отчетов*

Дадим команду File > New Data Object (Файл ➤ Новый объект данных) и в окне типов связи (рис. 6.20) выберем строку Direct Data View (Прямой обзор данных).

В следующем окне (рис. 6.21) будет предложено выбрать соединения с СУБД, доступные на данный момент в среде *Delphi*. В нашем случае это единственная связь RvDataSetConnection1. Теперь структуру установленной связи (она получила название DataView1) можно посмотреть в правой части *Rave*-редактора.

Подготовим табличную форму отчета с помощью *Rave*-Мастера. Для этого надо выполнить команду Tools ➤ Report Wizards ➤ Simple Table (Сервис ➤ Мастер отчетов >

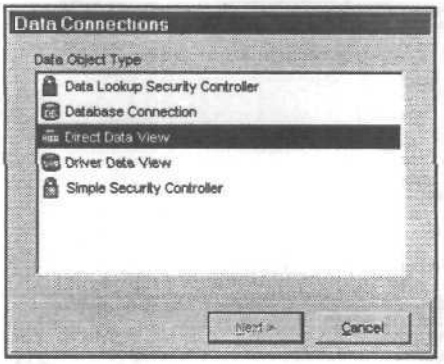


Рис. 6.20. Выбор типа связи

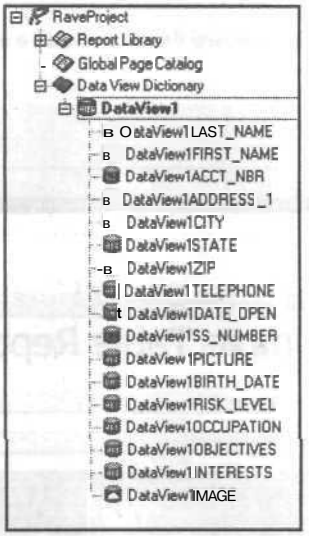


Рис. ff.21. Структура связи с СУБД

Простая таблица) и выбрать в диалоговом окне название объекта DataView1. В следующем окне (рис. 6.22) будет предложено отметить поля таблицы, которые желательно видеть в отчете.

На следующих этапах работы Мастера заданные поля можно будет отсортировать, ввести название отчета, указать границы печатаемой страницы и задать шрифты различных разделов. После нажатия в последнем окне кнопки Generate отчет будет создан и размещен в центре виртуальной страницы (рис. 6.23).

Отметим, что в одном проекте может быть несколько различных отчетов. Список доступных отчетов можно посмотреть в правом окне в элементе Report Library (Библиотека отчетов). Пока там доступен единственный отчет Report1. Назовем его MyReport.

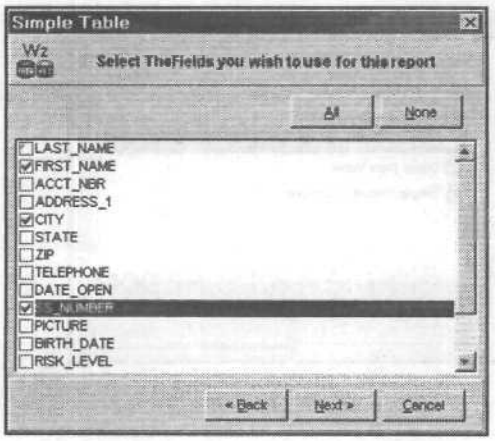


Рис. 6.22. Выбор полей для включения в отчет

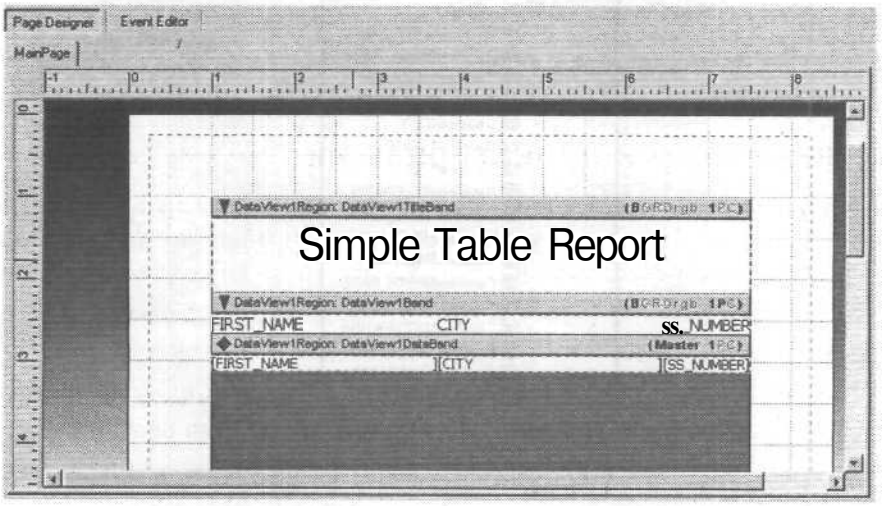


***Рис.* fi.23.** *Проектирование отчета*

Теперь созданный шаблон с нашим отчетом (он называется *Rave*-проект) надо сохранить в подходящей панке с помощью стандартный команды File ➤ Save (Файл ➤ Сохранить)с названием, например, TestRave. Файл проекта получит расширение .rav.

Вернемся в Delphi и добавим на форму компонент TRvProject (*Rave*-Проект).

Свойство Project File (Файл проекта) настроим на подготовленный файл TestRave.

Поместим на форму кнопку и в обработчике нажатия запишем следующий оператор:

```
RvProject1.ExecuteReport('MyReport');
```

В нем происходит обращение к методу ExecuteReport компонента TRvProject, по которому строится и вызывается для просмотра и печати указанный отчет.

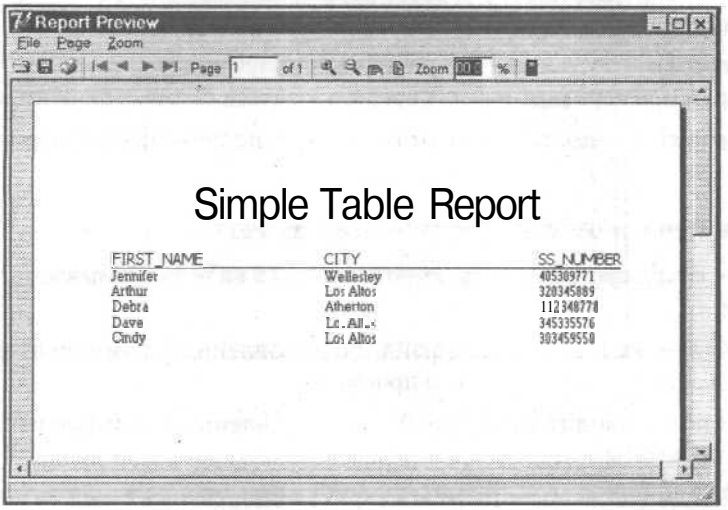


Рис. 6.24. Просмотр отчета перед выводом на печать

Визуальный Rave-проектировщик

Данная среда проста в эксплуатации и в плане пользовательского интерфейса схожа с системой Delphi. В ходе проектирования отчета разработчику доступны наборы компонентов следующих панелей:

- Drawing — графические элементы оформления (линии, полосы, фигуры);
- Bar Code — штрих-кодовые графические элементы;
- Standard — стандартные элементы отчета — одно- и многостраничные подписи, картинки, пользовательские средства настройки шрифтов;
- Report — компоненты, обеспечивающие связь отчета с базами данных;
- Zoom — средства масштабирования виртуальных страниц;
- Color — цветовые настройки отчета;
- Lines — средства построения линии;
- Fills — элементы заполнения фона;
- Fonts — средства настройки шрифтов различных частей отчета;
- Alignment — компоненты, позволяющие гибко выравнивать части отчета.

Rave-компоненты Delphi

Доступные в *Delphi* компоненты создания отчетов Rave можно поделить на следующие группы.

Компоненты связи с источниками данных

TRvCustomConnection — поставляет в отчет данные из обычных файлов или массивов в памяти программы;

TRvDataSetConnection — поставляет в отчет данные, получаемые от таблиц *BDE* (TTable). В нашем примере можно было воспользоваться этим компонентом;

TRvQuerySetConnection — поставляет в отчет данные, получаемые от запроса *BDE* (TQuery).

Компоненты печати отчета

TRvNDRWriter — сохраняет отчет в двоичной форме (в виде так называемого *NDR*-потока);

TRvRenderPreview — выводит отчет (файл, подготовленный компонентом TRvNDRWriter) в окне предварительного просмотра;

TRvRenderPrinter — выводит отчет (файл, подготовленный компонентом TRvNDRWriter) на печать;

TRvSystem — объединяет возможности этих трех компонентов в стандартной визуальной среде просмотра, настройки и печати.

Компоненты генерации отчета в других форматах

TRvRenderPDF — преобразует *NDR*-поток *в PDF*-формат с поддержкой графики;

TRvRenderHTML — преобразует *NDR*-поток в формат *HTML 4.0 с* поддержкой графики;

TRvRenderRTF — преобразует *NDR*-поток в *RTF*-формат с поддержкой графики;

TRvRenderText — преобразует *NDR*-поток в текст.

Средства анализа данных и принятия решений

Зачем нужен анализ данных

Технологии анализа данных, особенно способные работать в реальном режиме времени (*OLAP, On Line Analytical Processing*), сегодня переживают очень бурное развитие. Это связано с возросшими возможностями компьютеров и широким распространением систем автоматизации предприятий и финансовой деятельности, что вызывает потребность в анализе имеющейся информации.

Продукты *OLAP* обычно оперируют понятием *многомерного куба*. Например, имеется набор данных из четырех полей: названий складов, названий товаров, суммарной стоимости этих товаров на соответствующем складе (такое поле может быть вычисляемым) и количества товаров на определенном складе. При этом к каждому складу и каждому товару в таблице базы данных может относиться несколько записей. С помощью *OLAP*-системы можно сформировать следующее.

- *Одномерное* представление данных. По каждому товару суммируется его количество по всем складам (получается одна строка) и указывается общая стоимость этих товаров.
- *Двумерное* представление данных. Формируется таблица, столбцы которой соответствуют товарам, а строки — складам. В каждой ячейке таблицы указывается общая стоимость товаров соответствующего вида на соответствующем складе.

  При этом может возникнуть потребность поменять местами столбцы и строки. В таком случае в строках отображается информация о стоимости данного товара для каждого из складов, причем строки могут группироваться по категориям товаров.
- *Трехмерное* представление. Например, каждый из складов может быть привязан к нескольким поставщикам. В таком случае возникает потребность просмотра данных по стоимости или количеству товара в трехмерном пространстве «Поставщики — Склады — Товар* с возможностью быстрой смены этих координат. Фактически создается трехмерный куб с указанными измерениями.

В реальной жизни возникает немало случаев, когда трех измерений недостаточно. Например, у каждого товара может быть несколько групп потребителей, и тогда уже возникает потребность наглядного представления информации в четырехмерном виде, что в нашем мире невозможно даже теоретически. Поэтому *OLAP*-продукты позволяют быстро настраивать визуальное представление двумерных таблиц и трехмерных кубов, выбирая варианты разметки осей координат из списка доступных полей.

Несколько ограниченным набором подобных возможностей обладает и система *Delphi 7*. На панели Decision Cube (Многомерный куб) расположено шесть компонентов, на основе которых можно выполнять многомерный анализ данных и расширять имеющиеся базовые возможности программным путем.

Пример

Рассмотрим вышеописанный пример на практике. Допустим, сформирована таблица базы данных, каждая запись которой имеет следующую структуру полей:

- ID — ключевое поле;
- Sklad — название склада;
- Tovar — название товара;

- Cena — общая стоимость данного товара на складе;
- Num — число единиц данного товара на складе.

Создадим новое приложение и разместим на нем компонент TDecisionQuery — наследник класса TQuery, позволяющий формировать произвольные запросы к базам данных.

Вызовем редактор запросов командой Desicion Query Editor (Редактор запросов анализа) из контекстного меню. В открывшемся диалоговом окне выберем имя базы данных в раскрывающемся списке Database (База данных), а имя нужной таблицы — в списке Table (Таблица). Все поля, входящие в эту таблицу, появятся в списке List of Avialable Fields (Список доступных полей). Выберем строку Tovar и переместим это поле в правую часть окна в раздел Dimensions (Измерения куба) с помощью кнопки со стрелкой. Теперь необходимо определить, какое значение будет показываться в строке под каждым из товаров. Прежде всего это должна быть общая стоимость конкретного типа товара на всех складах. Выберем строку Cena и щелкнем в нижней части диалогового окна на кнопке со стрелкой, относящейся к списку Summaries (Итоги). Появится небольшое меню, в котором будет предложено выбрать способ подсчета итогов.

- Пункт sum определяет вычисление суммы всех значений поля Cena, относящихся к каждому товару.
- Пункт count позволяет определить число имеющихся в таблице значений поля Cena, относящихся к каждому товару.
- Пункт average вычисляет среднее значение поля Cena, относящееся к каждому товару.

Выберем пункт sum.

Интересно также узнать, на скольких складах хранится тот или иной товар. Для этого в список Summaries добавим поле Sklad, применив к нему функцию count (рис. 6.25).

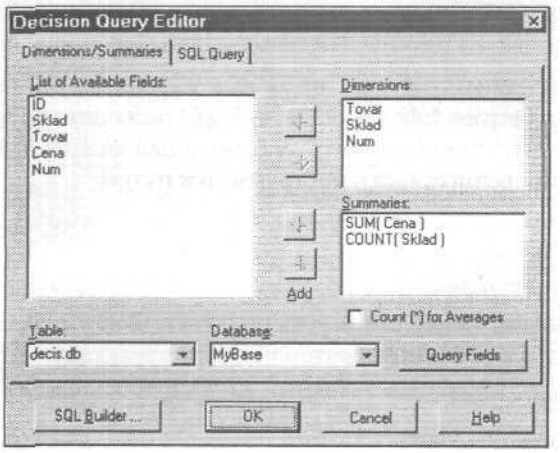


РИС. 6.25. *Выбор сведений, включаемых в сводный отчет*

ЗАМЕЧАНИЕ Из данного окно можно также вызвать ранее рассмотренный построитель запросов SQL Builder, щелкнув на соответствующей кнопке.

Теперь вернемся в Проектировщик и разместим на форме компонент TDecisionCube.  Этот компонент выполняет все вычислительные функции, связанные с построением многомерных кубов (в нашем случае, пока что, одномерной таблицы). Укажем в его свойстве DataSet исходный набор данных — объект DecisionQuery1. Отметим, что в качестве такого набора может выступать произвольный объект класса TDataSet, что позволяет выполнять анализ информации, поступающей из практически любых источников.

Далее требуется добавить третий компонент — TDecisionSource, который является промежуточным звеном между элементами отображения данных на  форме и компонентом построения аналитических кубов, выполняя ряд промежуточных преобразований при передаче информации. В свойстве DecisionCube надо указать ссылку на аналитический компонент DecisionCube1.

Для формирования пользовательского интерфейса поместим на форму компонент TDecisionPivot (элемент управления внешним представлением аналитических данных) и компонент TDecisionGrid (двумерную таблицу для   отображения многомерных данных). Связь с промежуточным обработчиком данных DecisionSource1 выполняется через свойство DecisionSource.

На этом процесс создания аналитического модуля закончен. Свойству Active компонента TDecisionQuery присваиваем значение True, и сразу по цепочке происходит формирование итоговых данных. На основе запроса в объекте DecisionQuery1 формируется промежуточный набор данных, далее он передается в объект DecisionCube1, где выполняются все необходимые вычисления. Затем промежуточные данные приводятся к виду, пригодному для использования в элементах отображения информации, и, наконец, этими данными заполняются объекты DecisionPivot1 и DecisionGrid1,

Очень важное отличие *OLAP*-технологии от подходов, связанных с простым выполнением запросов *SQL,* состоит в том, что данные, входящие в многомерный куб, рассчитываются заранее, и поэтому их отображение «под произвольными углами» — в нужных пользователю ракурсах уже не занимает много времени,

ВНИМАНИЕ Процесс формирования многомерного куба довольно длительный и весьма ресурсоемкий, особенно когда анализируются большие наборы данных, где число значений по каждому измерению велико. Потребность в памяти также возрастает в геометрической прогрессии. Ход этого процесса отображается заполнением шкалы во временном окне, если значение свойства ShowProgressDialog для компонента TDecisionCube установлено равным True.

Программу можно откомпилировать и запустить. Ее окно будет выглядеть примерно так, как показано на рис., 6.26.

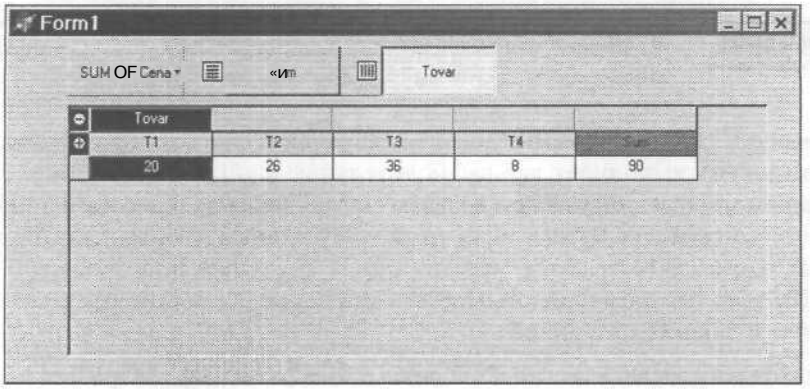


Рис. 6.26. *Отображение итогового отчета*

Рядом с желтым кружком, позволяющим при помощи щелчка сворачивать или разворачивать окно с таблицей, содержащей результаты анализа по конкретному измерению, указывается название поля набора данных, по которому проводится анализ, — Tovar. Такое же действие выполняется при нажатии или отжатии кнопки Tovar на объекте DecisionPivot1. Эту кнопку можно перетаскивать в рамках этого объекта между двумя панелями. Первая из них означает, что данное измерение (ось) должно располагаться по вертикали, вторая — по горизонтали.

В первой строке приводятся названия товаров, во второй — сумма стоимости каждого товара по всем складам. В конце строки приводится итоговая сумма всех значений (столбец Sum). Если требуется посмотреть, на скольких складах хранится каждый товар, надо щелкнуть на кнопке SUM OF Cena объекта DecisionPivotl и выбрать в открывшемся меню другое значение — COUNT OF Sklad (Число складов). При этом сразу изменится и содержимое таблицы.

Добавим в редакторе запроса объекта DecisionQuery1 в список Dimensions еще одно измерение — Sklad. Теперь выбранный нами итог (например, сумма стоимостей товаров) будет отображаться уже в двумерной таблице, одна из осей которой соответствует названиям складов, а другая — названиям товаров. Чтобы выполнить пересчет, требуется вновь установить значение поля Active компонента TDecisionQuery равным True. Это значение автоматически сбрасывается при каждом изменении настроек запроса.

Теперь таблица стала двумерной. Обратите внимание, что суммы по столбцам и строкам всегда должны совпадать, что напоминает требования бухгалтерской отчетности (рис. 6.27).

Свернув таблицу по полю Sklad, получим созданное ранее «потоварное» представление. Свернув таблицу по полю Tovar, мы узнаем, на какую сумму хранится товаров на каждом складе. Поменять местами вертикальную и горизонтальную оси в таблице можно, поменяв с помощью мыши местами кнопки Sklad и Tovar на объекте

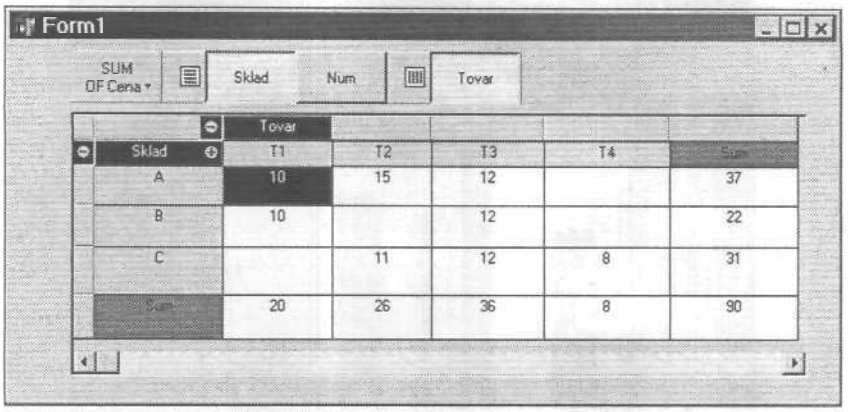


Рис. 6.27. *Представление итоговых данных в виде двумерной таблицы*

DecisionPivot1. Можно также установить, сколько типов товаров хранится на каждом из складов, изменив в этом объекте значение кнопки SUM OF Cena на COUNT OF Sktad и так далее (рис. 6.28).

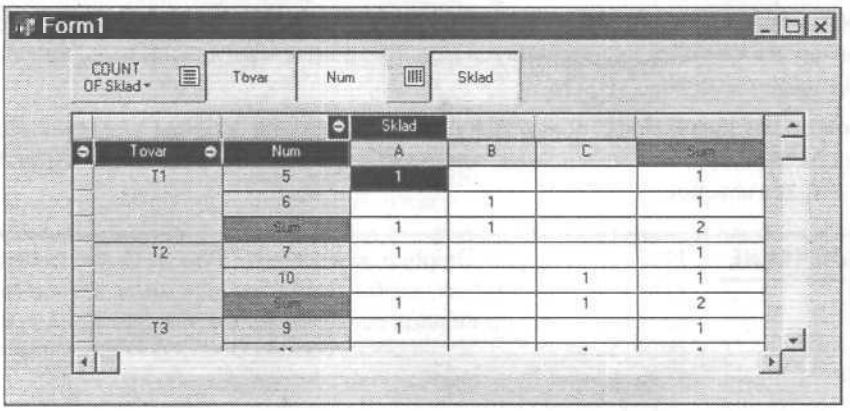


Рис. 6.28. Полный обзор сведений, на основе которых сформирован итоговый отчет

На панели Decision Cube имеется еще один компонент — TDecisionGraph, предназначенный для графического отображения многомерной информации. Он является наследником компонента TChart — пользовательской диаграммы.

После размещения его на форме и задания в свойстве DecisionSource ссылки на объект DecisionSource1 на графике сразу появится изображение, наглядно показывающее, каковы суммарные стоимости товаров по каждому из складов. Если сменить координатные оси с помощью объекта DecisionPivot1, можно узнать, каковы суммарные стоимости товаров на складах по каждому из товаров (рис. 6.29).

Внешний вид графика можно настроить стандартным способом, о котором рассказывалось в разделе, посвященном компоненту TChart.

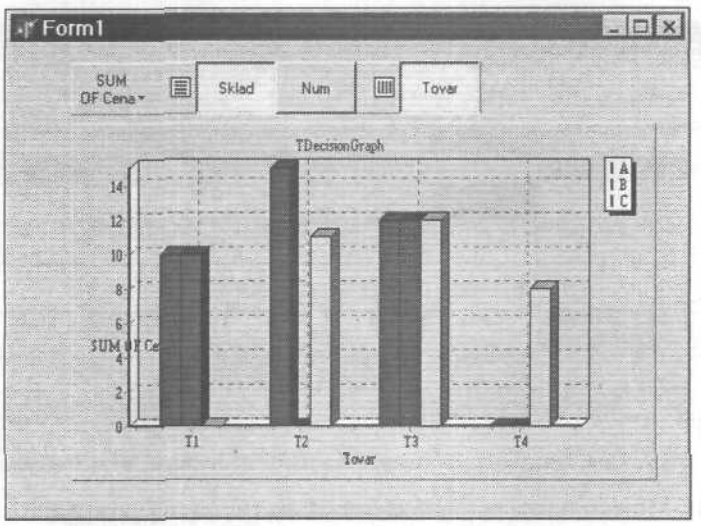


Рис. 6.29. Представление итоговых данных в виде ***диаграммы***

Число товаров на складе для эксперимента можно добавить и как поле для анализа (в список Summaries), и как третье измерение, благодаря чему можно посмотреть, например, на каких складах хранятся максимальные и минимальные запасы и для каких видов товаров.

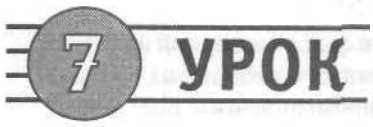
Возможности компонентов аналитической обработки данных системы *Delphi 7* очень полезны при создании самых разных аналитических приложений для любых отраслей деятельности.

ЗАМЕЧАНИЕ Начиная с версии Delphi 6, компонент TDecisionCube поставляется в исходных текстах, чтобы разработчики могли более глубоко разобраться в принципах создания аналитических приложений.

Что нового мы узнали?

В этом уроке мы научились

- применять средства визуального проектирования;
- создавать запросы к базам данных;
- записывать выражения на *языке SQL;*
- генерировать и форматировать отчеты;
- выполнять анализ данных.



Работа с клиент-серверными СУБД

- Принципы работы с клиент-серверными СУБД
- Работа с СУБД InterBase
- Расширенная поддержка СУБД InterBase 6
- Работа с SQL-серверами (панель dbExpress)

Принципы работы с клиент-серверными СУБД

Зачем нужны клиент-серверные СУБД

До сих пор мы рассматривали работу с базами данных в файл-серверной архитектуре. Сегодня этот подход актуален только для автономных и небольших сетевых приложений с десятком пользователей. Современные промышленные программы создаются на более мощных, так называемых *SQL*-серверах. Такие СУБД выполняются на отдельном компьютере и берут на себя большую часть работы по обслуживанию запросов пользователей, поиску, отбору, сортировке записей в таблицах и другим операциям.

ЗАМЕЧАНИЕ В файл-серверной архитектуре такую работу выполняют драйверы, представляющие собой фактически полноценную систему управления базами данных. В клиент-серверной архитектуре драйверы требуются только для установки и поддержания надежной связи с программой СУБД, выполняющейся в сети.

Такие СУБД обладают набором дополнительных возможностей:

- обеспечивается значительно более высокая сохранность данных;
- ведется контроль действий пользователей, обеспечивается разграничение их полномочий. Например, можно указать, к каким таблицам и в каком режиме (только для просмотра или с правом изменения) могут обращаться конкретные пользователи.

Принципы работы с подобными СУБД в системе *Delphi 7* не очень отличаются от принципов работы с файл-серверными базами данных. Благодаря использованию промежуточного модуля *BDE* разработчику не важно, где находится конкретная таблица (TTable). Надо только предварительно указать в настройках, какая СУБД будет применяться для доступа к той или иной таблице.

При работе с *SQL*-серверами использовать компонент TTable для прямого доступа к таблицам базы недостаточно. Это связано с более жесткими требованиями клиент-серверных СУБД к клиентским программам. Такие СУБД могут работать на других компьютерах, нежели клиентское приложение, и необходимо выполнять контроль надежности связи через сеть, проверять права программы, пытающейся подклю-

читься к базе данных, и так далее. Кроме того, *SQL*-серверы способны хранить и обрабатывать не одну, а несколько баз данных, каждая из которых обладает собственным набором таблиц.

Компонент Источник данных (TDataSource)

Компонент обеспечивает связь между таблицами и другими физическими наборами данных и элементами управления на форме.

Создание источника данных уже рассматривалось. Этот компонент не требует сложных настроек. Отметим только такое его свойство, как State (тип TDataSetState), которое позволяет узнать, в каком состоянии находится указанный в свойстве DataSet набор данных (например, в режиме редактирования, добавления, фильтрации и так далее).

Программисту в практической работе пригодится метод

```
function IsLinkedTo(DataSet: TDataSet): Boolean;
```

Этот метод определяет, связан ли источник данных с набором данных DataSet, заданным в качестве параметра.

Событие OnDataChange возникает, когда происходит редактирование одной из записей набора данных, событие OnStateChange — когда меняется состояние набора данных, событие OnUpdateData — перед тем, как будет выполнен метод Post по внесению сделанных обновлений в базу данных.

Работа с СУБД InterBase

В состав системы *Delphi 7* входит клиент-серверная СУБД *InterBase* (версия 6.5, не поддерживаемая ранними версиями *Delphi)* производства корпорации *Borland.* Этот продукт, начиная с версии 6, распространяется бесплатно, хотя лицензионное соглашение накладывает определенные ограничения на его использование. Версии *InterBase 5.x* остаются коммерческими.

СУБД *InterBase* устанавливается из начального окна установки продукта. Для этого надо выбрать пункт InterBase 6.5 и далее следовать командам процедуры установки (рис. 7.1).

ЗАМЕЧАНИЕ Настройка СУБД InterBase, ее сопровождение, администрирование, способы создания баз данных и таблиц в данной книге не рассматриваются. В дальнейшем считается, что во время создания и запуска демонстрационных программ доступ к этой СУБД разрешен.

Рис. 7.1. *Установка СУБДInterBase*

Компоненты для работы с СУБД InterBase

С этой СУБД (как и с любой другой) можно работать и с помощью стандартных компонентов панели Data Access (TDatabase, TSession, TTable и пр.), но компоненты панели InterBase предоставляют значительно более эффективные технологии доступа к базам данных *InterBase* благодаря прямому обращению к программному интерфейсу этой СУБД.

ПОДСКАЗКА При использовании СУБД InterBase рекомендуется применять компоненты панели InterBase.

Несколько баз данных в одном приложении

Рассмотрим следующий пример. Имеется база данных COMP, созданная в формате СУБД *InterBase*. В ней хранится одна таблица NEEDS, состоящая из двух полей: числового поля PartID, хранящего код записи (поле PartNo из таблицы Parts базы данных DBDEMOS), и строки (поле Info, 50 символов), содержащей описание детали в свободном формате. Требуется связать эту таблицу с ранее рассмотренным примером, заносить данные в нее во время добавления новой записи, а затем просматривать хранимую информацию.

Среда *Delphi 7* устроена настолько удобно, что работа с несколькими базами данных в разных форматах ничем не отличается от работы с одной базой данных. Просто соответствующие компоненты доступа настраиваются разными способами, а так как компоненты, отображающие информацию, работают с промежуточным звеном — источниками данных, не зависящими от конкретных СУБД, — то никаких серьезных изменений в структуру приложения вносить не потребуется.

Доступ к базе данных InterBase

Поместим в ранее созданный для работы с таблицей Parts модуль данных новый компонент TIBDatabase. В свойстве DatabaseName надо указать полный путь поиска и имя файла базы данных, например C:\ib65\COMP.GDB.

Работа с СУБД *InterBase* требует обязательной поддержки транзакций. Для этого поместим в модуль данных компонент TIBTransaction, ответственный за управление транзакциями. В его свойстве DefaultDatabase (База данных по умолчанию) укажем созданный объект IBDatabase1. В свою очередь, в свойстве DefaultTransaction этого объекта надо указать транзакционный объект IBTransaction1, чтобы установить между ними связь.

Компонент TIBTransaction расширяет возможности по управлению транзакциями, В частности, если начало и завершение транзакции при использовании компонентов с панели Data Access выполнялось с помощью методов компонента TDatabase, то при работе с СУБД *InterBase* за это отвечает компонент TIBTransaction.

Теперь можно добавить в модуль данных компонент TIBTable, позволяющий организовать доступ к конкретной таблице указанной базы данных. В свойство Database заносится значение IBDatabase1, в свойсво TableName - имя таблицы NEEDS (в этот момент СУБД *InterBase* может запросить имя пользователя и пароль для доступа к базе данных). Далее надо создать виртуальные образы полей в Просмотрщике модуля данных, а именно выполнить команду Add all fields (Добавить все поля) для раздела Fields объекта IBTable1.

Чтобы иметь возможность обращаться к этой таблице наравне с другими, поместим на форму еще один компонент TDataSource, в котором в свойстве DataSet укажем значение IBTable1. Теперь таблица из другой базы данных свободно доступна в любой точке нашей программы.

Ввод значений в таблицу InterBase

Поместим на форме текстовое поле Edit1 с подписью Описание детали и кнопку Button1. В обработчик щелчка на кнопке надо вставить следующий текст:

```
if Edit1.Text <> '' then
   begin
      // начало транзакции:
      DataModule2.IBTransaction1.StartTransaction;
      try
         // дополнительная проверка, открыта ли нужная
таблица
         if not DataModule2.IBTable1.Active then
            DataModule2.IBTable1.Open;
         // вставка записи:
         DataModule2.IBTable1.Insert;
         // в первое поле заносится
```

```
            // значение ключевого поля из таблицы Parts
            DataModule2.IBTable1.Fields[0].AsInteger :=
            DataModule2.Parts.Fields[0].AsInteger;
            // во второе поле заносится значение поля ввода
Edit1
            DataModule2.IBTable1.Fields[1].AsString :=
Edit1.Text;
            // информация записывается в таблицу NEEDS:
            DataModule2.IBTable1.Post;
            // успешное завершение транзакции:
            DataModule2.IBTransaction1.Commit;
            // ошибка:
        except
            DataModule2.IBTransaction1.RollBack;
            raise;
        end;
    end;
```

Обработка подключения к базе данных

При запуске программы пользователю каждый раз задается вопрос о его правах: имени и пароле. Это неудобно, поэтому лучше поместить нужные данные в программный код. Проще всего поступить так.

1. Значение свойства LoginPrompt (Запрос прав пользователя) установить равным False.
2. В свойстве Params (параметры подключения к серверу, тип TStrings) указать две строки:

```
user_name=SYSDBA
password=masterkey
```

Значения имени пользователя и пароля следует изменить на те, что заданы администратором. Теперь при запуске программы диалоговое окно запроса параметров подключения появляться не будет.

Отображение в запросе новой информации

Вызовем Построитель запросов (SQL Builder) для ранее созданного запроса Query1 и в раскрывающемся списке Database выберем имя СОНР базы данных *InterBase*. В списке Table будет присутствовать единственная таблица NEEDS. Она добавляется к уже существующим таблицам и становится полностью равноправной с ними. Ее поле PartID связывается с нолем PartNo таблицы Parts, а поле Info помечается флажком. Оно будет отображаться в запросе, после чего созданный запрос можно сохранить. Как мы видим, никакой принципиальной разницы при работе с таблицами одной или разных баз данных в системе *Delphi 7* нет.

Чтобы поле отображалось на экране в списке DBGrid, надо перенастроить его столбцы, добавив новый столбец для поля Info.

Дополнительные компоненты панели InterBase

На панели InterBase имеется ряд компонентов, совпадающих по своему назначению и принципам использования с рассмотренными ранее компонентами панели Data Access, но выполняющие соответствующие функции гораздо эффективнее. Прежде всего, это компонент Запрос (TIBQuery).

Он соответствует компоненту TQuery, но предназначен для работы только с таблицами в формате InterBase. Этот компонент не имеет визуального редактора создания запроса и работает на основании текстового описания запроса на языке *SQL*, занесенного в свойство *SQL*.

ПОДСКАЗКА Можно использовать визуальный построитель запросов для компонента TQuery, чтобы создать нужный запрос без знания *SOL* Далее сохраните этот запрос в текстовом виде в файле (как это сделать, описывалось выше), а потом вставьте текст на языке SQL в свойство SQL компонента TIBQuery.

Компонент Хранимая процедура (TIBStoredProc) соответствует компоненту TStoredProc, компонент Обновление базы данных (TIBUpdateSQL) — компоненту TUpdateSQL.

Компонент **Набор** данных (TIBDataSet)

Данный компонент, наследник компонента TDataSet, позволяет выполнить команду *SQL*. Как правило, в практической работе используется оператор SELECT, но данный компонент позволяет выполнять и другие действия, например модификацию или удаление записей в наборе данных СУБД InterBase. Класс TIBDataSet имеет следующие свойства.

Таблица 7.1. *Свойства класса TIBDataSet*

Свойство	Назначение
DeleteSQL	Оператор SQi для удаления записей
InsertSQL	Оператор *SQL* для добавления записей
ModifySQL	Оператор SQi для изменения записей
Params	Значения для запроса с параметрами
Prepared	Имеет значение True, если запрос подготовлен для исполнения
RefreshSQL	Оператор SQi для обновления записей
SelectSQL	Оператор SQi для отбора записей
StatementType	Способ выполнения запроса

Основных методов два:

```
procedure Prepare;
procedure UnPrepare;
```

Первый обеспечивает подготовку всех запросов к выполнению (он выполняется над закрытым набором данных). Второй метод сбрасывает состояние готовности.

Компонент Оператор SQL (TIBSQL)

Этот компонент предназначен для выполнения конкретного запроса *SQL,* который не создает набора данных для приложения, а просто выполняет определенные действия в базе данных. Содержит набор специфических вспомогательных свойств, ориентированных на особенности работы СУБД *InterBase.*

Компонент Информация о базе данных (TIBDatabaseInfo)

Этот компонент позволяет получить *информацию* о *текущей* версии используемой базы данных, в частности, версию ее внутренней структуры, которая постоянно оптимизируется корпорацией *Borland* для обеспечения максимально быстрого доступа к данным. Соответствующая база данных указывается в свойстве Database. Подобная информация очень полезна при создании крупных приложений, когда разработчикам требуется выполнять оптимизацию производительности своих программ. Класс TIBDatabaseInfo имеет свойства, представленные в табл. 7.2.

Таблица 7.2. *Свойства класса TIBDatabaseInfo*

Свойство	Назначение
Allocation	Число размещенных страниц базы данных
BackoutCount	Число обновлений версий записи
BaseLevel	Версия базы данных
CurrentMemory	Объем памяти, используемый сервером базы данных
DBFileName	Название файла базы данных
DBSQLDialect	Идентификатор диалекта SQL
DeleteCount	Число операций удаления в базе данных с момента последнего подключения к ней
Fetches	Число операций считывания из буфера базы данных
ForcedWrites	Имеет значение 1, если запись в базу происходит в синхронном режиме
InsertCount	Число операций добавления в базу данных с момента последнего подключения к ней
Marks	Число операций записи в буфер базы данных
MaxMemory	Объем максимально доступной памяти
NoReserve	Имеет значение 0, если резервируется память для создания архивной версии измененных записей
NumBuffers	Текущее число буферов базы данных
ODSMajorVersion	Главная версия поддерживаемой дисковой структуры

Таблица 7.2. Свойства класса TIBDatabaseInfo (продолжение)

Свойства	Назначение
ODSMinorVersion	Вспомогательная версия поддерживаемой дисковой структуры
PageSize	Число байтов на страницу базы данных
Readonly	Имеет значение 1, если в базу данных можно вносить изменения, и 0, если база данных открыта только для чтения
Reads	Число считанных страниц базы данных
ReadSeqCount	Число последовательных считываний по каждой таблице базы данных
SweepInterval	Число транзакций, которое будет выполнено, прежде чем старые версии записей будут удалены
UpdateCount	Число операций обновления базы данных с момента последнего подключения к ней
UserNames	Список всех пользователей, подключенных к базе на данный момент
Version	Идентификационная строка, описывающая версию базы данных
Writes	Число страниц, записанных в базу данных

Компонент Мониторинг работы запросов SQL (TIBSQLMonitor)

С помощью компонента TIBSQLMonitor можно осуществлять контроль за ходом выполнения *SQL*-запросов из приложения к базе данных *InterBase,* что позволяет выявлять узкие места в работе программы.

Компонент очень прост. После его размещения в модуле данных надо определить обработчик события OnSQL, которое будет генерироваться каждый раз во время обращения из программы к СУБД *InterBase.* Заголовок обработчика выглядит следующим образом.

```
procedure TDataModule2.IBSQLMonitor1SQL(EventText:
String);
```

Параметр EventText описывает конкретное событие, связанное с обращением к СУБД *InterBase.*

ЗАМЕЧАНИЕ Чтобы событие OnSQL генерировалось, надо установить равным True значения свойств TraceFlags в каждом объекте класса TIBDatabase.

Компонент Отложенные события (TIBEvents)

Когда с базой данных работают несколько пользователей, нередко в сложной системе требуется отслеживать действия других клиентов и реагировать на изменения, вносимые ими в базу. В этом помогает компонент TIBEvents.

Он связывается с конкретной базой данных *InterBase* через свойство Database, а список обрабатываемых (отслеживаемых) событий заносится в свойство Events. В названиях событий надо учитывать регистр. Эти события могут храниться в

СУБД *InterBase* в виде очереди отложенных сообщений и рассылаться программам не сразу, а по мере возможности, например когда нагрузка на сервер понижается до некоторого уровня.

Метод CancelEvents позволяет отменить обработку всех оставшихся отложенных сообщений. Начать поочередную обработку их можно с помощью метода QueueEvents. Чтобы обрабатывать не все сообщения от СУБД *InterBase* (их может быть очень много), можно зарегистрировать на сервере только ограниченный набор событий, занесенных в свойство Events. Для этого предназначен метод RegisterEvents.

Когда от СУБД *InterBase* приходит уведомление о некотором случившемся событии, приложение получает сообщение OnEventAlert. Именно в его обработчике и выполняются все действия по отслеживанию сообщений, поступающих от СУБД *InterBase*.

Заголовок обработчика выглядит следующим образом.

```
procedure TDataModule2.IBEvents1EventAlert(Sender: TObject;
EventName: String;
EventCount: Integer;
var CancelAlerts: Boolean);
```

Поскольку даже зарегистрированных событий в *InterBase* может быть очень много при активной работе сотни пользователей, то в обработчик передается событие, наиболее часто случавшееся с момента последнего вызова этого обработчика (оно описано в параметре EventName), а параметр EventCount содержит число этих событий.

Если требуется временно приостановить получение всех информационных сообщений от СУБД *InterBase,* в параметр CancelAlerts надо записать значение True. Чтобы восстановить генерацию сообщений, необходимо вызвать метод QueueEvents.

Компонент Информация InterBase (TIBExtract)

Компонент предназначен для получения подробной информации о текущей базе данных. Это сведения о структуре таблиц, запросов, индексов и других объектов. Для использования компонента TIBExtract надо настроить свойства Database и Transaction на соответствующую базу данных, после чего вызвать метод *ExtractObject()*, в параметрах ObjectType и ObjectName которого должен быть указан тип запрашиваемой информации:

```
procedure ExtractObject(ObjectType: TExtractObjectTypes;
  ObjectName: String = ""; ExtractTypes: TExtractTypes = []);
```

Значение, полученное в результате работы метода, записывается в виде массива строк в свойстве Items.

Кроме того, в момент подключения данного компонента к базе данных выполняется автоматическая запись глобальной информации о ней в свойство

DatabaseInfo: TIBDatabaseInfo

Это всевозможная информация о количестве доступных буферов, числе считанных или записанных страниц, версии структуры диска и тому подобные сведения.

Компонент Клиентский набор данных InterBase (TIBClientDataSet)

В *Delphi 7* появилось несколько наследников класса TClientDataSet, поддерживающих кэширование обновлений и рассчитанных на работу с механизмом конкретной СУБД. Для *InterBase* — это данный компонент. Его рекомендуется всегда применять совместно с компонентом TIBDatabase.

Расширенная поддержка СУБД InterBase 6

В поставку системы *Delphi 7* входит набор компонентов панели InterBase Admin (Администрирование InterBase). Эти компоненты позволяют управлять работой сервера *InterBase* непосредственно из программы, настраивать его, и даже инсталлировать и удалять. Последнее необходимо для крупных сетевых приложений, которые должны самостоятельно установить СУБД на сервер и удалить ее с сервера, не привлекая к этому пользователей продукта.

Новая версия СУБД *InterBase 6* имеет расширенный программный протокол и позволяет выполнять различные действия по ее настройке и конфигурированию.

ВНИМАНИЕ Рассматриваемые в этой главе компоненты работоспособны только с версией СУБД InterBase 6 и выше.

Иерархия компонентов InterBase Admin

Класс TIBCustomService — это базовый класс для всех компонентов группы IBService, участвующих в настройках различных служб *InterBase 6.* В частности, он определяет сервер сети, на котором запускается конкретная служба. Свойства и методы класса TIBCustomService приведены в табл. 7.3 и 7.4.

Таблица 7.3. *Свойства класса TIBCustomService*

Свойство	Назначение
Active	Имеет значение True, если служба активна
LoginPrompt	Имеет значение True, если необходим запрос на ввод информации о пользователе при его подключении к базе данных
Params	Параметры базы данных
Protocol	Вид используемого сетевого протокола
ServerName	Название сервера, на котором исполняется служба

Таблица 7.4. Методы класса TIBCustomService

Метод	Назначение
procedure Attach;	Подключение к базе данных
procedure Detach;	Отключение от базы данных

Класс TIBCustomService поддерживает следующие события.

- Событие QnAttach возникает после подключения к базе данных.
- Событие OnLogin возникает в момент запроса данных о пользователе. Это сообщение надо обрабатывать, если ввод имени и пароля будет выполняться программно.

Класс TIBControlService — наследник класса TIBCustomService, на основе которого создаются классы группы IBService, ответственные за конкретные службы. Его основное свойство IsServiceRunning. Оно имеет значение True, если служба запущена. Его основной метод выполняет запуск службы:

```
procedure ServiceStart;
```

Класс TIBControlAndQueryService — наследник класса TIBControlService. Его свойства и методы описаны в табл. 7.5 и 7.6.

Таблица 7.5. Свойства класса TIBControlAndQueryService

Свойство	Назначение
BufferSize	Размер буфера
Eof	Имеет значение True, если встретился конец файла

Таблица 7.6. Методы класса TIBControlAndQueryService

Метод	Назначение
function GetNextChunk: String;	Получение следующего блока данных
function GetNextLine: String;	Получение следующей строки данных

Класс TIBBackupRestoreService — наследник класса TIBControlAndQueryService. Класс TIBSetup — базовый класс для компонентов, ответственных за установку и удаление элементов *InterBase 6*. Его свойства приведены ниже.

Таблица 7.7. Свойства класса TIBSetup

Свойство	Назначение
ErrorContext	Пользовательские данные для функции ошибок
MsgFilePath	Полный путь поиска для файла с сообщениями сервера базы данных
Progress	Процент выполнения процесса установки
RebootToComplete	Имеет значение True, если по завершении установки необходимо перезагрузить компьютер
StatusContext	Пользовательские данные для функции состояния

Для класса TIBSetup генерируются следующие события.

- Событие Ол Error возникает в случае ошибки в ходе инсталляции.
- Событие OnStatusChange отражает изменение состояния процесса инсталляции. Как правило, оно генерируется при изменении свойства Progress.
- Событие OnWarning возникает при необходимости выдачи предупреждения в ходе инсталляции.

Компонент Конфигурация сервера (TIBConfigService)

Компонент TIBConfigService (наследник класса TIBControlService) позволяет программно настраивать различные характеристики и режимы работы баз данных. Его основное свойство — DatabaseName. Оно содержит название базы данных. Методы класса TIBConfigService приведены ниже.

Таблица 7.8. Методы класса TIBConfigService

Метод	Назначение
procedure ActivateShadow;	Активизация резервной базы данных
procedure BringDatabaseOnline;	Подключение базы данных в реальном масштабе времени
procedure SetAsyncMode(Value: Boolean);	Перевод базы данных в работу в асинхронном режиме (с активной буферизацией)
procedure SetPageBuffers(Value: Integer);	Установка числа буферных страниц
procedure SetReadonly(Value: Boolean);	Перевод базы данных в работу в режиме «только чтение»
procedure SetSweepInterval(Value: Integer);	Установка числа транзакций, после которых производится удаление старых версий измененных записей
procedure ShutdownDatabase(Options: TShutdownMode; Wait: Integer);	Выключение базы данных через указанное число секунд

Компонент Архивирование базы данных (TIBBackupService)

СУБД *InterBase 6* позволяет создавать архивные (резервные) копии баз данных в фоновом режиме незаметно для пользователей. Это дает возможность программисту встраивать в приложения режимы автоматического архивирования информации без привлечения дополнительного обслуживающего персонала.

Компонент TIBBackupService наследует свойства и методы классов TIBControlAndQueryService и TIBBackupRestoreService.

Таблица 7.9- Свойства класса TIBBackupService

Свойство	Назначение
BackupFile	Файл для архивирования
Blocking Factor	Фактор блокирования при использовании ленточных архивных устройств
DatabaseName	Название архивируемой базы данных
Options	Метод архивирования

Компонент Восстановление базы данных (TIBRestoreService)

При необходимости автоматического (из программы) восстановления базы данных надо воспользоваться компонентом TIBRestoreService. Помимо свойств BackupFile и DatabaseName, аналогичных свойствам компонента TIBBackupService, данный компонент имеет дополнительные свойства.

Таблица 7.10. Свойства класса TIBRestoreService

Свойство	Назначение
Options	Метод восстановления
PageBuffers	Размер буфера страницы в килобайтах
PageSize	Размер страницы в килобайтах

Компонент Проверка состояния базы данных (TIBValidationService)

Когда сервер *InterBase* работает с несколькими базами данных, расположенными на различных компьютерах, вполне реальна ситуация возникновения сбоев в сети или на отдельных компьютерах. При этом происходит частичное или полное прекращение доступа к некоторым базам данных. Проверить текущее состояние конкретной базы данных поможет компонент TIBValidationService (наследник класса TIBControlAndQueryService). Его свойства и методы приведены в табл. 7.11 и 7.12.

Таблица 7.11. Свойства компонента TIBValidationService

Свойство	Назначение
DatabaseName	Название базы данных, подлежащей проверке
GlobalAction	Поддержка глобальных транзакций
LimboTransactionInfo	Информация об отложенных транзакциях
Options	Метод проверки базы данных

Таблица 7.12. *Методы компонента TIBValidationService*

Метод	Назначение
function FetchLimboTransactionInfo;	Получение информации об отложенных транзакциях
function FixLimboTransactionErrors;	Восстановление отложенных транзакций после ошибок

Компонент Статистика работы с базой данных (TIBStatisticalService)

Для сбора статистики работы с конкретной базой данных *InterBase* предназначен компонент TIBStatisticalService (наследник класса TIBControlAndQueryService).

У этого компонента два свойства: стандартное свойство DatabaseName и свойство Options, имеющее специфический тип TStatOption и определяющее, какую информацию компонент запрашивает у сервера. Оно может иметь следующие значения;

- DataPages (запрашивается путь к файлу базы данных с точки зрения сервера);
- DbLog (запрашивается протокол работы);
- HeaderPages (запрашивается информация о заголовках страниц базы данных);
- IndexPages (запрашивается статистика по пользовательским индексным страницам);
- SystemRelations (запрашивается статистика по системным таблицам).

Компонент Протокол работы (TIBLogService)

СУБД *InterBase 6* ведет протокол работы, который хранится в файле interbase.log. Записи вносятся в протокол в стандартном текстовом формате. При желании несложно написать программу, которая будет выполнять анализ этого протокола, для чего надо использовать компонент TIBLogService (наследник класса TIBControlAndQueryService), не имеющий собственных свойств и методов.

Компонент Управление доступом пользователей (TIBSecurityService)

Когда с базой данных работает много пользователей, а СУБД *InterBase 6* встроена в приложение, желательно настраивать разграничение доступа пользователей непосредственно из этого приложения. Компонент TIBSecurityService (наследник класса TIBControlAndQueryService) содержит все необходимые свойства и методы для управления доступом пользователей к СУБД *InterBase 6* (табл. 7.13 и 7.14).

Таблица 7.13. Свойства компонента TIBSecurityService

Свойство	Назначение
FirstName	Имя (первая часть имени) пользователя
GroupID	Идентификатор группы пользователей
GroupName	Название группы пользователей
LastName	Фамилия (последняя часть имени) пользователя
MiddleName	Отчество (средняя часть имени) пользователя
Password	Пароль пользователя
SecurityAction	Вид действия (добавление, удаление, модификация информации о пользователе)
UserID	Идентификатор пользователя
UserInfo	Полная информация о пользователе в формате структуры TUserInfo
UserName	Имя пользователя для подключения к базе данных

Таблица 7.14. *Методы компонента TIBSecurityService*

Метод	Назначение
procedure AddUser;	Добавление нового пользователя
procedure DeleteUser;	Удаление текущего пользователя
procedure DisplayUser (UserName: String);	Запись информации о пользователе в свойство UserInfo
procedure DisplayUsers;	Получение информации обо всех пользователях
procedure ModifyUser;	Изменение информации о пользователе

Компонент Лицензирование (TIBLicensingService)

Большинство СУБД, в том числе и *InterBase*6, распространяются по принципу лицензирования с ограничением числа пользователей, одновременно подключенных к системе. Например, версии СУБД, входящие в поставку *Delphi 7,* допускают не более двух пользователей, одновременно работающих с *InterBase.* Небольшие компании обычно покупают систему *InterBase* с лицензией на 10 пользователей. В случае расширения системы приобретать новую СУБД не обязательно — достаточно приобрести *лицензию* (сертификат) на дополнительное число пользователей (например, еще на 20). Тогда в сумме СУБД сможет работать с 30 пользователями.

В СУБД *InterBase 6* такой процесс расширения возможностей можно автоматизировать. Для этого предназначен компонент TIBLicensingService (наследник класса TIBControlService), позволяющий добавлять или удалять сертификаты пользователей в системе (табл. 7.15 и 7.16).

Таблица 7.15. Свойства компонента TIBLicensingService

Свойство	Назначение
Action	Вид действия (добавление или удаление лицензии)
ID	Идентификатор лицензии
Key	Ключ лицензии

Таблица 7.16. Методы компонента TIBLicensingService

Метод	Назначение
procedure AddLicense;	Добавление лицензионного сертификата
procedure RemoveLicense;	Удаление лицензионного сертификата

Компонент Информация о сервере (TIBServerProperties)

Компонент TIBServerProperties, наследник класса TIBCustomService, позволяет получить от СУБД *InterBase6* подробную информацию о сервере баз данных, включая параметры настройки, текущую версию, лицензионные данные и так далее (табл. 7.17 и 7.18).

Таблица 7.17. Свойства компонента TIBServerProperties

Свойство	Назначение
ConfigParams	Параметры конфигурации сервера
DatabaseInfo	Информация о структуре базы данных
LicenseInfo	Информация о лицензии базы данных
Options	Свойства сервера (все свойства компонента, объединенные в структуру TPropertyOption)
VersionInfo	Информация о версии сервера

Таблица 7.18. Методы компонента TIBServerProperties

Метод	Назначение
procedure Fetch;	Получение всей информации о сервере
procedure FetchDatabaseInfo;	Получение информации о базе данных
procedure FetchLicenseInfo;	Получение информации о лицензиях
procedure FetchConfigParams;	Получение информации о параметрах конфигурации сервера
procedure FetchVersionInfo;	Получение информации о версии базы данных

Компонент Инсталляция компонентов сервера (TIBInstall)

Процесс добавления или обновления версий СУБД *InterBase 6* выполняется обращением к возможностям компонента TIBInstall (табл. 7.19 и 7.20).

Таблица 7.19. *Свойства компонента TIBInstall*

Свойство	Назначение
DestinationDirectory	Конечный каталог установки
InstallOptions	Настройки процесса установки
Source Directory	Исходный каталог установки
Suggested Destination	Предлагаемый конечный каталог установки
UnInstallFile	Полный путь поиска и имя файла, который содержит протокол установки. Этот протокол используется в процессе удаления

Таблица 7.20. *Методы компонента TIBInstall*

Метод	Назначение
procedure GetOptionDescription (Option): String;	Получение описания указанного компонента, заданного параметром Option. Возможные значения: TCmdOption (команда управления); TExamplesOption (пример); TGuiOption (настройки интерфейса); TMainOption (основные характеристики)
procedure GetOptionName (Option): String;	Получение названия указанного компонента
procedure GetOptionSpaceRequired (Option): Long;	Получение сведений о пространстве на жестком диске, необходимом для установки указанного компонента
procedure InstallCheck;	Проверка, готов ли компонент для выполнения процедуры установки
procedure InstallExecute;	Запуск процесса установки

Компонент Удаление компонентов сервера (TIBUnInstall)

Этот компонент выполняет действия, обратные процессу установки, — он удаляет указанные составляющие СУБД.

Основное свойство компонента — UnInstallFile (файл, в котором хранится протокол процесса установки). Имеются и два метода: UnInstallCheck для проверки, готов ли компонент для выполнения удаления, и UnInstallExecute для запуска этого процесса.

Пример получения протокола работы

Как уже говорилось, анализ протокола работы сервера может дать немало полезной информации, например о запуске серверов, о клиентских подключениях, о дате и времени и прочей. Для анализа протокола разместим на пустой форме два компонента — кнопку TButton и компонент ведения протокола TIBLogService. В свойстве Protocol надо указать подходящий протокол, по которому можно подключиться к серверу (например, *TCP),* в свойстве ServerName — имя сервера, хранящееся в сетевых настройках текущего компьютера. Если используется операционная система *Windows9x,* а сервер подключен локально, то необходимо указать название локального сервера localhost. После этого значение свойства Active устанавливается равным True и происходит подключение к СУБД.

Реакция на щелчок может быть следующей.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
// если служба запущена
if IBLogService1.IsServiceRunning then
   begin
   // считываем протокол в переменную S,
   // которая может быть, например, типа TStrings
   while not IBLogService1.Eof do
      S.Add( IBLogService1.GetNextLine );
   // отсоединяем службу от сервера
   IBLogService1.Detach;
   end
// запуск службы
else IBLogService1.ServiceStart;
// обработка протокола
end;
```

Работа с SQL-серверами (панель dbExpress)

Общие положения

В связи с тем что система *Delphi 7* позволяет генерировать код не только для *Windows,* но и для *Linux, в* среду разработки потребовалось ввести компоненты, позволяющие работать с реляционными данными независимо от типа СУБД и операционной системы.

На панели dbExpress доступен набор компонентов, дающих возможность работать с произвольными *SQL*-серверами — серверами, способными обрабатывать реляционные таблицы баз данных с помощью операторов языка *SQL*.

В набор dbExpress входят драйверы для большинства популярных *SQL*-серверов. Эти драйверы представляют собой небольшую клиентскую прослойку, предоставляющую программисту единый стандартизованный интерфейс в рамках *Delphi 7* для доступа к различным СУБД. Такие драйверы реализованы как для *Windows*, так и для *Linux*.

Вместе с тем предоставление программистам универсальных возможностей накладывает определенные ограничения на функциональные характеристики компонентов панели dbExpress. Эти компоненты работают с так называемыми однонаправленными наборами данных. Они позволяют обращаться к *SQL*-серверам максимально быстро, практически не требуя дополнительных расходов на собственное выполнение. Но разработчики при этом могут применять в программах только однонаправленные курсоры, так как наборы данных, представляемые с помощью компонентов dbExpress, не хранятся в памяти и не имеют временных копий (точнее, не буферизуются).

Однонаправленные курсоры позволяют пробегать по набору данных лишь в одном направлении (только от начала к концу или от конца к началу), что в определенной степени снижает гибкость применения компонентов dbExpress. В частности, в отличие от компонента TDataSet для навигации по набору данных доступны только два метода Next и Prior, а возможности редактирования отдельных записей недоступны, так как это требует наличия буфера для временного хранения измененного значения. Поэтому свойство CanModify наборов данных панели dbExpress всегда имеет значение False.

Не допускается также использовать поля соответствия, которые требуют организации буферов в памяти, и фильтры. Такие ограничения снижают количество реально доступных компонентов, предназначенных для визуального отображения наборов данных на форме. Не удастся использовать компоненты, которые позволяют редактировать наборы данных и работать с их образом в памяти. Это прежде всего компонент Таблица данных.

Другие компоненты, например Поле редактирования данных, Флажок данных и т. п., допускается применять только в качестве объектов для отображения содержимого соответствующих полей таблиц базы данных, но не для изменения их значений. А компонент Навигатор позволит перемещаться по соответствующему набору данных только в одну сторону, несмотря на то, что пользователю доступны все кнопки: вперед, назад, к началу, к концу.

Компонент SQL-связь (TSQLConnection)

Данный компонент предназначен для установления непосредственной связи с *SQL*-сервером. Все остальные компоненты панели dbExpress работают на его основе.

Применение компонента TSQLConnection, как и большинства других компонентов *Delphi 7,* требует минимальных усилий по программированию — вполне можно обойтись визуальной настройкой свойств. Прежде всего надо выбрать подходящий драйвер в раскрывающемся списке свойства DriverName. Во время работы программы этот список можно получить динамически с помощью метода GetDriverNames. Когда соответствующий драйвер выбран (его название обычно совпадает с названием коммерческого *SQL*-сервера), значения свойств LibraryName (динамическая библиотека драйвера) и VendorLib (динамическая библиотека поддержки работы с СУБД на клиентской стороне) автоматически примут нужные значения. Библиотеки LibraryName и VendorLib поставляются **компанией** разработчиком соответствующей СУБД (*Linux*-драйверы для LibraryName также включены в *Kylix*).

Рассмотрим несложный пример. В стандартную поставку *InterBase* (она входит в комплект *Delphi 7)* включена демонстрационная версия базы данных employee.gdb. Организуем *с* ней связь и отобразим поля на форме с; помощью КОМПонентов dbExpress.

Разместим на новой форме компонент TSQLConnection и выберем в свойстве DriverName значение InterBase. При этом свойство LibraryName примет значение dbexpint.dll, а свойство VendorLib — значение GDS32.DLL.

ЗАМЕЧАНИЕ Для того чтобы донный пример работал, необходимо, чтобы сервер InterBase был запущен.

Следующий шаг — выбор базы данных, доступной для текущего *SQL*-сервера. Эта база выбирается в раскрывающемся списке ConnectionName (для *InterBase* по умолчанию — IBLocal). Конкретные параметры соединения указываются в свойстве Params — оно представляет собой таблицу строк (компонент TValueListEditor). В частности, для сервера *InterBase* можно указать имя пользователя и пароль,чтобы не вводить их при каждом новом подключении вручную, а также полный путь к базе данных.

Если имя и пароль заданы в настройках заранее, то значение свойства LoginPrompt желательно установить равным False, чтобы при запуске программы каждый раз не возникало окно с запросом пароля. С помощью свойства TableScope можно определить, какие типы таблиц доступны пользователю. По умолчанию считается, что пользователь может работать с обычными (несистемными) таблицами и запросами (подсвойстваtsTable и tsView). Теперь значение свойства Connected, устанавливающего связь с сервером, надо установить равным True, и если все параметры были заданы правильно, то *SQL*-сервер становится доступным из **разрабатываемой** программы (рис. 7.2).

В ходе работы с *SQL*-сервером часто возникает ситуация, когда одни наборы данных закрываются, а другие открываются. Закрытие всех наборов данных выполняется при отключении от самой базы данных, но этот процесс и процесс повторного к ней подключения достаточно длительный. Чтобы быстро закрыть все наборы данных без отключения от сервера, надо записать в свойство KeepConnection компонента TSQLConnection значение True, а затем вызвать метод

```
procedure CloseDataSets;
```

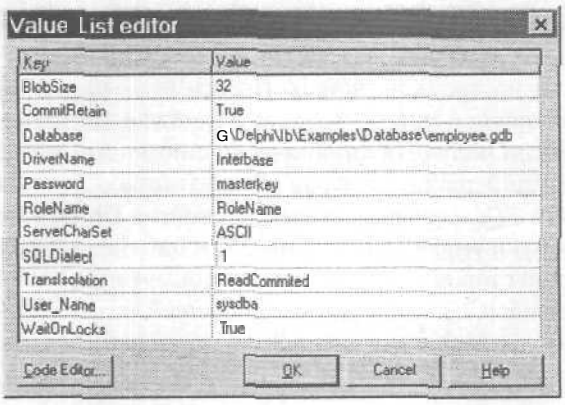


Рис. 7.2. Настройка SQL-соединения

Данный компонент предоставляет дополнительные возможности по работе с доступными наборами данных. Эти наборы представлены в виде массива, хранящегося в свойстве

```
DataSets[Index: Integer]: TCustomSQLDataSet;
```

Число *сто* элементов записано в свойстве

```
DataSetCount: Integer;
```

Важно отметить, что в массиве DataSets [ ] доступны только открытые наборы данных.

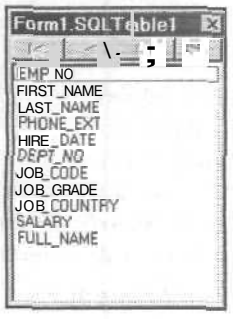
Для передачи серверу команды, например *SQL*-выражения для создания таблицы, служит метод

```
function Execute(const SQL: string; Params: TParams;
ResultSet:Pointer=nil): Integer;
```

Первый параметр хранит строку — *SQL*-выражение, последний — указатель на результирующий набор (если он формируется сервером).

Компонент SQL-Таблица (TSQLTable)

Для доступа к конкретной таблице базы данных используется этот компонент. **Он** напоминает компонент Таблица (TTable).



Прежде всего в свойстве SQLConnection в раскрывающемся списке выбирается доступный объект класса TSQLConnection, связывающий *SQL*-таблицуприложения с конкретной таблицей базы данных, а затем в свойстве TableName выбирается нужная таблица (EMPLOYEE).

Список доступных пользователю полей этой таблицы формируется с помощью стандартного редактора коллекций, вызываемого двойным щелчком на размещенном на форме объекте TSQLTable (вся работа с коллекцией полей ведется с помощью контекстного меню).

После этого свойство Active можно установить равным True, чтобы сделать таблицу активной.

Для связи таблицы с визуальными элементами служит стандартный компонент Источник данных (TDataSource). В его свойстве DataSet просто указывается соответствующий набор данных — в нашем случае таблица SQLTable1. Как уже говорилось, если связи с базой данных установлены с помощью компонентов dbExpress, использовать некоторые визуальные компоненты для отображения содержимого таблиц нельзя.

Добавим к форме два компонента, предназначенные для визуализации содержимого таблицы: Навигатор (TDBNavigator) и Надпись данных (TDBText). В каждом из них указывается источник данных (свойство DataSource), а в Надписи — дополнительно название соответствующего отображаемого поля данных (оно выбирается в раскрывающемся списке в свойстве DataField). Теперь приложение полностью работоспособно. Его можно запустить и пробежаться по всем записям таблицы, нажимая на кнопку Навигатора Next. Не следует забывать, что работа в данном случае происходит с однонаправленным набором данных, поэтому попытка переместиться не вперед, а назад (например, нажав кнопку Навигатора Prev) или немедленно к началу или концу набора данных (кнопки First/Last), приведет к ошибке,

Компонент SQL-Запрос (TSQLQuery)

Если возникает необходимость представления нескольких полей из различных таблиц, связанных друг с другом по ключевым полям, проще всего воспользоваться данным компонентом.

Связь с базой данных происходит, как и в случае SQL-Таблицы, с помощью свойства SQLConnection.

В свойстве CommandText указывается *SQL*-оператор, формирующий результирующий набор данных. Он может выглядеть, например, так:

```
select JOB_CODE from EMPLOYEE
```

(здесь название поля — JOB_CODE, а название таблицы — EMPLOYEE).

В остальном SQL-Запрос практически ничем не отличается от компонента SQL-Таблица. Он активизируется записью в свойстве Active значения True. Готовый запрос можно указать в свойстве DataSet объекта TDataSource.

Компонент SQL-набор данных (TSQLDataSet)

Компонент представляет собой обобщенный вариант компонентов TSQLTable и TSQLQuery. Он содержит несложный редактор *SQL-команд,* вызываемый при обращении к свойству CommandText (рис. 7.3).

В списке Tables выбирается таблица, а в списке Fields — нужные поля этой таблицы. В свойстве SortFieldNames дополнительно можно задать порядок сортировки значений выбранных полей (рис. 7.4).

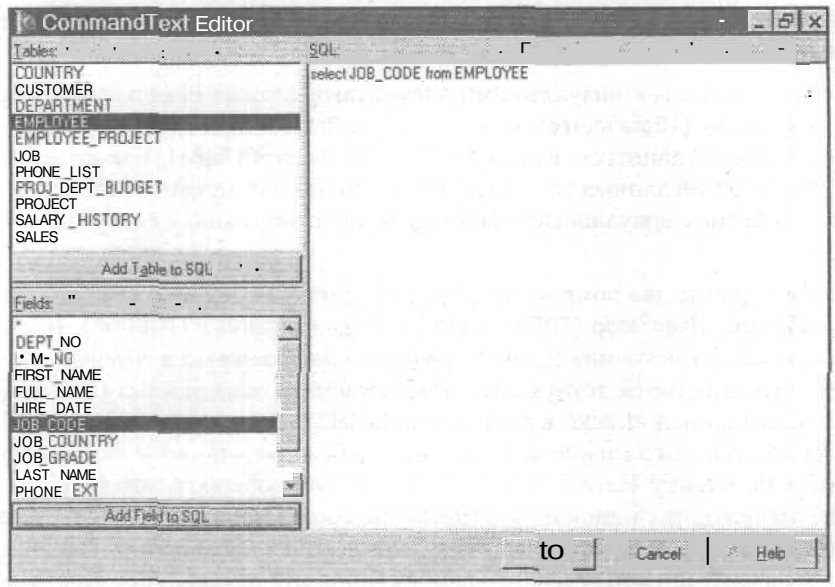


Рис. 7.3. *Редактирование SQL-команд*

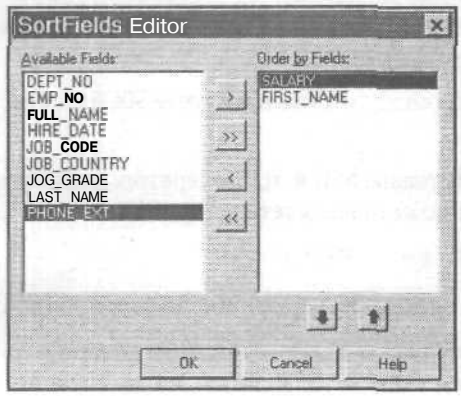


Рис. 7.4. *Определение порядка сортировки выбранных полей*

Для передачи параметров запросу или хранимой процедуре, которые вызываются через свойство CommandText, служит свойство

```
Params: TParams;
```

Удобнее всего настраивать это свойство на этапе проектирования с помощью визуального редактора, в трех полях которого надо определить: тип каждого параметра

(DataType), способ использования этого параметра (ParamType — служит ли он для принятия или передачи данных и т. д.) и значение параметра (Value).

Параметры можно настраивать и в ходе работы программы, в частности с помощью метода

```
function ParamByName(const Value: string) r TParam;
```

Например, параметру *SQL-запроса* UserName значение можно присвоить так:

```
SQLDataSet1.ParamByName('UserName').AsString := Edit1.Text;
```

Каждый из рассмотренных компонентов, предназначенных для доступа к таблицам, имеет достаточно важное свойство ObjeclView. Оно унаследовано от класса TDalaSet и позволяет обеспечить работу не только с реляционными, но и с объектными источниками информации, в которых записи организованы не в виде таблиц, а в виде иерархических структур. Если его значение равно True, то отдельные элементы свойства Fields могут представлять не конкретное поле, а ссылку на массив вложенных или нижестоящих в иерархии полей. Если это значение равно False, то вложенные поля идут вслед за своим родителем. Некоторые драйверы не поддерживают возможность представления данных в иерархическом виде. Например, для набора данных TBDEDataSel свойство ObjectView всегда равно False.

Компонент Хранимая SQL-процедура (TSQLStoredProc)

Данный компонент позволяет выполнять процедуры, хранимые на *SQL-сервере*. Он значительно проще, чем ранее рассмотренный компонент TStoredProc. Все, что требуется для его активизации, — задать способ связи с сервером с помощью компонента TSQLConnection и выбрать в раскрывающемся списке свойства StoredProcName название подходящей хранимой процедуры.

В дальнейшем TSQLStoredProc можно использовать как поставщика информации для набора данных, так как он является наследником класса TDataSet.

Компонент Простой набор данных (TSimpleDataSet)

Вышеприведенные компоненты панели dbExpress, как уже говорилось, обеспечивают только основную и довольно ограниченную функциональность при работе с наборами данных, так как не создают их временных образов в памяти и не буферизуют информацию. Данную проблему в определенном смысле решает компонент TSimpleDataSet. Он функционирует как TSQLDataSet, предоставляя быстрый доступ к *SQL-серверам*, но в то же время работает и как TDataSet, кэшируя информацию в памяти. В этом компоненте отсутствует свойство ProviderName, характерное для компонента TClientDataSet. Так как TSimpleDataSet имеет собственный

буфер памяти для редактирования данных, то указывать внешнего провайдера, выполняющего обновление изменений, не требуется.

Компонент TSimpleDataSet поддерживает все основные функции и возможности класса TDataSet, так как является его наследником. Это, в частности, возможность агрегирования данных, создание автоматически вычисляемых полей, фильтров и другие функции.

ЗАМЕЧАНИЕ Компонент TSimpleDataSet заменил в системе Delphi 7 компонент TSQLClientDataSet

Компонент SQL-монитор (TSQLMonitor)

В интенсивно работающих приложениях важно знать, какой из модулей системы наиболее активно обращается к серверу и какие запросы и команды требуют наибольшего расхода времени и загрузки процессора. В распределенной системе отслеживать такие вещи достаточно сложно, поэтому в *Delphi 7* для этих целей создан компонент TSQLMonitor. Он сохраняет протокол работы *SQL-соединения* в обычном текстовом файле.

После того как данный компонент размешен на форме, его надо связать с объектом класса TSQLConnection, который указывается в свойстве SQLConnection.

Протокол хранится в списке строк TraceList, а если значение свойства AutoSave равно True, то протокол автоматически записывается в файл, название которого и полный путь заданы в свойстве FileName.

Если же свойство AutoSave имеет значение False, то во время работы программы сохранить всю информацию из свойства TraceList в заданный файл можно с помощью метода

```
procedure SaveToFile(AFileName: string);
```

Текст протокола может выглядеть примерно так:

```
INTERBASE - isc_dsql_allocate_statement
SELECT 0, '', A.RDB$OWNER_NAME, A.RDB$RELATION_NAME,
A.RDB$VIEW_SOURCE,
A.RDB$SYSTEM_FLAG FROM RDB$RELATIONS A WHERE
(A.RDB$SYSTEM_FLAG <> 1 OR
A.RDB$SYSTEM_FLAG IS NULL) ORDER BY A.RDB$RELATION_NAME
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
```

```
INTERBASE - isc_commit_retaining
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_allocate_statement
select COUNTRY from COUNTRY
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
INTERBASE - isc_commit_retaining
INTERBASE - isc_dsql_free_statement
```

Обработка транзакций в модели dbExpress

Обработка транзакций выполняется разными *SQL*-серверами по-разному. Например, СУБД *MySQL* вообще не поддерживает транзакции. В компоненте TSQLConnection имеется несколько базовых механизмов поддержки транзакционной модели. При их использовании надо помнить, что другие компоненты системы (различные провайдеры данных или клиентские наборы) имеют собственные средства поддержки транзакций, поэтому применять TSQLConnection следует аккуратно, не нарушая логики работы других объектов.

Транзакция запускается методом

```
procedure StartTransaction(TransDesc: TTransactionDesc);
```

компонента TSQLConnection. В качестве параметра указывается идентификатор данной транзакции, чтобы в приложении можно было одновременно выполнять несколько транзакций.

```
type TTransactionDesc = packed record
      TransactionID    : LongWord;
      GlobalID         : LongWord;
      IsolationLevel   : TTransIsolationLevel;
      CustomIsolation  : LongWord;
    end;
```

Важнейшим элементом в этой структуре является параметр IsolationLevel, определяющий, как данная транзакция изолируется от других транзакций приложения. Он имеет тип TTransIsolationLevel:

```
type TTransIsolationLevel = (xilDIRTYREAD,
xilREADCOMMITTED, xilREPEATABLEREAD, xilCUSTOM);
```

Таблица 7.21. Значения параметра IsolationLevel

Значение	Вид изоляции
xilDIRTYREAD	Из текущей транзакции видны все изменения, выполняемые в наборах данных другими транзакциями, даже если они не завершены. Данная возможность недоступна при использовании СУБД Oracle
xilREADCOMMITTED	Из текущей транзакции видны только те изменения, которые выполнены завершенными транзакциями. При этом возможно, что из-за работы незавершенных транзакций информация о наборах данных будет противоречивой
xilREPEATABLEREAD	Гарантируется, что данные, доступные в ходе работы текущей транзакции, непротиворечивы — учитывается состояние наборов данных до запуска текущей транзакции. При этом есть вероятность, что к моменту завершения транзакции эти данные все равно будут изменены. Хотя в ходе выполнения транзакции ошибок не возникнет, они могут появиться при попытке физического обновления базы данных в момент завершения транзакции
xilCUSTOM	Текущая транзакция следит за наборами данных собственными методами, заданными разработчиком, и использует поле CustomIsolation (пока такая возможность драйверами dbExpress не поддерживается)

Поле GlobalID требуется при работе с СУБД *Oracle*.

Свойство InTransaction компонента TSQLConnection принимает значение True, если транзакция успешно запущена. Такой запуск может быть неудачным в случаях, когда, например, СУБД не поддерживает множественные транзакции и в приложении уже имеются незавершенные транзакции.

Для завершения транзакции вызывается метод

```
procedure Commit(TransDesc: TTransactionDesc);
```

с тем же параметром. В случае, если в ходе транзакции выявлены ошибки, например произошел сбой и сформированы неверные данные, то откатить эту транзакцию обратно (отменить все выполненные в ее ходе действия над набором данных) можно с помощью метода

```
procedure Rollback(TransDesc: TTransactionDesc);
```

получающего все тот же единственный параметр.

Схематичный пример выполнения транзакции с помощью TSQLConnection

```
procedure TForm1.TransactionButtonClick(Sender: TObject);
var
```

```
    Amt: Integer;
    TD: TTransactionDesc;
begin
    // проверяем возможность запуска транзакции
    if not SQLConnection1.InTransaction then
        begin
        TD.TransactionID := 1;
        TD.IsolationLevel := xilREPEATABLEREAD;
        SQLConnection1.StartTransaction(TD);

        try

            // выполняем два SQL-запроса с помощью объектов
            // MyDataSet и UserDataSet (тип TSQLDataSet) -
            // обновляем определенные таблицы и
            // переводим 100 у.е. с одного счета на другой

            Amt := 100,-
            MyDataSet.Params.ParamValues['Money'] := Amt;

            Amt := UserDataSet.Params.ParamValues['Money']
              - 100;
            UserDataSet.Params.ParamValues['Money'] := Amt;

            MyDataSet.ExecSQL;
            UserDataSet.ExecSQL;

            // пытаемся завершить транзакцию
            SQLConnection1.Commit(TD);
        // исключительная ситуация возникает,
        // если не удалось выполнить SQL-команду
        except

            // восстанавливаем состояние
            // наборов данных MyDataSet и UserDataSet,
```

```
                // существовавших к началу транзакции

                SQLConnection1.Rollback(TD);

            end;
        end;
    end;
```

Что нового мы узнали?

В этом уроке мы научились

И взаимодействовать с клиент-серверными СУБД;

0 обрабатывать данные, полученные от сервера СУБД;

0 программно управлять работой сервера СУБД.

урок 8 Взаимодействие приложений

- Принципы обмена информацией между приложениями Windows
- Динамически подключаемые библиотеки
- Работа с потоками
- Использование объектов COM
- Создание системы COM на базе транзакционного сервера MTS
- Создание распределенных приложений на основе технологии CORBA

Вступление

Б последующих главах акцент предлагаемых читателю примеров несколько смещается. От вопросов детального ручного программирования мы перейдем к вопросам визуальной разработки приложений. Такой подход требует минимальных усилий по написанию программного кода и позволяет полностью сосредоточиться на реализации логики создаваемого приложения.

Зачем это делается? Ниже рассматриваются вопросы, которые могли бы потребовать от программиста специальных знаний и изучения справочной информации. Это необходимо при создании распределенных приложений *CORBA*, *DCOM* или Интернета, для работы с базами данных на основе примеров, поставляемых в составе стандартных библиотек разработчика *SDK* (*Software Development Kit)*. Такие примеры обычно написаны на языке программирования *C++* (Си++), который поддерживается такими популярными продуктами, как *Microsoft Visual C++* и *Borland C++ Builder*.

Однако эти примеры, как правило, выполнены в низкоуровневом стиле, раскрывающем детали реализации конкретной технологии. Они занимают тысячи строк кода, что не позволяет разобраться в них прикладному программисту за разумное время. Если на базе *SDK* требуется создать большое и сложное приложение, обеспечивающее эффективную работу с базами данных и серверами приложений, то 90% усилий программистов уйдет не на реализацию технического задания, а на выявление ошибок среди огромного объема запутанного кода, реализующего рутинные аспекты различных технологий.

Система *Delphi 7* обладает уникальными возможностями по автоматизации вспомогательных процессов. В большинстве рассматриваемых примеров потребуется собственноручно написать лить несколько десятков вспомогательных, уточняющих операторов. Быстро выполнив всю работу по созданию заготовки приложения неограниченной сложности в визуальной среде, можно сосредоточиться на реализации нужных функций. Поэтому мы рассмотрим, прежде всего, визуальные средства генерации шаблонов приложений для работы с распределенными базами данных, с объектами, распределенными в локальной сети или Интернете. Благодаря таким возможностям системы *Delphi 7* специфические нюансы конкретных технологий будут скрыты, что позволит легко использовать эти технологии на уровне визуальной настройки и комбинирования компонентов.

Принципы обмена информацией между приложениями Windows

Совместная работа нескольких приложений

Система *Windows* исходно задумывалась как многозадачная. Это означает, что в ней одновременно могут работать несколько задач. Это действительно так. Напри-

мер, на ПК с процессором *Pentium/233* и достаточным объемом ОЗУ (64 Мбайт) можно запустить интенсивную вычислительную задачу и одновременно продолжать работу в редакторе *Word* или электронной таблице *Excel,* ожидая результаты расчетов, выполняющихся в фоновом режиме. А в системе *Windows NT* такие возможности значительно расширены и к ним добавлены специальные средства управления выполняющимися программами.

Однако запускать несколько задач одновременно не всегда имеет смысл. Чаще всего требуется, чтобы такие задачи взаимодействовали между собой, обменивались информацией, причем на основе стандартного механизма, не требующего от программиста дополнительных усилий по определению способа такого взаимодействия.

Программы, обменивающиеся информацией, как правило, не равноправны. Одна из них выступает в роли *сервера,* рассылающего обработанную информацию (по аналогии с сервером баз данных), другая (или несколько других) — в роли *клиента* (по аналогии с клиентской программой, получающей наборы данных по запросу к СУБД). Однако в большинстве случаев приложения *Windows* могут работать и как серверы, и как клиенты.

Технология DDE

В *Windows* имеется несколько технологий, позволяющих организовать эффективное взаимодействие группы приложений. Еще в первых 16-разрядных версиях *Windows* была реализована технология *DDE* (*Dynamic Data Exchange, динамический обмен данными)*. С ее помощью программа-сервер может обмениваться информацией (текстовыми строками) с программами-клиентами, подключенными к этому серверу с помощью протокола *DDE*. Абсолютное большинство приложений корпорации *Microsoft* поддерживают этот протокол, что позволяет обращаться к ним из других программ, передавать и получать информацию.

В системе *Delphi 7* для поддержки технологии *DDE* созданы четыре компонента, расположенные на панели System (Системные).

Таблица 8.1. *Компоненты панели System*

Компонент	Кнопка	Назначение
TDdeClientConv		Устанавливает связь с серверам DDE
TDdeClientItem		Конкретный объект, содержимое которого передается (или принимается) за один сеанс связи с сервером DDE. Обеспечивает транзакционную работу по обмену информацией
TDdeServerConv		Сервер DDE
TDdeServerItem		Объект стороны сервера, содержимое которого предназначено для обмена

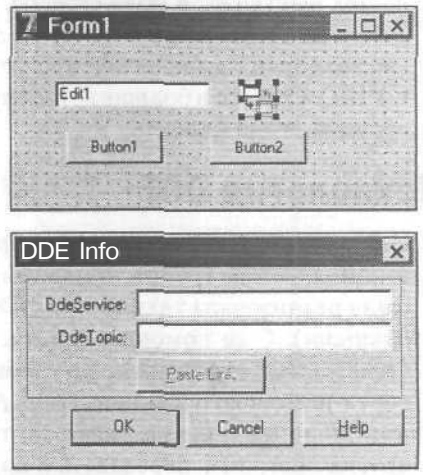
Компонент TDdeClientConv обладает двумя важнейшими свойствами, лежащими в основе протокола *DDE*. Свойство DdeService описывает серверное приложение, с которым устанавливается связь, а свойство DdeTopic — так называемый «топик»

(*topic*), характеризует имя серверного объекта, непосредственно участвующего в обмене информацией.

С помощью компонентов TDdeServerConv/TDdeServerItem можно включить в свою программу богатые возможности обмена данными, хотя для этих целей компания *Microsoft* предлагает значительно более мощную и гибкую технологию *OLE*. Тем не менее поддержка механизма *DDE* в продуктах корпорации сохраняется, и в некоторых случаях использовать этот протокол обмена значительно проще, чем все остальные.

Сервером *DDE* является, например, электронная таблица *Excel*. Написав несколько строк кода, можно организовать простой обмен содержимым между программой и ячейками страниц Excel 97.

Разместим на форме текстовое поле, две кнопки (Послать и Принять) — и компонент TDdeClientConv.



Основная задача при использовании компонентов *DDE* — установить связь между сервером и клиентом и, вообще, выяснить, поддерживает ли некоторое приложение этот протокол. Проще всего сделать это так. Запустите программу-сервер (в нашем случае — *Excel 97),* выберите произвольную ячейку и скопируйте ее содержимое в буфер обмена *Windows*. Затем перейдите к Инспектору объектов и для любого из свойств DdeService/DdeTopic вызовите редактор связи с сервером DDEInfo, щелкнув на кнопке вызова в соответствующей строке Инспектора.

Если программа, данные из которой были скопированы в буфер, способна работать как сервер *DDE*, то кнопка Paste Link (Установить связь) доступна. В противном случае она окажется в сером «отключенном» состоянии.

После щелчка на этой кнопке в поля DdeService/DdeTopic будут внесены названия сервера (*Excel*) и объекта — первого листа таблицы [Книга1]Лист1, если в программе *Excel* был открыт новый файл. Теперь надо подтвердить связь щелчком на кнопке OK.

Обмен данными может происходить в обе стороны: как в сторону электронной таблицы, так и от нее к клиентскому приложению. Это приложение управляет работой сервера, отдавая ему соответствующие команды.

Чтобы послать данные серверной программе, используется следующий метод.

```
function PokeData(Item: string; Data: PChar): Boolean;
```

Первый параметр описывает принимающий объект стороны сервера (этот объект обычно выделен или имеет фокус). Для некоторых программ можно указать принимающий элемент явно. В частности, для программы *Excel* можно указать номер ячейки в формате «RnCm». Здесь где n — номер строки, начиная с 1, а m — номер столбца, начиная с 1. Например, ячейка с координатами (5,12) обозначается как R5C12.

Второй параметр — передаваемая строка в формате PChar.

Сделаем так, чтобы по щелчку на кнопке Послать с помощью этого метода в ячейку электронной таблицы с координатами (2,3) (программа Excel должна быть запущена заранее) записывалось содержимое поля Edit1 нашей программы (рис. 8.1):

```
procedure TForm1.Button1Click(Sender: TObject);
var ToExcel: array[0..100] of char;
begin
StrPCopy(ToExcel, Edit1.Text);
DDEClientConv1.PokeData('R2C3', ToExcel),-
end;
```

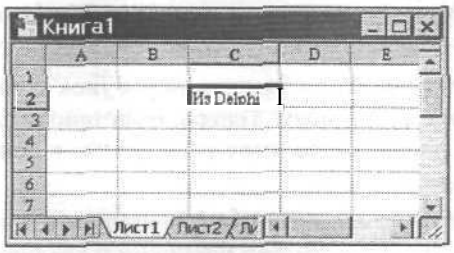


Рис. 8.1. Содержимое ячейки электронной таблицы передано из внешнего приложения

Чтобы принять информацию от программы-сервера, надо воспользоваться следующим методом.

```
function RequestData(const Item: string): PChar;
```

Единственный параметр — объект сервера, содержащий принимаемые данные (в нашем случае — ячейка таблицы).

Функция возвращает в формате PChar указатель на строку, которая хранит данные, полученные от сервера.

Для отображения в поле Editl нашей программы содержимого ячейки (1,1) надо сформировать следующий обработчик щелчка на кнопке Принять.

```
procedure TForm1.Button2Click(Sender: TObject);
var FromExcel: PChar;
begin
FromExcel := DDEClientConv1.RequestData('R1C1');
Edit1.Text := StrPas(FromExcel);
end;
```

Как видим, использование технологии *DDE* не вызывает особых сложностей. С помощью компонентов TDdeServerConv и TDdeServerItem можно расширить возможности своего приложения, чтобы оно также было способно работать как сервер *DDE*.

К недостаткам технологии *DDE* надо отнести отсутствие единого подхода к работе с различными серверами *DDE*. Например, чтобы узнать, как обращаться к ячейкам электронной таблицы *Excel*, надо основательно покопаться в документации. То же относится и ко многим другим стандартным программам.

Технология OLE

Технология *Object Linking and Embedding* (*Связывание и внедрение объектов*) обладает значительно большими возможностями, нежели механизм *DDE*. Типичный пример использования этой технологии — добавление рисунка в документ текстового процессора *Word* (Вставка > Рисунок > Из файла). Это действие можно выполнить двумя способами.

1. В ходе выбора рисунка в диалоговом окне устанавливается флажок Связать с файлом. В этом случае в документ физически добавляется только ссылка на рисунок (место его хранения и используемый графический формат). При внесении изменений в исходный файл с рисунком эти изменения отображаются в документе, а если этот файл удалить, то программа *Word* не сможет ничего отобразить. Такой способ объединения двух объектов называется *связыванием*.
2. Если флажок Связать с файлом сброшен, то рисунок копируется из исходного файла и вставляется в документ. Теперь, если оригинальный файл изменен или удален, копия рисунка останется в документе в неизменном виде. Такой подход называется *внедрением*.

В случае связывания объем документа *Word* увеличивается незначительно, но требуется контролировать наличие файла с рисунком и следить за его изменениями. В случае внедрения существенно увеличивается объем документа, так как изображение хранится внутри него.

Объект, который встраивается в документ или связывается с ним, называется *контейнером OLE* (OLE Container). В нашем случае это рисунок. В системе *Delphi 7* на панели System (Системные) имеется компонент TOleContainer, позволяющий встраивать объекты в программу или связывать их с ней.

Технология *OLE* обладает и множеством других возможностей. В частности, это *автоматизация OLE* (OLE Automation), позволяющая программно управлять другими приложениями, вызывая их методы, доступные через интерфейс *OLE*. При этом, конечно, требуется, чтобы приложение поддерживало автоматизацию *OLE*.

Технология *OLE* расширяет технологию *DDE*. Если ранее можно было только обмениваться данными и понятия «объект» не существовало, то в рамках технологии *OLE* программист работает с программами и данными как с полноценными объектами, обладающими не только свойствами, но и методами, к которым можно обращаться. Практически все офисные приложения *Windows* поддерживают автоматизацию *OLE*. Например, к таковым относится браузер *Internet Explorer*, работой которого несложно управлять программно.

Основные недостатки при использовании технологии *OLE* — необходимость знать (как и в случае *DDE*) описание доступных свойств и методов обрабатываемых объектов, которые в браузере и редакторе отличаются, и отсутствие общего программного интерфейса.

Эта проблема была полностью снята в технологии *COM (Component Object Model*, объектная модель компонента), основанной на технологии *OLE*. Технология *COM* подробно рассматривается в последующих главах.

Однако вернемся к объектам *OLE* и посмотрим, как можно использовать компонент TOleContainer. После размещения компонента на форме в его контекстном меню доступны, в частности, два пункта: Insert Object (Вставить объект) и Paste Special (Специальная вставка). С помощью первого пункта определяется объект *OLE,* который размещается на форме (рис. 8.2).

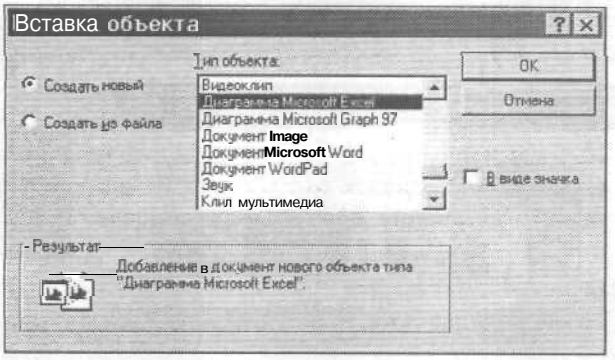


***Рис**. 8.2. Стандартное диалоговое окно для вставки объектов OLE*

В списке указываются все типы объектов *OLE,* которые зарегистрированы в конкретной версии *Windows.* Например, если выбрать пункт Диаграмма Microsoft Excel (при условии, что в системе установлена электронная таблица *Excel),* то в окне компонента отображается некая начальная диаграмма (рис. 8.3).

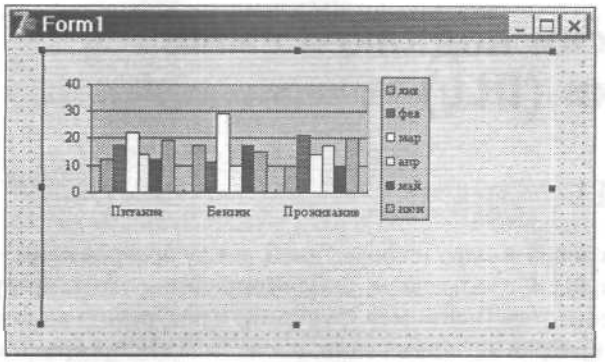


Рис. 8.3. Диаграмма Excel как объект OLE на форме

Если дважды щелкнуть на ней, то в рамках границ компонента TOleContainer запускается программа *Excel. С* ее помощью можно выполнить редактирование этой диаграммы (рис. 8.4).

Можно также вставить готовый объект из файла. Тогда в диалоговом окне вставки объекта надо установить переключатель Создать из файла и указать нужный файл. Флажок Связь определяет режим добавления документа в программу (связывание или внедрение).

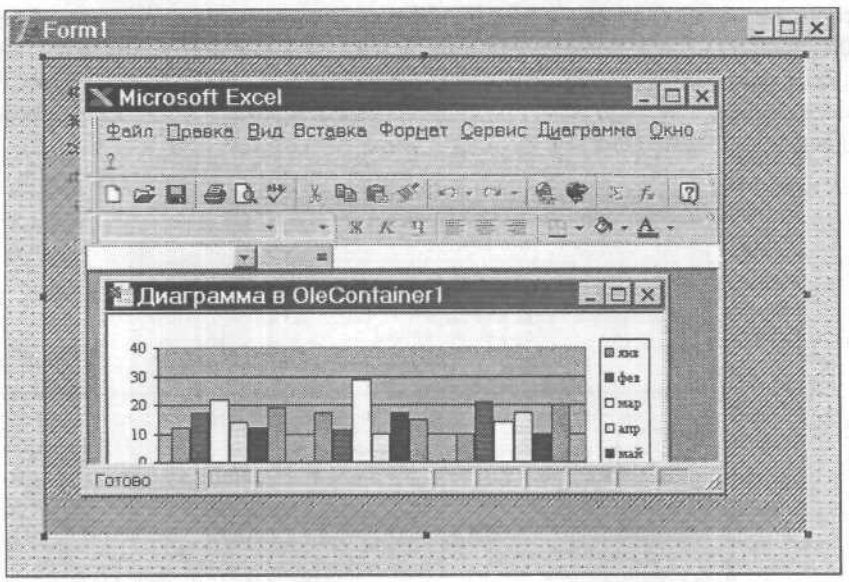


Рис. S. 4. Редактирование диаграммы в рамках формы средствами сервера OLE

Динамически подключаемые библиотеки (DLL)

Что такое DLL

Самый простой способ обмена информацией между программами — это использование библиотек .DLL. Строго говоря, библиотека .DLL — это не программа, а хранилище программного кода (например, функций) и ресурсов (например, форм). Она подключается к программе *динамически,* после того как программа запущена.

Использовать библиотеки .DLL очень удобно. Пусть, например, имеется программа прогнозирования некоего значения на основании данных, вводимых пользователем. Создадим пользовательский интерфейс, а сам алгоритм расчета прогноза (достаточно сложный) выделим в отдельную функцию и поместим в библиотеку .DLL, подключаемую к программе.

Теперь, если разработчик улучшит алгоритм прогнозирования, то пользователям достаточно поменять только библиотеку .DLL, а вносить изменения в код, ответственный за интерфейс, вообще не придется. По такому принципу, в частности, устроен браузер *Netscape Navigator.*

Создание библиотеки DLL

Допустим, требуется создать библиотеку, в которой будет находиться функция суммирования двух чисел.

```
function Sum(X,Y: integer): integer;
```

Выполним команду File ➤ New > Other ➤ DLL Wizard (Файл ➤ Создать > Другое ➤ Мастер DLL). При этом создается пустая заготовка библиотеки. Текст модуля начинается не со слова **unit,** а со слова **library**. Добавим в нее описание функции Sum, а в конец модуля библиотеки — ключевое слово exports, после которого приводится список *экспортируемых* функций — функций данной библиотеки, которые доступны другим приложениям.

ВНИМАНИЕ Подпрограммы, названия которых не указаны в директиве exports, не могут вызываться из других программ и используются только внутри программы.

Целиком модуль запишется следующим образом.

```
library Project1;

uses
  SysUtils,
  Classes,-

{SR *.RES}

function Sum(X,Y: integer): integer;
begin
Result := X+Y
end;

exports Sum;

begin
end.
```

Вызов библиотеки DLL

Добавим в текущую группу проектов новый проект (обычное приложение), разместим на форме два текстовых поля, кнопку и одну надпись для вывода результата. (рис, 8.5).

В разделе interface опишем импортируемую функцию с ключевым словом external, за которым указывается строка — имя файла (библиотеки .DLL).

```
function Sum(X,Y: integer): integer;
external  'Project1.dll';
```

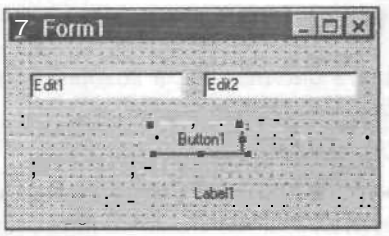


Рис. 8.5. Форма для приложения, которое будет обращаться к внешней библиотеке DLL

Обработчик щелчка на кнопке запишется следующим образом.

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Label1.Caption :=
IntToStr(Sum(StrToInt(Edit1.Text),StrToInt(Edit2.Text)))
end;
```

Теперь можно запустить приложение (предварительно надо откомпилировать библиотеку Project1.dll). После щелчка на кнопке выполняется суммирование введенных чисел с помощью функции, программный код которой хранится в динамической библиотеке (рис. 8.6).

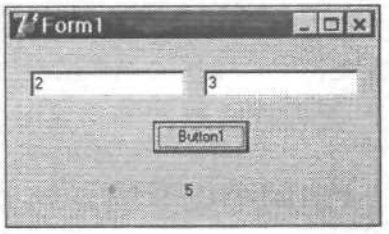


Рис. 8.6. Для расчета суммы вызывается функция, хранящаяся во внешней библиотеке

Добавление ресурсов в библиотеку

В библиотеке .DLL можно хранить не только программный код, но и ресурсы, в частности, формы *Delphi 7,* что значительно повышает полезность таких библиотек. Сделаем активным проект Project1.dll и выполним команду File ➤ New ➤ Form (Файл > Создать ➤ Форма). В текущем проекте появится новая форма (рис. 8.7). Добавим на нее одну кнопку, а обработчик нажатия сделаем таким.

```
procedure TForm2.Button1Click(Sender: TObject);
begin
ShowMessage('DLL!')
end;
```

Чтобы вызвать форму из другого приложения, ее надо создать динамически, так как обычные механизмы *Delphi 7* в библиотеках .DLL не работают. Для этого опишем новую процедуру.

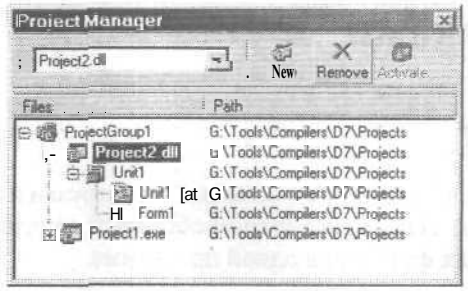


Рис. *8.7. Добавление формы, хранимой как ресурс в библиотеке DLL*

```
procedure ShowMyForm(AOwner: TComponent);
var MyForm: TForm2;
begin
MyForm := TForm2.Create(AOwner);
MyForm.ShowModal;
MyForm. Free;
end;
```

Эта процедура должна иметь параметр типа TComponent, необходимый для конструктора формы. Включим созданную процедуру в список экспорта.

```
exports Sum, ShowMyForm;
```

Далее переключимся на проект Project2.exe и укажем новую импортируемую процедуру.

```
procedure ShowMyForm(AOwner: TComponent);
external 'Project1.dll';
```

Добавим на форму еще одну кнопку и создадим такой обработчик щелчка.

```
procedure TForm1.Button2Click(Sender: TObject);
begin
ShowMyForm(Self);
end;
```

По щелчку на этой кнопке появится работающая форма, взятая из библиотеки .DLL (рис. 8.8).

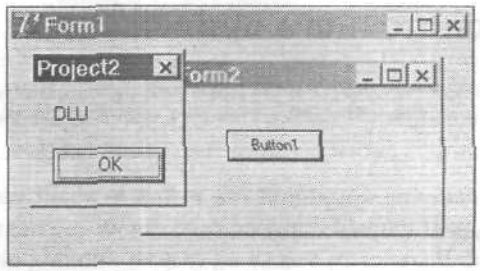


Рис. *8.8. Открытие формы, которая хранится в библиотеке DLL*

Работа с потоками

Что такое поток

В предыдущей главе была рассмотрена возможность загрузки и выполнения функций из внешних библиотек. Теперь рассмотрим обратный случай — «одновременное» выполнение нескольких функций в одной программе.

Слово «одновременное» не зря взято в кавычки. В реальности, параллельно выполнять несколько программ можно, очевидно, только при наличии у компьютера двух и более процессоров. Однако система *Windows* позволяет имитировать такую одновременную работу: каждой программе операционная система выделяет небольшой квант времени (несколько миллисекунд), в течение которого выполняется данная программа, а затем происходит переключение к следующей программе. Так как квант времени очень мал, для человека такое мгновенное переключение незаметно. В результате создается иллюзия одновременной работы нескольких программ.

Псевдопараллельно могут выполняться не только обычные программы, но и так называемые *потоки* — код, который входит в состав программы, пользуется ее ресурсами и выполняется в ее адресном пространстве. Строго говоря, любая программа — это всегда один главный поток и ноль или более вспомогательных.

Использовать потоки удобно для самых разных целей. Они дают возможность выполнять задачи в фоновом режиме и позволяют пользователю сосредоточиться на главной работе. Например, бухгалтер может вводить хозяйственные проводки, а в фоновом режиме в это время выполняется расчет баланса.

С появлением недорогих многопроцессорных персональных компьютеров технологии параллельных вычислений будут, без сомнения, все активнее совершенствоваться, поэтому знакомство с ними надо начинать уже сегодня.

ЗАМЕЧАНИЕ Корпорация Borland не рекомендует создавать в одной программе более 16 процессов, если эта программа работает но однопроцессорном компьютере.

Создание многопоточного приложения

Далее мы рассмотрим пример, в котором в рамках программы, содержащей формы с двумя областями рисования (TPaintBox), запускаются два потока, каждый из которых выполняет рисование в своей области и со своей скоростью.

Создадим новое приложение командой File ➤ New ➤ Application (Файл ➤ Создать > Приложение). Подготовим форму с двумя областями рисования TPaintBox размером 100x100 пикселов. Разместим также на форме кнопку, по щелчку на которой запускаются потоки (рис. 8.9).

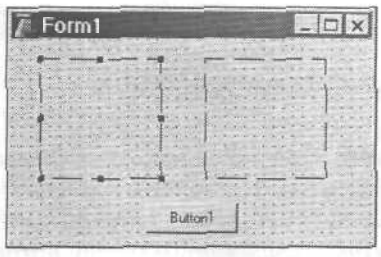


***Рис. 8.9.** Форма, подготовленная для двухпоточного приложения*

Добавим в проект класс-поток командой File ➤ New ➤ Other > Thread Object (Файл ➤ Создать ➤ Другое > Потоковый объект). В диалоговом окне ввода названия класса укажем TMyThread. Этот класс будет наследником базового класса-потока TThread. Возникнет файл с пустым описанием этого класса.

Перейдем к модулю главной формы и опишем в классе TForm1 новый метод CPaint, который получит в качестве параметра холст и нарисует на нем квадрат размером 10x10 пикселов. Место расположения квадрата и его цвет выбираются случайным образом.

```
procedure TForml.CPaint( Cv: TCanvas ),-
var b: byte;
    c: TColor;
    r: TRect;
begin
Cv.Brush.Style := bsSolid;
c := random(256);
b := random(256);
c := c or (b shl 8);
b := random(256);
c := c or (b shl 16);
Cv.Brush.Color := c;
r.left := random(90);
r.top  := random(90);
r.right := r.left+10;
r.bottom := r.top+10;
Cv.Rectangle(r);
end;
```

На деталях реализации этой процедуры останавливаться не станем.

Добавим еще два метода (процедуры без параметров), которые будут обращаться к процедуре CPaint для вывода квадрата либо на холст объекта PaintBox1, либо на холст объекта PaintBox2.

```
procedure TForm1.Paint1;
begin
CPaint( PaintBox1.Canvas );
end;
```

продолжение

```
procedure TForm1.Paint2;
begin
CPaint( PaintBox2.Canvas );
end;
```

Эти процедуры необходимы, чтобы вызывать их из класса-потока, причем специальным образом. Дело в том, что, когда методы классов *Delphi 7* (в нашем случае — метод Rectangle) одновременно вызываются из нескольких потоков, это может приводить к самым неожиданным конфликтам между потоками. Например, проблемы могут быть связаны со слишком быстрым чередованием обработки процесса вывода на экран, когда системе требуется обрабатывать объекты классов TBrush, TPen. Эти классы давно известны как источники ошибок, вызванных неаккуратным обращением с ними. Поэтому в классе TThread имеется метод гарантированно безопасного выполнения таких методов. Этот метод выполняет синхронизацию работы всех потоков.

```
type TThreadMethod = procedure of object;
procedure Synchronize(Method: TThreadMethod) ,-
```

В качестве параметра он получает название метода без параметров. Для этого нам и понадобилось создать процедуры Paintl и Paint2. А как определить внутри потока, какой из этих методов надо вызывать? Добавим в класс TMyThread переменную Box1.

```
...
public
   Boxl: boolean;
   end;
```

При создании экземпляра потока занесем в эту переменную значение True, если надо вызывать метод Paintl. Вся логика работы потока описывается *в* его основном методе.

```
procedure Execute;
```

Нам надо переопределить этот метод следующим образом.

```
procedure TMyThread.Execute;
begin
while not Terminated do
  if Boxl then Synchronize(Form1.Paintl)
          else Synchronize(Form1.Paint2)
end;
```

Свойство класса Terminated автоматически примет значение True, когда программа получит команду на завершение. В зависимости от значения переменной Boxl выполнится отрисовка одного квадрата (вызов соответствующего метода класса TForml), после чего продолжится цикл произвольной длительности.

Теперь надо добавить в описание класса TForm 1 две новые переменные — два будущих потока.

```
T1, T2: TMyThread;
```

Опишем обработчик щелчка на кнопке Button1. .

```
procedure TForm1.Button1Click(Sender: TObject);
begin
Tl := TMyThread.Create(true);
T1.Box1 := true;
T1.Priority := tpLower;

T2 := TMyThread.Create(true);
T2.Box1 := false;
T2.Priority := tpLowest;

T1.Resume;
T2.Resume;
end;
```

Поток создается с помощью конструктора Create, который имеет один параметр. Он имеет значение True, если поток начинает работу после вызова метода Resume, В противном случае (False) поток начнет работу сразу же после создания.

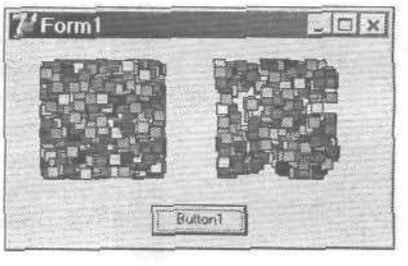
Далее задаются значения переменной Boxl и свойства Priority — приоритета выполнения потока. Приоритет определяет, как часто *Windows* выделяет процессу кванты времени, то есть фактически задает быстродействие этого процесса.

Свойство Priority имеет тип TThreadPriority и может принимать одно из следующих значений.

***Таблица** 8.2. Значения свойства Priority*

Значение	Приоритет потока
tpIdle	Поток выполняется, только когда системе Windows больше нечего делать
tpLowest	Приоритет на два пункта ниже нормального
tpLower	Приоритет на один пункт ниже нормального
tpNormal	Нормальный приоритет
tpHigher	Приоритет на один пункт выше нормального
tpHighest	Приоритет на два пункта выше нормального
tpTimeCritical	Максимальный приоритет

В раздел interface модуля Unit1 нужно добавить ссылку на модуль Unit2, где описан класс TMyThread, а в раздел implementation модуля Unit2 — ссылку на модуль Unitl. Программу можно откомпилировать и запустить. После щелчка на кнопке начнется заполнение двух областей рисования небольшими прямоугольниками, причем в правой области это будет происходить заметно медленнее.



Использование объектов COM

Что такое технология COM

Технология *COM* описывает модель объекта и способы взаимодействия таких объектов и программ. Объект *COM* напоминает компонент *Delphi 7.* Он представляет собой законченный объект со своими свойствами и методами, который может легко встраиваться в приложения или распространяться как отдельный программный продукт. Здесь надо сделать следующее замечание. *COM*-объект представляет собой либо *DLL-библиотеку,* либо *EXE-приложение Windows,* которые можно создавать в любой системе программирования, способной поддерживать нужный формат представления. Ранее мы считали объектом любой экземпляр класса, подготовленного с помощью системы *Delphi. COM*-объект (а также объект любой из рассматриваемых далее схожих технологий распределенного взаимодействия, например *CORBA* или *CGI*)отличается от таких объектов. *COM*-объект дополнительно обладает интерфейсом — внешним формальным описанием своих свойств и методов, которые могут использоваться другими *COM*-объектами. Таким образом, *COM*-объект напоминает скорее компонент *Delphi.* Из таких объектов, как из готовых кубиков, можно складывать приложения и крупные системы, работающие в компьютерной сети. Операционная система *Windows*базируется на технологии *COM* и ее сетевых расширениях (*DCOM, COM+* — они обеспечивают работу и взаимодействие объектов, физически выполняющихся на разных компьютерах). *COM*-объекты выполняют самые разные функции, причем создаются и уничтожаются системой автоматически (по мере возникновения запросов от пользователей, других программ и выполнения нужных действий), что значительно облегчает их использование. Достаточно только описать интерфейс *COM*-объекта и запрограммировать его реализацию, все остальное выполнят компилятор и *Windows.*

ВНИМАНИЕ Готовящаяся к выпуску в ближайшие годы платформа Windows .NET полностью переработана в архитектурном плане и предлагает качественно новую объектную модель — модель .NET-объекта. Массовое внедрение группы технологий .NET начнется, по прогнозам ведущих аналитических компаний, в 2004-2005 годах. В это же время предполагается резкий спад популярности COM-подхода.

Поэтому если планируется создание распределенной системы на базе Windows, надо решить, на какой срок эксплуатации она рассчитана. Если предполагаетсяее существенная переработка в течение ближайших пяти лет, то можно использовать COM-подход как наиболее простой и эффективный. Если же задумывается значительно более долгосрочный проект, то лучше основываться на таких подходах, как CORBA, Интернет-технологии(все они далее рассматриваются в данной книге). Выглядит очень перспективной и .NET-архитектура, но на момент выхода Delphi 7 она существовала только в бета-версии. Ее поддержка ожидается в следующих версиях продуктов Borland.

Одна из важных составляющих *.NET* — служба *Web Services*, реализация которой уже отдельно осуществлена корпорацией *Microsoft*. Создавать приложения *Web Services* можно и с помощью *Delphi 7* — этому вопросу посвящен соответствующий раздел, и хочется порекомендовать: обратите на *Web Services* самое серьезное внимание, как на одну из альтернатив подходу *COM*!

На основе технологии *COM* был создан ряд расширений, например серверы автоматизации (как в примере с автоматизацией *OLE),* активные серверные страницы *ASP (Active Server Pages)* и др. Одно из наиболее известных расширений технологии *COM* — элементы *ActiveX,* первоначально создававшиеся для использования в Интернете.

Основное различие технологий *COM и OLE* — возможность автоматического общения между компонентами *COM* и приложениями и наличие базового интерфейса, с помощью которого программа может выяснить, поддерживает ли конкретный объект *COM* функцию, нужную приложению (например, вычисление синуса или формирование определенного набора данных).

Составные части технологии COM

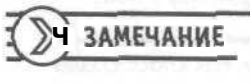
При создании приложения *COM* используются следующие понятия.

Таблица 8.3. Понятия технологии COM

Понятие	Что это такое
Интерфейс COM	Описывает методы и свойства, доступные программам, обращающимся К объекту. Объект COM может иметь один или несколько интерфейсов COM и содержит их описание и реализацию
Сервер COM	Законченный модуль кода (EXE или DLL), в котором хранится программный код одного или нескольких объектов COM
Клиент COM	Программный код в котором происходит обращение к интерфейсу COM с запросом на выполнение услуг сервера COM. Клиент COM знает, что ему надо получить от сервера COM, но не знает, как сервер будет это реализовывать и, вообще, где сервер физически расположен. Некоторая аналогия для клиента COM — рассмотренный ранее гример контейнера OLE

Интерфейс COM

Интерфейс *COM* позволяет клиентам *COM* общаться с сервером *COM* на основе стандартного механизма *публикации* интерфейса. После того как интерфейс *COM* опубликован (стандартным способом зарегистрирован в операционной системе), изменять его нельзя, что гарантирует одинаковую работу объекта *COM в* любых условиях.



ЗАМЕЧАНИЕ Идея интерфейсов COM аналогично идее реализации интерфейсной части (interface) в модулях Паскаля, когда другим разработчикам доступен для использования закрытый код и открытое описание его возможностей.

Базовый интерфейс IUnknown, который имеется у любого объекта COM, позволяет узнать, какие еще интерфейсы COM доступны для клиента COM Все эти интерфейсы наследуют характеристики интерфейса IUnknown.

ЗАМЕЧАНИЕ Название каждого интерфейса начинается с заглавной буквы I (Interface).

Уникальность интерфейса обеспечивается его глобальным идентификатором *Globally Unique Identifier (GUID)* длиной 16 байтов, а каждый объект COM имеет идентификатор интерфейса *IID (Interface Identifier)* на основе *GUID.* Идентификатор *GUID* требуется, чтобы избежать проблем при появлении интерфейсов COM с одинаковыми именами. Например, многие **разработчики** могут независимо друг от друга создать интерфейс COM с именем IGame, однако в каждом конкретном объекте *COM* он будет выполнять разные функции. Для того чтобы различать интерфейсы *COM* не по именам, и были введены идентификаторы *GUID.*

Благодаря наличию стандартных интерфейсов объект COM может быть реализован на любом языке программирования (в том числе и в системе *Delphi 7).*

Интерфейс ILJnknown содержит метод QueryInterface, возвращающий ссылку на другие доступные интерфейсы, а также методы AddRef и Release, которые увеличивают и уменьшают счетчик ссылок на конкретный интерфейс, когда к нему происходит обращение клиента *COM.* Например, пусть сервер *COM* содержит объект *COM,* имеющий интерфейс ICos, в котором содержится метод, вычисляющий косинус. При каждом обращении из разных программ к этому интерфейсу для вычисления косинуса счетчик увеличивается, а когда интерфейс освобождается (косинус вычислен и его значение передано клиенту COM), счетчик уменьшается. Как только значение счетчика становится равным нулю, то есть к интерфейсу больше нет обращений, соответствующий объект COM может быть удален из памяти до следующего запроса к его интерфейсу.

Сервер COM

Когда клиент COM обращается к серверу *COM,* он передает ему *идентификатор класса CLSID,* представляющий собой *GUID,* который ссылается на подходящий объект COM. Сервер COM создает специальный объект — фабрику классов (IClassFactory), — который занимается непосредственно созданием и загрузкой (производством) экземпляра нужного объекта COM, выполняющего конкретные действия его интерфейса, указанные в запросе клиента *COM.*

Фабрика классов «выпускает» объект *COM,* реализующий один или несколько интерфейсов *COM,* а также экземпляр специального класса CoClass, который обеспечивает возможность обращения к объекту COM на основе интерфейсов *COM.*

ЗАМЕЧАНИЕ В реальной работе этот класс называется так же, как класс создаваемого объекта (например, TTestObj), только вместо начальной буквы Т будет использована приставка Со — CoTestObj.

Серверы COW реализуются тремя способами.

1. В виде библиотеки .DLL. При этом объект СОЛ/ выполняется в адресном пространстве обратившегося к нему приложения.
2. Б виде приложения .EXE, которое выполняется в собственном адресном пространстве, но на одной машине с клиентом *COM*.
3. Б виде библиотеки .DLL или приложения .EXE, которые загружаются и работают на иной машине, нежели клиент *COM* (технология *DCOM*).

Расширения технологии COM

Ниже описаны некоторые расширения технологии *COM*.

Таблица 8.4. Расширения технологии COM

Типрасширения	Что это такое
Серверы автоматизации	Объекты COM, которые могут программно управляться из Других приложений. Например, большинство офисных приложений Windows, а также браузер Microsoft Internet Explorer, являются серверами автоматизации
Контроллеры автоматизации	Так называются клиенты COM, которые управляют серверами автоматизации И имеют дополнительные возможности настройки такого управления
Элементы ActiveX	Серверы COM, встраиваемые в приложения и содержащие средства для своей визуальной настройки. В некоторой степени аналогичны компонентам Delphi 5, которые можно настраивать с помощью Инспектора объектов и собственных редакторов
Библиотеки типов	библиотеки, хранящие описание объектов и их интерфейсов
Активные серверные страницы (ASP)	Компоненты ActiveX, которые предназначены для создания Web-страниц, активно взаимодействующих с пользователем
Активные документы	Объекты COM, которые поддерживают технологию OLE, методику перетаскивания и визуальное редактирование. Таковы, например, документы редактора Word
Визуальные мультипроцессные объекты	Объекты COM, которые могут использоваться в одновременно выполняющихся процессах

Пример создания объекта COM

Сейчас мы рассмотрим, как создается типичный сервер *COM* и как его использовать. Сервер *COM* будет содержать один объект *COM с* одним пользовательским (созданным разработчиком) интерфейсом *COM*. Этот интерфейс будет содержать единственный метод – функцию суммирования двух чисел.

Так как объекты *COM* могут быть созданы с помощью самых разных языков программирования, структуру их интерфейсов принято описывать на языке *IDL (Interface Definition Language, язык определения интерфейсов),* который немного напоминает язык программирования C++. В системе *Delphi 7* имеется возможность

настроить просмотр таких описаний как на языке *IDL*, так и на языке Паскаль. Для этого надо дать команду Tools > Environment Options (Сервис ➤ Настройка среды) и на вкладке Type Library (Библиотека типов) установить переключатель Pascal (Паскаль) на панели Language (Язык представления интерфейса) (рис. 8.10).

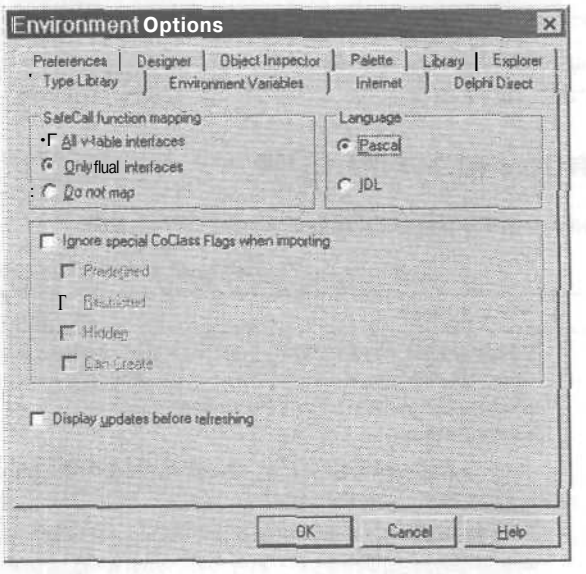


РИС. *8.10. Выбор языка описания интерфейса COM*

Создание сервера COM

Дайте команду File ➤ New > Other (Файл ➤ Создать > Другое) и на вкладке ActiveX выберите значок ActiveX Library (Библиотека объектов ActiveX). Такая библиотека представляет собой сервер COM, хранящий набор объектов *COM*.

Первоначальный код такой библиотеки генерируется автоматически. Теперь в нее надо добавить объект *COM*. Для этого служит команда File > New x Other (Файл ➤ Создать > Другое) — на вкладке ActiveX выбирается значок COM Object (Объект COM). В появившемся диалоговом окне задается ряд характеристик будущего объекта (рис. 8.11).

Имя класса, описывающего создаваемый объект, указывается в поле Class Name (Имя класса). Запишем здесь значение TestObj. При этом в недоступном для редактирования поле Implemented Interfaces (Реализованные интерфейсы) автоматически появится название соответствующего интерфейса ITestObj.

Способ создания объекта *COM* указывается в поле Instancing (Создание экземпляров). В раскрывающемся списке выбирается одно из следующих значений.

- Internal (Внутреннее). Объект создается только внутри приложения специальным методом этого приложения (например, документы редактора создаются только при работе с этим редактором).

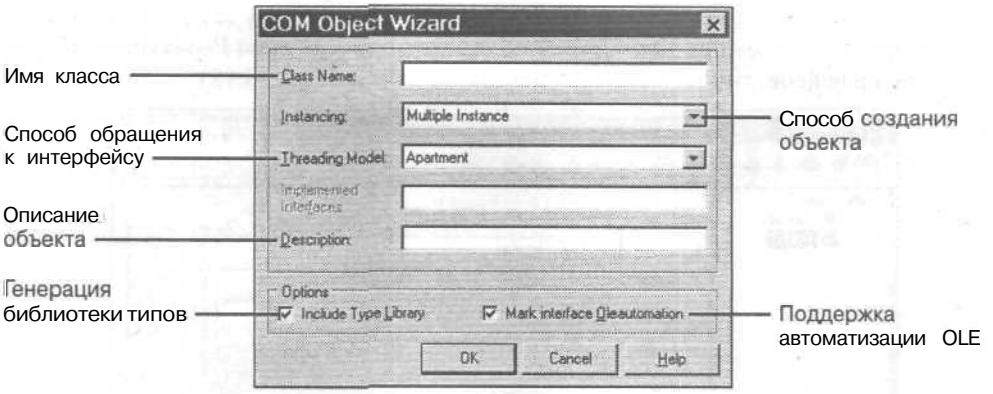


Рис. 8.11. *Окно предварительной настройки объекта COM*

- Single Instance (Один экземпляр). При запросе к серверу *COM* создается один экземпляр объекта. Следующий запрос к интерфейсу этого объекта рассматривается только после того, как текущий запрос обработан. Например, если использовать редактор, работающий по такому принципу, то для каждого нового документа потребуется запустить новую копию программы-редактора.
- Multiple Instance (Несколько экземпляров). Для одного приложения может быть создано несколько копий объекта. Выберем этот пункт списка.

Раскрывающийся список Threading model (Модель потоков) определяет способ обращения клиентских приложений к интерфейсу *COM,* Здесь можно выбрать одно из следующих значений.

- Single (Одиночная). Запросы обрабатываются по очереди.
- Apartment (Штучная). Для каждого запроса создается поток, в котором работает только один объект. Для каждого объекта может быть запущен только один поток, но внутри сервера *COM* могут работать несколько потоков для разных объектов. Для нашей программы выберем этот пункт списка.
- Free (Свободная). Для каждого запроса создается поток, в котором выполняется копия объекта *COM,* независимо от числа уже имеющихся копий этого объекта внутри данного потока.
- Both (Двойная). Поддерживаются модели Apartment (Штучная) и Free (Свободная).

В поле Description (Описание) задается краткое описание создаваемого объекта. Флажок Include Type Library (Включить библиотеку типов) желательно установить, чтобы для объекта была сгенерирована библиотека типов. Флажок Mark interface OleAutomation (Указать поддержку автоматизации OLE) устанавливается, если объект может работать как сервер автоматизации *OLE.* Эту возможность рекомендуется поддерживать всегда, благо она не требует от разработчика дополнительных усилий.

После щелчка на кнопке OK формируются интерфейс (пока без регистрации) и описание класса TTestObj. После этого на экране появится окно Редактора библиотеки интерфейсов, типов данных, объектов и функций (рис. 8.12).

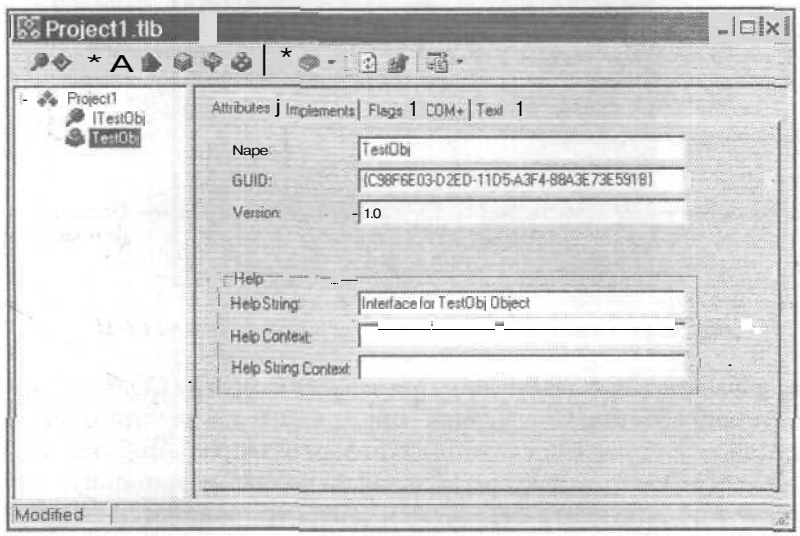


РИС. 8.12. *Редактирование интерфейсов сервера COM*

ЗАМЕЧАНИЕ Этот Редактор можно вызвать в любой момент командой View X Type Library (Вид > Библиотека типов).

Несмотря на обилие кнопок, работать с этим редактором несложно. На левой панели расположено дерево, отображающее взаимосвязи между объектами, имеющимися в библиотеке, и их интерфейсами, на правой — набор характеристик текущего выделенного объекта.

Строка TestObj описывает сам объект, строка ITestObj — один из его интерфейсов, которых у объекта *COM* может быть несколько. Выделив строку ITestObj, надо щелкнуть на кнопке New Method (Создать метод для интерфейса) и в новом поле ввести название метода GetSum.

В правой части редактора надо перейти на вкладку Parameters (Параметры функции), в раскрывающемся списке Return Type (Тип возвращаемого значения) выбрать тип Integer. Если бы в **настройках** среды мы указали не Паскаль, а язык *IDL*, то сейчас пришлось бы задавать все типы параметров в соответствии с требованиями этого языка. В нижнем списке указываются параметры метода. В нашем случае их два (два слагаемых). Щелкните на кнопке Add (Добавить). Теперь дважды щелкните на столбце Name (Имя) и введите имя параметра (первого слагаемого), например S1. В столбце Type (Тип), где определяется тип параметра, выберите в раскрывающемся списке значение Integer. Аналогичным способом добавляется и второй параметр — S2 (рис. 8.13).

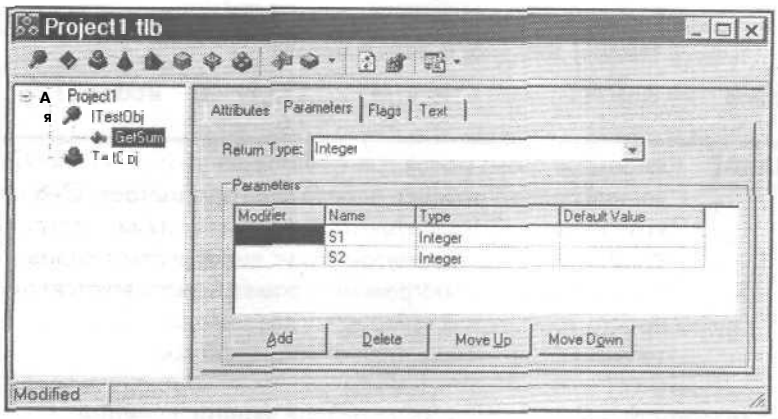


Рис. 8.13. Задание параметров вызова метода, включаемого в интерфейс COM

Незаполненными остались поля первого столбца списка Modifier (Модификатор), определяющие, каким способом параметр передается в метод. Если значение не указано, считается, что параметр передается по значению. Другие возможные значения поля Modifier (Модификатор) приведены в табл. 8.5.

Таблица 8.5. Способы передачи параметров методу объекта COM

Значение	**Способ передачи** параметра
none	Неизвестный (используется в технологии DCOM)
out	По ссылке. Параметр предназначен только для получения значения из метода, но не для передачи
optionalout	Необязательный параметр типа out
var	Параметр может использоваться как для передачи, так и для получения значения и передается по ссылке
optionalvar	Необязательный параметр типа var
optional	Необязательный параметр для передачи значения а метод. Должен иметь тип Variant
retval	Параметр используется для возврата значения. Он должен быть последним в списке параметров. Используется в исключительных случаях

ЗАМЕЧАНИЕ Объекту можно добавить не только новые методы, но и новые свойства. Это выполняют с помощью кнопки New Properly (Создать свойство). При этом надо указать нужный тип свойство в раскрывающемся списке Type [Тип] на вкладке Attributes (Атрибуты).

Интерфейс создан. Теперь осталось подготовить его реализацию, то есть указать, как конкретно будет работать метод GetSum. Для этого надо щелкнуть на кнопке Refresh Implementation (Обновить реализацию).

Теперь надо перейти в модуль Unit1, где хранится описание объекта TTestObj. Выяснится, что в классе TTestObj добавился новый метод:

```
function GetSum(S1, S2: Integer): Integer; stdcall;
```

ЗАМЕЧАНИЕ Ключевое слово stdcall применяется для указания способа вызова данной подпрограммы и передачи ей параметров. Оно означает, что функция будет вызываться в соответствии с соглашениями Windows API, а параметры будут передаваться справа налево. Конечно, в тексте программы параметры указываются как обычно, просто компилятор генерирует для методов с разными дополнительными описаниями разный машинный код.

В части реализации можно найти пустое описание данной функции:

```
function TTestObj.GetSum(S1, S2: Integer): Integer;
begin

end;
```

В ее тело надо добавить единственную строку:

```
Result := S1 + S2;
```

Серверный объект *COM* создан. Теперь его требуется зарегистрировать в *Windows*. Для этого в Редакторе интерфейсов надо щелкнуть на кнопке Register Type Library (Зарегистрировать библиотеку типов), в результате чего выполняется компиляция, создается библиотека .DLL и содержащийся в ней сервер *COM* регистрируется в системе.

Создание клиента COM

После того как сервер *COM* зарегистрирован, к нему возможно обращение от любого числа запущенных клиентских программ. Объект TTestObj создается внутри их адресных пространств.

Добавим в текущую группу проектов еще один (второй) проект, который будет описывать возможное клиентское приложение. Пусть оно содержит на форме два поля для ввода чисел, кнопку, при щелчке на которой произойдет обращение к серверу *COM,* и надпись для отображения результата обращения к методу GetSum интерфейса ITestObj (рис. 8.14).

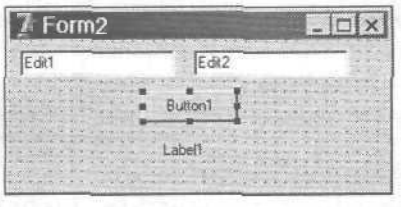


Рис. 8.14. Форма для приложения, которое будет производить обращение к серверу COM

Опишем в части public класса TForm2 две переменные: II и TestObjInterface.

```
type
   TForm2 = class(TForm)
      Edit1: TEdit;
      Edit2: TEdit;
      Button1: TButton;
      Label1; TLabel;
      procedure FormCreate(Sender: TObject);
      procedure Button1Click(Sender: TObject1 ;
   private
      { Private declarations }
   public
      { Public declarations }
      II, TestObjInterface: ITestObject;
   end;
```

С их помощью выполняется обращение к серверу *COM*. Чтобы клиентское приложение знало о наличии соответствующего интерфейса, в списке подключаемых модулей надо указать модуль Project1_TLB, автоматически сгенерированный Редактором интерфейсов:

```
uses
   Windows , Messages , SysUtils , Classes ,
   Graphics, Controls, Forms, Dialogs, StdCtrls,
   Project1_TLB;
```

Связь с сервером можно установить в момент создания формы. В соответствующем обработчике создается доступ к соответствующему интерфейсу путем создания Со-класса, описывающего общий интерфейс IUnknown, и вызова нужного интерфейса с помощью стандартного метода QueryInterface:

```
II := CoTestObject.Create;
II.QueryInterface(ITestObj,TestObjInterface);
```

Теперь к интерфейсу ITestObj можно обращаться с помощью переменной TestObjInterface.

Последний шаг — создание обработчика щелчка на кнопке. Он будет совсем простым. Сумма введенных значений вычисляется путем прямого обращения к методу GetSum интерфейса, хранящегося в объекте TestObjInterface:

```
procedure TForm2 .Button1Click(Sender: TObject);
. begin
   label1.Caption := IntToStr(
    TestObjInterface.GetSum(StrToInt(Edit1.Text),StrToInt(Edit2.Text))
);
end;
```

Теперь клиентское приложение можно откомпилировать и затем запустить несколько копий соответствующей программы. Каждая из них окажется работо-

способной. Вычисление суммы двух чисел производится независимо от состояния других приложений, так как каждая копия программы хранит внутри себя и копию объекта *COM,* который выполняется в собственном адресном пространстве (рис.8.15).

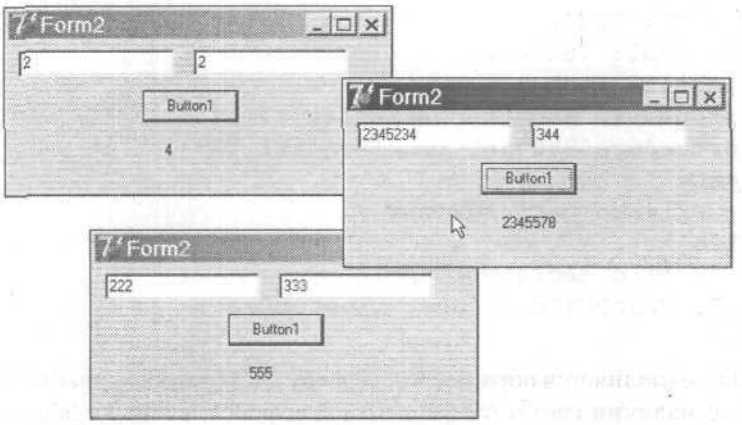


***Рис. 8.15.** Выполнение нескольких экземпляров программы, обращающейся для вычисления значений к серверу COM*

На основании описанного примера легко создать и значительно более сложные приложения. Достаточно описать в Редакторе интерфейсов новые методы (и, возможно, новые интерфейсы, которых у объекта *COM* может быть несколько) и зарегистрировать измененные интерфейсы. После этого обращаться к нужным интерфейсам можно из любых программ.

Создание системы COM на базе транзакционного сервера MTS

Особенности распределенных приложений COM

В данной главе рассматривается способ создания распределенных приложений на базе распределенной технологии *DCOM (Distributed COM)* с использованием сервера транзакций *MTS (Microsoft Transaction Server).* Транзакции в контексте распределенных вычислений понимаются несколько шире, чем при работе с СУБД. Транзакция включает в себя не просто логически законченную операцию над наборами данных, а логически законченный блок произвольных программных действий, в том числе включающих в себя обращения к базам данных (которые, в свою очередь, могут являться транзакциями СУБД).

Сервер транзакций *MTS,* который входит в стандартную поставку *Web*-сервера *Personal Web Server,* позволяет организовать совместную работу серверов *COM* и клиентов *COM,* возможно, выполняющихся на разных компьютерах. Он предоставляет весь комплекс услуг по безопасной (гарантированной) обработке транзакций, оптимальному распределению ресурсов и автоматическому управлению серверными компонентами *COM.*

Пример создания сервера COM и клиента COM на базе MTS

Постановка задачи

Этот пример похож на создание сервера *COM* в предыдущей главе. Только теперь к готовому серверу можно будет обращаться с любых компьютеров сети. Для такого сетевого распространения интерфейсов недостаточно только возможностей операционной системы. Именно поэтому используется сервер транзакций, отдельные компоненты которого работают на разных компьютерах и синхронизируют свою работу. Регистрация объекта *COM* на одном из компьютеров в сети приведет к автоматическому распространению информации об интерфейсах этого объекта по всей сети.

В данном примере создается сервер *COM,* который обладает собственным интерфейсом (с единственным методом-функцией AddNumbers для сложения двух чисел) и регистрируется в сервере *MTS,* а также клиентская программа, обращающаяся к интерфейсу в сервере *MTS* для суммирования двух введенных пользователем чисел.

ВНИМАНИЕ При реализации примера требуется, чтобы в системе был запущен и работал сервер MTS. Он входит в поставку бесплатно распространяемого персонального Web-сервера Microsoft Personal Web Server.

Создание пустого проекта

Создание проекта, ответственного за сервер *COM,* выполняется командой File ➤ New ➤ Other (Файл ➤ Создать ➤ Другое) с последующим выбором на вкладке Multitier (Многоуровневые приложения) значка Transactional Data Module (Транзакционный модуль данных), по аналогии с объектом COM.

ВНИМАНИЕ Предварительно необходимо подготовить пустую библиотеку COM командой File ➤ New > Other (Файл > Создать ➤ Другое) с последующим выбором на вкладке ActiveX значка ActiveX Library (Библиотека ActiveX).

Запускается Мастер создания транзакционного объекта. В поле CoCtass Name (Имя Со-класса) указывается имя нового класса (например, MTSTest), ответственного за формирование уже описанного выше сопроводительного класса — хранилища интерфейсов объекта и фабрики объектов.

Остальные значения менять не надо. Б иоле Threading model (Модель потоков) задается способ вызова сервером *MTS* соответствующего объекта *COM,* Значение Apartment (Штучная) позволяет запускать одновременно несколько процессов, обслуживающих один и тот же интерфейс.

В разделе Transaction Model (Модель транзакций) указывается способ поддержки транзакций сервером. Для обычного объекта *COM,* не выполняющего сложных действий и не обращающегося к базам данных, поддержка транзакций не нужна. В этом случае можно установить переключатель Does not support transactions (Поддержка транзакций отсутствует).

Объект COM MTSTest получаст интерфейс с именем IMTSTest и реализуется в системе *Delphi 7* в виде класса TMTSTest (рис. 8.16).

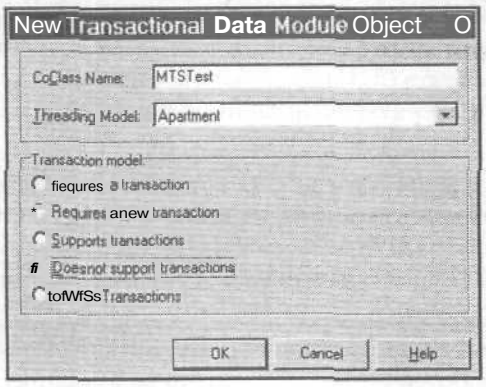


Рис. 8.16. *Создание транзакционного объекта*

Новый проект создается щелчком на кнопке OK. Модуль Unit1, сгенерированный системой *Delphi 7,* содержит пустое описание класса TMTSTest.

Проектирование интерфейса

Интерфейс для сервера *COM* создается с помощью редактора интерфейсов Type Library (Библиотека типов), уже рассмотренного ранее. В интерфейс IMTSTest добавляется новый метод AddNumbers. Это выполняется тем же способом, что и при разработке объекта COM TestObj (рис. 8.17).

В конце работы надо щелкнуть на кнопке Refresh Implementation (Обновить реализацию), чтобы сгенерировать программный код описания класса. Теперь весь проект, в том числе и созданный интерфейс, надо сохранить командой Save All (Сохранить все).

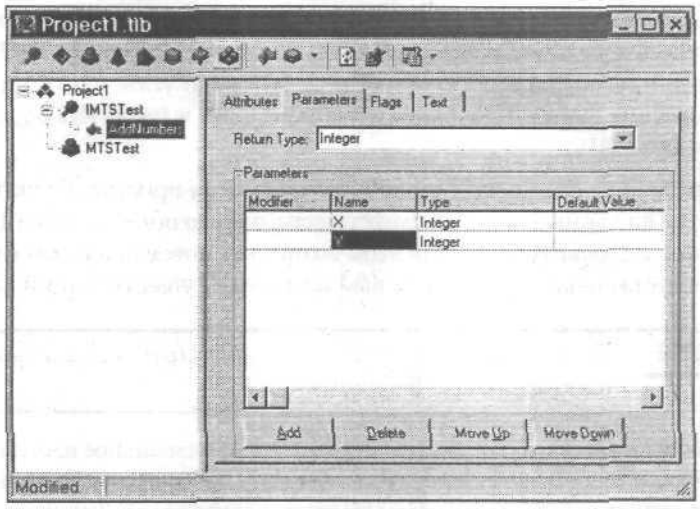


Рис. 8.17. *Определение методов, включаемых в интерфейс объекта MTS*

Создание реализации метода

Редактор интерфейсов автоматически создал описание нового метода в модуле Unit1:

```
function AddNumbers(X, Y: Integer): Integer; safecall;
```

ЗАМЕЧАНИЕ Донный метод имеет дополнительный описатель safecall, в отличие от описателя stdcall при создании обычного обьекто COM. Описатель safecall обеспечивает «безопасное» выполнение функции и дополнительный контроль ее работы, что по определению требуется для распределенных систем с поддержкой транзакций.

В части реализации модуля, где создана заготовка данной функции, необходимо описать, что эта функция делает (реально добавляется только один оператор присваивания):

```
procedure TMTSTest.AddNumbers(S1, S2: Integer);
begin
   Result := S1 + S2
end;
```

Теперь проект можно откомпилировать, нажав клавиши CTRL+F9. Б каталоге, где был сохранен проект, создастся библиотека Project1.dll (библиотека ActiveX). В ней содержится объект MTS MTSTest.

Регистрация объекта в сервере транзакций MTS

Для дальнейшей работы требуется, чтобы был запущен сервер транзакций *MTS*. Прежде всего сервер *COM* надо зарегистрировать. Это выполняется, как и в случае

обычного сервера *COM,* щелчком на кнопке Register Type Library (Зарегистрировать библиотеку типов). Система *Delphi 7* выполнит компиляцию сервера *COM* и сообщит о его успешной регистрации. Теперь объект MTS надо установить в сервер *MTS.* Для этого в системе *Delphi 7* выполняется команда Run ➤ Install MTS Objects (Запуск > Установка объектов MTS).

В появившемся окне со списком имеющихся в данном проекте объектов подходящего формата (на данный момент, скорее всего, одного объекта MTSTest) надо установить флажок в строке MTSTest. При этом возникнет новое диалоговое окно с уточнением места установки объекта: в новый или в уже существующий пакет,

ЗАМЕЧАНИЕ Объекты в сервере *MTS* хранятся сгруппированными в *пакеты,* напоминающие библиотеки.

На вкладке Into New Package (В новый пакет) введите произвольное имя пакета, например MTSTestPack, в поле Package Name (Имя пакета) и щелкните на кнопке OK. В окне установки против имени объекта теперь появится имя пакета. Для завершения установки щелкните на кнопке OK (рис. 8.18).

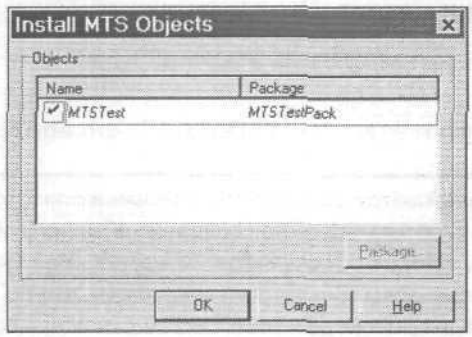


Рис. 8.18. *Указание имени и описания пакета*

А как узнать, добавился ли созданный объект *MTS* в сервер транзакций? Если это так, то он стал доступен всем приложениям в сети, где действуют компоненты текущего *MTS*-сервера. Для этого надо переключиться в окно сервера *MTS,* щелкнуть на значке My Computer (Мой компьютер), а затем — на значке Packages Installed (Установленные пакеты). В списке доступных пакетов должен быть указан и пакет MTSTestPack. Если его там нет, надо щелкнуть на кнопке Refresh (Обновить), чтобы сервер *MTS* проверил, не проводились ли новые установки пакетов и объектов (рис. 8.19).

Добраться до созданного интерфейса, доступного в сервере *MTS,* можно щелчком на значке MTSTestPack, затем на значке Components (Компоненты) и далее на значке Project1.MTSTest. Имя объекта *MTS* в сервере *MTS* формируется именно так, с предварительным указанием имени библиотеки DLL. Потом надо последовательно щелкнуть на значках Interfaces (Интерфейсы), IMTSTest, Methods (Методы), и только после этого в окне наконец появится список всех методов объекта MTSTest (рис. 8.20).

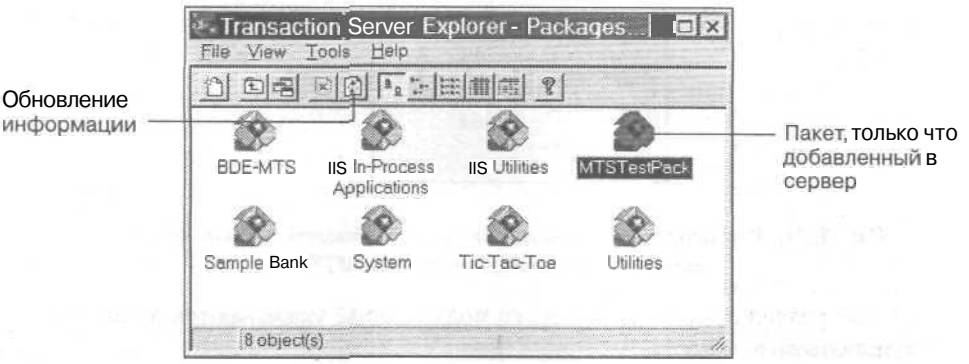


Рис. 8.19. Список пакетов, установленных на текущем сервере MTS

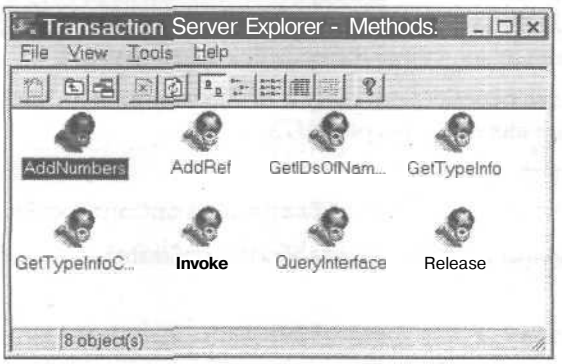


РИС. 5.20. *Список методов объекта MTS при просмотре с помощью сервера MTS*

Во время отладки приложения, взаимодействующего с сервером *MTS*, надо в начальном окне настроек сервера щелкнуть правой кнопкой мыши на значке My Computer (Мой компьютер) и выбрать в контекстном меню пункт Properties (Свойства). На вкладке Options (Настройки) задайте в поле Transactions will timeout in (Максимально допустимая продолжительность транзакции) значение 0, указывающее на неограниченную продолжительность транзакции. Это необходимо сделать, потому что во время отладки программы продолжительность транзакции может оказаться очень большой.

Создание клиентского приложения

Сервер *COM*, способный выполнять нужные вычисления (суммирование двух чисел), готов и зарегистрирован в сервере *MTS*. Теперь нам надо научиться обращаться к нему из клиентских программ.

С помощью Менеджера проектов создадим в текущей группе новый проект. Расположим на форме два текстовых поля Edit1 и Edit2, надпись Label1 и кнопку Button1 (рис. 8.21).

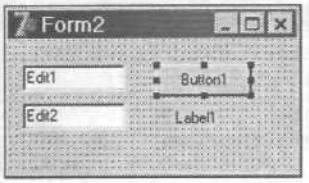


Рис. 8.21. *Форма программы, которая будет обращаться к объектам, зарегистрированным на сервере MTS*

В заголовке раздела реализации нового модуля Unit2 указываются дополнительные подключаемые модули:

```
implementation uses MtsObj, Mtx, ComObj, Project1_TLB;
```

Стандартные модули MtsObj, Mtx и ComObj ответственны за работу объектов *COM* и общение с сервером *MTS*, а модуль Project1_TLB, сгенерированный при компиляции объекта MTSTest, содержит описание интерфейса этого объекта.

Далее надо описать две переменные: одну для обращения к интерфейсу IMTSTest, а вторую для установки связи с сервером *MTS*:

```
var MTST: IMTSTest;
    TransactionContextEx: ITransactionContextEx;
```

В момент создания формы (обработчик события OnCreate) эти переменные нужно инициализировать:

```
procedure TForm2.FormCreate(Sender: TObject);
begin
   TransactionContextEx := CreateTransactionContextEx;
   OleCheck(TransactionContextEx.CreateInstance(
    CLASS_MTSTestIMTSTest,MTST) ) ,-
end;
```

Связь с сервером *MTS* устанавливается первым из операторов тела процедуры, а вызов метода CreateInstance осуществляет инициализацию переменной MTST. В трех параметрах этого вызова указываются следующие данные.

1. Идентификатор интерфейса — CLASS_MTSTest (тип TGUID). В исходном тексте автоматически сгенерированного модуля Project1_TLB можно найти описание этого идентификатора (возможный вариант приведен далее):

```
const
   CLASS_MTSTest: TGUID = '{720B0B03-53C7-11D3-A3F4-
     B8135F21131F}';
```

2. Имя интерфейса — IMTSTest.
3. Переменная MTST, в которую будет записана ссылка на реализацию данного интерфейса.

ЗАМЕЧАНИЕ Вызов метода CreateInstance помещен в параметр процедуры OleCheck, так как он не описан явно как safecall (безопасный формат вызова Windows API). При возникновении ошибки во время вызова данного метода надо возбудить исключительную ситуацию, что и делает метод OleCheck.

Логика работы программы закладывается в обработчик щелчка на кнопке Button1. Из текстовых полей берутся два числа, производится обращение к интерфейсу сервера *COM,* запускаемого сервером *MTS.* Объект *COM* выполняет нужные вычисления (суммирование) и возвращает результат, который передается в клиентскую программу сервером *MTS.* Этот результат отображается в виде надписи Label1:

```
procedure TForm2.Button1Click(Sender: TObject);
var N: integer;
begin
   N :=
MTST.AddNumbers(StrToInt(Edit1.Text),StrToInt(Edit2.Text));
   Label1.Caption := IntToStr(N);
end;
```

Клиентскую программу надо откомпилировать. После этого запустить несколько копий программы на разных компьютерах. Необходимо только, чтобы все эти компьютеры были объединены в сеть и на них работала служба транзакций *MTS* (рис. 8.22).

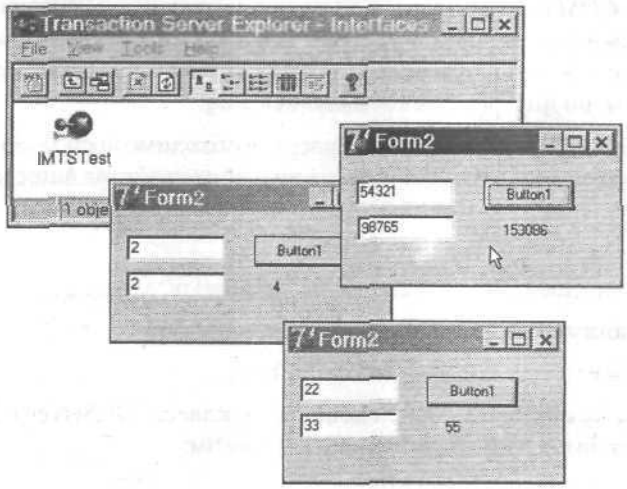


Рис. 8.22. *Запуск нескольких экземпляров программы, использующих объект COM, доступ к которому обеспечивает сервер MTS*

ВНИМАНИЕ Процессустановки связи с сервером транзакций (выполнение метода FormCreate) весьма продолжителен и на маломощном компьютере может занимать несколько десятков секунд.

По описанной технологии, почти не отличающейся от создания простого сервера *COM,* можно разрабатывать крупные сетевые приложения, вся логика работы которых заключается в компонентах сервера *MTS* со своими интерфейсами, а клиентским программам достаточно только обращаться к ним за нужными функциями. При этом программист освобождается от решения вопросов, связанных с распределением объектов по компьютерам сети, управлением загрузкой этих объектов на уровне файлов, контролем правильности их работы и от многих других задач, решение которых берет на себя сервер транзакций *MTS.*

При использовании системы *Delphi* /разработчик может полностью сосредоточиться на реализации нужных прикладных алгоритмов, совершенно не утруждая себя проблемами использования создаваемых объектов на локальном компьютере или запуска их в сети.

Панель COM+

Компонент Администратор COM-каталогов (TCOMAdminCatalog)

Этот компонент позволяет приложению работать как контроллеру автоматизации при обращении к *COM+* каталогам, С помощью компонента TCOMAdminCatalog автоматизируется администрирование *COM+ приложений* и служб; он позволяет считывать и записывать информацию из *COM+* каталогов, содержащих конфигурационные файлы, по интерфейсу ICOMAdminCatalog.

Для подключения к *COMAdminCatalog*серверу необходимо воспользоваться одним из двух следующих методов или установить значение свойства Autoconnect равным True.

```
procedure Connect;
procedure ConnectTo(svrIntf: ICOMAdminCatalog);
```

Интерфейс для доступа к каталогу хранится в свойстве:

```
DefaultInterface: ICOMAdminCatalog;
```

Другие свойства компонента унаследованы от класса TOleServer (*OLE*-сервер). Самые важные из них — тип соединения с каталогом:

```
ConnectKind: TConnectKind;
```

и название машины, на которой запущен *COM+* сервер:

```
RemoteMachineName: string;
```

Методы TCOMAdminCatalog позволяют устанавливать *COM+* приложения и *COM+* компоненты, запускать различные службы, а также соединяться с удаленными серверами и управлять ими.

Создание распределенных приложений на основе технологии CORBA

Что такое CORBA

Технология *COM* с использованием сервера *MTS* внешне, с точки зрения прикладного разработчика, во многом напоминает рассматриваемую далее технологию *CORBA, и* это действительно так. Даже создание серверов *COM* и клиентов *COM* напоминает создание соответствующих объектов *CORBA.* А в качестве своеобразного аналога сервера транзакций *MTS* выступает так называемый *брокер CORBA.*

Одно из существенных отличий технологии *CORBA* от *MTS/COM* состоит в том, что серверы *CORBA* представляют собой обычные исполнимые *EXE*-файлы, которые запускаются при каждом обращении к соответствующему интерфейсу.

Серверы *COM* являются библиотеками .DLL, которые загружаются в сервер транзакций *MTS* и выполняются только в его рабочем пространстве процессов. Эти технологии имеют и другие принципиальные отличия и реализованы совершенно по-разному. Внешняя схожесть процесса разработки распределенных программ связана только с прекрасными возможностями системы Delphi 7, маскирующей огромный объем рутинной работы, связанной с поддержкой нюансов, специфичных для каждой из технологий.

Технология *CORBA (Common Object Request Broker Architecture,* общая архитектура брокеров объектных запросов) — это специальная технология, позволяющая создавать распределенные приложения, работающие на нескольких компьютерах и, в отличие от компонентов *COM,* способные выполняться под управлением разных операционных систем, а не только *Windows.* Технология *CORBA* организует совместную работу объектов, предоставляющих другим программам свои интерфейсы, характеризующие возможности этих объектов.

С помощью технологии *CORBA* создаются серверные объекты *CORBA,* которые могут быть запущены на любых компьютерах сети, при условии, что в этой сети установлен *брокер объектных запросов (Object Request Broker).* Это специальная программная система, которая отслеживает и синхронизирует работу всех объектов *CORBA,* организует процесс обмена сообщениями между ними, перезапускает их в случае сбоев компьютера, оптимально распределяет загрузку и организовывает связь клиентских программ с серверными объектами. Все это удается выполнять благодаря большому набору встроенных служб, например службе объектных транзакций *OTS (Object Transaction Service).*

После того как интерфейс объекта зарегистрирован в системе *CORBA,* он становится доступен любым другим программам. Не правда ли, этот подход очень напоминает работу сервера *MTS*?

ВНИМАНИЕ Брокер — это не некий программный процесс, а коллекция библиотек стандартных функций CORBA и набор сетевых ресурсов, позволяющих организовать распределенную работу программ, написанных на разных языках программирования.

В сети может быть запущено множество объектов *CORBA*. Где им лучше выполняться — совместно на одном сервере или на выделенном компьютере, — решает брокер (точнее, один из его ресурсов). Он анализирует поступающие от клиентских программ требования, перераспределяет их соответствующим объектам на основании зарегистрированных интерфейсов, дожидается результата и возвращает его программе, установившей связь с объектом *CORBA,*

Отличительная особенность технологии *CORBA* — возможность совместной работы объектов, созданных для разных операционных систем (не обязательно для *Windows).* Хотя с помощью системы *Delphi 7* можно создавать только объекты *CORBA* для *Windows,* технологии *CORBA* поддерживаются сегодня на большинстве вычислительных платформ. Например, объекты *CORBA* могут быть запущены и в *Windows,* и под управлением *UNIX,* и на больших компьютерах. При этом клиентская программа запрашивает выполнение определенных действий, обращаясь к любому зарегистрированному в сети интерфейсу.

Брокер при необходимости автоматически организует связь с брокерскими службами, работающими в других операционных системах, и те самостоятельно находят нужный объект, на каком бы компьютере он ни располагался. А конечный пользователь даже не подозревает, где реально выполняются нужные ему вычисления.

ЗАМЕЧАНИЕ Система Delphi 7 позволяет создавать для Windows и серверы CORBA, и клиенты CORBA. В поставку также входит брокер VisiBroker.

Пример создания сервера CORBA и клиента CORBA

В прежних версиях *Delphi* поддержка технологии *CORBA* была тесно объединена с поддержкой технологии *COM* (на уровне визуальных средств формирования интерфейсов). Это, в частности, выражалось в использовании редактора библиотеки типов, применявшегося преимущественно при описании интерфейсов для *COM*-объектов.

В *Delphi 7* включен специальный компилятор IDL2PAS, автоматически генерирующий исходные тексты серверного и клиентского приложения на Паскале на основе описания интерфейса, заданного на языке *IDL.* Такой подход хорош тем, что опирается на платформно-независимый язык *IDL* и дает возможность создавать распределенные *CORBA*-приложения на основе шаблонов, подготовленных для других операционных систем.

ЗАМЕЧАНИЕ При использовании компилятора IDL2PAS в приложении будут использованы новые модули corba.pas и orbpas30.pas/orbpas40.pas вместо старых corbaobj.pas и orbpas.pas.

Серверный CORBA-модуль

Рассмотрим пример создания *CORBA*-системы с применением новых возможностей *Delphi 7*. Эта система реализует функцию сложения двух чисел:

```
function AddNmb(const X: Integer; const Y: Integer):
Integer;
```

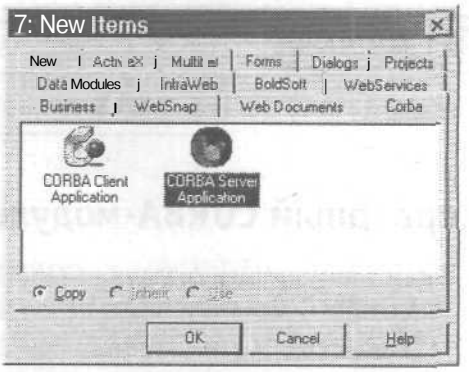
Прежде всего необходимо подготовить исходный файл server.idl, содержащий описание нужного нам интерфейса. Для этого необходимо знание языка *IDL* — он во многом напоминает язык Си. Вместе с тем его основные элементы настолько прозрачны, что понимание структуры простейшего интерфейса не составит труда.

Назовем *IDL*-модуль — Server и опишем в нем два интерфейса: IMyTest и MyTestFactory. Первый, основной интерфейс будет доступен другим *CORBA*-объектам, а второй интерфейс, MyTestFactory, нужен для автоматического создания копии соответствующего объекта при обращении к публикуемому интерфейсу. Этот интерфейс сохраним в файле server.idl.

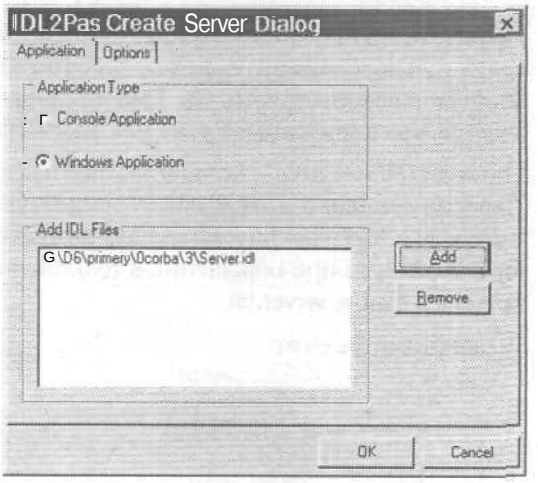
```
module Server
{
  interface IMyTest;

  interface IMyTest
  {
    long AddNmb( in long X, in long Y );
  };

  interface MyTestFactory
  {
    IMyTest CreateInstance(in string InstanceName);
  };
};
```

Для создания *CORBA*-сервера надо дать команду File ➤ New > Other и на закладке CORBA выбрать значок CORBA Server Application. Появится диалоговое окно, вызывающее компилятор IDL2PAS.

Этот компилятор можно также вызвать командой Tools ➤ Regenerate CORBA IDL files (Инструменты ➤ Перегенерировать CORBA IDL-файлы).



Следующий шаг — генерация на основе описанного выше интерфейса прикладных модулей *Delphi*, реализующих соответствующую функциональность. Кнопкой Add в диалоговом окне выбирается соответствующий *IDL*-файл (server.idl), описывающий наш интерфейс, а с помощью переключателя Application Type можно указать вид создаваемого приложения: консольное или *Windows* (выберем вариант *Windows*). На вкладке Options задаются дополнительные параметры генерации исходных текстов. Желательно, чтобы были установлены флажки Generate Skeleton Units и Generate Implementation Units, которые соответственно означают Генерировать шаблоны модулей и Генерировать модули реализации интерфейса.



После нажатия кнопки OK будут сгенерированы четыре файла на Паскале, названия которых совпадают с названием модуля *IDL*-файла, описывающего наш интерфейс (Server). Дополнения к названиям полученных файлов (они расположатся в одном каталоге с server.idl) означают следующее;

- i — модуль реализации интерфейса;
- s — модуль реализации шаблона интерфейса;
- c — модуль реализации класса;
- impl — модуль, реализующий функциональность интерфейса.

В файле server_impl.pas надо указать, что делает единственный описанный в интерфейсе метод AddNmb:

```
function TIMyTest.AddNmb(const X : Integer; const Y :
Integer): Integer;
begin
   Result :=X+Y
end;
```

Автоматически подготовленная процедура InitCorba этого модуля выполняет инициализацию и связь с *CORBA*-системой, а также создает и формирует соответствующий серверный интерфейс. Вызывается она в момент создания формы сервера. В нашем случае текст модуля будет выглядеть так:

```
unit Unit1;
interface uses
   Windows, Messages, SysUtils, Classes, Graphics,
Controls, Forms, Dialogs, Corba,
   Server_c, Server_i, Server_s, Server_impl;
type
   TForm1 = class(TForm)
      procedure FormCreate(Sender: TObject) ;
      private
      { private declarations }
      protected
      { protected declarations )
         MTest: IMyTest; // skeleton object
         procedure InitCorba;
      public
      { public declarations ]
   end;
var Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.InitCorba;
begin
   CorbaInitialize;
   MTest := TIMyTestSkeleton.Create('CORBA MyTest',
    TIMyTest.Create);
   BOA.ObjIsReady(MTest as _Object);
end;
procedure TForm1.FormCreate(Sender: TObject);
begin
   InitCorba;
end;
end
```

Данное приложение можно откомпилировать и запустить. Оно будет представлять собой полноценный *CORBA*-сервер, реализующий интерфейс IMyTest.

Создание клиентского CORBA-приложения

Дадим команду File ➤ New ➤ Other, на панели *CORBA* выберем значок CORBA Client Application. Снова, как в прошлый раз, с помощью кнопки Add добавим соответствующий *IDL*-файл с описанием нужного интерфейса и нажмем кнопку OK. Будет автоматически сгенерирован клиентский *CORBA*-модуль, в котором потребуется дополнительно описать переменную:

```
vIMyTest: IMyTest;
```

В момент создания формы эта переменная послужит интерфейсом *CORBA*-серверу. Такая связь выполняется с помощью метода bind класса TIMyTestHelper. Теперь к этому интерфейсу можно обращаться из других частей программы, например при нажатии кнопки Button1. Полный текст данного модуля запишется так:

```
unit Unit2;
interface uses
   Windows, Messages, SysUtils, Classes, Graphics,
    Controls, Forms, Dialogs, Corba,
   Server_c, Server_i, StdCtrls;
type
   TForm1 = class(TForm)
      Button1: TButton;
      procedure Button1Click(Sender: TObject);
      procedure FormCreate(Sender: TObject);
      private
      { private declarations }
      protected
      vIMyTest: IMyTest;
      public
      { public declarations }
   end;
var Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.FormCreate(Sender: TObject);
begin
```

```
    vIMyTest := TIMyTestHelper.bind;
end;
procedure TForm1.Button1Click(Sender: TObject);
var n: integer;
begin
    n := vIMyTest.AddNmb(2,3) ;
    ShowMessage(IntToStr(n))
end;
end
```

ЗАМЕЧАНИЕ Для того чтобы компиляция серверного и клиентского приложения проходило нормально, в список подключаемых модулей необходимо добавить ссылку на автоматически сгенерированные файлы — в нашем случае это Server_c и Server_i.

Тестирование CORBA-проекта

Запуск и использование объектов CORBA

В реальных проектах сервер *CORBA* запускается на более мощных компьютерах, чем ПК, на котором расположена клиентская программа *CORBA,* что позволяет более эффективно обрабатывать запросы. Однако тестовый пример вполне может работать и на локальном компьютере, под управлением *Windows 9x.* Брокеру все равно, расположены ли соответствующие объекты на одном компьютере или на ста разных, главное, чтобы они зарегистрировали свои интерфейсы.

А как произойдет регистрация интерфейса только что созданного сервера *CORBA?* Автоматически. Хотя технология *CORBA* предусматривает несколько способов такой регистрации, в том числе и динамический, система *Delphi7* создает весь код, необходимый для регистрации заданного интерфейса, без участия человека.

Перед запуском серверного модуля на компьютере должен быть запущен «интеллектуальный агент» SmartAgent — ресурс брокера, который автоматически обнаруживает объекты с интерфейсами, нужными клиентским программам. Он вызывается командой Пуск > Программы > VisiBroker ➤ Visibroker Smart Agent. На экране появляется пустое окно агента.

Далее надо запустить с помощью, например, Проводника сервер *CORBA* (Project1.exe, появится тоже пустое окно) и для проверки его работы две (или пять, или больше) копий клиента *CORBA* Project2.exe.

В каждой из клиентских программ можно ввести два числа и мгновенно получить их сумму, щелкнув на кнопке Button1 (рис. 8.23). При этом суммирование реально выполняется, как уже говорилось, сервером *CORBA,* а работать он может и на дру-

гом компьютере, обрабатывая запросы любого числа клиентов *CORBA*. Производительность серверного объекта и скорость его ответа на запросы клиентов определяется только ресурсами того компьютера, на котором он запущен.

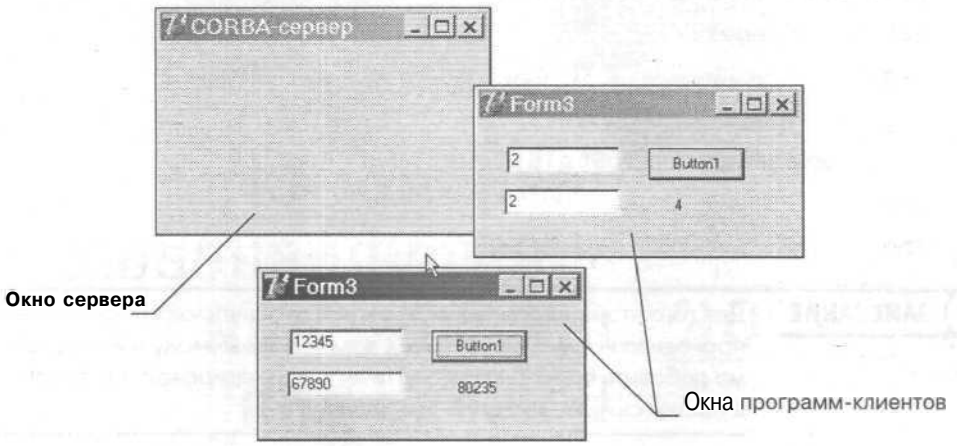


Рис. 8.23. *Выполнение вычислений при помощи сервера CORBA*

ЗАМЕЧАНИЕ Рассмотренные новые возможности Delphi *7* ориентированы на новую версию брокера объектных запросов VisiBroker 4.0. С ней могут быть несовместимы CORBA-приложения, разрабатывавшиеся для версии VisiBroker 3.0/3.3.

Немаловажно, что к созданному таким образом *CORBA*-серверу можно обращаться из клиентских *Java*-программ.

Что нового мы узнали?

В этом уроке мы научились

- передавать данные между приложениями;
- конструировать и применять динамические библиотеки;
- организовывать выполнение потоков;
- использовать в работе технологию *COM;*
- создавать распределенные приложения;
- проектировать распределенные приложения на основе технологии *CORBA.*

Технология многоуровневых приложений Borland для работы с СУБД

- Основные принципы создания многоуровневых **приложений,** работающих с СУБД
- Создание многоуровневых приложений, работающих с СУБД с использованием транзакционного сервера **MTS**
- Оригинальные возможности **Delphi** по созданию многоуровневых приложений
- Проектирование распределенных приложений, использующих протоколы TCP/IP и HTTP
- Использование множественных удаленных модулей данных
- Использование технологии XML

Основные принципы создания многоуровневых приложений, работающих с СУБД

В предыдущих главах рассматривался способ создания клиент-серверных приложений. Такие приложения называют *двухуровневыми*, потому что их составляющие работают на двух уровнях: клиентском и серверном.

Система *Delphi 7* позволяет создавать более сложные и значительно более эффективные программные конфигурации — *многоуровневые* системы, когда отдельные компоненты клиент-серверного приложения выполняются на разных компьютерах и связываются друг с другом через локальную сеть или Интернет.

Чаще всего применяется *трехуровневая* модель, когда многоуровневое приложение делится на три части.

- Клиентская программа. Реализует пользовательский интерфейс и посылает запросы на выполнение нужных действий. Речь не обязательно идет о получении наборов данных; удаленные компоненты могут проводить интенсивные вычисления, скажем, создание аналитических отчетов. При этом требования к ресурсам компьютера, на котором выполняется клиентская программа, минимальны, так как вся логика системы реализуется компонентами, работающими на значительно более мощных серверах сети. Такие клиентские программы называют «тонкими» *(thin),* потому что они выполняют минимальный объем работы и не предъявляют особых требований к процессору и объему оперативной памяти.
- Сервер приложений. Программное обеспечение промежуточного слоя. Синхронизирует работу всех компонентов системы и организует связь между ними. В рассмотренных выше примерах такую роль играли сервер *MTS* и брокер объектных запросов *CORBA.*
- Удаленный сервер баз данных. СУБД, выполняющаяся на выделенном компьютере, обрабатывающая запросы от сервера приложений и по этим запросам возвращающая наборы данных или вносящая изменения в таблицы. Данный сервер не работает напрямую с клиентскими программами, что повышает безопасность информации, хранимой в системе.

В некоторые многоуровневые приложения добавляются дополнительные службы, например сервер транзакций Интернета или различные системы обмена информацией между СУБД и программами, выполняющимися под управлением операционных систем, отличных от *Windows.* При этом, благодаря тому что удается выделить детали логики приложения в отдельные компоненты, клиентские программы могут одновременно обращаться к одним и тем же компонентам для решения одинаковых задач, а серверы приложений снимают все проблемы по синхронизации работы таких компонентов, их автоматической загрузке и перераспределению работы между серверами сети, если один из них выходит из строя.

Состав многоуровневого приложения

Многоуровневое приложение основывается на следующих компонентах.

- Удаленные модули данных. Серверы *COM* или *CORBA*, предоставляющие клиентской программе доступ к компонентам — поставщикам информации.
- Поставщики информации. Специальные объекты, возвращающие по запросу нужную информацию (результат вычислений или набор данных). Играют роль сервера приложений.
- Компоненты связи. Службы, обеспечивающие связь между всеми составляющими системы.
- Клиентские наборы данных. Специализированные объекты клиентской программы для обработки получаемых наборов данных.

Механизм работы многоуровневого приложения

1. Клиентская программа соединяется с сервером приложений. Если он не запущен, то запускается автоматически.
2. Клиентская программа запрашивает данные у сервера приложений.
3. Сервер приложений обращается к СУБД за получением набора данных, запаковывает его и отсылает клиентской программе.
4. Клиентская программа распаковывает «посылку», преобразует ее в локальный набор данных и отображает в соответствующих компонентах (например, в компоненте TDBGrid).
5. Пользователь вносит изменения в локальный набор данных (удаляет, изменяет или добавляет записи). Процесс изменений протоколируется клиентской программой.
6. Реагируя на действия пользователя, клиентская программа посылает запакованный протокол изменений серверу приложений.
7. Сервер приложений распаковывает посылку и формирует транзакцию серверу баз данных на внесение изменений в соответствующие таблицы. Если при этом возникает проблема выполнения такой транзакции (например, запись, которую один пользователь пытается удалить, другой пользователь изменил), сервер пытается согласовать содержимое клиентского набора данных с содержимым серверного набора данных, формируя набор ошибочных записей.
8. Информация о таких записях отправляется обратно в клиентскую программу.
9. Клиентская программа пытается решить проблему с ошибочными записями, взаимодействуя с пользователем.
10. Клиентский набор данных обновляется.

Упакованный набор данных

Передаваемые между объектами многоуровневого приложения наборы данных представлены в упакованном формате типа OleVariant. Этот тип аналогичен типу Variant, но несовместим с ним по внутренней структуре и предназначен только для использования в компонентах *COM*. Такие упакованные наборы данных называются *дельта-наборами*, а соответствующий им параметр в различных методах всегда обозначается Delta.

Компонент Поставщик данных (TDataSetProvider)

Этот невизуальный компонент наиболее важен при создании многоуровневых приложений. Он организует обмен информацией, упакованной в *пакеты*, между клиентскими наборами данных и внешними источниками информации. Большую часть такой работы поставщик данных берет на себя, а пользователь, используя его свойства и методы, может гибко контролировать этот процесс.

После размещения компонента на форме достаточно, в принципе, задать значение только одному его свойству — DataSet (например, указать конкретную таблицу базы данных).

Компонент TDataSetProvider наследует ряд свойств и методов от вышестоящих классов. Так, класс TCustomProvider обеспечивает базовую функциональность для всех *классов—поставщиков информации*. Более конкретные характеристики хранятся в классе TBaseProvider.

Таблица 9.1. Свойства класса TCustomProvider

Свойство	Назначение
Data	Упакованный набор данных (класс OleVariant). Доступен только для чтения
Exported	Имеет значение True, если клиентская программа может обращаться к данному классу напрямую. Это свойство получает значение False, если между клиентом и сервером приложений имеются дополнительные программные службы (например, система контроля доступа или шифрования информации)

Таблица 9.2. Методы класса TCustomProvider

Метод	Назначение
function ApplyUpdates(const Delta: OleVariant; MaxErrors: Integer; out ErrorCount: Integer); OleVariant;	Синхронизация упакованного набора с клиентским набором данных. Функция возвращает записи, которые согласовать не удалось, в виде упакованного набора
procedure Execute(const CommandText: WideString, var Params, OwnerData: OleVariant);	Выполнение запроса, хранимой процедуры или оператора SQL
function GetParams(var OwnerData: OleVariant); OleVariant;	Массив значений параметров для текущего поставщика

Таблица 9.2. Методы класса TCustomProvider (продолжение)

Метод	Назначение
function GetRecords(Count: Integer; out RecsOut: Integer; Options: Integer); OleVariant;	Получение упакованного набора данных поставщика с числом записей, указанным в параметре Count
function RowRequest(const Row; OleVariant; RequestType: Integer; var OwnerData: OleVariant): OleVariant;	Получение информации о конкретной записи набора данных поставщика. Вид требуемой информации указывается в параметре RequestType, который фактически имеет тип TFetchOptions, приведенный к типу Integer

Таблица 9.3. События класса TCustomProvider

Событие	Условие генерации
BeforeApplyUpdates AfterApplyUpdates	Возникают, соответственно, до и после синхронизации данных
BeforeExecute AfterExecute	Возникают, соответственно, до и после выполнения запроса
BeforeGetParams AfterGetParams	Возникают, соответственно, до и после получения значений параметров
BeforeGetRecords AfterGetRecords	Возникают, соответственно, до и после получения упакованного набора данных.
BeforeRowRequest AfterRowRequest	Возникают, соответственно, до и после получения информации о конкретной записи
On Data Request	Возникает при запросе данных. Программист может настроить свойство DataSet поставщика, например указать подходящий фильтр для получения нужного набора

Таблица 9.4. Свойства класса TBaseProvider

Свойство	Назначение
Options	Информация о том, как будет использоваться набор данных (тип TProviderOptions). Описывает нюансы работы с подчиненными таблицами, способы обработки набора данных, Blob-значений и так далее
Resolver	Ссылка на объект TCustomResolver, который создается поставщиком автоматически. Этот объект выполняет процедуру разрешения конфликтов между клиентским и серверным наборами данных
UpdateMode	Способ поиска записей для их обновления в процессе работы метода ApplyUpdates. Возможные значения: upWhereAll (используются все поля); upWhereChanged (используются только измененные ключевые поля и их исходные значения); upWhereKeyOnly (используются только ключевые поля)

Таблица 9.5. События класса TBaseProvider

Событие	Условие генерации
BeforeUpdateRecord AfterUpdateRecord	Возникает, соответственно, до и после обновления записи. Используется, например, для ведения протокола корректности обновлений
OnGetData	Поставщик получил набор данных, но еще не послал его клиентской программе. На этом этапе можно, например, выполнять шифрование информации
OnUpdateData	Поставщик приступает к синхронизации данных. В обработчике этого события можно выполнять, например, предварительное тестирование данных
OnUpdateError	Обнаружена невозможность синхронизации двух записей (одна из них, например, была удалена другим пользователем). В обработчике можно реализовать собственный процесс разрешения подобных коллизий

Таблица 9.6. Собственные свойства класса TDataSetProvider

Свойство	Назначение
Constraints	Имеет значение True, если клиентской программе сообщается дополнительная информация о целостности передаваемого набора данных
ResolveToDataSet	Имеет значение True, если обновления фиксируются в локальном наборе данных. В противном случае обновления фиксируются в наборе данных на сервере

Таблица 9.7. События класса TDataSetProvider

Событие	Условие генерации
OnGetDataSetProperties	Передача сведений о наборе данных клиентской программе. В обработчике этого события можно подготовить дополнительную пользовательскую информацию. Это может быть, к примеру, число записей, время создания набора и прочее
OnGetTableName	Объект, решающий проблемы синхронизации, пытается сформировать название набора данных. Если набор представляет собой объект TTable, то при этом происходит обращение к свойству TableName

Компонент Клиентский набор данных (TClientDataSet)

Данный компонент, прямой наследник класса TDataSet, предназначен для поддержки наборов данных, независимых от конкретной СУБД и вообще от типа источника информации. Кроме того, подобный набор данных может быть распределенным: группы записей могут располагаться на разных компьютерах или, наоборот, представлять собой обычный «плоский» файл.

Этот компонент используется, как правило, совместно с поставщиком данных. Он обладает большим числом характеристик, охватывающих всевозможные способы физического представления соответствующей набору информации.

Необходимо отметить также свойство CurValue (тип Variant) класса TField, описывающее конкретное поле набора данных. Оно используется для получения текущего значения поля и разрешения возможных конфликтов с учетом изменений, внесенных в него другими пользователями, и может отличаться от значений свойств OldValue (Значение до согласования) и NewValue (Значение после согласования). Свойства CurValue, OldValue и NewValue могут использоваться только в классе TClientDataSet.

Таблица 9.8. Свойства класса TClientDataSet

Свойство	Назначение
Active	Имеет значение True, если данные доступны
ActiveAggs	Список (TList) активных итоговых полей
Aggregates	Список всех итоговых полей
AggregatesActive	Имеет значение True, если итоговые поля должны вычисляться
CanModify	Имеет значение True, если разрешается изменять записи в текущем наборе данных
ChangeCount	Число изменений, произведенных в наборе данных
CloneSource	Ссылка на набор данных (TClientDataSet), который делает текущие данные доступными другим пользователям путем их копирования
CommandText	Это новое и очень полезное свойство. Оно описывает оператор SQL, который может быть выполнен на сервере с помощью метода Execute
Data	Ссылка на представление набора данных в упакованном формате (OleVariant) для передачи по сети
DataSetField	Ссылка на главную таблицу (отношение Master/Detail)
DataSize	Число байтов, требуемое для хранения значения в свойстве Data
DataSource	Ссылка на таблицу-хозяина, если данный набор является подчиненным в отношении Master/Detail
Delta	Упакованный (OleVariant) протокол изменений, произведенных над набором данных. Применяется для отправки серверу приложений, который автоматически выполняет и синхронизирует все нужные изменения в базе данных
Fetch On Demand	Имеет значение True, если загрузка данных из базы выполняется только по запросу пользователя. Полезно использовать, если таблица содержит данные большого размера (например, Blob-поля с рисунками)
FileName	Имя файла, в котором хранятся данные текущего набора
Filter	Фильтр для отбора записей
Filtered	Имеет значение True, если используется свойство Filter
FilterOptions	Параметры работы фильтра
GroupingLevel	Уровень группирования записей
HasAppServer	Имеет значение True, если установлена связь с сервером приложений
IndexDefs	Массив определений индексов для текущего набора данных
IndexFieldCount	Число полей, привязанных к текущему индексу
IndexFieldNames	Названия полей, используемых как текущий индекс

продолжение

Таблица 9.8. Свойства класса TClientDataSet (продолжение)

Свойство	Назначение
IndexFields	Массив полей, используемых как текущий индекс
IndexName	Название альтернативного индекса
KeyExclusive	Имеет значение True, если будет использоваться диапазон отбора записей
KeyFieldCount	Ограничение на максимальное число полей, используемых при поиске записей на основе ключа, состоящего из нескольких полей
KeySize	Размер текущего ключа
LogChanges	Имеет значение True, если изменения в наборе данных предварительно протоколируются (свойство Delta), а потом отсылаются серверу приложений. В противном случае измененные данные сразу заносятся в свойство Data
MasterFields	Названия полей в главной таблице, по которым установлена связь Master/Detail
MasterSource	Источник данных — главная таблица
PacketRecords	Число записей в пакете, который был послан или получен от сервера приложений
Params	Параметры для выполнения запроса или хранимой процедуры на сервере приложений
ProviderName	Название объекта — поставщика данных (сервер приложений)
Readonly	Имеет значение True, если набор данных доступен в режиме «только чтение»
RecNo	Номер текущей записи в наборе данных
RecordCount	Общее число записей в наборе данных
RecordSize	Размер одной записи в байтах
RemoteServer	Ссылка на компонент, используемый для связи с сервером приложений
SavePoint	Текущее состояние протокола изменений. Можно использовать для управления транзакциями. Восстановление ранее сохраненного значения этого свойства выполняет «откат» назад состояния набора данных
StatusFilter	Отбор записей, имеющих указанное состояние (измененные, добавленные и другие)
StoreDefs	Имеет значение True, если индексы и определения полей хранятся вместе с набором данных

Таблица9.9. Методы класса TClientDataSet

Метод	Назначение
Procedure AddIndex(const Name, Fields: string; Options: TIndexOptions; const DescFields: string = ; const CaseInsFields: string = ; const GroupingLevel: Integer = 0);	Добавление нового индекса к набору данных
Procedure AppendData(const Data: OleVariant; HitEOF: Boolean);	Добавление новых записей, полученных от сервера приложений в упакованном формате, к набору данных
Procedure ApplyRange;	Установка диапазона записей

Таблица 9.9. Методы класса TClientDataSet (продолжение)

Метод	Назначение
function ApplyUpdates(MaxErrors: Integer); Integer;	Посылка протокола изменений серверу приложений, чтобы он внес эти изменения в базу данных
function BookmarkValid(Bookmark: Tbookmark): Boolean;	Возвращает значение True, если указанная закладка существует
Procedure Cancel;	Отмена незафиксированных изменений в наборе данных («откат» транзакции)
Procedure CancelRange;	Отмена всех установленных диапазонов
Procedure Cancel Updates;	Отмена всех изменений, незафиксированных в протоколе
ProcedureCloneCursor(Source :TclientDataSet; Reset: Boolean; KeepSettings: Boolean = False);	Формирование дополнительного набора данных (например, временного) на основе текущего
Function CompareBookmarks(Bookmark1, Bookmark2: TBookmark): Integer;	Сравнение двух закладок
function ConstraintsDisabled: Boolean;	Возвращает значение True, если используются дополнительные описания набора данных
function CreateBlobStream(Field: TField; Mode: TblobStreamMode): TStream;	Создание потока для обработки Blob-поля
Procedure Create DataSet;	Создание нового пустого набора данных
function DataRequest(Data: OleVariant): OleVariant;	Метод генерирует событие OnDataRequest (запрос данных от объекта-поставщика) и позволяет создавать собственные алгоритмы фильтрации записей в момент передачи упакованного набора данных
Procedure DeleteIndex(const Name: string);	Удаление указанного индекса
Procedure EnableConstraints; Procedure DisableConstraints;	Включение или отмена процесса обработки дополнительной информации о наборе данных
Procedure EditKey;	Разрешается модификация буфера ключей поиска
Procedure EditRangeStart; Procedure EditRangeEnd;	Установка начала и конца диапазона отбора записей
Procedure EmptyDataSet;	Удаление всех записей из набора данных
Procedure Execute;	Запуск оператора SQL на сервере приложений
Procedure Fetch Blobs;	Получение значений Blob-полей от сервера приложений
Procedure FetchDetails;	Получение наборов данных, вложенных в текущий набор от сервера приложений
function FetchParams;	Инициализация свойства Params значениями, полученными от сервера приложений
function FindKey(const KeyValues: array of const): Boolean;	Поиск записи, содержащей указанные значения

продолжение

Таблица 9.9. Методы класса TClientDataSet(продолжение)

Метод	Назначение
Procedure FindNearest(const KeyValues: array of const);	Поиск записи, которая содержит значения, наиболее близкие к указанным
function GetCurrentRecord(Buffer: PChar): Boolean;	Копирование текущей записи в буфер (побайтово)
function GetFieldData(FieldNo: Integer; Buffer: Pointer): Boolean; function GetFieldData(Field: Tfield; Buffer: Pointer): Boolean;	Получение значения конкретного поля записи по ее номеру или описанию
function GetGroupState(Level: Integer): TgroupPosInds;	Положение текущей записи внутри указанной группы
Procedure GetIndexInfo(IndexName: String);	Получение информации о текущем индексе
Procedure GetIndexNames(List: TStrings);	Список названий всех доступных индексов
function GetNextPacket: Integer;	Получение следующего блока записей от сервера приложений
function GetOptionalParam(const ParamName: string): OleVariant;	Получение пользовательской информации, добавленной к набору данных
procedure GotoCurrent(DataSet: TClientDataset);	Установка указателя в наборе данных в такое же положение, в каком этот указатель находится в наборе данных, созданном с помощью метода CloneCursor
function GotoKey: Boolean;	Перемещение указателя к записи, определяемой текущим ключом
procedure GotoNearest;	Перемещение указателя к записи, наиболее близкой к определяемой текущим ключом
procedure LoadFromFile(const FileName: string -);	Загрузка набора данных из файла
procedure LoadFromStream (Stream: TStream);	Загрузка набора данных из потока
function Locate(const KeyFields: string; const KeyValues: Variant; Options: TlocateOptions): Boolean; function Lookup(const KeyFields: string; const KeyValues: Variant; const ResultFields: string): Variant;	Поиск записи
procedure MergeChangeLog;	Согласование изменений в протоколе с текущими данными набора
procedure Post;	Завершение транзакции. Запись данных в протокол или свойство Data
function Reconcile(const Results: OleVariant): Boolean;	Удаление информации об успешном внесении изменений из кэша
procedure RefreshRecord;	Обновление значений в текущей записи на основе набора данных сервера приложений
procedure RevertRecord;	Отмена всех модификаций, хранимых в протоколе, если они еще не переданы серверу приложений

Таблица 9.9- Методы класса TClientDataSet (продолжение)

Метод	Назначение
procedure SaveToFile(const FileName: string = ; Format TdataPacketFormat=dfBinary);	Сохранение набора данных в файле
procedure SaveToStream (Stream:TStream; Format TdataPacketFormat=dfBinary);	Сохранение набора данных в потоке
procedure SetKey;	Подготовка к модификации текущего установленного ключа или диапазона
procedure SetOptionalParam(const ParamName: string; const Value: OleVariant; IncludeInDelta: Boolean - False);	Добавление пользовательской информации к набору данных
procedure SetProvider(Provider:TComponent);	Указание объекта-поставщика
procedure SetRange(const StartValues, EndValues: array of const); procedure SetRangeStart; procedure SetRangeEnd;	Задание границ диапазона
function UndoLastChange(FollowChange: Boolean): Boolean;	Отмена последней операции редактирования, добавления или удаления записей
function UpdateStatus: TupdateStatus;	Получение информации о состоянии текущей записи

Создание многоуровневых приложений, работающих с СУБД с использованием транзакционного сервера MTS

Новые возможности

В одной из предыдущих глав рассматривалась технология создания распределенных приложений на основе объектов *MTS*, выполняющихся в оболочке сервера транзакций *MTS*. В системе *Delphi 7* имеется специальная возможность создания подобных объектов, ориентированных на работу с СУБД.

Рассмотрим следующий пример. Требуется создать серверный объект, который получает набор данных от СУБД (сервер баз данных и объект могут выполняться на разных компьютерах) и затем передает этот набор клиентскому приложению. Оно может использовать этот набор данных как обычный объект типа TDataSet с помощью произвольных элементов интерфейса, ориентированных на обработку данных.

ВНИМАНИЕ Недостаток использования сервера MTS для обработки баз данных заключается в том, что при каждом обращении к интерфейсу серверного объекта происходит его новое создание в адресном пространстве MTS, вследствие чего связь с СУБД каждый раз приходится организовывать заново, а это процесс достаточно медленный. С другой стороны, полное отключение обьекта MTS от СУБД после выполнения нужных действий позволяет снизить нагрузку на сервер баз данных. Применение того или иного подхода при создании многоуровневых приложений с доступом к данным во многом определяется характеристиками решаемой задачи, планируемой нагрузкой на сервер СУБД и транзакционный сервер, числом пользователей и другими факторами.

Процесс создания серверной и клиентской частей такого проекта во многом напоминает процесс создания уже рассмотренного приложения *MTS*.

Создание серверного объекта

Транзакционный модуль данных

Мастер создания транзакционных объектов Transactional object wizard в *Delphi 7* поддерживает две транзакционные технологии: *COM+* и *MTS*, — и способен создавать объекты на основе любой из них.

Выполним команду File ➤ New > Other (Файл ➤ Создать ➤ Другое) и на вкладке Multitier (Многоуровневые приложения) выберем значок Transactional Data Module (Транзакционный модуль данных). Соответствующий компонент во многом аналогичен компоненту MTS Object (Объект MTS), но имеет расширенные возможности поддержки работы с СУБД.

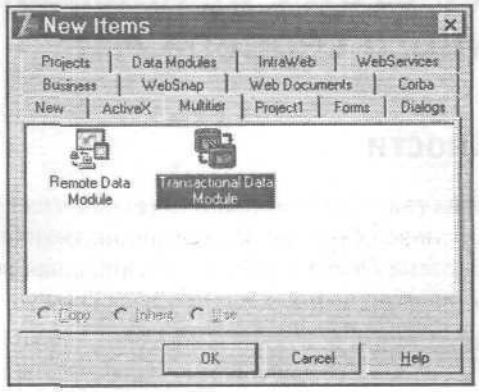


***Рис. 9.1.** Вызов Мастера создания транзакционного объекта*

После щелчка на кнопке OK Мастер создания транзакционного объекта попросит указать имя класса будущего объекта в текстовом поле CoClass Name (Имя Со-класса).

Допустим, это MTSDataTest. Значения остальных полей, о которых рассказывалось ранее, менять не надо. Управление транзакциями будет выполняться непосредственно в коде программы, при этом завершение транзакции сервера *MTS* автоматически повлечет за собой завершение всех операций, связанных с СУБД.

В соответствии с введенным именем объекта тип класса, реализующего работу данного объекта, получит имя TMTSDataTest, а интерфейс — IMTSDataTest.

Установление связи с СУБД

Предварительно требуется подготовить тестовую базу данных. Пусть это база данных *IBTest,* зарегистрированная в *BDE* под именем *IBMTS.* В ней присутствует таблица PROGRAM_LIST, состоящая из двух полей: ключевого поля ID и текстового поля - названия программы NAME длиной 80 символов. Эта база данных создана в формате СУБД *InterBase.* Для обращения к ней необходимо, чтобы был запущен сервер *InterBase.*

На вкладке Components (Компоненты) созданного объекта поочередно размещаются следующие компоненты.

1. Компонент TDatabase. Его надо настроить так, чтобы он установил связь с базой данных IBTest. В качестве псевдонима AliasName указывается IBMTS, а в качестве названия базы (DatabaseName) — название IBTEST.

   Данный компонент нужен для того, чтобы устанавливать связь с сервером InterBase, которому в исходном тексте программы передаются параметры (имя пользователя и пароль); в противном случае (например, если использовать только компонент TTable) при каждом обращении к объекту *MTS* будет открываться диалоговое окно с запросом соответствующих прав доступа.

   Поэтому для созданного объекта Database1 надо определить обработчик события OnLogin. Он выглядит следующим образом.

   ```
   procedure TMTSDataTest.Database1Login(Database:
    TDatabase; LoginParams: TStrings);
   begin
      LoginParams.Values['USER NAME'] := 'SYSDBA';
      LoginParams.Values['PASSWORD'] := 'masterkey';
      ObjectContext.SetComplete;
   end;
   ```

   Завершать каждый метод класса TMSTDataTest должен следующий оператор:

   ```
   ObjectContext.SetComplete;
   ```

   Он сообщает серверу *MTS,* что данная подпрограмма успешно завершила свою работу. Это требуется, чтобы сервер мог корректно обработать соответствующую транзакцию.

2. Компонент TTable. В свойстве DatabaseName выбирается значение IBMTS, а в свойстве TableName (при этом может потребоваться подключение к СУБД) — таблица PROGRAM_LIST.

3. Компонент TDataSetProvider (панель OataAccess). Данный компонент необходим в качестве промежуточного звена (сервера приложений) при использовании серверного модуля, обрабатывающего наборы данных в СУБД. Он выполняет функции передачи таких наборов от сервера клиентской программе и обратно. Пользователь на своем компьютере работает с локальной копией запрошенного набора данных, например внося в него изменения. Затем он передает набор данных обратно серверу приложений TDataSetProvider, который выполняет анализ сделанных изменений (он может, в частности, накапливать небольшие порции изменений) и передает результирующий набор серверу баз данных.

Созданному объекту DataSetProvider1 надо задать только новое значение свойства DataSet (Набор данных) — строку Table1, ссылку на объект-таблицу. Теперь в модуле установлена связь TDatabase ➤ TTable > TDataSetProvider. Общение клиентской программы и сервера *MTS* с сервером InterBase выполняется именно объектом — поставщиком данных.

ВНИМАНИЕ Ни один из этих компонентов не следует делать активным (свойство Active не должно равняться True). Сервер MTS при каждом обращении к соответствующему интерфейсу создает новую копию серверного объекта, который после выполнения нужных действий уничтожается. Иными словами, в промежутках между запусками никакой дополнительной информации о предыдущих состояниях серверного объекта не запоминается. Поэтому сохранение информации о текущей (созданной во время проектирования программы) связи с СУБД невозможно — программа каждый раз запрашивает информацию о подключении заново. Все же, в зависимости от особенностей используемой СУБД и серверной архитектуры, возможны и исключения.

Проект надо сохранить и перейти к формированию интерфейса.

Создание интерфейса

У серверного объекта *MTS*, предназначенного для работы с СУБД, должны быть как минимум две функции внешнего интерфейса: одна — для передачи клиентскому приложению набора данных от СУБД, а вторая — для внесения в СУБД изменений, сделанных пользователем в локальной копии набора данных.

Первый метод интерфейса — назовем его GetCustomRecords — должен иметь хотя бы один параметр, через который возвращается число реально передаваемых записей набора данных. Для этого надо создать метод с помощью Редактора интерфейсов и добавить новую строку в списке Parameters (Параметры) на одноименной вкладке с помощью кнопки Add (Добавить). В этой строке надо указать: имя параметра - RecsOut, тип — Integer, а также задать в качестве модификатора значение out, действие которого аналогично действию ключевого слова var в Паскале, с тем исключением, что внутри метода данный параметр не может применяться, кроме как для изменения его значения. Слово out означает, что при выполнении метода

GetCustom Records в переменную, указанную в качестве первого параметра, заносится некоторое значение. Тем самым, в качестве первого параметра может быть указана только переменная типа integer.

Значение, возвращаемое самим методом, должно иметь тип OleVariant. Это специальный тип, предназначенный для передачи упакованного содержимого набора данных (рис. 9.2).

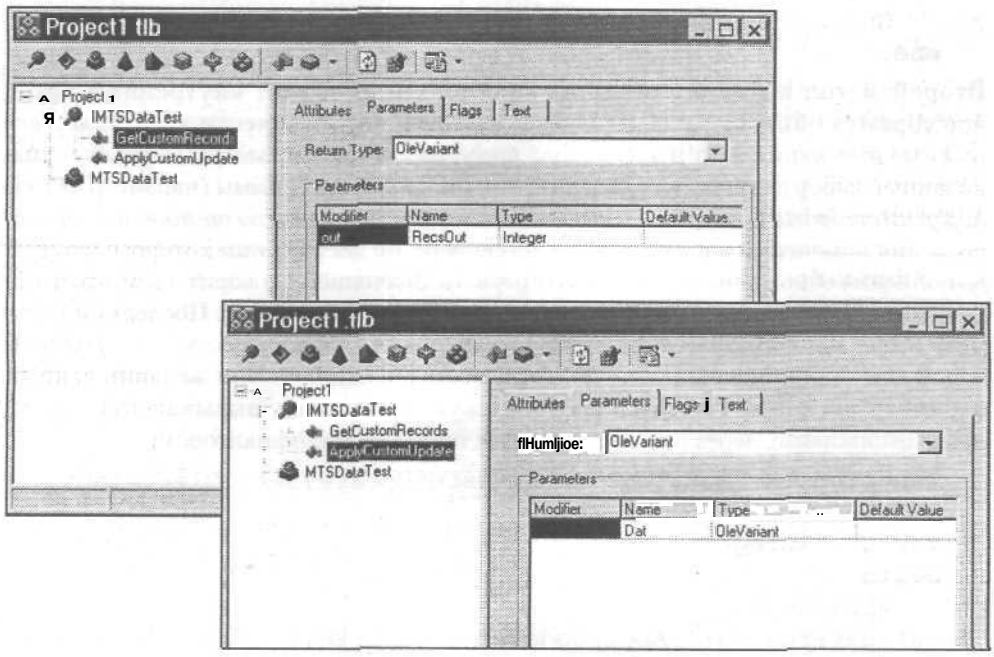


РИС. 5.2. *Определение интерфейса метода, предназначенного для получения набора записей и для передачи измененного набора записей*

Следующим добавляется метод ApplyCustomUpdate (название выбирается произвольно). Он должен получить от клиентской программы упакованный набор данных (тип OLeVariant) как параметр и вернуть его серверу в качестве значения (рис. 9.2).

Далее нужно зафиксировать сделанные изменения в Редакторе интерфейсов с помощью кнопки Refresh Implementation (Обновить реализацию).

Реализация интерфейса

Теперь необходимо описать, что же делают эти методы. Для этого надо перейти в редактор исходных текстов и найти там функцию GetCustomRecords. Она должна состоять из трех операторов. Первый выполняет переход к начальной записи набора данных; второй — записывает упакованное содержимое набора (свойство Data объекта DataSetProvider1) в стандартную переменную Result, возвращающую значе-

ние функции; третий оператор — стандартная команда успешного завершения работы метода серверного объекта *MTS*.

```
function TMTSDataTest.GetCustomRecodrs(out RecsOut:
Integer): OleVariant;
begin
   DataSetProvider1.DataSet.First;
   Result := DataSetProvider1.Data;
   ObjectContext.SetComplete;
end;
```

Второй метод выглядит столь же просто. Он вызывает внутренний метод ApplyUpdates объекта DataSetProviderl, который автоматически выполняет все нужные изменения в СУБД. У метода ApplyUpdates три параметра. Первый — упакованный набор данных, поступивший от клиентской программы (параметр метода ApplyCustom Update). Второй — максимально допустимое число ошибок в процессе внесения изменений в набор данных на сервере, по достижении которого процесс выполнения обновлений необходимо прервать. Значение -1 говорит о том, что необходимо выполнить все изменения независимо от числа ошибок. Последний параметр имеет модификатор out поэтому на его месте должна указываться переменная. В нее запишется общее число встретившихся ошибок. При желании данный параметр несложно передать в клиентскую программу, вызывающую метод ApplyCustomUpdate, через новый параметр с таким же модификатором:

```
function TMTSDataTest.ApplyCustomUpdate(Dat: OleVariant):
OleVariant;
var EC: integer;
begin
   ApplyCustomUpdate :=
DataSetProvider1.ApplyUpdates(Dat,-1,EC);
   ObjectContext.SetComplete;
end;
```

На этом создание серверного объекта завершено. Все изменения надо сохранить и выполнить регистрацию созданного интерфейса кнопкой Register Type Library (Зарегистрировать библиотеку типов) на панели Редактора интерфейсов. При этом выполнится компиляция созданного проекта и появится сообщение об успешной регистрации сервера *ActiveX*.

Созданный объект надо подключить к серверу *MTS*, который должен быть уже запущен. Это выполняется командой системы *Delphi 7* Run ► Install MTS Objects (Запуск > Установить объекты MTS). Мастер инсталляции запросит название пакета, в который данный объект устанавливается на сервере *MTS*. Можно ввести, например, название MTSDataTestPack. Более подробно процесс установки серверного компонента на сервер *MTS* был описан ранее.

Создание клиентского приложения

Для клиентского приложения мы рассмотрим два подхода к работе с серверным объектом: использование стандартного элемента управления TDBGrid с панели Data

Controls (Элементы управления данными), связанного с базой данных визуальным способом, и программный вызов методов, позволяющих вносить изменения в таблицу базы данных.

К текущей группе проектов добавляется новый проект Project2, а на форме размещаются следующие компоненты:

- текстовое поле для задания нового значения элементу таблицы базы данных (PROGRAM_LIST);
- кнопка для занесения этого значения в таблицу;
- кнопка для выхода;
- элемент управления TDBGrid для автоматического отображения содержимого таблицы.

Клиентский набор данных

Как уже говорилось, клиентское приложение работает с копией набора данных (в данном случае — с копией таблицы PROGRAM_LIST), которая получает свое наполнение от сервера в упакованном виде. В системе *Delphi 7* имеется уже рассмотренный компонент TClientOataSet (панель DataAccess), предназначенный именно для такого использования. Его надо разместить на форме приложения, пока что не изменяя значений свойств.

Для того чтобы к этому клиентскому набору данных могли обращаться элементы управления с панели Data Controls (Элементы управления данными), необходимо предварительно подготовить промежуточный источник данных (компонент TDataSource), указав в его свойстве DataSet значение ClientDataSet1 (название клиентского набора данных, сгенерированное по умолчанию). Теперь на форму можно поместить компонент TDBGrid (рис. 9.3). Его свойство DataSource должно иметь значение DataSource1 (автоматически созданное название размещенного на форме источника данных).

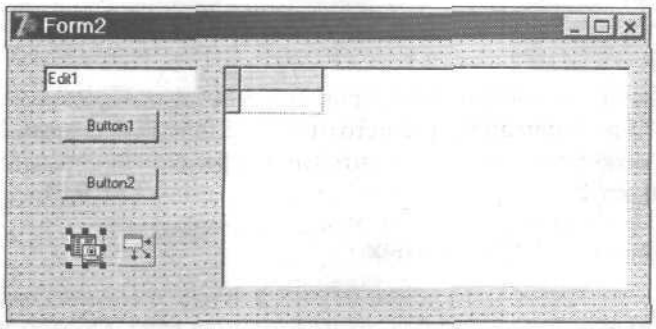


Рис. 9.3. Подготовленная форма для работы с базой дачных через сервер MTS

Подготовка клиентского приложения к работе

В момент создания главного окна клиентской программы надо установить связь с сервером *MTS* через интерфейс IMTSDataTest и получить набор данных для объекта

ClientDataSet1 с помощью метода GetCustomRecords. Для этого необходимо сделать следующее.

1. Подключить новые модули к части реализации:

```
implementation uses MtsObj , Mtx, ComObj, Project1_TLB;
```

2. Описать две переменные, первая из которых, MTSTD, обеспечивает интерфейс IMTSDataTest, а вторая, TransactionContextEx, описывает связь с сервером *MTS*:

```
var MTSTD: IMTSDataTest; TransactionContextEx:
  ITransactionContextEx;
```

3. Реализовать обработчик события главной формы OnCreate. Результат, возвращаемый методом GetCustomRecords, имеет тип OleVariant. Его необходимо присвоить свойству Data клиентского набора ClientDataSetl, после чего с ним можно работать как с обычным набором данных. При этом связь элемента управления DBGrid1 через источник данных DataSourcel установится автоматически. В переменную RO запишется число реально полученных записей таблицы:

```
procedure TForm2.FormCreate(Sender: TObject);
var RO: integer;
begin
   TransactionContextEx := CreateTransactionContextEx;
   OleCheck(TransactionContextEx.CreateInstance(
     CLASS_MTSDataTest,IMTSDataTest,MTSTD));
   ClientDataSetl. Data  := MTSTD.GetCustomRecords(RO)
end;
```

Для обработчика щелчка на кнопке Button2 достаточно указать команду закрытия.

```
procedure TForm2.Button2Click(Sender: TObject);
begin
   Close
end;
```

Теперь приложение можно откомпилировать и запустить. Процесс установки связи с сервером *MTS* и получения через него от СУБД *InterBase* содержимого таблицы PROGRAM_LIST может потребовать значительного времени. Все значения из этой таблицы отображаются в элементе DBGridl автоматически (рис. 9.4).

Редактирование набора данных

Допустим, что по щелчку на кнопке Button1 в текущую запись набора данных в поле NAME надо внести значение из текстового поля Edit1. Обработчик щелчка на кнопке Buttonl запишется следующим образом.

```
procedure TForm2.Button1Click(Sender: TObject);
begin
   // переход к первой записи в наборе
  ClientDataSet1.First;
```

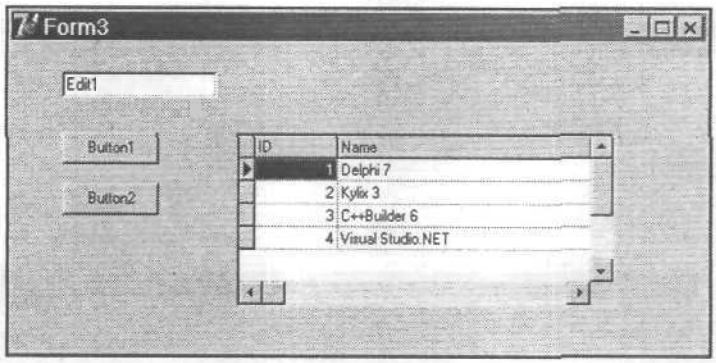


Рис. 9.4. Доступ к СУБД InterBase через сервер MTS

```
  // режим редактирования
   ClientDataSet1.Edit;

   // полю таблицы с именем NAME
   // присваивается значение из поля Edit1
   ClientDataSet1['NAME'] := Edit1.Text;

   // изменения сохраняются и фиксируются
   ClientDataSet1.Post;

   // измененный набор данных отсылается серверу MTS,
   // чтобы он внес нужные изменения в базу данных
InterBase
   MTSTD.ApplyCustomUpdate(ClientDataSet1.Delta);
end;
```

В качестве параметра метода ApplyCustomUpdate указывается свойство Delta клиентского набора данных. Оно представляет собой упакованное содержимое этого набора в формате OleVariant.

ВНИМАНИЕ Хотя элемент управления TDBGrid позволяет редактировать содержимое, по окончании редактирования необходимо обновлять набор данных серверного компонента, вызывая метод ApplyCustomUpdate.

Теперь можно запустить программу, ввести в поле значение Win2000 и щелкнуть на кнопке Button1. Если теперь запустить вторую копию этого приложения (Project2.exe), то в таблице появится уже измененное значение первой записи таблицы (рис. 9.5).

Чтобы в дальнейшем приложения работали с более-менее согласованными значениями записей, хранящихся в таблицах базы данных, текущее содержимое клиентского набора данных надо периодически обновлять, например путем повторных вызовов метода GetCustomRecords.

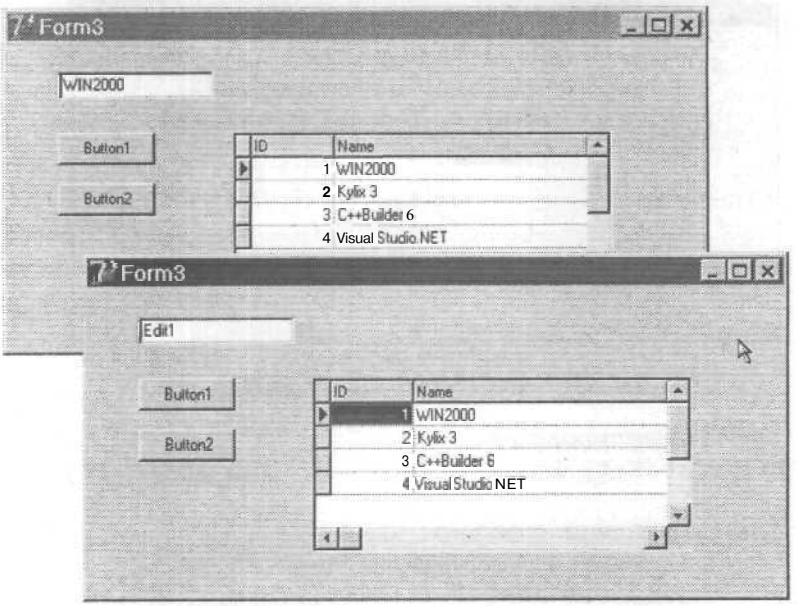


Рис. 9-5. Одновременная работа с таблицей базы данных

Оригинальные возможности Delphi по созданию многоуровневых приложений

Визуальное создание распределенных приложений с доступом к данным

Мы только что рассмотрели пример, в котором доступ к данным осуществлялся с помощью серверного объекта, установленного на сервере транзакций *MTS*. При этом пришлось прибегнуть к ручному кодированию, чтобы получить доступ к нужному набору данных из клиентского приложения. Используя другие технологии, можно обойтись вообще без ручного программирования и без развертывания службы *MTS*. Для этого применимы, например, такие технологии, как *CORBA*, *OLEnterprise*, технологии Интернета и др. Но все они также требуют установки дополнительного программного обеспечения (серверных и клиентских приложений *CORBA* или *OLEnterprise*), которое не входит в поставку системы *Delphi 7*, а распространяется отдельно. Единственное исключение — технология *DCOM*, которая всгроена в *Windows 98*, *Windows2000* и *Windows NT*.

Далее будет рассмотрен ряд примеров визуального создания приложений, обращающихся к базам данных в сети с помощью всех этих технологий — *DCOM*, *CORBA*, Интернета и новых средств *Windows 2000* и *Microsoft.NET*. Эти примеры основываются на наборе компонентов, расположенных на панели DataSnap.

Создание многоуровневого приложения COM

Технология *DCOM* реализована только для платформ *Windows*, поэтому сервер будет представлять собой обычное приложение *Windows*. Оно создается командой File ➤ New Application (Файл > Создать приложение). Далее в проект необходимо добавить новый модуль данных, обеспечивающий удаленную связь с клиентскими программами. Это выполняется командой File > New > Other (Файл ➤ Создать ➤ Другое) с последующим выбором на панели Multitier (Многоуровневые приложения) значка Remote Data Module (Удаленный модуль данных).

В ответ на запрос названия интерфейса этого модуля можно ввести, например, имя DCOMTest и щелкнуть на кнопке OK (рис. 9.6).

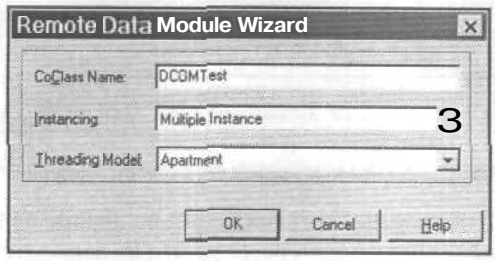


Рис. 9.6. *Задание основных характеристик удаленного модуля данных*

В окне модуля данных размещаются следующие компоненты.

1. Компонент TTable. В качестве базы данных (свойство DatabaseName) указывается IBTest (псевдоним IBMTS), а в свойстве TableName (название таблицы) – PROGRAM_LIST.

ПОДСКАЗКА В самом начале можно добавить компонент TDatabase и сформировать обработчик события OnLogin, чтобы не задавать каждый раз параметры доступа к СУБД (имя пользователя и пароль) вручную.

   Делать таблицу активной (устанавливать свойство Active равным True) не надо.

2. Компонент TDataSetProvider. Это основной компонент, выполняющий всю рутинную работу сервера приложений. В его свойстве DataSet надо указать значение Table1, после чего в объекте Table1 свойство Active сделать равным True. Связь с СУБД в данном случае можно установить заранее, так как сервер будет постоянно находиться в памяти компьютера, обслуживая запросы пользовательских программ. В этом его работа существенно отличается от работы сервера *MTS*, который для каждого запроса клиентской программы запускает новый процесс (создает новую копию объекта *COM*).

На этом создание сервера закончено! Теперь его надо зарегистрировать в системе — в той версии *Windows*, которая установлена на компьютере, где сервер приложений будет выполняться. Для этого он запускается с параметром /regserver. При запуске из

системы *Delphi 7* параметр указывается в поле Parameters (Параметры) диалогового окна, вызываемого командой Run ➤ Parameters (Запуск > С параметрами), как показано на рис. 9.7.

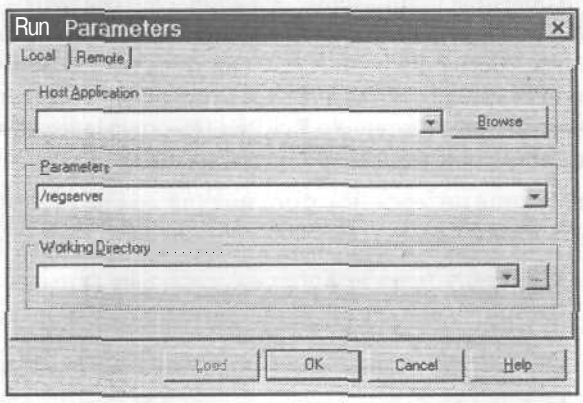


***Рис. 9.7.** Регистрация сервера путем его запуска с параметрами*

Регистрация проходит незаметно. Никаких окон на экране не появляется, просто соответствующая информация о новом сервере *DCOM* заносится в Реестр *Windows,* Для того чтобы сервер выполнялся как обычное приложение, параметр /regserver надо удалить из строки запуска.

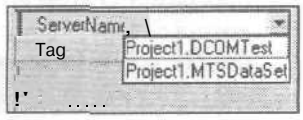
ЗАМЕЧАНИЕ Чтобы отменить регистрацию [удалить информацию о регистрации из системы), надо запустить соответствующий сервер с параметром /unregserver.

Создание клиентской программы

К текущей группе проектов добавляется новый проект. На форме последовательно размещаются следующие компоненты.

1. Компонент TDCOMConnection с панели DataSnap. Он обеспечивает все аспекты связи пользовательской программы с сервером приложений на основе технологии *DCOM.* Если щелкнуть на его свойстве ServerName, то в раскрывшемся списке отобразится набор доступных серверов *COM.* Среди них может быть и объект, зарегистрированный в *MTS,* если он не был удален из сервера транзакций.

   Если изменить значение свойства Connected (установка связи с сервером приложений), присвоив ему значение True, то сервер приложений (программа Project1.exe) запустится автоматически. Закроется она тоже автоматически, если значение свойства Connected опять сделать равным False.



2. Компонент TClientDataSet с панели Data Access. Б его свойстве RemoteServer (Удаленный сервер) указывается название объекта DCOMConnection1, а при щелчке на свойстве ProviderName (Название поставщика) раскрывшийся список содержит единственный пункт DataSetProvider1 — это имя удаленного поставщика, объекта DataSetProvider1, размещенного в модуле данных предыдущего проекта. При его выборе автоматически запустится сервер приложений. Закрывать сервер нельзя, так как тогда связь с этим объектом *COM* прервется.

ЗАМЕЧАНИЕ

После того как разработчик создал запрос на подключение к серверам DCOM (обратился к свойству ProviderName), система Delphi 7 самостоятельно проделала все нужные действия для получения от Windows списка зарегистрированных серверов DCOM, который хранится в Реестре. Когда на основании этой информации выбран конкретный поставщик, соответствующий сервер автоматически запускается с помощью протокола COM,

3. Компонент TDataSource. Как уже говорилось, данный компонент является стандартным при разработке приложений, работающих с базами данных. Он позволяет элементам управления с панели Data Controls (Элементы управления данными) работать с СУБД независимо от типа исходного набора (источника) данных. В его свойстве DataSet указывается название набора ClientDataSetl.

4. Элемент управления TDBGrid. Он предназначен для формирования пользовательского интерфейса. Разместив на форме компонент TDBGrid и указав в его свойстве DataSource значение DataSource1 (объект, соответствующий предыдущему компоненту), надо сделать равным True свойство Active объекта ClientDataSetl (клиентского набора данных). При этом сервер приложений запросит имя пользователя и пароль для подключения к СУБД *InterBase*. После ввода соответствующих значений в элементе DBGrid1 на форме появится содержимое таблицы PROGRAM_LIST (рис. 9.8).

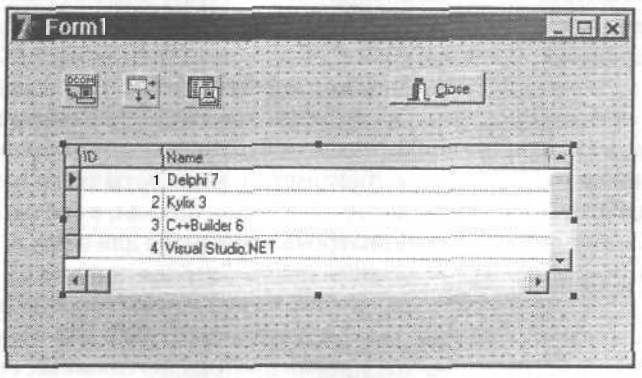


***Рис. 9.8.** Доступ к данным с помощью технологии DCOM*

Данное клиентское приложение можно откомпилировать и запустить, возможно в нескольких копиях.

Компонент Простой брокер объектов (TSimpleObjectBroker)

С помощью данного компонента, представляющего собой несложный пример брокера распределенных объектов, опрашиваются функциональные способности серверов приложений. Это полезно, например, в случае отказа одного из компьютеров сети. Если возможности приложения, работавшего на отказавшем компьютере, использовались через брокер, он автоматически формирует связь этого объекта с работоспособным сервером, выбирая его в списке доступных в сети серверов по некоторому алгоритму.

Этот список (свойство Servers, тип TServerCollection) представляет собой коллекцию объектов TServerItem. Если значение свойства LoadBalanced имеет значение True, брокер предлагает объекту случайный сервер из числа доступных, в противном случае брокер выберет самый первый сервер из списка. Таким образом, данный компонент представляет собой несложную систему балансировки загрузки. На его основе можно создавать свои собственные брокеры, специфичные для конкретных задач.

Протоколы Интернета

Содержимое последующих глав книги опирается на базовые знания об устройстве Интернета и его основных технологиях и протоколах.

Интернет — это глобальная компьютерная сеть, в которой каждый компьютер имеет свой адрес (так называемый *IP-адрес)* и может передавать и принимать сообщения от компьютеров с другими *IP*-адресами. Такой адрес обычно записывается четырьмя числами со значениями от 0 до 255, разделенными точками, например:

```
127.0.0.1
194.100.95.20
```

Подключение к Интернету в домашних условиях обычно осуществляется с помощью модема, в служебных целях чаще используется выделенная линия, позволяющая принимать и передавать большие объемы информации. Компьютеры, подключенные к Интернету, обмениваются данными, которые можно трактовать по-разному в зависимости от *сетевого уровня,* на котором эти данные рассматриваются. Низкие уровни ориентированы на конкретные особенности оборудования, а высокие — на конкретные области применения. Каждому уровню соответствует понятие *сетевого протокола,* который представляет собой набор правил, в соответствии с которыми программы формируют последовательности команд для обмена сообщениями. Низкоуровневые протоколы предназначены для передачи простых потоков байтов компьютеру с указанным *IP*-адресом, высокоуровневые протоколы позволяют передавать текстовые и двоичные файлы, страницы *HTML* для просмотра в программах-броузерах и прочие объекты. Уровни протоколов связаны с сетевыми уровнями. Наиболее полно описывает семь сетевых уровней модель взаимодействия открытых систем *ISO/OSI,*одобренная Международным институтом по стандартизации. Рассмотрим каждый из них более подробно.

Физический уровень

Компьютеры на уровне линий связи и сетевого оборудования, преобразующие сигналы в цифровое представление и обратно, обмениваются битами.

Уровень соединения

На уровне соединения, который чаще всего определяется сетевыми картами компьютера, происходит выделение блоков битов в *кадры,* которые считаются стандартными в рамках используемой локальной или глобальной сети. На этом уровне также дополнительно осуществляется контроль ошибок передачи информации.

Сетевой уровень

Сетевой уровень отвечает за передачу по сети кадров, снабженных дополнительными сведениями о приемнике и получателе информации, хранимой в кадре. Такой блок информации называется *пакетом.* В Интернете для реализации функций сетевого уровня используется протокол *Internet Protocol (IP).*

Транспортный уровень

Транспортный уровень отвечает за доставку полученных пакетов конкретной программе и за отправку пакетов — передачу их на сетевой уровень. Начиная с этого высокого уровня все блоки информации называются *сообщениями.*

Сеансовый уровень

Здесь происходит формирование сеансов связи между двумя компьютерами (например, в момент подключения программы к сети) — выполняется настройка сеанса связи на физические и программные характеристики сети. В Интернете функции сеансового уровня выполняет транспортный уровень.

Уровень представления

Данный уровень отвечает за способ представления данных, поступающих к пользователю («приемником» данных может служить монитор, принтер или файл). Официально в Интернете уровня представления не существует.

Прикладной уровень

На этом уровне создаются приложения: программы электронной почты, броузеры, системы распределенной обработки данных и другие. Практически все приложения, рассматриваемые в настоящей книге, относятся к данному уровню.

Технологии Интернета базируются на наборе протоколов *TCP/IP.* Здесь *TCP* — *TransportControl Protocol* — протокол транспортного управления. Этот набор протоколов охватывает все аспекты передачи и обработки информации в глобальной сети. Часто технологию *TCP/IP* называются просто протоколом *TCP/IP,* хотя в общем случае это больше, чем просто протокол.

ЗАМЕЧАНИЕ На основе технологии TCP/IP можно создавать и локальные сети, тогда они называются корпоративными сетями интранет.

Скорее, имеет смысл говорить о технологии *TCP/IP* для передачи информации в сетях. В частности, на трансаортном уровне в *TCP/IP* можно выделить два основных протокола (в реальности их больше): *TCP,* ориентированный на соединение, и *UDP (User Datagram Protocol, протокол пользовательских датаграмм).* В первом случае перед передачей данных требуется установить связь между начальной и конечной точками маршрута. Во втором случае предварительной установки соединения не требуется. Данные разбиваются на независимые блоки (датаграммы) и просто посылаются по конкретному адресу. Расчет делается исключительно на возможности сети, не ожидается даже уведомлений о неудачных попытках доставки. Более того, датаграммы не всегда поступают к программе-получателю в том же порядке, в каком они были посланы. Упорядочивать их должно само клиентское приложение.

Другими словами, протокол *TCP* чаще всего используют, когда требуется обеспечить надежный уровень передачи информации, а протокол *UDP* применяют, если главным требованием является высокая скорость передачи данных по сети, а требования к качеству передачи невысоки.

Понятие порта

Большинство компонентов, ориентированных на создание приложений Интернета, используют понятие *порта.* По аналогии с аппаратными портами компьютера (последовательные порты *COM, к* которым обычно подключают модем или мышь, параллельные порты *LPT,* к которым подключают принтер или электронный ключ защиты) в *Windows* существует понятие *программного порта,* применяемое на транспортном сетевом уровне. К таким портам «подключаются» программы Интернета, чтобы передать или принять определенную информацию (например, файл или электронное письмо).

Такие порты имеют свои номера, а их назначение описано в файле SERVICES, расположенном в каталоге WINDOWS. Например, порт 21 используется для работы протокола передачи файлов *FTP.* Большинство номеров портов зарезервировано для стандартных служб, а те номера, которые отсутствуют в списке, можно использовать для собственных нужд (например, для отладки сетевых программ). При этом используемый номер желательно выбрать достаточно большим, чтобы исключить совпадения со стандартными службами.

Создание многоуровневого приложения TCP/IP

В системе *Delphi 7* работа с протоколом *TCP/IP* реализована на основе технологии сокетов *(Sockets).* Сокеты позволяют работать с протоколом *TCP/IP* на уровне про-

граммных интерфейсов (*API*), обращаясь к возможностям *TCP/IP* непосредственно из прикладной программы.

Каждый сокет представляет собой виртуальное окончание сетевого соединения с другим компьютером, сформированное конкретной прикладной программой. Другими словами, сокет — это аналог потока (или просто файла), который надо открыть (установить соединение). Затем в него можно записывать данные (посылать их на другой компьютер) или считывать их (принимать от другого компьютера). По окончании работы сокет надо закрыть.

ЗАМЕЧАНИЕ В Windows робота с сокетами обеспечивается технологией Winsock, реализованной в виде библиотеки Winsock.dll. Для работы с серверами приложений в сетях TCP/IP в поставку Delphi 7 входит дополнительное программное обеспечение.

Чтобы организовать связь клиентского приложения с сервером по протоколу *TCP/IP,* надо использовать компонент TSocketConnection.

В свойстве Address указывается *IP*-адрес удаленного компьютера, где работает сервер приложений (при отладке можно указать адрес локального компьютера, 127.0.0.1). Важнейший параметр — свойство ServerGUID, хранящее идентификационный номер (*GUID*) сервера приложений, с которым надо установить связь. Данный номер генерируется при создании объекта DCOMConnection1. Переключившись на этот объект (который должен быть неактивным — значение свойства Connected равно False), можно скопировать содержимое свойства ServerGUID в буфер обмена и затем вставить его в одноименное свойство объекта SocketConnectionl. При этом в свойстве ServerName автоматически появится имя соответствующего сервера - Project1.DCOMTest.

Для организации связи по протоколу *TCP/IP с* помощью сокетов на компьютере-сервере надо запустить специальную программу, ответственную за обмен данными с сервером приложений через сокеты. Эта программа называется SckfSrvr.exe и расположена в каталоге \8in, вложенном в папку, куда была установлена система *Delphi 7.* После ее запуска эта программа отображается в виде значка на панели индикации *Windows.*

Окно программы можно развернуть двойным щелчком на этом значке. В поле GUID необходимо внести значение, указанное в свойстве ServerGUID, и щелкнуть на кнопке Apply (Применить) (рис. 9.9).

Менять значения полей в программе SckfSrvr.exe можно, только если она запушена не в режиме регистрации. Флажок меню Connections ➤ Registered Objects Only (Соединения > Только регистрация объектов) должен быть сброшен. Если этот флажок установлен, его надо сбросить, программу закрыть и перезапустить в нормальном режиме.

Затем свойство Connected объекта SocketConnectionl надо сделать равным True. Если сервер приложений не был запущен ранее, он при этом запустится автоматически. В свойстве RemoteServer объекта ClientDataSetl указывается новый вид связи

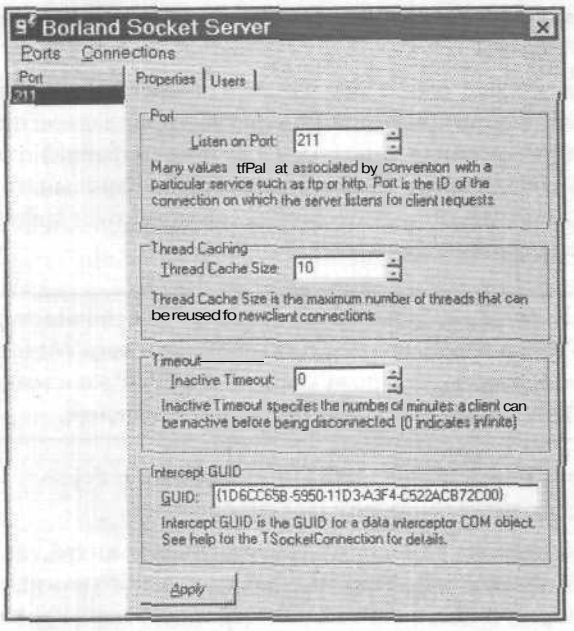


Рис. 9.9. Настройка сервера сокетов

SocketConnection1, после чего данный объект можно сделать активным, занеся в свойство Active значение True.

ВНИМАНИЕ В этот момент надо переключиться на сервер приложений, который запросит имя пользователя и пароль для подключения к СУБД.

В элементе управления DBGrid1 на форме сразу же отобразится содержимое таблицы PROGRAM_LIST. Теперь клиентский проект готов к компиляции и запуску. Копии исполняемого файла могут запускаться на разных компьютерах. Требуется только, чтобы эти компьютеры были соединены по протоколу *TCP/IP, в* программах стояли правильные значения *IP*-адресов, а на сервере была также запущена программа ScktSrvr.exe.

Создание многоуровневого приложения HTTP

Созданная система поддерживает связь сервера с клиентскими программами с помощью технологий *DCOM* или *TCP/IP,* которые позволяют напрямую обмениваться информацией между приложениями через сеть. Б ряде случаев подобный подход может иметь существенные недостатки, так как требует наличия прямой сетевой связи между клиентскими компьютерами и сервером. В крупных корпоративных сетях, которыми пользуются десятки и сотни пользователей, предъявляются повышенные требования к безопасности и защите передаваемой информации. В подоб-

ных ситуациях для связи с сервером приложений лучше использовать компонент TWebConnection с панели Data Snap.

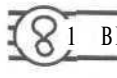
Данный компонент организует связь клиентских программ *с* сервером по протоколу *HTTP,* активно применяемому в Интернете. Этот протокол имеет встроенные механизмы безопасности и позволяет, в частности, применять *брандмауэры (firewall)* для защиты от попыток несанкционированного проникновения в локальную сеть.

Чтобы клиентское приложение могло общаться с сервером по протоколу *HTTP* через брандмауэр, что гарантирует высокую степень защиты данных, необходимо, чтобы в сети был установлен *Web*-сервер. Данные от сервера приложений поступают к клиентским программам и обратно через *Web*-сервер и средства защиты. В такой ситуации *Web-сервер* выступает в роли клиентской программы для сервера приложений, а для клиентских программ конечных пользователей — в роли полноценного сервера, что добавляет системе дополнительный уровень надежности.

Для реализации подобной технологии в системе *Delphi* появился новый компонент TWebConnection. Разместив его на форме клиентского приложения, в его свойстве ServerGUID надо указать идентификатор зарегистрированного сервера приложений Project1.DCOMTest, как описано в предыдущем разделе. При этом соответствующее имя сервера должно автоматически появиться в свойстве ServerName.

Работа компонента TWebConnection основывается на специальной библиотеке сервера *HTTP* httpsrvr.dll, которая обрабатывает запросы *HTTP* и преобразует их в обращения к серверу приложений *COM.* Данную библиотеку (она, как и программа ScktSrvr.exe, расположена в каталоге \Bin системы *Delphi 7)* надо скопировать в виртуальный каталог *Web*-сервера, допускающий вызовы серверных приложений (например, в каталог \cgi-bin). В свойстве URL объекта WebConnection1 надо указать полный адрес данной библиотеки, например:

```
http://testbed/cgi-bin/httpsrvr.dll
```



ВНИМАНИЕ К этому времени Web-сервер должен быть запущен.

Теперь осталось изменить свойство Connected объекта WebConnection1, присвоив ему значение True. Произойдет запуск сервера приложений. Далее надо указать в свойстве RemoteServer объекта ClientDataSet1 значение WebConnection1 и сделать этот объект активным, занеся значение True в свойство Active. Сервер приложений запросит имя пользователя и пароль для подключения к СУБД *InterBase, и* после их ввода в элементе DBGrid1 на форме появится содержимое таблицы PROGRAM_LIST.

В *Delphi 7 имеется* новый компонент TSOAPConnection, расширяющий возможности TWebConnection. Он поддерживает протокол *SOAP* (высокоуровневый протокол на базе *XML*), позволяя использовать в качестве транспортных протоколов не только *HTTP,* но и *SSL,* что обеспечивает шифрование информации при передаче. Для использования TSOAPConnection достаточно указать в свойстве URL строку, которая определяет адрес *SOAP*-сервера. Такой сервер должен предоставлять интерфейс IAppServerSOAP в виде услуги *Web Services* (о них рассказывается в следующих гла-

вах), чтобы клиентские приложения могли программно обращаться к серверу (удаленному поставщику данных).

Создание многоуровневого приложения ADO

В систему *Delphi* 7 включен специальный компонент TRDSConnection (наследник класса TCustomConnection), который играет роль компонента TDataSetProvider, выступая в качестве сервера приложений. Он используется, когда надо объединять и обрабатывать данные, расположенные на разных компьютерах сети и формируемые разными программными процессами.

В таких случаях этот компонент надо применять вместо TADOConnection, а для связи с ним в компонентах—наследниках класса TADODataSet следует обращаться к свойству RDSConnection (вместо свойства Connection).

Если не планируется использовать специализированные программные компоненты, выполняющиеся на других компьютерах, а требуется только организовать доступ к распределенным данным, то значение свойства ServerName, которое определяет специальный объект—фабрику данных (по аналогии с фабрикой объектов *COM),* можно оставить без изменений. Этот объект выполняет запрос к данным с помощью своего метода Query, который в качестве параметров обращается к свойствам ConnectionString или CommandText (оператор *SQL* или название свойства объекта *ADO,* которое будет хранить результирующий набор данных).

Б первом случае (оператор *SQL)* для получения итогового набора данных надо обратиться к следующему методу:

```
function GetRecordset(const CommandText: WideString;
   ConnectionString: WideString = ''): _Recordset;
```

Например:

```
ADODataSet1.CommandText := 'SELECT * FROM Employee';
ADODataSet1.RecordSet :=
RDSConnection1.GetRecordset(ADODataSet1.CommandText,'');
```

Использование множественных удаленных модулей данных

В одном проекте *Delphi У* допускается использовать несколько модулей данных, что упрощает групповую разработку приложений. Для того чтобы к наборам данных можно было обращаться через сервер приложений (как было описано выше для одного модуля), достаточно просто добавлять такие модули в проект командой File x New > Other (Файл > Создать > Другое). На вкладке Multitier выберите значок Remote Data Module (Удаленный модуль данных).

ВНИМАНИЕ Не путайте эту команду с командой File ➤ New ➤ Data Module [Файл ➤ Создать ➤ Модуль данных), которая создает локальный модуль данных, не предназначенный для использования в распределенной системе. Команда создания удаленного модуля данных важно тем, что автоматически формирует интерфейс этого модуля, доступный из клиентских программ, и генерирует исходный код, регистрирующий этот интерфейс. При этом в свойствах Remote Server или Server Name клиентских компонентов в раскрывающемся списке доступны зарегистрированные интерфейсы всех модулей данных одного сервера (их названия начинаются с названия исполнимого файла сервера, а через точку следует название конкретного модуля данных, например Project1.DataModule1).

Множественная **связь**

В стандартном примере Demos/Midas/SharedConn показано, как создать простейший сервер, поставляющий данные из таблицы Vendors базы *DBDEMOS*, и соединиться с ним с помощью компонента TDCOMConnection. Особенность данного примера в том, что сервер использует несколько модулей данных, но они не равнозначны — есть один родительский модуль MainRDM, ответственный только за организацию сессий, и модуль данных ChildRDm — потомок MainRDM. *Delphi 7* разрешает создавать сколько угодно таких потомков главного модуля данных. Структуру интерфейсов родительского модуля и модуля-наследника можно просмотреть с помощью редактора библиотеки типов (команда View ➤ Type Library).

Данный подход выгодно отличается от рассмотренного выше тем, что для взаимодействия с таким сервером не надо устанавливать отдельные соединения с каждым удаленным модулем данных, что заметно тормозит работу системы, если подобных модулей и соединений много. Достаточно установить связь только *с* родительским модулем. Для этого в клиентской программе надо задействовать компонент TSharedConnection с панели DataSnap.

Он настраивается на работу совсем просто. В свойстве Parent Connection указывается название доступного компонента, обеспечивающего удаленную связь с сервером. В его свойстве Server Name указывается название родительского модуля данных, например ShareServer.MainRDM. А интерфейс конкретного подчиненного модуля, ответственного за поставку данных, указывается в свойстве ChildName (выбирается из списка доступных зарегистрированных значений).

Структура нового клиентского приложения отличается тем, что используется только один компонент, непосредственно ответственный за удаленное соединение (TDCOMConnection). Взаимодействие со всеми подчиненными модулями данных происходит через промежуточные компоненты TSharedConnection, которые надо подготовить для каждого клиентского набора данных (вместо TDCOMConnection). Такой подход упрощает перенастройку крупных приложений при необходимости внесе-

ния изменений в применяемые компоненты удаленной связи, которых может быть довольно много. Другой вариант оптимизации числа клиент-серверных соединений рассматривается ниже.

Брокер соединений

Выше был рассмотрен ряд примеров по созданию многоуровневых приложений, которые используют компоненты, поддерживающие взаимодействие на основе различных протоколов и технологий (*CORBA, DCOM, TCP/IP, RDS*). Иногда доступ к серверу приложений из клиентской программы выполняют несколько клиентских наборов данных — для каждого из них требуется подготовить соответствующий компонент-соединение.

Упростить этот подход поможет компонент Брокер соединений (TConnectionBroker) с панели DataSnap, Например, если в приложении есть два клиентских набора (TClientDataSet), связанных с двумя компонентами TDCOMConnection, которые ссылаются на один сервер, то от одного из этих компонентов TDCOMConnection с помощью Брокера соединений можно избавиться так.

1. Добавим на форму компонент TConnectionBroker.
2. В его свойстве Connection выберем первый компонент *DCOM*-соединения, например DCOMConnection1.
3. В свойстве RemoteServer второго клиентского набора данных (ClientDataSet2) укажем значение ConnectionBroker1.
4. Сделаем этот набор активным (свойство Active установим равным True).
5. Удалим ставший ненужным компонент DCOMConnection2.

Шаги 3-4 надо также повторить для первого клиентского набора ClientDataSet1. Таким образом, клиентские наборы данных отвязываются от конкретного типа соединения, и настройка приложения на новый протокол работы значительно упрощается. Интерфейс сервера приложений теперь можно получать централизованно с помощью метода Брокер соединений:

```
function GetServer: IAppServer; override;
```

Локальная связь

В некоторых случаях возникает потребность взаимодействия клиентских компонентов (TClientDataSet) с различными *поставщиками данных* собственного приложения. Чтобы не перегружать работу серверного приложения

ненужными дополнительными внешними связями с самим собой, лучше использовать компонент Локальное соединение (TLocalConnection) с панели DataSnap. Настраивать его не надо, только после размещения на форме в клиентском наборе данных в свойстве Remote Server надо укапать этот компонент, а затем с помощью его метода

```
function GetServer: IAppServer; override;
```

получить прямой доступ *к* прикладному интерфейсу сервера IAppServer, программно предоставляющему все возможности по получению и обработке наборов данных.

Использование технологии XML

В данной главе были рассмотрены различные компоненты, позволяющие состыковывать с помощью самых разных стандартных протоколов клиентские программы с серверными модулями, обращающимися к базам данных. Нижеследующая часть посвящена ряду компонентов DataSnap и новым возможностям *Delphi 7* по работе с форматом *XML*. В частности, в системе добавилось несколько компонентов, позволяющих преобразовывать информацию в формат *XML*, и специальная утилита XmlMapper.

Что такое XML

В число нововведений системы *Delphi* входят компоненты, поддерживающие язык *XML* (язык расширенной разметки, стандарт которого был принят совсем недавно — в 1998 г.). На первый взгляд он напоминает язык описания гипертекста *HTML,* активно используемый для создания *Web*-страниц.

В чем особенность *XML?* С помощью тегов (определенной системы разметки) он позволяет описывать смысловое содержание документа. При этом язык *XML* не имеет фиксированного набора тегов. Их можно свободно придумывать для собственных нужд, расширяя возможности системы. Хотя имеются зарезервированные наборы тегов для конкретных областей применения, ничто не мешает использовать те же названия в собственных приложениях и для других целей.

Сегодня язык *XML* поддерживают практически все ведущие компании, производящие программное обеспечение. В частности, язык *XML* нашел очень широкое применение в системе *Windows 2000,* поэтому реализация этого языка имеется и в системе *Delphi 7.*

Язык *XML* позволяет описывать сколь угодно сложно организованные данные и их структуру. В примерах, рассмотренных ранее, мы передавали упакованные наборы данных в формате OleVariant. Теперь же мы можем пересылать их между сервером приложений и базой данных в формате *XML.*

XML-преобразования

Возможность преобразования наборов данных из различных внутренних форматов (например, таблиц баз данных) в стандартизованный формат *XML* (и наоборот) очень полезна. Такие средства позволяют стыковать работу приложений, созданных разными производителями, и расширять функциональные возможности унаследованных (морально устаревших, но активно эксплуатируемых) систем, не меняя их исходных текстов, а просто добавляя утилиты и программы конвертации данных и делая эти данные общедоступными с помощью стандарта *XML*. Для визуального преобразования наборов данных из внутренних форматов в *XML* и обратно предназначена утилита XmlMapper.

Как работать с утилитой XmlMapper

Данная утилита (xmlmapper.exe) расположена в подкаталоге Bin каталога, в который установлена система *Delphi 7*. Назначение этой программы — преобразование информации из набора данных в формат *XML* и наоборот, а также автоматическая генерация правил такого преобразования (например, как представлять дату, сколько знаков отводить для дробных чисел, какую кодировку использовать для представления текстов). С помощью подобных правил удается хранить данные в независимом от конкретного приложения формате и при необходимости обращаться к ним из других программ. Вручную набирать эти правила в формате *XML* достаточно трудоемко, поэтому утилита XmlMapper предоставляет разработчику визуальные средства.

Рассмотрим пример создания *XML*-документа на основе таблицы clients.dbf, входящей в состав базы данных *DBDEMOS* из набора примеров *Delphi 7*. Создадим *COM*-сервер, который обращается к таблице Clients. Назовем его, например, DataTestl, откомпилируем и зарегистрируем в системе.

Для корректной работы утилиты XmlMapper требуется, чтобы в системе были зарегистрированы две динамические библиотеки: midas.dll и msxml.dll. Их можно зарегистрировать самостоятельно, например с помощью командных строк:

```
regsvr32 rasxinl.dll
regsvr32 midas.dll
```

Обе эти библиотеки входят в поставку *Delphi 7* и при установке копируются в системный каталог Windows/System. Стандартная утилита regsrv32.exe расположена в этом же каталоге.

После запуска средства XmlMapper командой Tools ➤ XML Mapper (Утилиты > XML Mapper) в его окне на панели Datapacket надо вызвать контекстное меню и в нем выбрать пункт ConnectTo Remote Server (Подключиться к удаленному серверу). На экране появится окно со списком доступных (зарегистрированных в системе) *COM*-серверов, связанных с базами данных. Б раскрывающемся списке Remote Server Name выбирается подходящий сервер (в нашем случае — что-то вроде Project1.DataTestl), нажимается кнопка Connect, в результате чего сервер запускается. При этом стано-

вится доступным раскрывающийся список Providername. В нем выбирается нужный поставщик данных (например, Project1.DataTest1:DataSetProvider1), доступный через данный *COM*-сервер. После нажатия на кнопку Get Datapacket на панели Datapacket отображается иерархическая структура соответствующего набора данных.

Следующий шаг — определение тех полей набора, которые требуется преобразовать в формат *XML*. Для этого достаточно дважды щелкнуть мышью над каждым из нужных полей (например, FIRST_NAME и TELEPHONE), в результате чего они появятся на панели Transformation в поле Selected Fields.

Для подготовки и создания набора данных в *XML*-формате надо дать команду Create ➤ XML From Datapacket (Создать ➤ XML-данные из пакета данных), нажать кнопку OK в промежуточном диалоговом окне — и на левой панели (XML Document) возникнет *XML*-структура преобразовываемого набора данных.

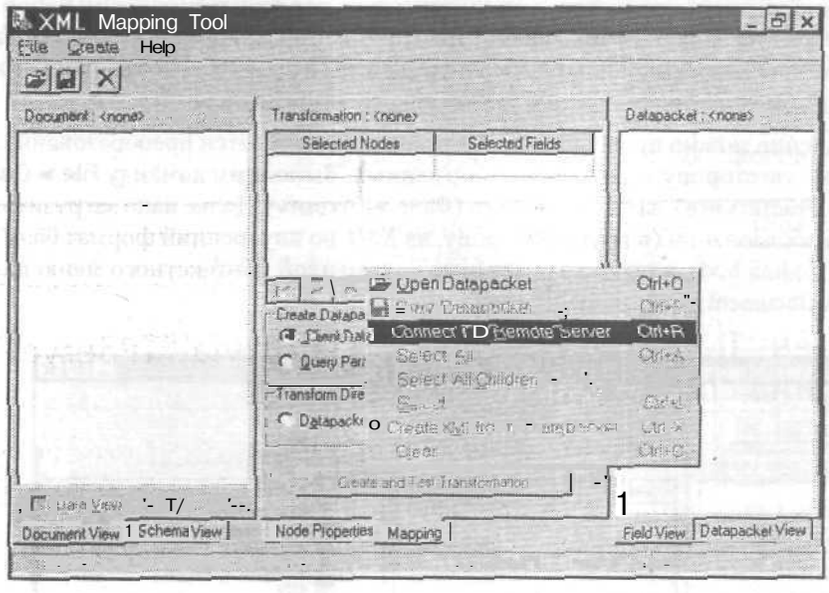


Рис. 9.10. *Подключение к удаленному серверу из утилиты XmlMapper*

На вкладке Node Properties можно определить дополнительные правила преобразования каждого поля (например, изменить тип данных, его вид, название и т. п.).

На панели Transformdirection надо установить переключатель Datapacket to XML (направление преобразования — из внутреннего формата в *XML*). Чтобы сохранить правила преобразования, в контекстном меню панели Transformation выберем пункт Create, а затем Save (сразу возможность сохранения недоступна) и сохраним сгенерированные правила в файле a.xtr. На этом шаге также полезно сохранить правила преобразования в направлении XML to Datapacket (для примера, в файле b.xtr).

На вкладке DTD панели Schema View (Просмотр схемы XML-документа) с помощью контекстного меню командой Save Schema можно сохранить *XML*-схему в файле. На

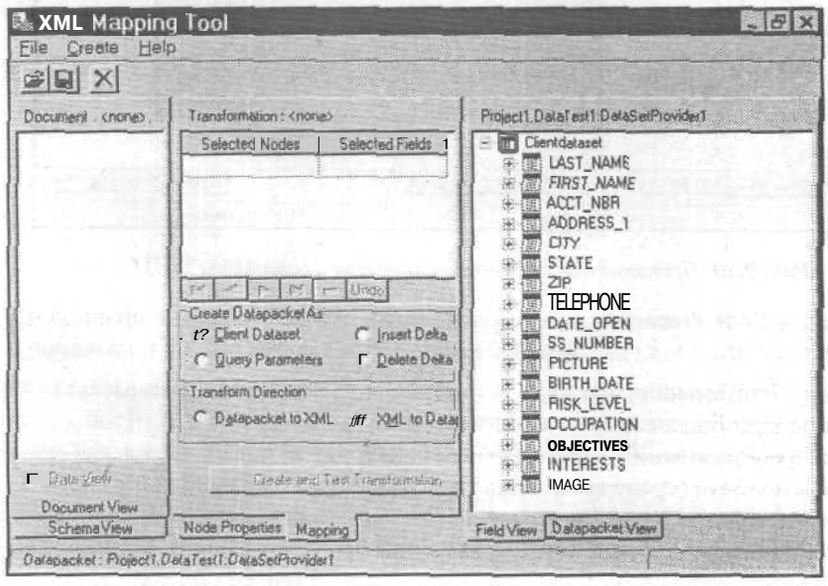
двух других вкладках (XDR и XML-Schema) можно сформировать описание схемы в других форматах.

Доступные форматы схем:

- DTD (Document Type Definition) — определение типа логически законченного *XML*-документа;
- *XML*-схема (XSD) — описание отличий и дополнений к DTD для данного *XML*-документа;
- XDR (Reduced XML Data) — дополнительный способ описания *XML*-схем;
- XTR (XML Transformation) — правила преобразования данных.

Завершающий шаг — непосредственное преобразование набора данных в *XML*-формат. Для этого надо нажать кнопку Create and Test Transformation — на экране появится дополнительное тестовое окно, описывающее результирующий набор данных в формате, заданном правилами трансформации. С помощью кнопки Save в этом окне сохраним выбранный набор в файле a.xml в произвольно выбранном каталоге. Файл можно просмотреть в любом текстовом редакторе.

В завершение можно проверить, правильно ли выполняется преобразование данных в другую сторону — из *XML* в набор данных. Выполним команду File ➤ Clear All (Файл > Очистить все), затем File ➤ Open (Файл > Открыть). Далее надо загрузить правила преобразования (в другую сторону, из *XML* во внутренний формат базы данных) из файла b.xtr, а затем сами данные — командой контекстного меню панели Open XML Document, выбрав файл a.xml.



РМС. *9.11. Структура полученного от сервера набора данных*

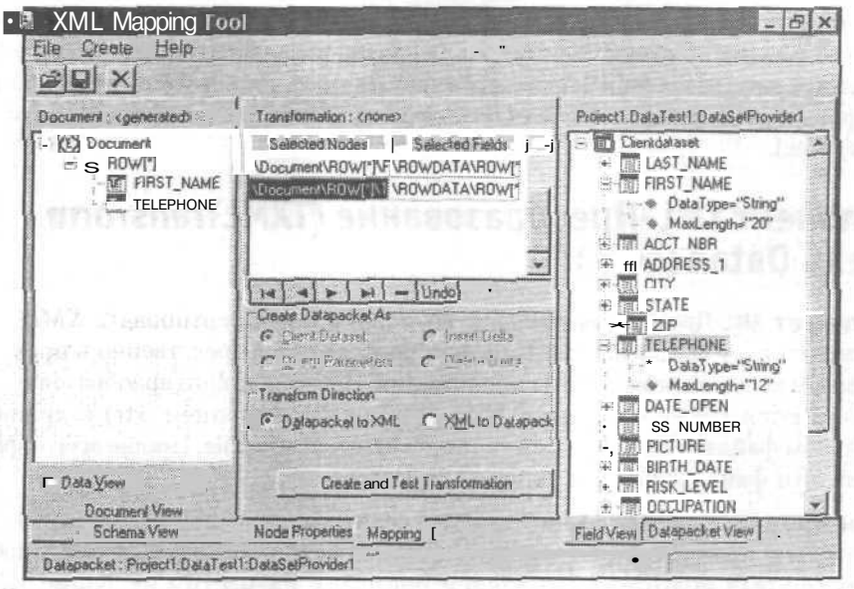


Рис. 9.12. Определение полей, связанных с преобразованием данных

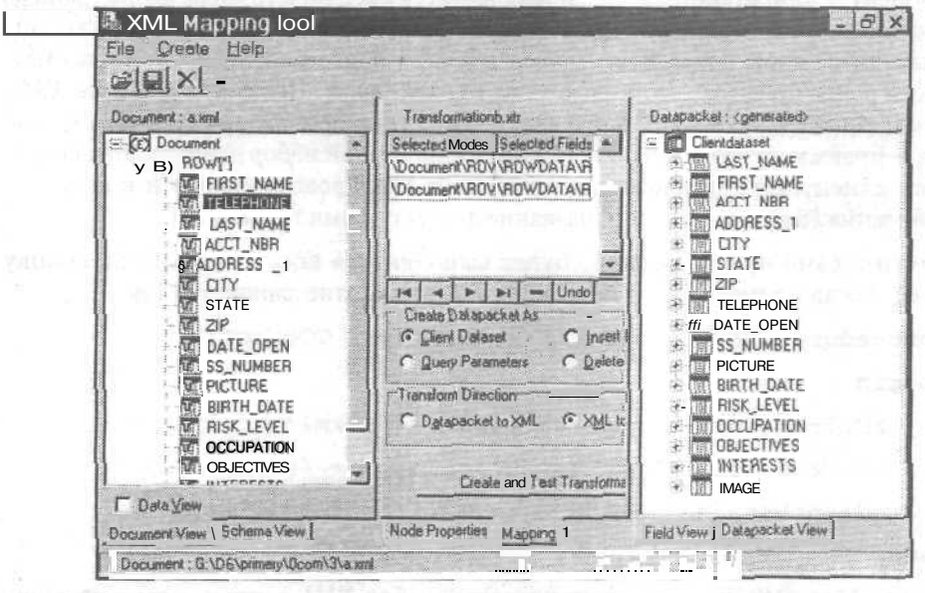


***Рис. 9.13.** Результат преобразования*

Двойным щелчком на каждом из полей укажем, какая информация должна передаваться в результирующий пакет данных. Центральная панель заполняется соответствующими правилами преобразования, а правая — структурой конечного набора данных.

Для выполнения самого преобразования надо определить направление передачи данных — установить переключатель панели Transformation Direction в положение XML to Datapacket, в разделе Create Datapacket As указать значение ClientDataset (клиентский набор данных) и нажать кнопку Create and Test Transformation. Данные сохраним в файле b.xml.

Компонент XML-Преобразование (TXMLTransform, панель DataAccess)

Компонент XML-Преобразование дает возможность конвертировать *XML-документы* в наборы и пакеты данных и наоборот непосредственно в программном коде во время работы приложения. Правила такого преобразования задаются также в формате *XML* (файл с расширением .xtr) и хранятся в отдельном файле — он задается в свойстве Transformation File. Проще всего сформировать этот файл с помощью утилиты XmlMapper.

Рассмотрим применение данного компонента на конкретном примере. Создадим клиентскую программу для работы с *COM-сервером*, обращающимся к таблице Clients (он был подготовлен в раннем примере). Разместим на форме таблицу TDBGrid, свяжем ее с источником данных — компонентом TDataSource, далее добавим компонент — клиентский набор данных TClientDataSet, ничего в нем не настраивая, и свяжем источник данных с этим набором данных. Затем скопируем в каталог, где находится проект, два файла — a.xml и b.xtr, — подготовленные в ходе рассмотренного ранее примера с помощью утилиты XmlMapper. Первый, в формате *XML*, описывает исходную информацию для соответствующего набора данных, а второй файл — правила преобразования из *XML* в клиентский набор данных. Добавим на форму клиентского приложения компонент XML-Преобразование и в свойстве TransformationFile укажем b.xtr — название файла с этими правилами.

Допустим, само преобразование будет выполняться после нажатия на кнопку Button1. Тогда примерный текст реакции на это нажатие запишется так:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
   XMLTransform1.SourceXMLFile := 'a.xml';
   XMLTransform1.TransformationFile := 'b.xtrм;
   ClientDataSet1.XMLData := XMLTransform1.Data;
end;
```

Свойство SourceXMLFile определяет исходный набор *XML*-данных, а само преобразование происходит в момент присваивания свойству XMLData клиентского набора данных значения свойства Data объекта XML-Преобразование (точнее, непосредственно в момент обращения к свойству Data). Если теперь запустить приложение и нажать на кнопку Buttonl, в таблице появится информация, записанная в *XML*-файле (значения из двух полей FIRST_NAME и TELEPHONE).

Компонент Поставщик XML-данных (TXMLTransformProvider, панель DataAccess)

Для самых разных целей, например для взаимодействия с внешними приложениями и передачи им информации в открытом прозрачном виде, данные из внутренних баз данных программы, как уже говорилось, желательно представлять в стандартизованном *XML*-формате.

В большинстве случаев процесс обмена *XML*-информацией можно упростить. Проще всего применить компонент Поставщик XML-данных. Он, в частности, позволяет представлять *XML*-документ как обычную таблицу данных (при условии, что структура документа это допускает).

Вернемся к предыдущей клиентской программе и добавим в нее Поставщик XML-данных (TXMLTransformProvider). В его свойстве XMLDataFile укажем файл, в котором хранятся исходные *XML*-данные (например, ранее сформированный файл a.xml).

Свойства TransformRead и TransformWrite фактически представляют собой вложенные компоненты XML-Преобразование и используются для указания правил преобразования при получении этим провайдером *XML*-данных (свойство TransformRead) и при поставке им таких данных (свойство TransformWrite).

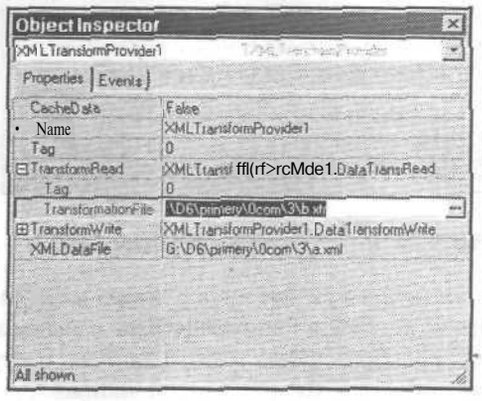


Рис. 9.14. Настройка XML-поставщика

Укажем в свойстве TransformRead (подсвойство TransformFile) соответствующий файл преобразования b.xtr.

Если теперь обратиться к свойству ProviderName клиентского набора данных, то в раскрывающемся списке появится название XML-Поставщика (XMLTransformProvider1). Выберем его. После того как клиентский набор данных активизирован (в свойство Active запишем значение True), элемент DBGrid1 сразу же отобразит соответствующую информацию из *XML*-файла.

Компонент XML-Клиент (TXMLTransformClient, панель DataAccess)

Поставщик *XML*-данных, как было только что показано, может брать данные *непосредственно* из XML-файла. В тех случаях, когда поставщиком информации выступает, например, *COM*-объект и получаемые от него данные надо представить в *XML*-формате, следует воспользоваться компонентом XML-Клиент.

Сформируем связь с *COM*-сервером (например, с DataTest1) с помощью компонента TDCOMConnection. После размещения на форме компонента TXMLTransformClient свяжем его с этим сервером через свойство RemoteServer.

В свойстве ProviderName в раскрывающемся списке выбирается провайдер DataSetProvider1 этого сервера.

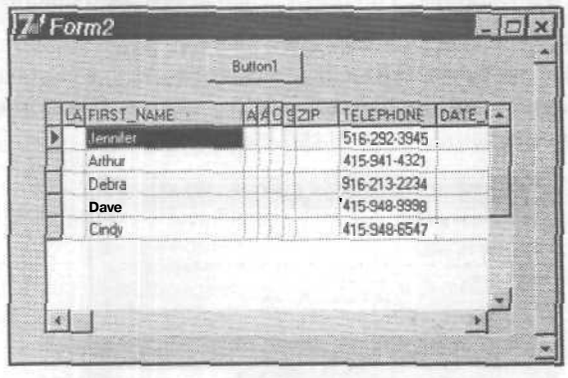


Рис. 9.15. *Прием данных в формате XML*

Чтобы преобразовать поставляемую им информацию в формат *XML*, в свойстве TransformFile вложенного компонента TransformGetData зададим значение, определяющее файл с правилами преобразования набора данных, поступающего от *COM*-сервера (это может быть файл a.xtr — с учетом направления преобразования Datapacket to XML).

Для формирования *XML*-данных на основе принимаемой информации служит следующий метод XML-Клиента (параметр PublishTransformFile — имя файла с правилами преобразования):

```
function GetDataAsXml(PublishTransformFile: string):
string;
```

В качестве результата функция возвращает строку, представляющую собой законченный *XML*-документ. Если параметр не указан, то он определяется свойством TransformGetData.TransformFile.

Компонент XML-документ (TXMLDocument, панель Internet)

После того как приложение (как правило, клиентское или предназначенное для связи двух независимых систем) получает информацию в *XML*-формате, эту информацию необходимо определенным образом преобразовать и обработать — не обязательно в виде таблицы базы данных, а, например, в каком-то внутреннем формате другой программы. Если на предприятии работает несколько бухгалтерских программ, способных сохранять хозяйственные проводки в *XML*-формате, и принято решение о переходе на единую систему учета, все *XML*-документы желательно привести к единому виду. Первоначально они наверняка отличаются друг от друга способами представления экономической информации, количеством и названиями отдельных полей, их описанием и формой представления, глубиной вложенности данных, иерархической структурой документа и другими атрибутами. Специально для обработки и согласования *XML*-данных в *Delphi 7* введен компонент XML-документ,

Данный компонент не визуальный. Он обрабатывает *XML*-данные, либо записанные во внешнем файле (свойство FileName), либо представленные непосредственно в виде текстовой строки в памяти (свойство XML). Эти свойства взаимоисключающие: если одному из них задано значение, то другое должно быть пустым.

В свойстве DOMVendor в раскрывающемся списке можно выбрать одну из зарегистрированных в системе *DOM*-моделей *XML*. В *Windows* это, как правило, MSXML.

ПОДСКАЗКА DOM (Document Object Model) — объектная модель документа, определяющая методы разборки и редактирования XML-документа. Разные компании предлагают свои DOM-методы, поэтому способы нахождения нужного раздела в XML-документах могут отличаться.

Возьмем для примера следующий *XML*-документ (заранее подготовленный в файле x.xml), содержащий курсы акций различных компаний:

```
<?xml version='1.0' standalone='yes' ?>
<!DOCTYPE StockList SYSTEM "x.dtd" >
<StockList>
   <Stock>
      <name>Borland</name>
      <price>15.375</price>
      <symbol>BORL</symbol>
      <shares>100</shares>
   </Stock>
```

продолжение

```
    <Stock>
        <name>Pfizer</name>
        <price>42.75</price>
        <symbol>PFE</symbol>
        <shares>25</shares>
    </Stock>
</StockList>
```

Описание этого документа в формате *DTD* можно создать с помощью утилиты XmlMapper.

Корневой узел документа называется StockList (список акций), далее следуют отдельные узлы Stock (акция), описывающие конкретные характеристики каждой акции — название компании (name), цену акции (price), символическое название на бирже (symbol) и число акций для покупки или продажи (shares).

После размещения *XML*-документа на форме укажем файл x.xml в свойстве FileName и переведем значение свойства Active в значение True. Если никаких предупреждающих сообщений не возникнет, значит, этот *XML*-документ готов для дальнейшей обработки.

Самое важное его свойство — это DocumentElement. Оно предоставляет доступ к корневому узлу всей структуры документа и имеет тип IXMLNode. Для получения доступа к списку наследников корневого узла надо воспользоваться свойством ChildNodes. Оно имеет тип IXMLNodeList — список объектов уже упомянутого типа IXMLNode (из чего следует, что каждый из объектов-узлов в свою очередь тоже может указывать на подобный список). Все они в совокупности формируют иерархическую структуру *XML*-документа.

Для лоступа к конкретным значениям узла Stock можно воспользоваться обращением к свойству Nodes; разрешается указывать как номер нужного элемента (индексация начинается с нуля), так и его название в *XML-структуре* (например, name).

Свойство NodeType (тип TNodeType) определяет тип узла (элемент, атрибут, содержимое, инструкция) и множество других характеристик *XML*-документа. С помощью свойства HasChildNodes можно проверить, является ли данный элемент документа конечным и с определенным значением (HasChildNodes = False) или же промежуточным узлом на дереве (HasChildNodes = True). Значение узла можно получить либо в самом общем виде с помощью свойства NodeValue, содержащего значения типа Variant, либо с помощью свойства Text, представляющего значение в текстовом виде.

Пример

Чтобы выбрать из представленного выше *XML*-документа значение поля name первой «записи* *(Borland),* можно использовать следующую строку:

```
s := XMLDocument1.DocumentElement.ChildNodes.Nodes[0].
  ChildNodes['name'].Text,-
```

Свойство *XML*-документа DocumentElement определяет корневой узел StockList. Его наследники (свойство ChildNodes) — список узлов Stock (тип IXMLNodeList). К любому из них можно обратиться с помощью свойства Nodes, у которого в свою очередь будут узлы-наследники name, price, symbol и shares (свойство ChildNodes, в котором в качестве параметра указывается строка с названием нужного узла). Результат в виде текстовой строки получается через свойство Text. Можно использовать сокращенную запись:

```
s r= XMLDocument1.DocumentElement.ChildNodes[0].
  ChildNodes['name'].Text;
```

Так как обращение XMLDocument1.DocumentElement.ChildNodes[N] определяет конкретный (*N*-й) узел из списка наследников, он имеет тип TNodeType и его можно индексировать сразу, напрямую.

ВНИМАНИЕ Так как схемы разборки (DOM-модели) разных компаний способны работать с одними и теми же документами по-разному, при создании приложения может потребоваться настройка конкретного разборщика. Это выполняется уточнением значения свойства ParseOptions XML-документа. В частности, значение вложенного свойства po PreserveWhite Spa ce желательно установить равным False, если документ содержит пробелы и символы перевода строк, которые можно игнорировать.

Еще один пример

Рассмотрим вариант обработки данных, получаемых от *COM*-сервера (об этом процессе рассказывалось при описании компонента TXMLTransformClient).

После того как с помощью метода GetDataAsXml получена строка (потенциальный *XML*-документ), ее надо представить в соответствующем виде. Для этого разместим на форме компонент TXMLDocument, а в обработчике нажатия на кнопку запишем код, в котором выбирается значение любого доступного поля, например FIRST_NAME:

```
procedure TForm1.Button1Click(Sender: TObject);
var s: string;
begin
  s := XMLTransformClient1.GetDataAsXml('');
  XMLDocument1.LoadFromXML(s);
  Label1.Caption :=
    XMLDocument1.DocumentElement.ChildNodes.Nodes[0].
    ChildNodes['FIRST_NAME'].Text;
end;
```

Напрямую записывать одну строку (String) в свойство XML (тип TStrings) нельзя: в этом свойстве каждый элемент должен соответствовать одной строке *XML*-документа. Разбор «длинной» строки и представление *XML*-документа в построчном виде выполняет следующий метод:

```
procedure LoadFromXML(const XML: string); overload;
```

Что нового мы узнали?

В этом уроке мы научились

- ☑ создавать серверные транзакционные объекты;
- 0 получать доступ к СУБД *InterBase* через сервер *MTS*;
- 0 развертывать многоуровневое *COM*-приложение;
- В организовывать доступ к данным по протоколам *TCP/IP* и *HTTP*,
- 0 использовать в одном приложении разные технологии доступа;
- 0 готовить и передавать данные в формате *XML*.

урок Программирование для Интернета

- Введение в программирование для Интернета
- Панели Indy
- Создание клиентских и серверных служб Интернета
- Передача данных через Интернет

Введение в программирование для Интернета

Возможности системы Delphi 7 по созданию приложений для Интернета

В системе *Delphi 7* имеется немало новых компонентов, предназначенных для работы с Интернетом, *WWWu* сетями *TCP/IP,* а возможности старых компонентов расширены.

Компоненты панели WebServices позволяют создавать *Web*-серверные приложения, общающиеся по независимому от операционной системы протоколу *SOAP.* Компоненты панели Internet ориентированы на создание Web-серверных модулей. Компоненты панели InternetExpress дают возможность создавать многоуровневые Интернет-приложения, работающие с базами данных. Для быстрого создания динамических *Web*-программ, позволяющих обрабатывать данные через Интернет, предназначены компоненты панелей WebSnap и IntraWeb.

На основе этих компонентов можно разработать серверы приложений, которые обмениваются наборами данных, упакованными и представленными не в виде объектов типа OleVariant, а в виде описания *XML.* Преимущество такого подхода состоит в том, что разработчику не требуется писать собственное клиентское приложение — достаточно использовать обычный браузер, например *Microsoft Internet Explorer.* В дальнейшем в книге при упоминании браузера подразумевается *Internet Explorer 5.0.*

В *Delphi 7* имеется набор *Indy,* в который входят пять панелей компонентов, значительно упрощающих создание Интернет-приложений. Данный набор содержит более высокоуровневые компоненты, позволяющие не вникать в тонкости Интернет-протоколов. Другая отличительная особенность этого набора — переносимость программ, созданных с помощью *Indy,* между платформами *Linux* и *Windows.* Это связано с поддержкой *Indy* как в *Delphi,* так и в *Kylix.* Кроме того, набор *Indy* содержит в себе как клиентские, так и серверные компоненты, что позволяет, в частности, создавать свои собственные *Web*-серверы (*HTTP*-серверы — именно серверы, а не серверные приложения!). Подобные программы работоспособны как при использовании *Web-серверов* для *Windows,* таких как *IIS,* так и при использовании *Web*-серверов для *Linux,* например *Apache.* При этом они не привязываются к конкретному *Web*-серверу и могут реализовываться как для *Windows,* так и для *Linux* на основе одного и того же исходного кода.

Создание собственного браузера

Чтобы сразу получить наглядный пример возможностей системы *Delphi 7* по разработке приложений Интернета, посмотрим пример *Webbrows* из числа стандартных

примеров, поставляемых вместе с системой. Он хранится в каталоге Delphi 7\Demos\Coolstuf.

Если запустить это приложение, то на первый взгляд может показаться, что перед нами браузер *Microsoft Internet Explorer*: видны те же кнопки и перемещаемые панели, и только по мелким деталям оформления можно понять, что это обычная программа, созданная с помощью системы *Delphi 7* (рис. 10.1).

Рис. 10.1. *Просмотр Web-страницы при помощи браузера, созданного средствами системы Delphi 7*

Панели инструментов созданы на основе компонентов TToolBar и TCoolBar. В нижней части главной формы расположен компонент TWebBrowser, выполняющий все функции стандартного браузера, связанные с отображением страниц *HTML*.

Все, что требуется для такой работы, — вызов метода Navigate. Этот метод выполняет обращение к узлу, название которого указано в качестве параметра этого метода. Навигация между страницами осуществляется щелчком мыши на ссылках и не требует специальных усилий по программированию. Обновление страницы или прекращение ее загрузки также выполняется с помощью нескольких простых методов компонента TWebBrowser. При желании на его основе несложно создать собственный браузер, внешним видом напоминающий *Netscape Navigator,* или полностью оригинальное средство просмотра.

Основной объем программного кода примера приходится на поддержание возможности работы со списком ранее посещенных страниц, что является неотъемлемым атрибутом современных браузеров.

ЗАМЕЧАНИЕ Данный компонент является наследником класса TOLEControl, что позволяет обращаться к нему как к серверу автоматизации OLE. Подобными возможностями обладают все современные браузеры.

При работе с браузерами часто применяется понятие адреса *URL* (*Uniform Resource Locator)* — универсального идентификатора ресурса, который указывает на конкретный объект в сети и, дополнительно, метод доступа к нему (как правило, название протокола). При путешествии в *WWW* обычно используется протокол *HTTP.*

Адрес *URL* записывается в следующем формате:

```
схема-доступа : местоположение-объекта
```

например:

```
http://www.vovan.ru/index.html
```

Панели Indy

Три новые панели *Indy* (сокращение от *Internet Direct)* предоставили разработчикам большой набор компонентов с упрощенным интерфейсом, позволяющий разрабатывать клиент-серверные программы для Интернета. Немаловажно, что новые компоненты поддерживают две операционные системы: *Windows* и *Linux,* — и применяются в *Delphi, C++Builder* и *Kylix.*

Набор Indy Clients содержит компоненты, ответственные за поддержку Интернет-протоколов на клиентских местах, набор Indy Servers — компоненты, ответственные за работу Интернет-серверов, набор Indy Misc — вспомогательные компоненты, Indy Intercepts — компоненты, поддерживающие функции сжатия и протоколирования, Indy I/O Handlers — компоненты, позволяющие работать с идентификаторами ввода/вывода при использовании сокетов.

Базовые TCP-компоненты

Компонент TCP-Сервер (TIdTCPServer)

Этот компонент — базовый для всех серверных систем, поддерживающих протокол *TCP.* Он наследует свойства и методы класса TIdComponent — базового класса всех компонентов *Indy,* на основе которых создаются клиентские и серверные приложения.

На базе TIdTCPServer можно разрабатывать и собственные оригинальные TCP-серверы. В стандартную поставку *Delphi* входит ряд таких серверов, например *HTTP,* новостных, предоставляющих информацию о времени и пр. Они реализованы в качестве других компонентов, которые будут рассмотрены позже.

Каждый TCP-сервер по своему определению способен поддерживать несколько соединений с удаленными машинами. Сервер контролирует и прослушивает состояние различных портов удаленных приложений через клиентские соединения, которые вместе с портами указываются в коллекции Bindings (тип TIdSocketHandles) в визуальном редакторе.

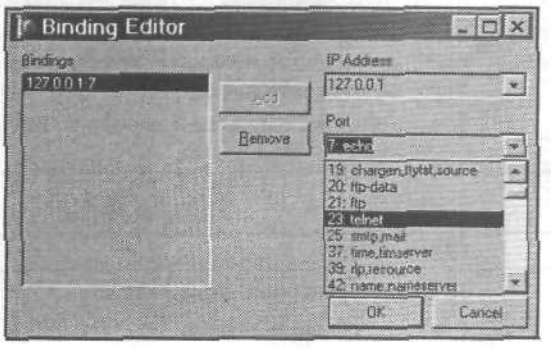


Рис. *10.2. Настройка TCP-сервера на клиентские соединения*

ЗАМЕЧАНИЕ Если сервер контролирует состояние портов компьютера, на котором он запущен, то IP-адрес этого компьютера можно не указывать. Достаточно лишь ввести номер порта после двоеточия, например так: «:8080»

Использование данного компонента рассмотрено далее в прикладном примере. Почти все компоненты панели Indy Servers в той или иной степени наследуют функциональность TIdTCPServer.

Компонент TCP-клиент (TIdTCPClient)

TCP-клиент наследует свойства и методы класса TIdTCPConnection, которые определяют конкретное *TCP*-соединение. Компонент TIdTCPClient можно использовать самостоятельно и как основу для создания всевозможных клиентов: почтовых, *FTP*- и других.

Использование данного компонента рассмотрено далее в прикладном примере. Почти все компоненты панели Indy Clients в гой или иной степени наследуют функциональность TIdTCPClient.

Класс Indy-поток (TIdThread)

Наследник стандартного класса TThread, используемый в многопотоковых компонентах *Indy*. Дополнен функциональными возможностями по контролю за состоянием потоков.

Таблица 10,1. Свойства компонента TIdThread

Свойства	Назначение
Data	Хранилище данных потока
StopMode	Определяет действие, которое будет предпринято при вызове метода Stop: smTerminate — поток завершается; smSuspend — лоток временно приостанавливается
Stopped	Имеет значение True, если поток приостановлен. Изменение значения этого свойства не останавливает поток!
TerminatingException	Текст сообщения, сформированного в процессе работы потока (при вызове метода Execute)

Таблица 10.2. Методы класса TldThread

Метод	Назначение
procedure Start;	Запуск/возобновление работы потока
procedure Stop;	Приостановка работы потока
procedure Synchronize(Method: TThreadMethod);	Процедура Synchronize позволяет выполнить метод объекта в гарантированно безопасном режиме, с синхронизацией его работы с другими методами

Пример *создания* распределенной TCP-системы

Рассмотрим несложный пример распределенной серверной системы, работающей по протоколу *TCP*. Центральный сервер принимает текстовые сообщения от множества программ-клиентов и высылает им ответные подтверждения о том, что информация принята.

В первую очередь создадим TCP-сервер. Для этого разместим на новой форме компонент TIdTCPServer. В его свойстве Bindings укажем подходящий порт для прослушивания. Дополнительно добавим компонент Многострочное поле ввода (TMemo). В нем будут отображаться все поступающие на сервер сообщения.

Функциональность сервера реализуем в обработчике события OnExecute, которое возникает, когда один из клиентов пытается установить связь с сервером. Этот обработчик будет содержать в себе три оператора:

```
procedure TForm1.IdTCPServer1Execute(AThread: TIdPeerThread);
begin
  with AThread.Connection do
    begin
      Memo1.Lines.Add(AThread.Connection.CurrentReadBuffer);
      WriteLn('ok');
      Disconnect;
    end;
end;
```

В качестве параметра обработчику передается объект класса TIdPeerThread. Такой объект создается автоматически для каждой установленной клиент-серверной *TCP*-связи, порождающей в свою очередь отдельный выполняющийся на сервере поток. Нужное нам свойство Connection этого объекта, имеющее тип TIdTCPServerConnection и обеспечивающее связь с *TCP*-сервером, представляет собой простую программную оболочку. Принципы работы с ней примерно такие же, как при работе с клиентским компонентом TIdTCPClient.

Поступившую от клиентского приложения информацию можно получить разными способами, например прочитать из буфера с помощью метода

```
function CurrentReadBuffer: string;
```

Он используется в операторе, который сразу выводит соответствующую информацию на экран — в текстовую область Memo1. Далее клиентской программе отсылается подтверждение — строковая константа 'ok'. Затем происходит закрытие соединения. Для этого используются такие методы:

```
procedure WriteLn(const AOut: string); virtual;
procedure Disconnect; virtual;
```

Закрывать сформированное соединение очень важно, так как для каждого такого соединения на сервере запускается отдельный поток. При большом количестве незакрытых соединений существенно возрастает нагрузка на процессор.

С помощью менеджера проекта добавим в проект еще одно приложение (клиентское), разместим на форме кнопку, поле ввода и компонент TIdTCPClient. Для его настройки достаточно указать *IP*-адрес сервера: например в свойстве сервера Host, как это делалось раньше по аналогии с настройкой сокета. Также зададим порт в свойстве Port.

Чтобы введенная в поле Editl информация отсылалась серверу, надо сформировать обработчик нажатия на кнопку. Он будет состоять из четырех операторов:

```
IdTCPClient1.Connect;
IdTCPClient1.Write(Edit1.Text);
Edit1.Text := IdTCPClient1.ReadLn;
IdTCPClient1.Disconnect;
```

В первом операторе клиент устанавливает соединение с сервером, во втором посылает серверу текст из объекта Editl. В третьем с помощью метода

```
function ReadLn(const ATerminator: string; const ATimeout:
  Integer): string; virtual;
```

получает подтверждение с сервера (строку 'ok'), показывает это подтверждение в объекте Editl и в завершение закрывает связь.

Все программы проекта можно откомпилировать и запустить: один сервер и неограниченное количество клиентов. При нажатии на кнопку Button1 в любой из клиентских программ текст, введенный в поле Editl, появится в многострочном поле сервера. Практически одновременно текст в поле Editl заменяется поступившим от сервера подтверждением.

Компонент Простой TCP-сервер (TIdSimpleServer)

Реализованная в вышеприведенном примере базовая функциональность TCP-сервера содержится в однопотоковом компоненте TIdSimpleServer. Время ожидания ответа от клиентской программы, по истечении которого клиентское соединение закрывается, задается в свойстве AcceptWait. IP-адрес и рабочий порт сервера определяются свойствами:

```
BoundIP: string;
BoundPort: Integer;
```

Свойство

```
ListenHandle: TIdStackSocketHandle;
```

доступное только для чтения, определяет текущий соединенный с сервером сокет или значение Id_INVALID_SOCKET, если связь отсутствует. Прослушивание запросов от клиентских сокетов запускается методом

```
function Listen: Boolean; virtual;
```

Он возвращает значение True, если обнаружен запрос, требующий обработки. Установка связи с прослушиваемым сокетам выполняется вызовом метода

```
procedure Bind; virtual;
```

а идентификатор сокета возвращается в свойстве ListenHandle.

Компоненты UDP-клиент (TIdUDPClient) и UDP-сервер (TIdUDPServer)

Эти компоненты схожи по своим функциональным возможностям с компонентами TIdTCPClient и TIdTCPServer. Они поддерживают протокол UDP вместо TCP. UDP-протокол более простой и работает быстрее, но хуже защищен от возможных сбоев и потерь информации. Основное отличие заключается в том, что клиентские объекты реализуют в основном методы отправки UDP-пакетов, а не полноценное сокет-соединение.

Немного изменим предыдущий пример. К серверной программе добавим компонент TIdUDPServer. Его свойство Bindings, как и в случае с TIdTCPServer, представляет собой коллекцию объектов типа TIdSocketHandle. В каждом из них достаточно указать только IP-адрес и порт.

Создадим обработчик события OnUDPRead, которое возникает, когда UDP-сервер получает UDP-посылку от конкретного клиента. Получаемая информация имеет тип TStream и ее с помощью метода

```
procedure LoadFromStream(Stream: TStream);
```

класса TStrings можно сразу загрузить в многострочное текстовое поле:

```
procedure TForm1.IdUDPServer1UDPRead(Sender: TObject;
AData: TStream;
   ABinding: TIdSocketHandle);
begin
```

```
    Memo1.Lines.LoadFromStream(AData);
end;
```

Функциональность клиентской программы также не сложна. Разместим на форме такой программы поле ввода и компонент TIdUDPClient, а в обработчике нажатия на кнопку запишем один оператор:

```
procedure TForm2.Button1Click(Sender: TObject);
begin
    IdUDPClient1.Send(Edit1.Text);
end;
```

С помощью метода

```
procedure Send(AData: string);
```

данный объект отправит по установленному соединению (свойства Host и Port) содержимое поля ввода.

Запустим серверное и клиентское приложения, введем в поле ввода клиента любую текстовую строку, нажмем кнопку — и она отобразится в многострочном поле сервера.

Другие Indy-компоненты

Рассмотрим вкратце другие клиент-серверные компоненты панелей Indy Servers/Clients. Они являются наследниками функциональных характеристик компонентов TIdTCPServer/TIdTCPClient и в плане прикладного программирования отличаются небольшим набором дополнительных свойств.

Компонент TIdChargenServer предназначен для тестирования сетей в стрессовых режимах и автоматической генерации символьных посылок в соответствии со спецификацией *RFC864*. В качестве порта по умолчанию используется значение свойства DefaultPort.

Компонент TIdDayTimeServer позволяет создавать свои собственные серверы времени. Временная зона, в которой работает сервер, задается в свойстве

```
TimeZone: String;
```

Чтобы получить от сервера время, надо использовать в клиентском приложении компонент TIdDayTime. В его свойстве

```
DayTimeStr: String;
```

в момент установления соединения с сервером запишется полученное от сервера время и дата в виде текстовой строки в наглядном для пользователя виде.

ВНИМАНИЕ Данные компоненты рекомендуется применять в основном для отладки. Если требуется реализовать синхронизацию клиент-серверной системы, лучше воспользоваться компонентами TIdTimeServer/TIdTime.

Компонент TIdTimeServer — это серверный компонент, поставляющий информацию о текущем времени. Как правило, используется в локальных сетях. Свойство

```
BaseDate: TDateTime;
```

определяет начало отсчета времени сервером. Никаких дополнительных расширений функциональности данного компонента программисту выполнять не требуется.

ВНИМАНИЕ Спецификация RFC 868, которую реализует компонент TIdTimeServer, имеет временное ограничение до 2035 г.

На клиентской стороне применяется компонент TIdTime. Он должен быть соединен с серверным приложением, использующим активный компонент TIdTimeServer. Свойство

```
DateTime: TDateTime;
```

определяет текущие дату/время, полученные от сервера. Примерное время задержки ответа от сервера записывается в свойстве

```
RoundTripDelay: Cardinal;
```

Чтобы синхронизировать работу клиентского компонента с серверным, надо вызвать метод

```
function SyncTime: boolean;
```

который вернет значение True, если связь и синхронизация прошли успешно.

Компонент TIdDICTServer предоставляет Интернет-пользователям доступ к стандартным базам спецификаций *DICT,* которые представляют собой набор словарных определений для различных национальных языков. Для каждой из команд протокола *DICT,* функционирующего в режиме «запрос-ответ», в компоненте реализован обработчик соответствующего события.

Компонент TIdQOTDServer реализует совсем простой протокол, который определяет текущую часть дня (утро, день, вечер, ночь: спецификация *RFC 865)* в обработчике события OnCommandQOTD. Применяется обычно в дополнение к компоненту TIdDayTimeServer.

Клиентское приложение для QOTD можно создать с помощью компонента TIdQOTD. Оно содержит единственное важное свойство

```
Quote: String;
```

в которое записывается время дня при установлении связи с *QOTD*-сервером.

Компонент TIdEchoServer — это простой «эхо»-сервер для тестирования работы *IP*-соединений. Вся информация, посылаемая клиентами (TIdEcho), автоматически возвращается от сервера к ним обратно. Время ответа в миллисекундах клиентский компонент получает в свойстве

```
EchoTime: Cardinal;
```

Сама посылка произвольной тестовой строки серверу выполняется с помощью метода

```
function Echo(AText: String): String;
```

Компонент TIdFingerServer дает возможность создания собственных серверов для поиска информации о конкретных пользователях сети (протокол *Finger).* Обрабатывает событие On Command Finger, которое в качестве параметра передает запрошенное имя пользователя.

Клиентский компонент TIdFinger задает запрос в свойстве

```
CompleteQuery: String;
```

(в формате 'user@host') или в свойстве Query (сокращенная форма). Если значение свойства VerboseOutput равно True, то у сервера запрашивается детальная информация о пользователе.

Обращение к *Finger*-серверу и получение результата происходит с помощью метода

```
function Finger: String;
```

Компонент TIdGopherServer реализует протокол *Gopher (RFC 1436),* позволяя дистанционно работать с иерархически организованной системой документов. Форматирование и передача пользовательских меню осуществляется на основе методов ReturnGopherItem и Send DirectoryEntry.

Клиентский компонент TIdGopher запрашивает меню от сервера через метод GetMenu. Метод GetFile позволяет получить с сервера нужный файл, метод Search — выполнить поиск в базе документов в соответствии с заданным запросом.

Компонент TIdHostNameServer позволяет создавать свои собственные серверы имен, преобразующие *IP*-адреса, корректно обращаться к шлюзам и другим сетям и т. п. Вся функциональность реализуется в обработчиках событий (реакциях на стандартные команды спецификации *HOSTNAME RFC 953).*

Компонент TIdIRCServer дает возможность создавать собственные чат-серверы произвольной сложности (спецификация *IRC server, RFC 1459).* Пример подобного сервера будет рассмотрен при обсуждении работы с сокетом.

Компонент TIdIM AP4Server выполнен специально для почтовых приложений, предоставляющих пользователям доступ к «почтовым ящикам» из стандартных клиентских программ. TIdIMAP4Server реализует спецификацию *RFC 2060,* позволяющую работать с удаленными почтовыми папками. Вся его функциональность реализуется на уровне обработчиков стандартных команд протокола *IMAP4.*

Клиентский компонент TIdDNSResolver — наследник класса TIdUDPClient. Применяется для обращения к службе имен *DNS* (взаимное преобразование из текстовой записи *Web-адреса URL* в *IP-адрес* и обратно). Запрос на расшифровку/преобразование готовится в свойствах DNSAnList, DNSARList, DNSHeader, DNSNSList и DNSQDList. Результирующий текстовый запрос к *DNS*-серверу формируется автоматически в свойстве

```
QPacket: string;
```

при обращении к методу

```
procedure ResolveDNS;
```

В результате его работы заполняются новой информацией указанные выше свойства, имена которых начинаются с символа *DNS,* а текстовый ответ возвращается в свойстве

```
RPacket: string;
```

Еще одно характеризующее ответ сервера свойство RequestedRecords имеет тип

```
TRequestedRecords = set of TRequestedRecord;
```

В частности, если в качестве значения этого свойства указать константу cA, то заданный в запросе текстовый адрес сервера будет преобразован в псевдочисловое значение (в формате '*.*.*.*', например '250.1.2.3'), возвращаемое в свойстве

```
TIdDNSResourceItem.RData.HostAddrStr
```

где TIdDNSResourceItem — элемент (запись) коллекции DNSAnList, хранящий необработанный ответ сервера.

Наиболее простой способ использования данного компонента — метод

```
procedure ResolveDomain(const ADomain: string);
```

в качестве параметра которому задается доменное имя в текстовом формате. Заполнение всех полей DNS* при этом происходит автоматически.

Компонент TIdIcmpClient реализует протокол *ICMP.* Он не требует наличия сервера в прямом его понимании, а используется для трассировки маршрутов *IP*-пакетов и для так называемого пинга (*Ping*), позволяющего определить, в каком состоянии находится тот или иной компьютер в ГСР-сети,

Свойство ReceiveTimeout задает время ожидания ответа от сервера в миллисекундах, свойство

```
ReplyStatus: TReplyStatus;
```

содержит структуру, хранящую ответ удаленного компьютера. Посылка эхо-пакета и ожидание ответа выполняются вызовом метода

```
procedure Ping;
```

Можно также задать собственный обработчик события OnReply, возникающего при получении ответа па *ICMP*-посылку.

Компонент TIdIcmpClient основывается на классе TIdRawClient, являющемся наследником класса TIdRawBase. Класс TIdRawClient позволяет создавать нестандартные сетевые связи, в частности программно задавать *IP*-заголовок (такая возможность пока доступна только при работе в системе *Windows 2000).* В то же время данный класс использовать в приложениях напрямую не рекомендуется. Он предназначен прежде всего для создания на его основе собственных компонентов. Помимо стандартных свойств Host и Port, определяющих конкретные соединения, он позволяет задавать название реализуемого протокола (свойство Protocol), а также максимально допустимое время ожидания ответа от приложения, с которым установлена связь (свойство ReceiveTimeout),

Компонент TIdSNTP позволяет создавать клиентские компоненты, поддерживающие протокол *SNTP,* предназначенный для синхронизации времени через Интернет с

точностью от 1 до 50 миллисекунд, вне зависимости от того, насколько далеко физически расположены компьютеры, где выполняются соответствующие приложения.

Таблица 10.3. Свойства и методы компонента TIdSNTP

Описание	Назначение
AdjustmentTime	Временной сдвиг для локальной области, полученный от SNTP-сервера. Свойство доступно только для чтения
DateTime	Клиентское время
RoundTripDelay	Продолжительность временной задержки в процессе получения клиентского времени
function SyncTime	Обновление (синхронизация) часов на локальном компьютере. Возвращает значение True, если обновление локальных часов выполнено успешно

Компонент TIdMappedPortTCP является базовым для создания собственных прокси-серверов. Позволяет прослушивать порты *IP*-серверов и, в зависимости от поступающей информации о клиентах, разрешать или запрещать соединение с определенным сервером в системе. Координаты контролируемого сервера указываются в свойствах:

```
MappedHost: string; // IP-адрес
MappedPort: Integer; // порт
```

Функциональность компонента TIdMappedPortTCP задается в обработчике события OnBeforeClientConnect:

```
TBeforeClientConnectEvent = procedure (
  ASender: TComponent;
  AThread: TIdPeerThread;
  AClient: TIdTCPClient) of object;
```

Это событие возникает при попытке клиентской программы подсоединиться к прослушиваемому серверу. Параметр AThread определяет серверный поток, создаваемый для клиентского соединения, а параметр AClient — клиентский TCP-компонент.

Компонент TIdNNTPServer дает возможность создавать свои собственные новостные серверы с помощью набора обработчиков событий, возникающих при получении от клиентов команд спецификаций *RFC 977* и *2980* протокола *NNTP*.

Для создания собственных клиентских новостных программ предназначен компонент TIdNNTP. Он содержит большой набор свойств, методов и событий для поддержки протокола *NNTP*.

Компонент TIdTelnetServer реализует функциональность *Telnet-сервера,* позволяя подключаться к серверу клиентам с удаленных алфавитно-цифровых терминалов. Свойство

```
LoginAttempts: Integer;
```

задает максимально допустимое число неудачных попыток подключения к серверу, после которых соединение разрывается.

При установлении удачной связи клиенту высылается сообщение, заданное в свойстве

```
LoginMessage: String;
```

Метод

```
procedure DoConnect(AThread: TIdPeerThread); override;
```

формирует *Telnet*-соединение, инициализируя свойство Data, имеющее тип TTelnetData.

Компонент TIdWhoIsServer позволяет реализовать работу службы WhoIs, представляющей собой несложную систему запросов к базе данных по именам пользователей, зарегистрированных в глобальной системе имен. Обработка запроса выполняется в реакции на событие OnCommandLookup, которое в качестве параметра получает имя пользователя.

На клиентской стороне применяется компонент TIdWhoIs, который обращается к серверу с помощью метода

```
function WhoIs(const ADomain: string): string;
```

Запрос указывается в свойстве ADomain.

Серверный компонент TIdDISCARDServer реализует протокол *Discard Protocol (RFC 863)*. Применяется, как правило, для отладки программ и измерения производительности системы. Все данные, поступающие на сервер, сразу же уничтожаются.

Компонент TIdTunnelMaster предназначен для создания виртуальных *IP*-каналов. Фактически работает как прокси-сервер, позволяя дополнительно выполнять проверки *IP*-пакетов, проводить авторизацию пользователей, управлять *IP*-каналами, оптимизировать трафик в сети, собирать всевозможную статистику и организовывать так называемые *виртуальные частные сети (Virtual Private Networks)*.

Данный компонент неразрывно связан с компонентом TIdTunnelSlave — серверным объектом, поддерживающим *IP*-соединение со стороны клиента. Все потоки клиентских соединений обрабатываются компонентом TIdTunnelMaster, он выполняет предварительную обработку данных и передает их компоненту TIdTunnelSlave. Тот, в свою очередь, выполняет перераспределение обработанных клиентских потоков, поддерживая работу всевозможных сервисных соединений, например виртуальных *IP*-соединений.

Свойства и методы компонента TIdTunnelSlave приведены в таблице.

Таблица 10.4. *Свойства и методы компонента Прокси-клиент (TIdTunnelSlave)*

Свойства и методы	Назначение
Bindings	Список соединений (TIdTunnelSlave)
MasterHost MasterPort	Настройка связи с сервером TIdTunnelmaster
Socks4	Определяет, требуется ли авторизация клиентского соединения на сервере. Если значение свойства равно True, то информация, поступающая от клиента, должна содержать информацию о версии Socks, номере порта и IP-адресе прокси-сервера Socks, а также пользовательский идертификэтор. В противном случае номер порта для соединения с сервером берется из свойства DefaultPort

Таблица 10.4. Свойства и методы компонента Прокси-клиент (TIdTunnelSlave) (продолжение)

Свойства и методы	Назначение
procedure GetStatistics (Module: Integer; var Value: Integer); procedure SetStatistics (Module: Integer; Value: Integer);	Сбор статистики работы сервера. Первый параметр может принимать одно из следующих значений: NumberOfConnectionsType (число потоков или сервисных соединений), NumberOfPacketsType (число поддерживаемых пакетов данных), CompressionRatioType (степень сжатия данных), CompressedBytesType (число сжатых байтов), BytesReadType, BytesWriteType (число считанных и записанных байтов). Результирующее значение записывается во второй параметр. Для обновления какого-либо текущего параметра, например для его сброса, можно воспользоваться методом SetStatistics

Компонент TIdLogDebug предназначен для сбора всевозможной статистики о работе Интернет-компонентов и анализа работы распределенной системы, созданной с помощью компонентов *Indy*. Он является наследником класса TIdLogBase, который содержит несколько виртуальных методов для записи текстовых строк л протокол и способен, в частности, обрабатывать такие события в процессе соединения, как установка связи, передача или прием данных и разрыв соединения.

Компонент ведет подробный протокол работы объектов *Indy,* записывая собираемую информацию в файл. Если значение свойства LogTime равно True, то в записываемую в файл протокола информацию (в начало каждой строки) добавляются текущие дата и время. Процесс записи сообщения в файл начинается после того, как в свойство Active записывается значение True, и прекращается, когда это свойство становится равным False. Свойство

```
Target: TIdLogDebugTarget;
```

определяет объект, в который будут выводиться сообщения. Значения данного свойства могут быть такими:

- ltFile — данные записываются в файл;
- ltDebugOutput — данные записываются в стандартный отладочный поток *Windows* (Win32 API Debug Output);
- ltEvent — данные никуда не поступают, а только формируют события OnLogItem.

Компонент TIdNetworkCalculator предназначен для определения корректности и доступности *IP*-адресов в сети. Чтобы получить список всех доступных адресов в сети, надо воспользоваться методом

```
procedure FillIPList;
```

Он заносит список адресов в формате TStrings в свойство ListIP. Адрес локального компьютера в сети задается с помощью свойства

```
NetworkAddress: TIpProperty;
```

но так как, например, в глобальной сети этих адресов очень много, для отбора можно задать фильтр в свойстве

```
NetworkMask: TIpProperty;
```

Чтобы получить последний IP-адрес из списка всех доступных адресов, можно воспользоваться методом

```
function EndIP: String;
```

Для проверки, принадлежит ли конкретный адрес к текущей сети, применяется метод

```
function IsAddressInNetwork(Address: String): Boolean;
```

Общее количество доступных *IP*-адресов возвращается методом

```
function NumIP: Integer;
```

а самый первый доступный адрес — методом

```
function StartIP: String;
```

Компонент TIdVCard позволяет обрабатывать электронные бизнес-карты в соответствии со спецификациями *VCard 2.1* и *VCard3-0 (RFC 2425* и *2426;* www.imc.org/pdi/pdiproddev.html). Его основной метод

```
procedure ReadFromTStrings(s: TStrings);
```

позволяет заполнить все поля виртуальной карты из массива строк.

Протоколы *DICT, Gopher, Finger, Telnet, WhoIs* сегодня практически не используются, и соответствующие компоненты реализованы в наборе *Indy* преимущественно для сопровождения и стыковки со старыми, но пока еще работающими сетевыми службами. Примеры приложений, использующих эти и другие компоненты *Indy,* входят в стандартную поставку *Delphi 7* (каталог Delphi/Demos/Indy).

Панель Indy Misc

Здесь расположен набор дополнительных компонентов, расширяющих функциональные возможности ранее рассмотренных клиент-серверных компонентов в архитектуре *Indy*.

Компонент Антиблокировщик (TIdAntiFreeze)

Работа клиент-серверной модели *Indy* подразумевает монопольное использование ресурсов приложения. Например, когда какой-нибудь клиентский объект *Indy* выполняет обращение к серверу с помощью одного из своих методов, работа всего клиентского приложения приостанавливается до тех пор, пока этот объект не получит ответа на свой запрос. В такие моменты пользователю не доступны никакие элементы пользовательского интерфейса.

Компонент TIdAntiFreeze решает эту проблему — система *Indy* самостоятельно выполняет вызовы метода Application.ProcessMessage для анализа и обработки очереди сообщений *Windows*.

ЗАМЕЧАНИЕ В программе разрешается использовать только одну копию данного компонента.

Главные свойства и методы Антиблокировщика наследуются из класса TIdAntiFreezeBase. Активизируется объект с помощью стандартного свойства Active. Свойство ApplicationHasPriority, если его значение установить равным True, определяет, что приложение будет выполняться в приоритетном режиме по отношению к системе *Indy*. Свойство OnlyWhenIdle позволяет данному компоненту функционировать, только когда никаких реальных действий в приложении не происходит (оно находится в состоянии ожидания).

Компонент Преобразование даты/времени (TIdDateTimeStamp)

Этот компонент характеризуется множеством свойств и методов, позволяющих преобразовывать дату и время в различные форматы в соответствии с требованиями пользователя и операционной системы, а также различных клиентских программ. Позволяет различными способами устанавливать дату, изменять ее, прибавляя или вычитая различные временные промежутки от миллисекунд до лет, а также формировать значение даты и времени в самых различных видах (например, в виде дня недели или месяца, названия этого дня или месяца), показывать минуту часа, минуту дня, минуту года и т. д,

Компонент IP-информация (TIdIPWatch)

Компонент предоставляет различную информацию о текущем *IP*-адресе приложения — точнее, компьютера, на котором это приложение выполняется. Позволяет определить, подключен ли компьютер к сети и каков его *IP*-адрес.

Когда данный компонент становится активным, его *IP*-адрес автоматически записывается в свойство

```
CurrentIP: string;
```

Это строка, доступная только для чтения. Чтобы определить, подключен ли компьютер к сети, можно воспользоваться свойством

```
IsOnline: Boolean;
```

Надо отметить, что *IP*-адрес компьютера доступен вне зависимости от того, подключен ли компьютер к сети, так как при отсутствии подключения этот адрес равен стандартному значению 127.0.0.1. Конечно, для корректной работы данного компонента необходимо, чтобы на компьютере было установлено программное обеспечение, поддерживающее *TCP*-протокол.

При изменении значения свойства IsOnline (оно доступно только для чтения), генерируется сообщение OnStatusChanged. Таким образом можно отслеживать, в какие моменты компьютер подключается к сети или отключается от нее. Периодичность опроса состояния компьютера задается в свойстве

```
WatchInterval: Cardinal;
```

в миллисекундах. Наряду с анализом состояния компьютера по отношению к сети проверяется также, не изменился ли его *IP-адрес,* который может присваиваться динамически при периодических подключениях к глобальным сетям. В последнем

случае бывает полезно отслеживать ход таких подключений. Для этого значения свойства HistoryEnabled надо установить равными True. Тогда при изменении *IP*-адреса его предыдущее значение будет записываться в файл, заданный в свойстве

```
HistoryFilename: string;
```

Кроме того, список *IP*-адресов хранится в свойстве

```
IPHistoryList: TStringList;
```

Максимальная длина такого списка задается свойством MaxHistoryEntries, а последний *IP*-адрес можно получить с помощью свойства

```
PreviousIP: string;
```

Если требуется немедленно определить *IP*-адрес компьютера, не дожидаясь окончания интервала опроса, определяемого свойством WatchInterval, можно вызвать функцию, которая выполнит проверку подключения компьютера к сети и определит его адрес немедленно:

```
function ForceCheck: Boolean;
```

Чтобы восстановить всю историю *IP*-адресов (свойство PHistoryList), вызывается метод, загружающий этот список из файла, заданного свойством HistoryFilename:

```
procedure LoadHistory;
```

Сохранение этого списка в файле выполняется с помощью метода

```
procedure SaveHistory;
```

Компонент Менеджер потоков (TIdThreadMgrDefault)

При обращении клиентского приложения к любому серверу *Indy* на последнем создается отдельный поток (в рамках самого сервера), который закрывается только после того, как завершается соединение с клиентской программой. Для эффективного управления созданием и уничтожением всех серверных потоков предназначен данный компонент. Его достаточно разместить на форме, а затем в свойстве ThreadMgr серверного компонента (например, TIdTCPServer) выбрать этот Менеджер в раскрывающемся списке (рис. 10.3).

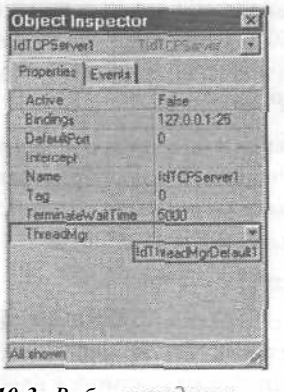


Рис. 10.3. *Выбор менеджера потоков*

Компонент Менеджер пула потоков (TIdThreadMgrPool)

Для некоторых видов серверов, реализующих, в частности, протоколы *HTTP, Time, Gopher,* оказывается более эффективным не создавать и уничтожать новые потоки, а постоянно хранить некоторое число потоков в пуле и активизировать их по мере необходимости. Таким серверам лучше использовать не Менеджер потоков, а Менеджер пула потоков. Размер пула задается в свойстве PoolSize.

Данный компонент более эффективен при использовании в интенсивно работающих серверных приложениях, но он может потребовать дополнительных расходов ресурсов на поддержку пула, а неправильное определение размера пула может существенно сказаться (в худшую сторону) на производительности всей системы.

Компонент HTTP-сервер (TIdHTTPServer) и создание Web-сервера

В данном разделе рассматривается способ создания простейшего *Web-сервера.* Точнее, *HTTP*-сервера, работающего как *Web-сервер в* прямом смысле этого определения *(IIS, PWS, Apache* и т. п.). В нижеследующем примере создается не просто прикладной программный модуль, выполняемый на сервере и выполняющий конкретные функции вроде формирования *HTML*-страницы на основе базы данных, а системное приложение, обрабатывающее *HTTP*-запросы из любых клиентских программ, поддерживающих *HTTP*-протокол, и возвращающее соответствующие им *HTML*-страницы для просмотра и отображения.

Создадим новую форму и разместим на ней компонент TIdHTTPServer (*HTTP-сервер).* В свойстве Bindings укажем *IP*-адрес компьютера в сети и порт, через который будут обрабатываться *HTTP*-запросы (как правило, это восьмидесятый порт — 80). В свойстве SessionTimeOut указывается время ожидания сервера в процессе передачи информации клиентскому приложению, если оно по каким-то причинам не способно принять соответствующие данные (например, оборвалась связь). При желании *в* свойстве ServerSoftware можно ввести строку, которая будет характеризовать название *Web*-сервера: Apache или Мой web-сераер 1.0.

Как ни удивительно, но вся функциональность нашего *Web*-сервера поместится в одной строчке: в обработчике события OnCommandGet, которое возникает, когда к серверу поступает запрос от клиентского приложения, например обычного браузера:

```
procedure TForm1.HTTPServerCommandGet(AThread: TIdPeerThread;
   RequestInfo: TIdHTTPRequestInfo;
   ResponseInfo:TIdHTTPResponseInfo),•
begin
   HTTPServer.ServeFile(AThread, ResponseInfo,
      'c:\MyWebServer\root'+RequestInfo.Document);
end;
```

Параметр AThread имеет тип TIdPeerThread. Он характеризует соединение с клиентской программой и рассматривался нами при создании ГСР-сервера. Параметр RequestInfo определяет характеристики запроса, а параметр ResponseInfo — характеристики ответа сервера.

Все, что требуется от создаваемого нами приложения, — это отправить запрошенный *HTTP*-файл клиенту, что и выполняется с помошью метода ServeFile. Первый параметр в нем — это объект, описывающий данное соединение (AThread). Второй параметр — информация об ответе сервера (она передается *в* оригинальном виде без изменений), а третий параметр — полный путь к запрошенному файлу. Этот путь, очевидно, проходит через каталоги компьютера, локального для *HTTP*-сервера. Такие каталоги пользователю браузера (или любой другой программы, клиентской по отношению к *HTTP*-серверу) не видны и не доступны.

Обратите внимание на запись пути к документу при вызове метода ServeFile. Путь не завершается наклонной чертой, так как свойство Document пользовательского запроса представляет собой строку, просто выделенную из полного обращения к серверу. Путь — это *IP*-адрес или символьное имя сервера, которое затем преобразовывается в *IP*-адрес системой доменных имен (например: http://www.MyWebServer.ru/index.html или http://225.1.2.3/index.html). Первая часть обращения представляет собой название протокола запроса (HTTP://), вторая — символьное имя сервера (оно может быть представлено и в цифровом виде). Следом за ним идет путь к документу, включая символ наклонной черты — /index.html! Поэтому указывать наклонную черту следом за локальным путем (c:\MyWebServer\root) не надо, она в запросе уже есть.

Переводить объект в активное состояние следует в момент создания формы следующим образом:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
   HTTPServer.Active := True
end;
```

Теперь программу можно откомпилировать и запустить, а в соответствующий каталог (в нашем случае — c:\MyWebServer\root\) записать набор *HTML*-файлов. Для наглядности будем считать содержимое файла index.htm таким:

```
<HTML>
<HEAD>
<TITLE>
Главная страница
</TITLE>
</HEAD>
<BODY bgColor=white>
<CENTER>
```

```
Это главная страница моего web-сервера]
</CENTER>
</BODY></HTML>
```

Теперь можно обратиться к такому серверу напрямую из браузера. *HTTP*-сервер, очевидно, должен быть запущен на подключенном к сети компьютере, настроен на его *IP*-адрес, и никакие другие *Web*-серверы на этом компьютере работать не должны.

Мы рассмотрели простейший пример создания *Web*-сервера. В дополнение к данному примеру надо сделать несколько замечаний. Подобные серверы, массово применяемые на практике, например *Apache,* обладают множеством дополнительных возможностей. В частности, в нашем случае в клиентской программе можно указать путь к документу index.html, соответствующий его реальному пути. Это запишется так:

```
http://www.MyWebServer.ru/MyWebServer/root/index.html
```

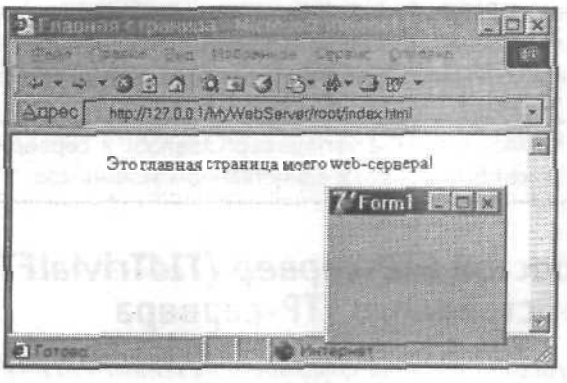


Рис. 10.4, *Самодельный Web-сервер в работе*

В реальной жизни предоставлять пользователям открытую информацию о структуре каталогов на локальном компьютере, где выполняется *Web-сервер,* нежелательно по самым разным причинам — прежде всего, по причине безопасности. Для ликвидации данного недостатка введено понятие виртуальных каталогов. Для этого путь к документу подвергается синтаксическому разбору и программа заменяет названия виртуальных каталогов реально существующими.

Для примера, предыдущий запрос в браузерах может указываться так:

```
http://www.MyWebServer.ru/www/index.html
```

а затем в самой серверной программе каталог *WWW* заменяется реальным каталогом MyWebServer/root. В случае, когда пользователь не указывает никакого *HTML-файла,* подразумевается, что происходит обращение к некоей *HTML*-странице по умолчанию. Для корневого каталога *Web*-сервера это, как правило, файл index.html или index.htm. Запрос по умолчанию обычно характеризуется тем, что свойство Document содержит либо пустую строку, либо один символ «/».

У *HTTP*-сервера могут запрашиваться не только *HTML*-страницы, но и другие файлы, например сценарии, написанные на Бейсике, *Pert* или иных языках программирования. В таком случае серверное приложение должно самостоятельно выполнять интерпретацию подобных сценариев, что само по себе достаточно трудоемкое занятие.

Немного более сложный и законченный пример *Web*-сервера можно найти в демонстрационном примере, входящем в стандартную поставку *Delphi*, — Demos/Indy/HTTPServer.

В ряде случаев для работы распределенной системы в условиях повышенной безопасности соединение с клиентскими программами желательно организовывать по так называемому безопасному протоколу *HTTPS*, представляющему собой расширение протокола *HTTP* и содержащему средства повышения надежности и защищенности передаваемой информации. Это требуется, в частности, для систем электронной торговли.

Чтобы реализовать в своей программе поддержку протокола *HTTPS*, надо прежде всего загрузить с сайта www.intelicom.si динамические библиотеки, поддерживающие функциональность Secure Sockets Layer. Затем необходимо добавить к приложению HTTP-сервера новый компонент TIdServerInterceptOpenSSL и выбрать его в свойстве Intercept объекта TIdHTTPServer (в раскрывающемся списке).

ВНИМАНИЕ Компонент TIdServerInterceptOpenSSL в серверном приложении может быть только в единственном экземпляре.

Компонент Простой FTP-сервер (TIdTrivialFTPServer) и создание собственного FTP-сервера

В данном разделе рассматривается создание простейшего *FTP*-сервера по аналогии с предыдущим примером создания *Web*-сервера. На основе этого примера можно создать свой собственный *FTP*-сервер, позволяющий принимать и передавать файлы с локального компьютера, подключенного к TCP-сети, по *TFTP*-протоколу (*TFTP — Trivia! FTP*). Никакого дополнительного программного обеспечения для этого не потребуется.

Создадим пустое приложение и разместим на форме компонент TIdTrivialFTPServer, хранящий в себе всю базовую функциональность *FTP*-сервера. Он связывается с конкретным *IP*-адресом с помощью свойства Bindings. Порт, используемый протоколом *TFTP* по умолчанию, — 69. Свойство BufferSize определяет значение буфера, в котором временно накапливается информация, поступающая через соединение с клиентским приложением. Перевод *FTP*-сервера в активное состояние осуществляется в момент создания формы:

```
procedure TfrmMain.FormCreate(Sender: TObject);
begin
    IdTrivialFTPServer1.Active := True;
end;
```

При обращении к серверу с запросом о передаче или приеме файла генерируется сообщение OnReadFile или OnWriteFile. В обоих случаях понадобится записать всего по одному оператору, присваивающему параметру этого обработчика AStream (тип TStream) соответствующее значение файлового потока. Такой поток либо открывается для чтения в случае, когда к *FTP*-серверуобращаются с запросом на поставку файла, либо этот файл создается на компьютере, где работает *FTP*-сервер, а его содержимое принимается из сети:

```
procedure TfrmMain.IdTrivialFTPServer1ReadFile(Sender:
  TObject;
    var FileName: String; const PeerInfo: TPeerInfo;
    var GrantAccess: Boolean; var AStream: TStream;
    var FreeStreamOnComplete; Boolean);
begin
    AStream := TFileStream.Create(FileName, fmOpenRead);
end;

procedure TfrmMain.IdTrivialFTPServer1WriteFile(Sender:
  TObject;
    var FileName: String; const PeerInfo: TPeerInfo;
    var GrantAccess: Boolean; var AStream: TStream;
    var FreeStreamOnComplete: Boolean);
begin
    AStream := TFileStream.Create(FileName, fmCreate);
end;
```

В качестве *FTP*-клиентадля наглядности лучше всего взять готовый пример, расположенный в каталоге Delphi 7/Demos/Indy/TrivialFTPClient. В нем используется компонент TIdTrivialFTP, представляющий собой клиентскую *FTP*-программу.

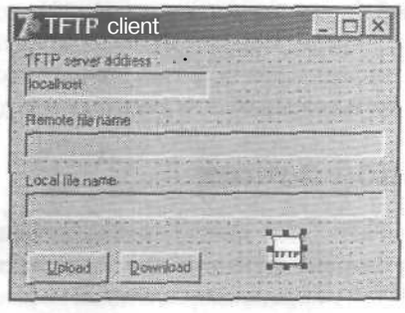


Рис. 10.5. *Проектирование FTP-клиента*

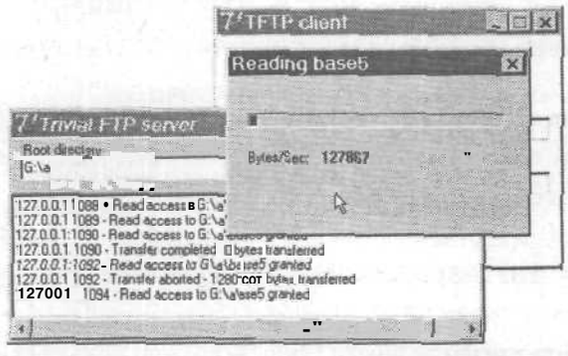
Он не требует при использовании особых настроек, только в свойстве ReceiveTimeout можно указать время ожидания в миллисекундах. Оно определяет, сколько времени

может ожидать клиентская программа ответа от *FTP*-сервера, например в случае плохой связи.

Загрузка файла с сервера и передача его серверу происходит в результате нажатия кнопок btnDownload и btnUpload. Весь процесс приема и передачи файлов выполняется автоматически с помощью двух методов Get и Put компонента TIdTrivialFTP. В качестве параметров им передаются: название файла, расположенного на удаленном *FTP*-сервере (ASourceFile), и название локального файла (ADest). Параметр AAppend принимает значение *True*, если происходит запись в конец уже существующего файла:

```
procedure Get(const ASourceFile: string; ADest: TStream);
procedure Put(const ASource: TStream; const ADestFile:
  string; const AAppend: Boolean);
```

Запустим созданный *FTP*-сервер и эту клиентскую программу. Для проверки работоспособности можно попробовать отправить файлы на локальный сервер и принять оттуда. Важно понять, что при этом выполняется не просто копирование файлов средствами операционной системы, а их передача между приложениями в пакетном режиме протокола *TCP*.



Рис, 10.6. Работа самодельного FTP-сервера

Полноценные HP-серверы устроены, конечно, гораздо сложнее. Все они требуют авторизации пользователей (ввода их имени и пароля), поддерживают работу с виртуальными каталогами и предоставляют множество других возможностей, например, выполнение простейших команд, позволяющих получить список файлов в своем виртуальном каталоге, создавать эти каталоги и т. д.

Работа с электронной **почтой**

Для приема и отправки электронной почты предназначены компоненты панели FastNet. Она обеспечивают базовые функции для поддержки необходимых протоколов, но не всегда удобны для создания законченных коммерческих приложений, в которых желательно предоставлять пользователю максимум комфортных возможностей. Дополнительно на панели Indy Clients имеется несколько компонентов, значительно облегчающих программисту создание клиентских (не серверных!) почтовых программ и обработку писем.

Компонент Прием почты (TIdPOP3)

Для подключения к почтовому серверу по протоколу *POP3* (прием почты) требуется указать название этого сервера (свойство Host), а также имя пользователя и пароль (свойства UserID и Password) (рис. 10.7).

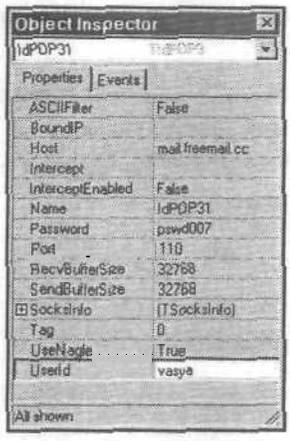


Рис. 10.7. *Настройка почтового клиента*

Их можно задавать и в процессе работы программы, например копируя в соответствующие строки из полей на форме. Следующий код выполняет проверку, подключен ли клиент к серверу, заносит название *POP3*-сервера, номер порта, имя пользователя и пароль в соответствующие свойства объекта TIdPOP3, после чего выполняет подключение к серверу, вызывая метод Connect

```
if POP.Connected then POP.Disconnect;
POP.Host := Pop3ServerName;
POP.Port := Pop3ServerPort;
POP.UserId := Pop3ServerUser;
POP.Password := Pop3ServerPassword;
POP.Connect;
```

Письма, поступившие конкретному адресату (он определяется именем пользователя и паролем), физически хранятся на почтовом *POP*-сервере. Получить их полный список можно с помощью метода

```
function CheckMessages: longint;
```

который возвращает клиентской программе общее число хранимых на сервере сообщений. Программа затем может принять любое из этих сообщений либо полностью, либо только заголовок. В заголовке хранится тема сообщения, почтовый адрес, с которого оно поступило, и различная служебная информация, например размер сообщения, наличие присоединенных файлов и др. Чтобы получить заголовок конкретного сообщения, можно воспользоваться методом

```
function RetrieveHeader(const MsgNum: Integer;- AMsg:
  TIdMessage):,Boolean;
```

MsgNum — номер сообщения из списка. Заголовок сообщения записывается в параметр AMsg.

Отметим, что нумерация сообщений при работе с компонентом TIdPOP3 начинается с единицы.

В следующем цикле происходит последовательное извлечение заголовков всех поступивших сообщений. При этом они поочередно автоматически расшифровываются и записываются в переменную MyMsg, которая имеет тип TIdMessage.

```
FMsgCount  := POP.CheckMessages;
for i :- 1 to FMsgCount do
   RetrieveHeader(i, MyMsg);
```

Для приема всего сообщения служит метод

```
function Retrieve(const MsgNum: Integer; AMsg:
  TIdMessage): Boolean;
```

После того как данный метод отработает, сообщение на сервере останется! Это особенность работы протокола *POP3*. Чтобы удалить конкретное сообщение, надо использовать метод

```
function Delete(const MsgNum: Integer): Boolean;
```

который реально помечает предназначенное для удаления сообщение на сервере, делая его невидимым для клиентских программ. Физическое удаление помеченных сообщений происходит при обращении к методу отключения от *POP3*-сервера:

```
procedure Disconnect;
```

Компонент Отправка почты (TIdSMTP)

Реализация отправки почты не сильно отличается от реализации приема. Для этого предназначен компонент TIdSMTP. Он расширен несколькими дополнительными свойствами. Свойство AuthenticationType определяет способ авторизации на *SMTP*-сервере, когда либо требуется ввод имени пользователя и пароля (значение atLogin), либо этого делать не обязательно (значение atNone). В свойстве MailAgent можно указать собственное оригинальное название программы, которая отправляет сообщения.

В следующем небольшом примере показывается, как содержимое многострочного текстового поля заносится в свойство Body объекта IdMsgSend (тип TIdMessage), определяющее текст сообщения, и затем это сообщение отправляется адресату на *SMTP*-сервер. Процесс отправки физически происходит в момент вызова метода

```
procedure Send(AMsg: TIdMessage); virtual;
```

которому передается единственная переменная, имеющая тип TIdMessage.

```
IdMsgSend.Body.Assign(Memo1.Lines);
// ... - заполнение других полей сообщения
SMTP.Connect; // соединение с SMTP-сервером
try
    SMTP.Send(IdMsgSend),-
finally
    SMTP.Disconnect;
end;
```

Кодировка пересылаемых данных

Иерархия кодировочных компонентов

При передаче различной информации с использованием популярных протоколов, например почтовых, нередко возникают проблемы, когда информация представляет собой двоичные данные (присоединенные файлы или текст, написанный не латинским шрифтом). Такие проблемы связаны с тем, что ряд протоколов поддерживает обработку только текстовой информации — первых 128 символов кодировочных таблиц. Протоколы обычно работают только с семибитными символами, и в каждом восьмибитном байте старший бит как бы обрезается, не учитывается.

Обойти проблему позволяет ряд стандартных технологий, которые представляют двоичные восьмибитные данные в семибитном виде — с помощью последовательности печатных символов. Результирующий файл, как правило, оказывается значительно больше исходного двоичного файла. Это связано с тем, что каждый символ файла исходно считается двоичным и для его представления требуется два текстовых символа. То есть, величина закодированного файла примерно в два раза больше, чем его обычный размер.

Шифрование двоичных файлов применяется, в частности, во всех почтовых программах, которые способны работать с обычной текстовой информацией, а для пересылки двоичных данных они преобразовываются в специальные форматы. Такая работа выполняется самими почтовыми программами, поэтому для разработки собственных приложений (тех же почтовых клиентов), которые помимо текстовой информации должны предоставлять пользователям возможность передавать и двоичные файлы, можно использовать готовые компоненты.

В *Delphi 7* для этого введен новый класс TIdCoder. На его основе создано девять компонентов, поддерживающих основные протоколы шифрования и кодирования. Класс TIdCoder — базовый. Напрямую его использовать в своих программах не рекомендуется — лучше воспользоваться одним из компонентов-наследников этого класса. В то же время класс TIdCoder сочетает в себе все основные свойства и методы, характерные для группы компонентов кодирования (табл. 10.5).

Таблица 10.5. *Свойства, события и методы класса TIdCoder*

Свойства или метод	Назначение
AddCRLF	Имеет значение True, если в конец результирующей строки добавляется символ EOL (перевод строки и возврат каретки, CR + LF)
AutoCompleteInput	По умолчанию имеет значение False. При этом функция CodeString, формирующая входную строку для методов кодирования, вызывает функцию GetNotification, возвращающую либо кодированные данные, либо одно из уведомлений о состоянии исходных данных. Если значение равно False, вызывается функция CompletedInput, что может потребоваться для предварительной обработки входных данных
BufferSize	Размер входного буфера при поставке данных кодировщику
ByteCount	Число байтов, которые обработал кодировщик. Доступно только для чтения
BytesIn	Число байтов, поступивших на вход кодировщика. Доступно только для чтения
BytesOut	Размер результирующих закодированных или раскодированных данных в байтах. Рекомендуется применять вместе со свойством BytesIn для проверки корректности работы кодировщика
FileName	Имя файла, который может потребоваться кодировщику при выполнении отдельных методов
IgnoreCodedData	Настройка работы события OnCodedData, которое вызывается, когда сформированы результирующие данные. Если значение данного свойства равно True, сообщение OnCodedData будет вызываться, когда выходные данные формируются полностью, а не изменяются или дополняются. Если значения данного свойства и свойства Ignore Notification равны True, то кодировщик на выходе выдаст пустую строку
Ignore Notification	Если значение данного свойства равно True, то при изменении выходной информации кодировщика никакие события генерироваться не будут
Key	Уникальный ключ, применяющийся в криптографических кодировщиках
OnCodedData	Это событие генерируется при изменении данных, обрабатываемых кодировщиком
Priority	Приоритет работы кодировщика в коллекции кодировщиков
TakesFileName	Требуется задать значение True, если при работе кодировщика требуется промежуточный файл (свойство FileName)
Takes Key	Принимает значение True, если для корректной работы кодировщика требуется задать значение ключа (свойство Key)
UseEvent	Данное свойство определяет, как будет работать система уведомления кодировщика о состоянии выходных данных. Если значение данного свойства равно True, то обработчики OnCodedData и On Notification будут вызываться, как только сформированы выходные данные. В противном случае выходная информация становится доступна с помощью метода CompletedInput, когда ее объем превышает значение, заданное в свойстве BufferSize. Эту же информацию можно получить, напрямую обратившись к методу CodeString

Таблица 10.5. Свойства, события и методы класса TIdCoder(продолжение)

Свойство или метод	Назначение
function CodeString (AStr: String): String;	Базовая функция для первичной кодировки данных. В качестве параметра задается исходная информация для кодирования. Результирующая строка представляет собой закодированную последовательность, затем символ «;» и заданную строку
procedure CodeStringFromCoder (Sender: TComponent; const sOut: string);	Данная процедура позволяет программисту переопределить способ кодирования данных. Ее первый параметр — компонент, сгенерировавший входные данные, а второй параметр — результирующая закодированная строка
function CompletedInput: String; virtual;	Функция выдает результирующее закодированное значение и формирует уведомляющее сообщение о завершении процесса преобразования данных
function GetCodedData: String; virtual;	Функция возвращает результирующее значение, полученное в процессе преобразования данных кодировщиком
function GetNotification: String; virtual;	Функция проверяет состояние находящейся в кодировщике информации на предмет ее изменения или дополнения. Если ее состояние изменилось, формируется уведомляющее сообщение
procedure Reset; virtual;	Сброс состояния кодировщика в исходное состояние и очистка всех внутренних буферов
procedure SetBufferSize (ASize: LongWord); virtual;	Процедура позволяет изменить размер буфера (свойство BufferSize)
procedure SetKey (const key: String); virtual;	Процедура предназначена для создания криптографического ключа (свойство Key)
On Notification	Стандартное событие, позволяющее контролировать процесс кодирования или раскодирования данных. Параметр обработчика принимает одно из следующих значений: CN_CODED_DATA (идет процесс кодирования данных); CN_DATA_START_FOUND (кодировщик приступил кобработке кодированных данных); CN_DATA_END_FOUND (обработка входных данных завершена); CN_CODING_STARTED (начат процесс кодирования); CN_CODING_ENDED (кодирование данных завершено); CN_NEW_FILENAME (кодировщик в качестве исходной информации взял данные из файла)

Класс TIdASCIICoder (*ASCII*-кодировщик) является прямым наследником класса TIdCoder. Он применяется, когда для преобразования данных разрешается использовать прямую перекодировочную таблицу. На этом классе базируются компоненты, реализующие технологии взаимно-однозначного преобразования данных. Данный класс, как и его предшественник, явно в программах не применяется. Он характеризуется дополнительным свойством — перекодировочной таблицей:

```
CodingTable: String;
```

Следующим в иерархии идет класс TId3To4Coder. На его основе созданы все компоненты, выполняющие преобразования между двоичными и *ASCII*-форматами.

Компонент Base64-кодировщик (TIdBase64Encoder)

Выполняет преобразование двоичных данных в семибитное текстовое представление в формате *Base64 MIME*. В этом формате используются 64 текстовых символа и символ-разделитель «=». *Base64* является неформальным стандартом для кодирования двоичных и восьмибитных сообщений в почтовых системах и новостных конференциях. Для обратного преобразования (раскодирования) применяется компонент TIdBase64Decoder.

Компонент UUE-кодировщик (TIdUUEncoder)

Этот компонент осуществляет преобразование двоичных данных в семибитное текстовое представление, формируя выходные данные в формате *UUE* в соответствии с требованиями *UUCP*-приложений (*Unix-to-Unix Copy Program*). Данный формат во многом напоминает Base64 и часто применяется для кодирования присоединенных файлов, а также для передачи больших объемов данных, разбиваемых на множество частей. Данный формат разрабатывался для системы *Unix,* поэтому реализующие его компоненты обладают небольшим набором свойств и методов, специфичных для этой платформы.

Таблица 10.6. *Свойства и методы класса TIdUUEncoder*

Описание	Назначение
Privilege	Уровень привилегированности для пользователя Unix. Разрешает или запрещает пользователю выполнять операцию кодирования. Для платформы Windows не нужен. Значение этого свойства включается в заголовок UUEncode-сообщения
TableNeeded	С помощью данного свойства можно создавать собственные таблицы перекодировки. По умолчанию имеет значение False, а для перекодировки применяется таблица UUCodeTable, входящая в модуль *IdCoder3To4.pas*
procedure SetCodingTable(NewTable: String); override;	Задает новую перекодировочную таблицу
procedure SetPrivilege(Priv: Integer);	Задает уровень привилегированности Unix-пользователя

Для выполнения обратного преобразования из текстового представления в двоичный вид служит компонент TIdUUDecoder. По своим характеристикам не отличается от компонента TIdUUEncoder.

Компонент IMF-раскодировщик (TIdIMFDecoder)

Применяется для декодирования сообщений в старом формате *Internet Message Format,* стандарт па который был принят в 1982 г.

Компоненты MD2, MD4, MD5-кодировщик (TIdCoderMD2, TIdCoderMD4, TIdCoderMD5)

Данные компоненты представляют собой реализацию алгоритмов шифрования по стандартам *RSA-MD2/4/5* Эти алгоритмы были опубликованы и распространяются свободно без

каких-либо ограничений. Они применяются в системах шифрования с открытым ключом, для создания электронной подписи и т. д.

Компонент Расширенный кодировщик (TIdQuotedPrintableEncoder)

Расширяет возможности *MIME*-кодирования, предлагая дополнительную схему кодирования *Quoted-Printable.* Кодирование символов по этой схеме удобнее, чем *Base 64,* в тех случаях, когда язык сообщения похож на английский (французский, немецкий, испанский и др.). Б этом случае символы, совпадающие по начертанию с английскими, вообще не кодируются, зато национальные символы представляются не двумя кодами (как в *Base 64),* а тремя. В среднем для европейских языков эта схема более экономична.

Пример кодирования

Рассмотрим несложный пример применения компонентов на практике. Создадим новую форму. На ней разместим два многострочных текстовых поля, две кнопки, один компонент IdBase64Encoder1 для преобразования текста, введенного в компонент Memo1, в данные в формате *Base64* — эти данные отображаются в правом многострочном поле Memo2. Такое преобразование выполняется при нажатии на левую кнопку, а при нажатии на правую кнопку данные с помощью компонента IdBase64Decoder1 раскодируются и выводятся в левое многострочное поле Memo1.

Реакция на нажатие левой кнопки запишется так:

```
procedure TForm1.Button1Click(Sender: TObject);
var
   ReturnedString : String;
   ReturnedInteger : Integer;
begin
   Memo2.Clear;
   IdBase64Encoder1.CodeString(Memo1.Text);
   ReturnedString := IdBase64Encoder1.CompletedInput;
   ReturnedInteger := StrToInt(Fetch(ReturnedString, ';'));
   Memo2.Text := Fetch(ReturnedString, ';')
end;
```

Б операторе

```
IdBase64Encoder1.CodeString(Memo1.Text);
```

происходит поставка входной информации для кодировщика IdBase64Encoder1. Результирующая строка формируется в следующем операторе вызовом метода CompletedInput. Для выделения из нее той части, которая представляет собой непосредственно закодированный текст, можно воспользоваться стандартной функцией:

```
function Fetch(var AInput: String; const ADelim: String = ' ';
   const ADelete: Boolean = True): String;
```

Она входит в модуль IdGlobal и выделяет из строки, заданной первым параметром, подстроку, начинающуюся с символа, заданного вторым параметром.

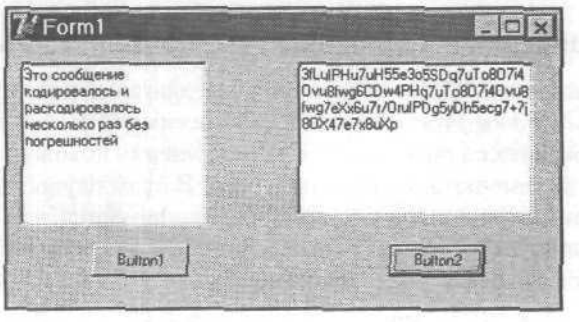


РИС. *10.8.* *MIME-перекодировщик*

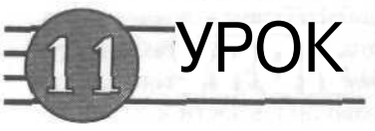
Обратное преобразование (из закодированного вида в обычный, текстовый), выполняется схожим образом:

```
procedure TForm1.Button2Click(Sender: TObject);
var
   ReturnedString : String;
   ReturnedInteger : Integer;
begin
   Memo1.Clear;
   IdBase64Decoder1.CodeString(Memo2.Text);
   ReturnedString := IdBase64Decoder1.CompletedInput;
   ReturnedInteger := StrToInt(Fetch(ReturnedString, ';'));
   Memo1.Text := Fetch(ReturnedString, ';');
end;
```

Что нового мы узнали?

В этом уроке мы научились

- быстро обеспечивать поддержку основных протоколов Интернета;
- создавать собственный *Web*-сервер;
- создавать собственный *FTP*-сервер;
- организовать работу электронной почты;
- реализовывать алгоритмы кодировки данных.



УРОК Программирование для Web-серверов

- Создание приложений Web-сервера
- Доступ к данным из приложений Web-сервера
- Быстрая разработка приложений Web-сервера с доступом к данным на основе технологии XML
- Создание приложений Web Services
- Создание Web-серверных приложений с помощью технологии WebSnap
- Принципы работы приложения WebSnap
- Быстрое создание WebSnap-приложения, работающего с базами данных
- Визуальное проектирование приложений Web-сервера с помощью технологии IntraWeb

Создание приложений Web-сервера

Web-программирование

Система *Delphi* 7 содержит множество особенностей, ориентированных на создание программ для *Web*-серверов, в том числе поддерживающих интерфейс *CGI*. *Web*-серверные приложения могут содержать любые невизуальные компоненты, что позволяет сконструировать сколь угодно сложную программную логику, реализующую функцию обращения к базам данных, преобразование документов в форматах *HTML* и *XML*. Комплексная архитектура, определяющая и поддерживающая внутренние взаимосвязи между этими компонентами, называется в *Delphi WebBroker* (*Web*-брокером). Среди важнейших особенностей этой архитектуры — возможность отладки серверных *COM*-модулей, а также важная возможность запуска таких модулей, если на компьютере не установлен стандартный *Web*-сервер. Для этого вместе с *Delphi* поставляется микро- Wei-сервер *TestSvr*. Он позволяет, в частности, отслеживать *HTTP*-запросы и возвращать информацию в формате *HTML*.

В одном Wei-приложении допускается создавать несколько Wei-модулей. Эта возможность весьма удобна, когда требуется выделить ту или иную функциональность приложения в отдельный блок.

Компонент TWebDispatcher позволяет обрабатывать ЯГГР-запросы большой сложности. На основе базового класса TWeb Form создаются конкретные компоненты, генерирующие по внешнему запросу *HTML*-формы, например компоненты TDataForm или TQueryForm, которые связываются с базами данных и представляют их содержимое в виде *HTML*-страниц. При этом формирование таких страниц может выполняться не только с помощью модуля, написанного средствами *Delphi,* но и с помощью сценарных языков наподобие *JavaScript*.

В *Delphi 7* появился специальный проектировщик *HTML*-страниц, позволяющий формировать их в визуальном редакторе и составлять свои собственные сценарии. Мастер создания Wei-приложений полностью переработан с учетом новых возможностей архитектуры *WebBroker*.

ВНИМАНИЕ Большинство проектов, созданных с помощью старых версий Delphi, в этой архитектуре смогут работать с определенными ограничениями.

Как уже говорилось, Wei-сервер обеспечивает работу *Web*-узлов Интернета (или локальной сети на базе протокола *TCP/IP),* позволяя множеству пользователей одновременно обращаться к одним и тем же страницам *HTML,* Wei-сервер рассылает копии этих страниц клиентским программам-браузерам. В то же время современные подходы к созданию приложений Интернета, ориентированных на бурно развивающую электронную коммерцию, требуют активного взаимодействия с пользователем и обеспечения тесной обратной связи с ним, по аналогии с обычными автономными офисными приложениями.

Для этого необходимо, чтобы содержание страницы *HTML,* отправляемой пользователю, не было простой копией файла на сервере, а создавалось динамически, программным путем, в зависимости от действий пользователя. Например, пользователь может заполнять формы и отправлять их серверу или работать через сервер с таблицей удаленной базы данных, в которой хранятся списки товаров. Такая схема работы напоминает клиент-серверную и распределенную архитектуры.

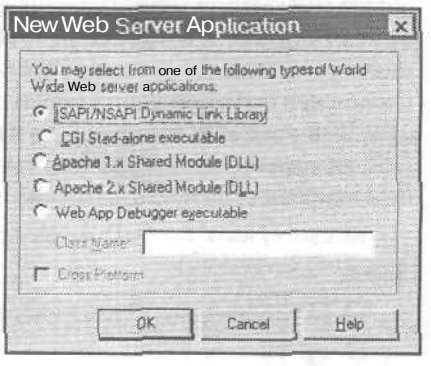
Отличие заключается в том, что, во-первых, передавать клиентской программе надо не массив значений, который она обработает сама, а уже готовую страницу *HTML.* Ее надо полностью подготовить на *Web*-сервере, вследствие чего нагрузка на сервер увеличивается. С другой стороны, нагрузка на компьютеры пользователей снижается, так как там требуется только отобразить страницу в окне браузера. Во-вторых, информация, особенно коммерческая, передаваемая но глобальной сети, должна быть хорошо защищена. Она может шифроваться промежуточными программными модулями или, по крайней мере, при ее передаче должна использоваться более надежная и безопасная версия протокола *HTTP,*

Сам *Web-сервер* (программа) не занимается генерацией страниц. Ему и так хватает работы по рассылке этих страниц тысячам посетителей *Web-узлов.* При поступлении запроса на выполнение нестандартной операции *Web-сервер* загружает в виде отдельного процесса один из специальных модулей. Этот модуль выполняет создание нужной страницы *HTML* и затем передает ее обратно *Web*-серверу, который отсылает ее соответствующему клиенту.

Такие модули пишутся на самых разных языках программирования и могут работать под управлением самых разных операционных систем. Так как мы рассматриваем среду *Delphi 7,* то и примеры будут относиться к системе *Windows.*

Создание заготовки Web-модуля

В системе *Delphi* 7есть Мастер, позволяющий путем нескольких уточняющих вопросов подготовить пустое *Web-приложение* (модуль, загружаемый *Web*-сервером в ответ на специфический запрос пользователя). Этот Мастер вызывается командой File ➤ New > Other (Файл > Создать ➤ Другое) с последующим выбором значка Web Server Application (Приложение Web-сервера). Мастер предложит выбрать один из четырех типов будущего серверного *Web*-модуля.



- ISAPI/NSAPI Dynamic Link Library. Создание библиотеки .DLL, которая автоматически загружается *Web*-сервером в случае необходимости. Данные для обработки передаются в библиотеку и возвращаются *Web*-серверу в виде специальной структуры с помощью программного интерфейса *ISAPI* фирмы *Microsoft* или *NSAPI* фирмы *Netscape.*

- CGI Stand-alone executable. Создание серверного консольного приложения. Оно по мере необходимости запускается Wei-сервером. Приложение получает параметры и передает результаты работы напрямую.
- Apache Shared Module (DLL). Создание подгружаемой библиотеки для Wei-сервера *Apache*. Он применяется, как правило, на платформе *Linux*.

ЗАМЕЧАНИЕ В прежних версиях поддерживался тип сервера Win_CGI Stand-alone executable в формате исполнимого приложения Windows. Он исключен из Delphi, начиная с седьмой версии системы.

Каждый из этих модулей запускается на сервере. При обращении к нему из браузера пользователя Wei-сервер загружает соответствующую программу, при необходимости передает ей параметры и ожидает окончания ее работы, после чего отсылает результаты обратно браузеру пользователя.

Если к *Web*-серверу обращается множество пользователей (тысячи), то для каждого из них запускается копия программного модуля, что может вызвать значительную перегрузку компьютера. В таких случаях лучше использовать модули в формате ISAPI/NSAPI, которые ускоряют запуск благодаря тому, что выполнены в формате .DLL и позволяют Wei-серверу использовать их функции во внутреннем рабочем пространстве. Форматы *CGI* требуютзапуска полноценной консольной программы или программы *Windows,* что предъявляет повышенные требования к компьютеру. Недостаток подобного подхода в том, что при каждом изменении модуля в формате .DLL надо перезапускать сам Wei-сервер, чтобы он выполнил перезагрузку этого модуля в свое адресное пространство.

Параметры и результаты

Wei-модуль получает от Wei-сервера набор параметров *(запрос),* указанных пользователем вручную или сгенерированных браузером автоматически. В зависимости от этих параметров модуль выполняет те или иные действия. Запрос передается в виде объекта, имеющего тип TWebRequest. Допустим, создан Wei-модуль MyTest.exe. Он должен быть помещен в подходящий виртуальный каталог Wei-сервера (условный Wei-узел www.my-site.ru), для которого установлены права на запуск программ *CGI* на сервере. Как правило, такой каталог называется cgi-bin. Данный модуль будет вызван, если пользователь наберет в адресной строке своего браузера такой текст:

```
http://www.my-site.ru/cgi-bin/MyTest.exe/PARAM1?f7
```

Здесь:

- http://www.my-site.ru — адрес Wei-узла;
- cgi-bin — каталог;
- MyTest.exe — запускаемый *Web*-модуль;
- /PARAM1 — название параметра;
- ? — символ, отделяющий название параметра от его значения;
- f7 — значение параметра PARAM1.

Модуль можно запускать и без параметров:

```
http://www.my-site.ru/cgi-bin/MyTest.exe
```

а также со значением, указываемым модулю без названия параметра:

```
http://www.my-site.ru/cgi-bin/MyTest.exe?c5
```

После того как на сервере выполнены нужные действия, зависящие от переданной информации, модуль возвращает серверу результат своей работы. Он записывается в объект, имеющий тип TWebResponse. Этот результат представляет собой строку, содержащую текст *HTML,* который и отображается браузером пользователя в качестве текущей страницы.

Пример создания Web-модуля

Содержание примера

При обращении к *Web*-модулю без параметров отображается пустая страница с коротким приветствием. Если указан параметр /DAY со значением пате, то сообщается название текущего дня недели; если значение равно full, отображается полная дата. Если задан параметр /TIME, то показывается текущее время.

Создание пустого Web-модуля

Web-модуль создается с помощью Мастера. Выберем консольный тип приложения — CGI Stand-alone executable.

Для добавления и редактирования обрабатываемых им действий надо вызвать редактор действий с помощью команды Action Editor из контекстного меню этого модуля. Для редактирования списка возможных действий применяется стандартный редактор коллекций, немного модифицированный для удобства визуальной настройки наиболее часто используемых свойств.

Пока что в списке действий (Actions) нет ни одного ответного действия модуля на поступление запроса от *Web*-сервера. Чтобы добавить новое действие, надо нажать на кнопку Add New редактора коллекции. В список добавляется новая строка, соответствующая новому созданному объекту класса TWebActionItem, а Инспектор объектов покажет свойства этого объекта, представленные на рис. 11.1 и в табл. 11.1.

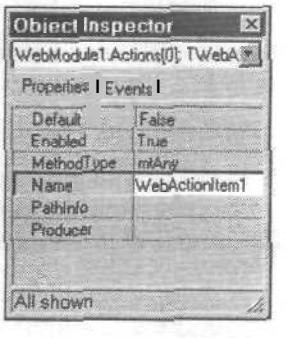


Рис. *11.1. Настройка свойств действия, выполняемого Web-модулем*

Таблица 11.1. Свойства объектов класса TWebActionItem

Свойство	Назначение
Default	Имеет значение True, если данное действие выполняется по умолчанию, когда в ответ на запрос не выполнено ни одно из других действий
Enabled	Имеет значение True, если Web-сервер может использовать данное действие
MethodType	Тип обрабатываемого запроса. По умолчанию имеет значение mtAny (все запросы)
Name	Имя объекта
PathInfo	Имя параметра, задание которого активизирует вызов дачного объекта
Producer	Имя поставщика Web, формирующего результат работы модуля

Основное свойство — это свойство PathInfo, в котором надо задать одно из обрабатываемых названий параметра (/DAY, /TIME) или не указывать ничего. Таким образом, для двух параметров и значения, принятого по умолчанию, надо подготовить три объекта в списке Actions.

У первого (пусть он называется DefaultAction) в свойстве PathInfo не указывается ничего (чтобы это действие вызывалось по умолчанию, значение его свойства Default надо установить равным True), у второго (DayAction) — задается строка /DAY, у третьего (TimeAction) — строка /TIME. Свойство Producer сохраняется пустым.

Реакция на запрос

Чтобы объект DefauLtAction мог реагировать на запрос *Web*-сервера, на вкладке Events (События) для этого объекта надо выбрать событие OnAction (оно единственное) и создать его обработчик двойным щелчком:

```
procedure TWebModule1.WebModule1DefaultActionAction(
    Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
begin

end;
```

Данный обработчик будет вызываться при обращении из браузера к *Web-модулю* без параметров, например с помощью следующей строки:

```
http://www.my-site.ru/cgi-bin/MyTest.exe
```

Параметр процедуры Request содержит всю информацию о запросе. В частности, в свойстве этого параметра Query хранится значение запроса, если оно было указано пользователем после знака ?. Например, пусть обращение выполнялось следующим образом:

```
http://www.my-site.ru/cgi-bin/MyTest.exe?проверка
```

В этом случае значение переменной Request.Query равно строке 'проверка'.

Результат работы модуля записывается в виде строки в свойство Content параметра обработчика Response. Так как при обращении к модулю без специальных параметров должна отображаться пустая страница, вся логика работы данного обработчика уместится в одном операторе присваивания:

```
Response.Content := '<body bgcolor=white><center>' +
   '<h3>привет!</h3></center></body>';
```

Обработка запросов с параметрами чуть сложнее. Чтобы сформировать название дня недели, надо узнать его номер с помощью стандартной функции DayOfWeek, которая принимает значения от 1 (воскресенье) до 7 (суббота). В качестве ее параметра надо указать структуру типа TDateTime, содержашую информацию о текущем дне. А такой день можно получить с помощью другой функции — Now.

Полное текстовое представление даты формируется обращением к стандартной функции DateToStr. Процедура, вызываемая при указании параметра /DAY (реакция на событие OnAction объекта DayAction), выглядит следующим образом:

```
procedure TWebModule1.WebModule1P1Action(Sender: TObject;
   Request: TWebRequest; Response: TWebResponse; var
   Handled: Boolean);
const DAY_NAMES: array[1..7] of string =
   ('воскресенье', 'понедельник', 'вторник',
   'среда', 'четверг', 'пятница', 'суббота');
begin
   if Request.Query = 'name' then
      Response.Content := '<center>сегодня ' +
         DAY_NAMES[DayOfWeek(Now)] + '</center>'
   else if Request.Query = 'full' then
      Response.Content := '<center>' + DateToStr(Now) +
'</center>'
end;
```

Вот пример обработчика запроса с параметром /TIME. В нем используется стандартная функция TimeToStr, преобразующая текущее время в текстовое представление.

```
procedure TWebModule1.WebModule1TimeActionAction(Sender:
  TObject;
   Request: TWebRequest; Response: TWebResponse; var
    Handled: Boolean);
begin
   Response.Content := '<center>сейчас ' + TimeToStr(Now)
    + '</center>'
end;
```

Запуск Web-модуля

Написав всего два условных оператора и четыре оператора присваивания, нам удалось создать с помощью системы *Delphi 7* законченный Wei-модуль, пригодный для размещения на Wei-сервере.

Приложение надо откомпилировать, переименовать (MyTest.exe) и разместить в каталоге Wei-сервера, соответствующем виртуальному каталогу cgi-bin с правами на запуск на сервере исполняемых *EXE*-модулей.

Если отсутствует полноценная возможность отладки Wei-приложений (нет доступа к *Web*-серверу, ограничено время в Интернете или возникло иное препятствие), то можно организовать работу Wei-сервера в локальном режиме, например установить на компьютер *Web*-сервер Microsoft Personal Web Server. Тогда можно обратиться из браузера к этому серверу (он должен быть запущен), набрав в адресной строке такой текст

```
http://127.0.0.1
```

Это *IP*-адрес локального компьютера. Соединение браузера с Web-сервером произойдет автоматически.

Есть и более привычный способ организации связи с локальным *Web*-сервером. В настройках протокола *TCP/IP* (значок Сеть на Панели управления) можно указать имя компьютера (вкладка Конфигурация DNS, поле Имя компьютера), например TestBed. Это название служит виртуальным синонимом текущего *IP*-адреса (127.0.0.1) компьютера, и теперь обращаться к запущенному *Web*-серверу можно, набрав в строке браузера следующий адрес:

```
http://TestBed
```

Для проверки работы *Web*-модуля в адресной строке браузера задаются различные параметры запуска:

```
http://TestBed/cgi-bin/MyTest.exe
http://TestBed/cgi-bin/MyTest.exe/DAY?name
http://TestBed/cgi-bin/MyTest.exe/DAY?full
http://TestBed/cgi-bin/MyTest.exe/TIME
```

Отладка без Web-сервера

В *Delphi 7* появилась новая возможность отладки Wei-приложений, не требующая наличия Wei-сервера. В поставку *Delphi* включен локальный сервер *Web App Debugger,* который имитирует работу полноценного *Web*-сервера, обрабатывая *HTTP*-запросы и передавая их приложению. Он прост в использовании, и для отладки приложений с его помощью не требуется устанавливать дополнительно никакого программного обеспечения.

Создадим новое Wei-приложение, только в качестве его типа в начальном окне мастера выберем значение Web App Debugger executable. В нижней части окна станет

доступным поле ввода, в котором надо указать произвольное название создаваемого приложения. Система *Windows* использует это название для регистрации приложения в качестве *COM*-сервера, что необходимо для корректной работы сервера *Web App Debugger*. Дальнейшее создание *Web*-модуля не отличается от принципов разработки, рассмотренных выше.

После того как *Web*-модуль подготовлен, можно приступать к его отладке. Для этого сначала надо установить точки прерывания в исходном коде, затем запустить модуль стандартной командой Run. Если теперь дать команду Tools ► Web App Debugger (Инструменты ► Web-Отладчик), на экране появится окно сервера *Web App Debugger*.

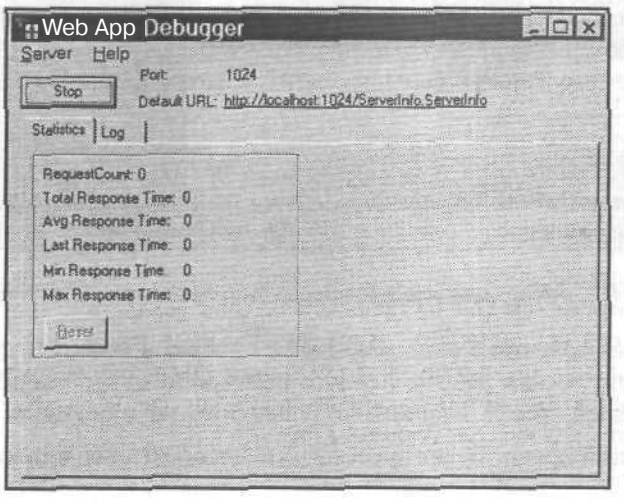


Рис. 11.2. *Сервер отладки Web-модулей*

Для его активации надо нажать на кнопку Start и затем щелкнуть на строке-ссылке, расположенной справа от нее. В результате на локальном компьютере запустится браузер, установленный по умолчанию, и в списке сформированного программой Web App Debugger окна появятся все приложения, зарегистрированные с ее помощью как *COM*-серверы.

Далее надо выбрать в этом списке текущее отлаживаемое приложение и нажать кнопку Go в окне браузера (рис. 11.3). Выбранное приложение запустится автоматически, а результат его работы будет показан в окне браузера. При этом возможна остановка работы программы, если в *Delphi* были заданы точки прерывания.

После того как приложение отлажено с помощью отладчика Web App Debugger, возникает потребность преобразования его в полноценное *Web*-приложение, способное работать с одним из стандартных серверов. Для такого преобразования надо выполнить следующие шаги.

1. К текущей группе проектов добавить новый проект.
2. В окне мастера Web Server Application выбрать соответствующий тип *Web-модуля*.

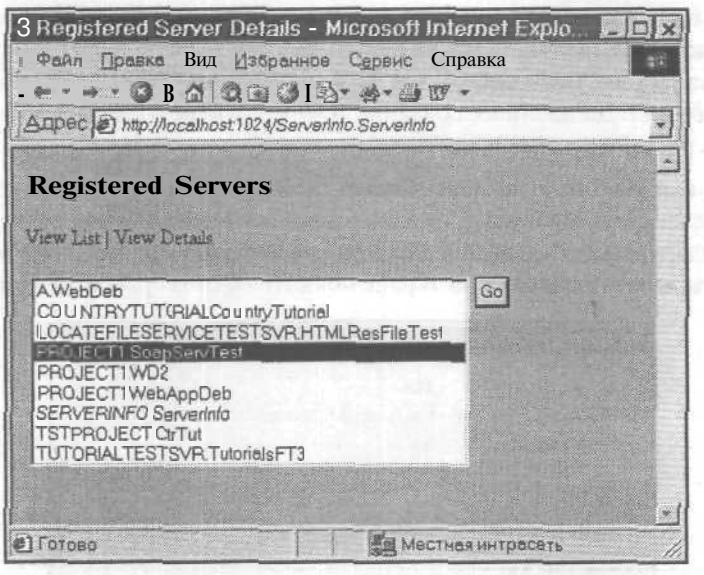


Рис. 11.3. *Выбор зарегистрированного Web-приложения для записка*

3. После переключения в окно менеджера проекта все модули, в которых была реализована логика работы первого приложения, перетащить при помощи мыши в новый проект. Главный модуль-форму перетаскивать не надо.

Результирующий проект будет представлять собой готовое приложение нужного типа.

Прием данных от Web-формы

В предыдущем примере мы разобрали, как создаются и запускаются Web-модули, имеющие различные параметры. Однако чтобы организовать более привычный пользовательский интерфейс со стандартными элементами управления, надо вызов этих модулей спрятать «внутрь» страницы *HTML.* Подготовим страницу *HTML,* на которой разместим следующий код;

```
<HTML>
<HEAD>
<TITLE>Тестовый пример</TITLE>
</HEAD>
<BODY>
<B>Введите Ваше имя и электронный адрес для подписки на
  наш список рассылки:</B>
<P><CENTER><form action='http: //TestBed/cgi-bin/
  getmail.exe' method='GET'>
```

```
<tablextrxtd>
Имя:</td><td> <input type='text' size='20' name='Name'></
  tdx/tr>
<trxtd>
Адрес почты:</td><td> <input type='text' size='20'
  name='Email'></td></tr>
<tr><td> </td><td>
<input type='submit' name='Button' value='Отправить'></
  td></tr>
</table>
</form>
</CENTER>
</BODV>
</HTML>
```

ЗАМЕЧАНИЕ Теги, ответственные за организацию пользовательского интерфейса (поля ввода, кнопки, переключатели и прочие элементы), описаны в любом справочнике по языку HTML

По щелчку на кнопке Отправить выполнится *Web*-модуль, указанный как значение атрибута ACTION в теге FORM.

```
http://TestBed/cgi-bin/getmail.exe
```

Информацию от формы он получит, как обычно в виде параметров. Если загрузить подготовленную страницу в браузер, то она будет выглядеть так, как показано на рис. 11.4.

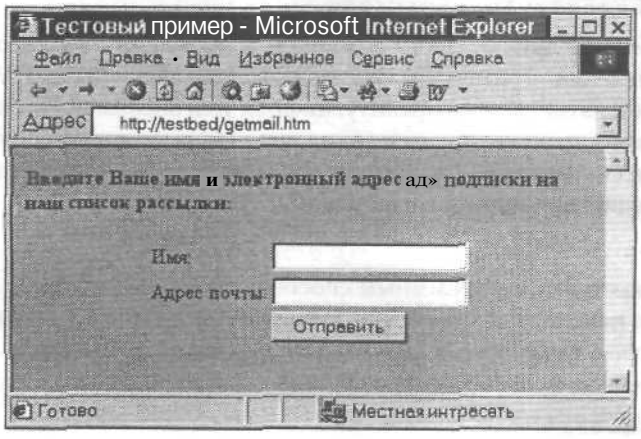


Рис. 11.4. *Страница HTML, содержащая форму, при просмотре о браузере*

С ее помощью, например, можно организовать регистрацию подписчиков на список рассылки электронных писем.

По щелчку на кнопке Отправить на сервере вызывается модуль getmail.exe, которому в качестве параметров передаются значения, введенные в два поля. Полученные сведения надо занести в базу данных по подписчикам, а на Web-сервере организовать специальную службу ведения этой базы и выполнения автоматической рассылки. Существует немало свободно распространяемых программ и бесплатных служб, позволяющих организовать такую работу без программирования. Поэтому в данном примере рассмотрим только технологию получения и обработки информации через форму, которая имеется на странице *HTML*.

Создадим Wei-модуль и сформируем в нем единственный обработчик события, у которого свойство PathInfo будет пустым. Этот обработчик запишется, например, так:

```
procedure TWebModule1.WebModule1WebActionItem1Action(
   Sender: TObject; Request: TWebRequest;
   Response: TWebResponse; var Handled: Boolean);
begin
with Request.QueryFields do
   Response.Content := '<html><body> Спасибо, ' +
      Values[Names[0]] + ', Ваш адрес ' +
      Values[Names[1]] + ' включен в список рассылки.</
        body></html>'
end;
```

Содержимое возвращаемой страницы (Response.Content) представляет собой строку, состоящую из комментария и введенных пользователем значений. Теперь необходимо выполнить разбор не значения Request.Query, а значения Request.QueryFields, которое содержит длинную строку, полученную от формы:

```
Name=%C2%E0%F1%FF&Email=vovan@vovan.ru&Button=
  %CE%F2%EF%F0%E0%E2%E8%F2%FC
```

Параметры отделены друг от друга символом &, названия элементов и соответствующие им значения — символом =. Если в качестве значения выступает символ, не относящийся к латинскому алфавиту, цифрам или знакам препинания (например, русская буква), то он кодируется двумя шестнадцатеричными цифрами, перед которыми ставится символ %. Так, в данном примере строка Вова, введенная в поле имени, представляется следующим образом:

```
Name=%C2%E0%F1%FF
```

Разбирать такую строку программным способом неудобно, поэтому вместо свойства Query лучше использовать свойство QueryFields (тип TStrings). Оно представляет собой массив строк, выделенных из исходной строки по символам &. То есть в нашем случае в этом свойстве будут три строки:

```
Name=%C2%E0%F1%FF
Email=vovan@vovan.ru
Button=%CE%F2%EF%F0%E0%E2%E8%F2%FC
```

Однако и в таком виде требуется программно определять позицию символа = и выделять остаток строки, что относится к чисто рутинной работе. Класс TStrings, рассмотренный ранее, позволяет автоматически анализировать и разбивать на пары строки, выполненные по схеме

```
название=значение
```

Свойство Values класса TStrings позволяет получить значение (заключительную часть) строки, начинающейся с подстроки, указанной в свойстве Value в качестве параметра. В приведенном выше коде выделяются значения первой и второй строк.

Если теперь поместить данный скомпилированный модуль (назвав его getmail.exe) в каталог /cgi-bin Web-сервера, то при вводе в поля значений Вова и vovan@vovan.ru и щелчке на кнопке Отправить на экране браузера появится сообщение, показанное на рис. 11.5.

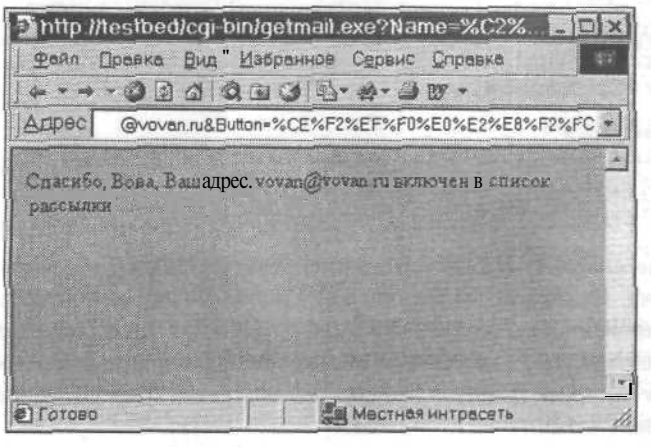


Рис. 11.5. *Результат обработки Web-модуле м данных, полученных в результате отправки формы*

В нашем примере никакой регистрации, конечно, не произошло, но, зная, как работать с базами данных, развить этот пример совсем несложно. Далее, в частности, мы рассмотрим вопросы создания *Web*-модулей, работающих с базами данных.

Поддержка шаблонов HTML

На *Web*-сервере файлы *HTML* нередко хранятся не в том виде, в каком они передаются браузеру пользователя на его компьютер. В них допускается использовать *шаблоны HTML,* с помощью которых можно осуществлять подстановку различных значений (параметров) в текст *HTML,* настраивая его на конкретного пользователя.

Шаблон *HTML* представляет собой обычный файл *HTML,* в котором помимо стандартных тегов имеются теги-шаблоны, выглядящие следующим образом:

```
<#название-шаблона параметр-1 = значение-1 параметр-2 =
  значение-2  .. >
```

Список параметров можно не указывать. Например, следующий тег-шаблон описывает текущую дату:

```
<#CurrentDate>
```

Если требуется, чтобы при посещении страницы *HTML* пользователю показывалась текущая дата, надо подготовить файл *HTML,* содержащий соответствующий шаблон (пусть он называется date.html). Его содержимое должно быть следующим:

```
<HTML>
<HEAD>
<TITLE>Приветствую на моем сайте!</TITLE>
</HEAD>
<BODY>
<h2><center>
Сегодня <#CurrentDate> !
</center></h2>
</BODY>
</HTML>
```

Преобразование шаблона *HTML в* обычный текст *HTML* осуществляется *с* помощью компонента TPageProducer (Поставщик страниц). Он размещается на панели Components (Компоненты) в окне Web Module 1. Содержимое заготовки страницы *HTML с* исходными тегами указывается в свойстве FileName, а сам файл должен размещаться в одном каталоге с запускаемым *Web*-модулем. Можно указать и полный путь поиска для этого файла.

Менять значение данного свойства можно и в процессе работы программы, например:

```
PageProducer1.FileName := 'date.html';
```

Допустим, данный объект (он получил название PageProducer1) используется обработчиком стандартного запроса без параметров (DefaultAction). В его свойстве Producer надо указать (выбрать в раскрывающемся списке) соответствующий объект PageProducerl. При этом свойство PathInfo автоматически получает значение /PageProducerl.

Теперь потребуется немного изменить генерацию ответа модуля на запрос без параметров (процедура WebModule1DefaultActionAction). Вместо того чтобы возвращать явно указанную строку в формате *HTML,* надо присвоить значение, возвращаемое поставщиком переменной Response.Content:

```
procedure TWebModule1.WebModule1DefaultActionAction(
  Sender: TObject;  Request: TWebRequest;
  Response: TWebResponse; var Handled; Boolean);
```

```
begin
   Response.Content := PageProducer1.Content
end;
```

В завершение остается только указать поставщику, как обрабатывать (чем заменять) новые нестандартные теги. Для этого надо для объекта PageProducer1 организовать обработку события OnHTMLTag:

```
procedure TWebModule1.PageProducer1HTMLTag(Sender:
TObject; Tag: TTag;
   const TagString: String; TagParams: TStrings; var
    ReplaceText: String);
begin

end;
```

Название нестандартного тега (без предваряющего символа #) передается в параметре TagString, а результирующее значение, сформированное на основе его анализа, записывается в параметр ReplaceText.

```
if TagString = 'CurrentDate'
   then ReplaceText := DateToStr(Now)
```

Теперь можно перекомпилировать *Web*-модуль, записать его в каталог /cgi-bin и запустить с параметром поставщика:

```
http://testbed/cgi-bin/project1.exe/PageProducer1
```

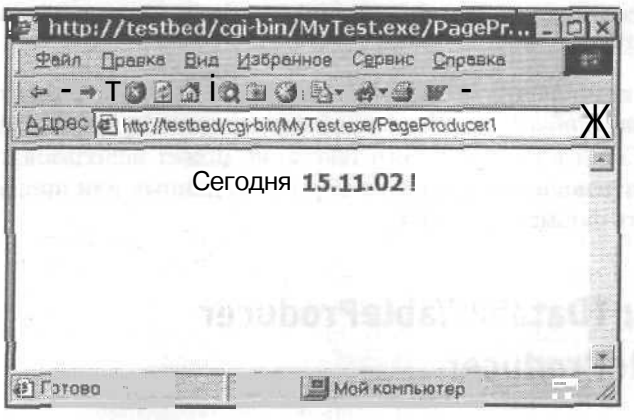


Рис. 11.6. Страница HTML, сформированная на основе шаблона

ЗАМЕЧАНИЕ Обычно подобные параметры для вызова того или иного поставщика страниц формируются в тексте HTML специальными командами, например средствами языка сценариев JavaScript.

Доступ к данным из приложений Web-сервера

Публикация данных на Web-сервере

Потребность в публикации информации из баз данных возникает на *Web-узлах* очень часто. Все сетевые магазины, системы электронной торговли, службы заказа товаров через Интернет построены по этому принципу. На компьютере, где установлен *Web*-сервер (или на другом компьютере, связанном с ним по сети), запушена СУБД. При посещении пользователем определенных страниц запускаются Web-модули, которые отображают данные из таблиц прямо в браузере (передавая страницы, созданные динамически). При этом нередко предлагается возможность задания параметров запроса к базе данных, чтобы получить от нее конкретные интересующие сведения. Для решения подобных задач система *Delphi 7* содержит набор соответствующих компонентов.

Способы публикации данных

Система *Delphi 7* предоставляет два способа публикации данных на Wei-сервере из СУБД.

1. С помощью компонентов TDataSetTableProducer и TQueryTable Producer. Этот подход отличается простотой и предназначен для автоматического отображения содержимого таблицы или результата запроса в формате *HTML*.

2. С помощью компонента TDataSetPage Producer. Этот подход характеризуется большей гибкостью. По умолчанию он позволяет представлять данные из СУБД в виде простого текста, но может использоваться и для предварительной программной обработки данных или произвольного сложного форматирования.

Компоненты TDataSetTableProducer и TQueryTableProducer

Подготовка компонентов для связи с СУБД

Поместим в окне WebModule1 компонент TQuery. С помощью построителя запросов подготовим следующий запрос:

```
SELECT Items.OrderNo, Parts.Description, MAX( Items.Qty )
  Items."MAX"
```

```
FROM "parts.db" Parts
   INNER JOIN "items.db" Items
   ON  (Items.PartNo = Parts.PartNo)
WHERE   (Items.PartNo = PArts.PartNo)
   OR  Parts.Cost BETWEEN 100 AND 150
GROUP BY Items.OrderNo, Parts.Description
HAVING  MAX(Items.Qty)  >= 5
```

Предварительно настроим объект Query1 на базу данных *DBDEMOS*. В его свойство SQL занесем текст данного запроса. Запрос Query1 надо сделать активным, установив значение свойства Active равным True.

Компонент для отображения содержимого таблицы

Теперь к модулю надо добавить компонент TDataSetTableProducer. В его свойстве DataSet указывается название запроса, содержимое которого требуется отобразить (Query1). Можно ввести и название заранее подготовленной таблицы (компонент TTable).

При двойном щелчке на этом компоненте запустится Мастер настройки внешнего вида информации из таблицы, выводимой в формате *HTML*. В нижней части Мастера отображается будущий вид таблицы в формате *HTML*, а в верхней расположены элементы управления для проведения всевозможных настроек.

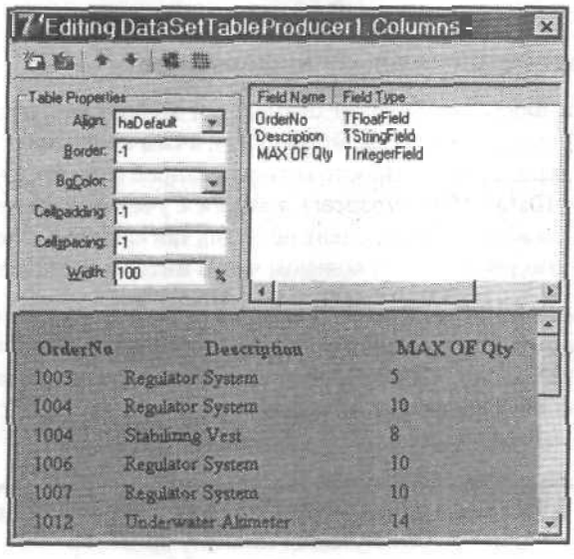


Рис. 11.7. *Настройка представления таблицы данных в формате HTML*

Таблица 11.2. Элементы управления представлением таблицы данных

Элемент управления	Параметр настройки
Align	Способ выравнивания всей таблицы в границах окна браузера
Border	Толщина каймы таблицы в пикселах
BgColor	Цвет фона таблицы
Cellpadding	Расстояние в пикселах от краев ячейки таблицы до ее содержимого
Cellspacing	Расстояние в пикселах между ячейками таблицы
Width	Ширина таблицы в процентах от общей ширины окна браузера

Список выводимых нолей приведен в правой верхней части окна Мастера. Для каждого из полей можно задать локальные настройки в Инспекторе объектов, в частности определить выравнивание (свойство Align), цвет столбца, название и цвет заголовка.

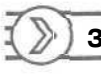
После настройки внешнего вида итоговой таблицы Мастер надо закрыть и добавить к списку обрабатываемых действий *Web*-модуля новое действие, назвав его WebDataAction. В качестве поставщика для этого действия (свойство Producer) указывается только что созданный объект DataSetTableProducerl. Для свойства PathInfo можно задать, например, значение /Set.

Компонент TDataSetTableProducer допускает еще несколько настроек внешнего вида итоговой таблицы. С помощью Инспектора объектов можно задать заголовок таблицы (свойство Caption), например Тестовый пример, а также информацию, выводимую перед таблицей (свойство Header) и после нее (свойство Footer).

Отображение параметризованной информации

Работа компонента TQueryTableProducer отличается только тем, что набор данных (название запроса) указывается в свойстве Query, а сам компонент обычно применяется, когда надо выполнить не просто готовый запрос (для этого можно использовать и компонент TDataSetTableProducer), а запрос с учетом параметров, заданных пользователем. Такая возможность очень полезна, так как позволяет организовать интерактивное взаимодействие с человеком через интерфейс браузера.

Допустим, что пользователя удаленной базы данных интересуют все компакт-диски, содержащие программы, в название которых входит определенный текст. Для этого в экранной форме (в файле *HTML)* надо создать текстовое поле и кнопку, посылающую Web-модулю информацию для запроса на поиск нужных сведений в базе данных.



ЗАМЕЧАНИЕ В Kylix вместо компонента TQueryTableProducer применяется компонент TSQLQueryTableProducer, ориентированный на работу с компонентами панели DBExpress.

Публикация данных с помощью компонента TDataSetPageProducer

Данный поставщик лишь немногим отличается от компонента TPageProducer, так как обрабатывает данные по схожему алгоритму. Для его работы надо заготовить файл *HTML* под названием, например, Dataset.html. Содержимое его может быть следующим:

```
<body>
это значение поля Name:<br>
<b><#NAME></b>
</body>
```

В данном случае обрабатываются и отображаются только значения из столбца NAME.

Также как обычный поставщик страниц, реагирующий на теги с префиксом # (теги, подлежащие замене), объект DataSetPageProducer1 при обнаружении тега, совпадающего с именем поля базы данных, заменяет его значением этого поля. При этом все остальные пользовательские теги, начинающиеся с символа #, пропускаются.

Форматирование значений для вывода на странице *HTML* осуществляется автоматически. Если требуется выполнить особенно красивое форматирование или нестандартное расположение значений из базы данных на *Web*-странице или, например, на основе значений из базы данных формировать разные страницы, то необходимо создать обработчик события OnMSG, в котором и будут выполняться нужные действия.

Такой обработчик, имитирующий работу по умолчанию (простой вывод данных в окно браузера), может выглядеть следующим образом:

```
procedure TWebModule1.DataSetPageProducer1HTMLTag(
   Sender: TObject; Tag: TTag; const TagString: String;
   TagParams: TStrings; var ReplaceText: String);

begin
   if TagString = 'NAME' then
      ReplaceText :=
DataSetPageProducer1.DataSet.FieldByName(TagString).AsString;

end;
```

Вместо тега #NAME в генерируемую строку подставляется текущее значение поля набора данных, связанного с поставщиком DataSetPageProducer1. Подставляемое поле имеет название, указанное в теге (NAME).

Перенос приложения в Web-архитектуру

В последнее время все большее число клиент-серверных и трехуровневых приложений переносится в Интернет-среду. Это связано, в частности, с возможностью

обращаться к приложениям в стандартизованном виде из так называемых тонких клиентов (браузеров), что существенно снижает требования к пользовательским компьютерам. Основная логика клиент-серверных приложений сосредоточена, как правило, в модуле данных. Но если она аккуратно разрабатывалась, если в таком модуле собраны десятки, а нередко и сотни таблиц баз данных и прочих вспомогательных объектов, связывающих эти таблицы, то вручную переносить готовые модули в заново разрабатываемые Web-серверные приложения весьма затруднительно. Для упрощения такого процесса в *Delphi 7* добавлен новый компонент Web-диспетчер (TWebDispatcher).

Допустим, имеется модуль данных, который надо перенести на Web-сервер. Для этого создадим пустой *Web*-модуль и с помощью Менеджера проекта добавим к такому только что созданному приложению готовый модуль данных из другого приложения. Это можно сделать, например, с помощью команд Копировать и Вставить в Менеджере проектов.

Дальнейшая последовательность действий очень важна. В первую очередь необходимо удалить пустой *Web*-модуль из проекта, а затем — добавить к модулю данных компонент *Web*-диспетчер. Такое добавление автоматически преобразует приложение в готовый *Web*-модуль.

Единственное достаточно важное свойство компонента TWebDispatcher, Actions, представляет собой коллекцию объектов TWebActionItem (они уже рассматривались при создании обычного *Web*-модуля). Такие объекты определяют реакции на обращение к приложению с различными параметрами из Wei-сервера. Далее надо создать обработчики событий, которые могут свободно обращаться к модулю данных, например для формирования *HTML*-страниц на основе данных из таблиц.

Таким образом мы быстро создали Web-модуль, сохранив старую, ранее сформированную логику работы и обращения к базам данных.

Быстрая разработка приложений Web-сервера с доступом к данным на основе технологии XML

Поддержка *XML* в системе *Delphi* позволяет дополнительно расширить возможности программы, работающей с базами данных, развернув ее в Интернете. На клиентских местах можно использовать браузеры, что не требует написания дополнительных клиентских программ.

На панели InternetExpress имеются два компонента (появившиеся в версии *Delphi 5), с* помощью которых можно создавать *Web*-приложения, ориентированные на обработку данных. Ранее уже рассматривался пример вывода содержимого таблицы в окно браузера. Однако создание более-менее приличного интерфейса с возможно-

стью навигации по таблице с помощью кнопок при красивом оформлении ее содержимого потребует значительных усилий и ручного кодирования.

Сейчас с помощью компонентов TXMLBroker и InetXPageProducer мы создадим *Web*-приложение, которое позволит работать с таблицей базы данных через Интернет из браузера. При этом не потребуется написать ни строчки кода — весь процесс разработки будет визуальным.

Отличие данного подхода от ранее рассмотренных состоит в том, что компонент TXMLBroker обменивается с сервером приложений пакетами данных (упакованными наборами данных). Он преобразует их в формат *XML* и передает браузеру в комбинации с тегами *HTML* к командами языка *JavaScript* для отображения пользовательского интерфейса. Этот же компонент принимает от браузера сведения о внесенных изменениях и отсылает их обратно серверу приложений. Непосредственно генерацией страниц *HTML* с нужным содержанием занимается компонент InetXPageProducer.

Подобное *Web*-приложение играет двойную роль. С одной стороны, это клиентская программа, использующая компоненты связи с сервером приложений, а с другой стороны — серверное приложение, выполняемое на *Web*-сервере и предоставляющее конечным пользователям информацию для отображения с помощью браузера. Такая длинная цепочка «сервер баз данных — сервер приложений — Wei-сервер — браузер» позволяет практически полностью снять нагрузку с компьютера пользователя (ему нужен только обычный браузер) и оптимально распределить ее между остальными уровнями.

Создание Web-приложения с доступом к базе данных

Создание заготовки Web-приложения

Дайте команду File ➤ New > Other (Файл ➤ Создать > Другое), выберите на вкладке New значок Web Server Application (Приложение Web-сервера) и щелкните на кнопке OK. На вопрос о типе *Web*-приложения укажите значение CGI Stand-alone executable (Консольная программа CGI). Более подробно о создании *Web*-приложений рассказывалось ранее. Созданный проект сохраните.

ВНИМАНИЕ Для дальнейшей работы необходимо, чтобы в системе был установлен сервер приложений, например Project1.DCOMTest, созданный в одном из предыдущих примеров.

Установка связи с сервером приложений

На панели *Web-модуля* разместим компонент TDCOMConnection с панели DataSnap. При желании можно использовать и другой способ установления связи с сервером приложений, например с помощью компонента TSocketConnection. В свойстве ServerName указывается имя сервера приложений — Project1.DCOMTest. Свойство ServerGUID при этом заполняется автоматически.

Доступ к набору данных

В предыдущих примерах на следующем шаге добавлялся компонент TClientDataSet — клиентский набор данных. Теперь его роль выполнит компонент TXMLBroker с панели InternetExpress. Он точно так же будет получать данные от поставщика (сервера приложений) и передавать измененную информацию. Однако, в отличие от компонента TClientDataSet, компонент TXMLBroker получает наборы данных не в формате OleVariant, а в формате XML, и для отображения данных взаимодействует не с элементами управления панели Data Controls (Элементы управления данными), а с браузером.

После того как компонент TXMLBroker размещен на панели *Web*-модуля, в его свойстве RemoteServer указывается название объекта, ответственного за связь с сервером приложений (DCOMConnection1), а в свойстве ProviderName (название поставщика) — DataSetProvider1. В момент щелчка на этом свойстве самостоятельно запускается соответствующий сервер приложений. Он автоматически закрывается, когда в раскрывающемся списке поставщиков выбирается его название.

Установка поставщика страниц

После того как связь с сервером приложений на основе *XML* сформирована, надо подготовить пользовательский интерфейс. Для этого добавляется компонент InetXPageProducer с панели InternetExpress палитры компонентов. Прежде всего в его свойстве IncludePathURL необходимо указать полный путь поиска для каталога, в котором хранятся библиотеки *JavaScript*, ответственные за работу интерфейса (кнопок и полей) в браузере пользователя. Исходно данные библиотеки (они имеют расширение имени .JS) расположены в подкаталоге \Source\Webmidas основного каталога системы *Delphi 7*, например C:\Delphi7\Source\Webmidas.

Таблица 11.3. Библиотеки JavaScript, определяющие интерфейс пользователя в браузере

Имя библиотеки	Назначение
xmldom.js	Разборщик XML-документов в соответствии с требованиями DOM-модели, написанный на JavaScript. Работает с браузерами Internet Explorer 5.0 и более поздними версиями
xmldb.js	Классы, управляющие пакетами данных в формате XML (при работе с базами данных)
xmldisp.js	Классы, связывающие библиотеку xmldb.js с визуальными компонентами, размещаемыми на HTML-странице
xmlerrdisp.js	Классы для обработки возникающих ошибок
xmlshow.js	Классы для отображения наборов данных в формате XML. Используются в процессе отладки

Добавление нового поставщика

В *Web*-приложение необходимо добавить как минимум одно действие (Action), в соответствии с которым будет вызываться поставщик данных. Для этого в окне *Web*-модуля щелчком правой кнопки мыши вызывается контекстное меню и в нем

выбирается пункт ActionEditor, после чего добавляется новый элемент. При этом генерируется новое действие WebActionItem1.

Для него надо указать Б Инспекторе объектов название поставщика страниц (свойство Producer) — InetXPageProducer1. В строке PathInfo (путь поиска в строке параметров вызова *Web*-модуля, в соответствии с которым выполняется тот или иной поставщик страниц) для удобства можно задать короткое значение, например \MID.

Создание пользовательского интерфейса

Теперь начинается заключительная стадия разработки *Web*-приложения — проектирование интерфейса страницы, отображаемой в браузере конечного пользователя. Выделив объект InetXPageProducerl, надо щелкнуть на кнопке запуска Мастера в строке свойства WebPageItems, определяющей составные части интерфейса. В результате запустится редактор *Web*-интерфейса будущей страницы.

В нижней части окна редактора на вкладке Browser (Браузер) отображается вид проектируемой страницы, а на вкладке *HTML* — соответствующий код *HTML*, доступный только для просмотра.

ЗАМЕЧАНИЕ Редактор Web-интерфейса работает с браузером Microsoft Internet Explorer версии 4 и выше.

После щелчка на кнопке New Item (Создать элемент) на панели редактора появится диалоговое окно, в котором предлагается выбрать один из двух типов экранной формы. Тип DataForm (Форма данных) позволяет сгенерировать страницу *HTML*, в которой отображается содержимое таблицы базы данных и появляются элементы управления для навигации по этой таблице. Тип QueryForm (Форма запроса) позволяет сгенерировать страницу *HTML*, содержащую поля и кнопки для организации пользовательских запросов к базе данных.

Отображение содержимого таблицы

После выбора значения DataForm в окне редактирования появится новая строка (новый объект) DataForm1. Она генерируется автоматически, если объект XMLBroker1, ответственный за связь с сервером приложений *DCOM*, активен.

При выборе объекта DataForm1 и последующем щелчке на кнопке New Item (Создать элемент) открывается диалоговое окно, позволяющее добавить к проектируемой форме *HTML* один из четырех типов элементов управления (рис. 11.8).

Элемент DataGrid (Таблица данных) — это основной элемент управления, отображающий содержимое таблицы (набора данных, получаемого от сервера приложений через поставщика DataSetProvider1) к окне браузера. Чтобы проектируемое окно выдало корректную информацию, надо запустить *Web*-сервер, а в свойстве XMLBroker объекта DataGrid1 указать значение XMLBrokerl. При этом на панели Browser (Браузер) появится пустой прообраз будущей таблицы.

Элемент DataNavigator (Навигатор) предназначен для навигации по таблице. Он содержит стандартный набор кнопок для перемещения к началу/концу таблицы, к

Рис. 11,8. Выбор типа элемента управления

следующей/предыдущей записи, для создания новой записи и других действий (рис. 11.9).

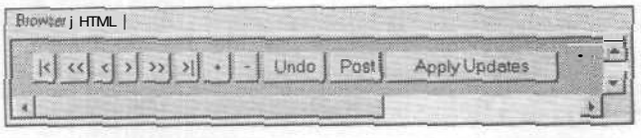


Рис. 11.9. *Стандартная панель навигатора*

Для корректной работы объекта DataNavigator1 в его свойстве XMLComponent надо указать значение DataGrid1. При этом с панели Browser (Браузер) исчезнет предупреждение о неготовности данного объекта.

ЗАМЕЧАНИЕ Данное предупреждение появляется только во время проектирования приложения s среде Delphi 7.

Элемент FieldGroup (Группа полей) предназначен для воспроизведения значений полей текущей аписи. При его добавлении и задании значения XMLBroker1 в свойстве XMLBroker в окне просмотра появляются текстовые поля, подписи к которым соответствуют названиям полей соответствующей таблицы (рис. 11.10).

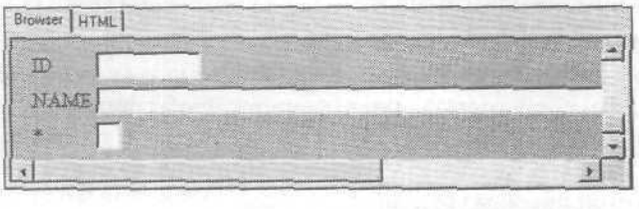


Рис. 11.10. *Элемент управления для выдачи сведений и редактирования текущей записи*

Элемент DataNavigator (Навигатор) может быть привязан и к данному полю (через свойство XMLComponent) для просмотра записей набора данных поодиночке.

Элемент LayoutGroup (Группа элементов) объединяет в одной группе несколько элементов из числа описанных выше.

Кроме того, вид каждого из четырех элементов управления поддается специфической настройке с помощью контекстного меню. Вы можете добавлять детали оформления, редактировать список отображаемых полей набора данных и т. д. При этом характеристики текущего выбранного элемента отображаются в правой верхней части окна редактора экранной формы.

Редактирование созданной страницы HTML

Чтобы выполнить пользовательскую настройку данной страницы (изменить цвета, форматы, заголовки, добавить дополнительную информацию и картинки, изменить стили и пр.), необходимо обратиться к свойству HTMLDoc объекта InetXPageProducer1. Редактор кода *HTML* вызывается щелчком на кнопке запуска Мастера. Первоначально в него загружается стандартный шаблон *HTML*. Его не требуется готовить вручную, и он автоматически заполняется всей необходимой информацией. Этот шаблон можно заменить другим, созданным самостоятельно. Полный путь поиска для нового шаблона задается в свойстве HTMLFile (рис. 11.11).

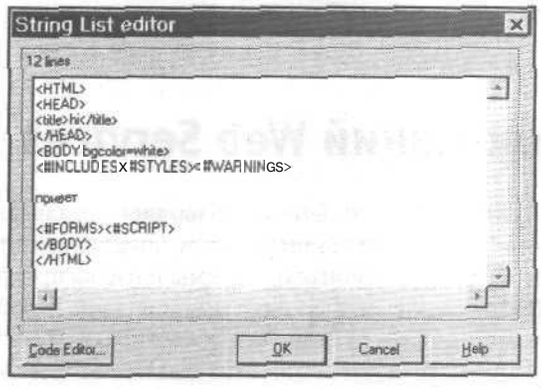


Рис. 11.11. Ручное редактирование шаблона будущей Web-страницы, содержащей данные формы

Создание и установка Web-модуля

Теперь можно выполнить компиляцию модуля. Результлруюшее приложение (файл .EXE) надо скопировать в виртуальный каталог *Web*-сервера\cgi-bin, а обращаться к нему следует при помощи следующей адресной строки:

```
http://testbed/cgi-bin/project1.exe/MID
```

Окно браузера примет вид, показанный на рис. 11.12. С помощью кнопок можно выполнять навигацию по записям удаленного набора данных, их просмотр и редактирование. При этом должен быть загружен ранее созданный сервер приложений с компонентом TDCOMConnection.

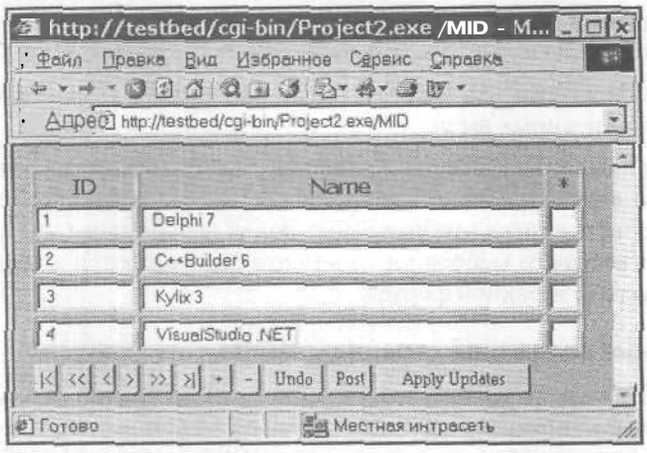


Рис. 11.12. Отображение в браузере страницы, созданной Web-модулем

Таким образом, вообще не прибегая к ручному программированию, а только используя компоненты *Delphi 7* с панели InternetExpress, можно создавать приложения, позволяющие публиковать произвольные данные в Интернете и организовывать тесное взаимодействие с пользователем.

Создание приложений Web Services

Новая программная архитектура *Web Services* позволяет создавать приложения, обменивающиеся информацией по различным сетям, локальным или глобальным. Эта технология не зависит от операционной системы и языка программирования, на котором разрабатываются соответствующие приложения. В ней используется ряд стандартных высокоуровневых протоколов, в частности *SOAP* (*Simple Object Access Protocol*). На его основе создаются приложения Web Services в рамках среды *Delphi*.

Высокоуровневый протокол *SOAP* основан на языке *XML*. С помощью *XML* описываются доступные интерфейсы удаленных объектов и способы вызова их методов, а в качестве протокола связи применяется обычно *HTTP*. Реально для такого описания применяется подмножество *XML*, специально разработанное для данных целей — *WSDL* (*Web Service Description Language*).

Возможности архитектуры *Web Services* не ограничиваются *SOAP* и *HTTP*, но компоненты, входящие в состав *Delphi* и поддерживающие такую архитектуру, построены именно на этих базовых протоколах. Кроме того, хотя технология *Web Services* не зависит от операционной системы, среда *Delphi* позволяет создавать приложения *Web Services* только для *Windows*, хотя они могут взаимодействовать с приложениями *Web Services*, выполняющимися на других платформах.

Сервер *Web Services* представляет собой программу, обрабатывающую различные запросы от клиентских приложений и возвращающую результаты работы различных процедур в соответствии со своими доступными интерфейсами. Клиентская программа (например, браузер) обращается к серверу по стандартному протоколу, например *SOAP,* получает результаты работы и отображает их на экране пользователя.

Сервер Web Services

Серверы *Web Services* создаются па основе так называемых *invokable* (вызываемых, призываемых по мере необходимости) интерфейсов. Для реализации вызываемого интерфейса требуется класс, который будет хранить его функциональность.

Эти интерфейсы сформированы особым способом. Когда во время работы программы поступает запрос от клиента па вызов некоторого метода, этот запрос динамически расшифровывается, а метод корректно обнаруживается и вызывается. Интерпретацией подобных запросов (они поступают в виде *SOAP*-сообщений) занимается специальный компонент, которьш определяет соответствующий вызываемый интерфейс, обращается к соответствующему методу и формирует ответное сообщение. К такому компоненту в свою очередь обращается компонент-диспетчер, который получает *SOAP*-сообщение, поступающее по протоколу *HTTP,* и передает его компоненту-обработчику. Компонент-диспетчер требуется для того, чтобы сервер мог одновременно обрабатывать большое число запросов от клиентских программ.

Вызываемые интерфейсы регистрируются в операционной системе автоматически при запуске сервера. Клиенту достаточно указать конкретный адрес этого сервера *Web Services,* например его *IP*-адрес. Определение вызываемого интерфейса и его реализацию (в виде класса) рекомендуется размещать в отдельных модулях *Delphi,* чтобы в дальнейшем на их основе создавать приложения *Web Services,* не привязанные к конкретной реализации.

Создание простейшего сервера Web Services

Процесс создания интерфейса *Web Services* и его реализация в принципе ничем не отличается от рассмотренных ранее принципов создания интерфейсов для других распределенных технологий, например *CORBA.*

Создадим новый модуль *Delphi* командой File ➤ New > Unit. В модуле вручную определим простейший интерфейс IMulValues, который должен быть наследником базового вызываемого интерфейса IInvokable (на основе этого интерфейса в *Delphi* создаются все интерфейсы *Web Services*):

```
unit Unit1;
interface
type
```

продолжение ⇨

```
    IMulValues = interface(IInvokable)
    end;
implementation
end.
```

Каждый интерфейс характеризуется уникальным идентификатором. *Delphi* позволяет сгенерировать такой идентификатор для нового интерфейса автоматически. Для этого надо разместить курсор на пустой строке, следующей за описанием интерфейса, и нажать **комбинацию** клавиш Ctrl + Shift + G:

```
type
    IMulValues = interface(IInvokable)
    E'{02F40B82-9971-11D5-A3F4-E3CF74A75C1B}']
    end;
```

Данный интерфейс будет предназначаться для вызова всего одной функции MulValues, выполняющей умножение двух чисел, заданных в качестве параметров, и возвращающей результат операции:

```
type
    IMulValues = interface(IInvokable)
    ['{02F40B82-9971-11D5-A3F4-E3CF74A75C1B}']
    function MulValues(V1, V2: Integer): Integer; stdcall;
    end;
```

Сохраним данный модуль в отдельном каталоге. В начале этого модуля надо указать директиву компилятора {$M+}:

```
{$M+}
unit Unit1;
interface
...
```

Это необходимо для обязательного резервирования памяти для стека, требуемого при обращении к удаленным процедурам.

Чтобы приложения *Web Services* могли обращаться к интерфейсу IMulValues, он должен быть зарегистрирован в системе в специальном реестре вызываемых интерфейсов. Такая регистрация позволяет однозначно идентифицировать класс, реализующий данный интерфейс. Подобная регистрация нужна и программе-клиенту - для корректного обращения к методам этого интерфейса.

С помощью *Delphi* подобная регистрация выполняется совсем просто. В интерфейсном разделе текущего модуля, в списке используемых стандартных модулей, надо указать модуль InvokeRegistry. Б нем хранится глобальная переменная InvRegistry (объект, определяющий список всех зарегистрированных вызываемых интерфей-

сов, соответствующих им классов реализации и процедур-фабрик, которые создают реальные объекты).

Для регистрации созданного интерфейса необходимо в разделе инициализации текущего модуля указать следующую команду:

```
initialization
InvRegistry.RegisterInterface(TypeInfo(IMulValues));
end.
```

Когда этот модуль будет включен в состав проекта, реализующего сервер *Web Services,* регистрация созданного интерфейса IMulValues выполнится автоматически.

После того как подготовлен вызываемый интерфейс, требуется создать класс, реализующий функциональность, задаваемую этим интерфейсом. Для этого в текущем модуле после описания интерфейса добавим описание класса, соответствующего этому интерфейсу. Он является наследником стандартного класса TInvokableClass, предназначенного для реализации функциональности базовых вызываемых интерфейсов, и соответствующего интерфейса IMulValues. В разделе реализации текущего модуля надо описать функциональность метода MulValues:

```
implementation
function TMulValues.MulValues(V1, V2: Integer): Integer;
begin
   Result := V1 * V2
end;
```

Последний шаг при подготовке описания и реализации вызываемого интерфейса — регистрация класса реализации. Для этого надо обратиться к уже упомянутой выше переменной InvRegistry и вызвать метод RegisterInvokab[eClass. который зарегистрирует соответствующий класс:

```
initialization
   InvRegistry.RegisterInterface(TypeInfo(IMulValues));
   InvRegistry.RegisterInvokableClass(TMulValues),-
end.
```

На этом процесс создания функциональности сервера *Web Services* практически закончен. Достаточно просто добавить созданный нами модуль к серверному приложению, например *Web*-модулю и дополнить его средствами поддержки *SOAP.*

Создание клиента Web Services

Все, что требуется для создания клиента *Web Services,* — это включение описания соответствующего интерфейса *Web Services и* создание объекта, способного по определенному транспортному протоколу посылать *SOAP*-сообщения *в* формате *XML* для вызова удаленных методов и принимать результаты их работы от сервера.

В *Delphi для* этих целей создан класс TRIO *(RIO* — аббревиатура от *Remote Invokable Object*, удаленный вызываемый объект). Данный класс является базовым для всех компонентов, которые обращаются к удаленным объектам, публикующим свой интерфейс. В прикладных приложениях класс TRIO напрямую не применяется, а используются объекты, основанные на его классах-наследниках. Для доступа к объектам *Web Services* в команде uses надо указать модули SOAPLinked и SoapHTTPClient,

Создадим новое приложение и поместим на форме одну кнопку. Перейдем к менеджеру проектов и добавим к текущему приложению модуль, в котором были описаны вызываемый интерфейс и класс реализации. Допустим, при нажатии на кнопку требуется обратиться к методу MulValues созданного нами интерфейса и получить результат его работы. Для этого в обработчике события нажатия на кнопку определим следующие переменные:

```
var InterfaceVariable: IMulValues;
    Code: Double;
    RIO : TLinkedRIO;
```

Переменная InterfaceVariable представляет собой созданный нами интерфейс. Чтобы он был доступен в программе, в разделе реализации текущего модуля надо указать ссылку на ранее подготовленный модуль, в котором реализована функциональность этого интерфейса. Переменная RIO предназначена для создания клиентского объекта, который будет обращаться к серверу *Web Services:*

```
RIO := TLinkedRIO.Create(nil);
```

После того как такой объект создан, он преобразуется к нужному нам интерфейсу с помощью оператора as:

```
InterfaceVariable := RIO as IMulValues;
```

Теперь к переменной можно обращаться напрямую, как к нужному нам интерфейсу:

```
Code := InterfaceVariable.MulValues(5,4);
```

Полученный результат будет показан в стандартном диалоговом окне:

```
ShowMessage(IntToStr(Round(Code)));
```

Надо отметить, что данный пример сочетает в себе функциональность как сервера *Web Services,* так и клиента *Web Services* — они реализованы в одном приложении. В момент запуска программы произойдет регистрация соответствующего интерфейса и класса, а при нажатии на кнопку сформируется клиентский объект Web Services, который вызовет удаленный метод MulValues.

ПОДСКАЗКА Класс TLinkedRIO *используется* обычно в отладочных целях. Он обращается к методам Web Services не по HTTP-протоколу, а просто вызывает их напрямую (в рамках одного приложения). Для создания реальных серверных систем надо применять компонент THTTPRIO — о нем рассказывается далее.

Пример создания работающей клиентской системы Web Services

Рассмотренный выше пример полезен в познавательном плане — он описывает все основные шаги, необходимые для создания системы *Web Services*. На практике услуги *Web Services* уже предоставляются довольно большим числом компаний, и следующий пример покажет, как обращаться к реальным службам *Web Services* через Интернет. Его автор — Боб Сварт, сотрудник консалтинговой компании *Gartner*.

Пример позволяет, обратившись к серверу *Web Services,* перевести числовое значение в словесное описание соответствующей денежной суммы (па английском языке). Описание данной услуги *Web Services* доступно по адресу:

```
http://powerofzen.com/cgi-bin/wordsforchecks.exe/wsdl/
  IWordsForCheck
```

Создадим новый проект и сохраним его под названием Clients.dpr в отдельном каталоге. Выполним команду File ➤ New ➤ Other (Файл ➤ Новый > Другое) и на вкладке Web Services выберем значок WSDL Importer (импорт документа WSDL).

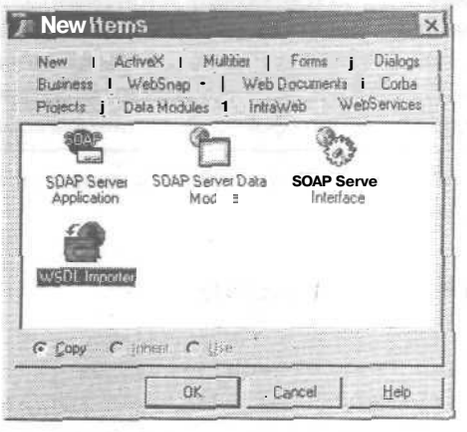


Рис. 11.13. Выбормастера WebServices

В появившемся диалоговом окне в поле Location of WSDL File or URL (местонахождение документа WSDL — имя файла или Интернет-адрес) надо ввести указанный выше адрес *URL* и нажать на кнопку Next. Мастер генерации *WSDL*-документа представит полученный интерфейс в окне просмотра.

Для завершения процесса создания программного интерфейса надо нажать кнопку Finish.

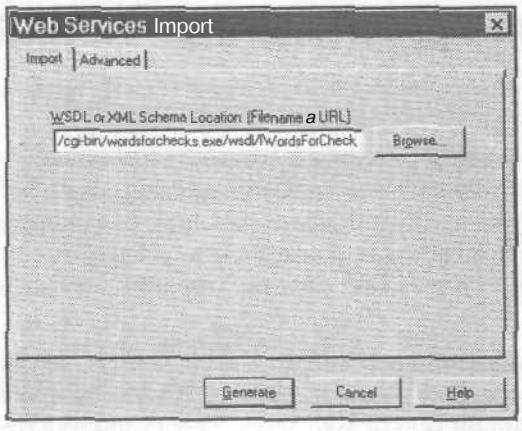


РИС. *11.14. Настройка адреса WSDL-документа*

ВНИМАНИЕ

К этому моменту, конечно, необходимо, чтобы компьютер, на котором выполняется данная работа, был подключен к Интернету — Delphi обратится за WSDL-документом к Web-серверу powerofzen.com. Кроме того, процесс получения документа может закончиться неудачей, если на компьютере установлен брандмауэр. В последнем случае надо поступить по-другому — указать адрес URL в браузере, сохранить полученный документ (XML-страницу) в файле, а в поле окна Web Services Importer указать не Интернет-адрес, а местонахождение нужного файла на локальном компьютере.

После того как генерация завершится, в проекте появится новый модуль, текст которого будет примерно таким:

```
Unit Unit2;
interface uses Types, XSBuiltIns;
type
   IWordsForCheck = interface(IInvokable)
      ['{E80150AA-9A27-11D5-A3F4-B6D59E41731B}']
      function GetWordsForCheck(const Value: Double):
        WideString; stdcall;
   end;

implementation uses InvokeRegistry;

initialization
   InvRegistry.RegisterInterface(TypeInfo(IWordsForCheck),
      'urn:UIWordsForChecks-IWordsForCheck', '');
end.
```

Различия возможны только в уникальном идентификаторе вызываемого интерфейса IWordsForCheck (обратите внимание, что он, как и положено, наследуется от интерфейса IInvokable). В описание IWordsForCheck включен единственный доступный метод GetWordsForCheck (он будет описан позже).

Теперь интерфейс IWordsForCheck надо сделать доступным в главном модуле Unit1. Переключимся на него и дадим команду File > Use Unit ➤ Unit2 (Файл > Использовать модуль > Unit2).

На главной форме разместим два поля ввода и кнопку, а также компонент THTTPRIO, реализующий функциональность *RIO* по протоколу *HTTP*. Первое поле Edit1 послужит для задания исходного числа, второе — для вывода текста — строки, содержащей словесное описание этого числа.

Полученный ранее *WSDL*-документ содержал описание интерфейса, но теперь нам надо определить координаты сервера, реализующего функциональность IWordsForCheck (указать адрес *URL*, по которому находится служба *Web Services)*. Уже упоминавшуюся строку

```
http://powerofzen.com/cgi-bin/wordsforchecks.exe/wsdl/
  IWordsForCheck
```

надо указать в свойстве WSDLLocation компонента THTTPRIO. Теперь требуется обратиться к свойству Service (доступные службы сервера) и выбрать значение IWordsForCheckservice (оно определено в *WSDL*-документе и в раскрывающемся списке будет единственным). В заключение необходимо обратиться к свойству Port (порт для работы по протоколу *SOAP)* — просто раскрыть список и выбрать в нем значение IWordsForCheckPort.

ВНИМАНИЕ Настройка компонента THTTPRIO, обращение к свойствам Services и Port И получение списка их значений закончится удачно, только если компьютер подключен к Интернету.

Теперь компонент THTTPRIO готов для работы. Обращаться к предоставляемому с его помощью интерфейсу IWordsForCheck будем в обработчике нажатия на кнопку:

```
procedure TForm1.Button1Click(Sender: TObject);
var WordsForCheck: IWordsForCheck;
    Number: Integer;
begin
    WordsForCheck := HTTPRIO1 as IWordsForCheck;
    Number := StrToInt(Edit1.Text);
    Edit2.Text := WordsForCheck.GetWordsForCheck(Number);
end;
```

Здесь, как и в предыдущем примере, объект HTTPRIO1 преобразовывается к нужному типу IWordsForCheck **(создавать** этот объект явно не надо, так как он уже существует в приложении в виде компонента), а затем вызывается его единственный удаленный метод GetWordsForCheck, который получает R качестве параметра число, а возвращает строку с его описанием. Откомпилируем и запустим приложение, введем в поле Edit1, например, число 256 и нажмем кнопку. В поле Edit2 появится результат:

```
TWO HUNDRED FIFTY SIX DOLLARS AND 00 CENTS
```

Расширим данный пример. Допустим, полученный текст надо перевести г английского языка на немецкий. Для этот можно воспользоваться соответствующей службой *Web Services*, которая называется BabelFish и используется в известной поисковой системе *AltaVista*.

Рис. 11.15. *Результат обращения к сервису IWordsForCheck*

Снова выполним импорт интерфейса *Web Services, ч* поле Location ofWSDL File or URL укажем координаты нужного *WSDL*-документа:

```
http://www.xmethods.net/sd/BabelFishService.wsdl
```

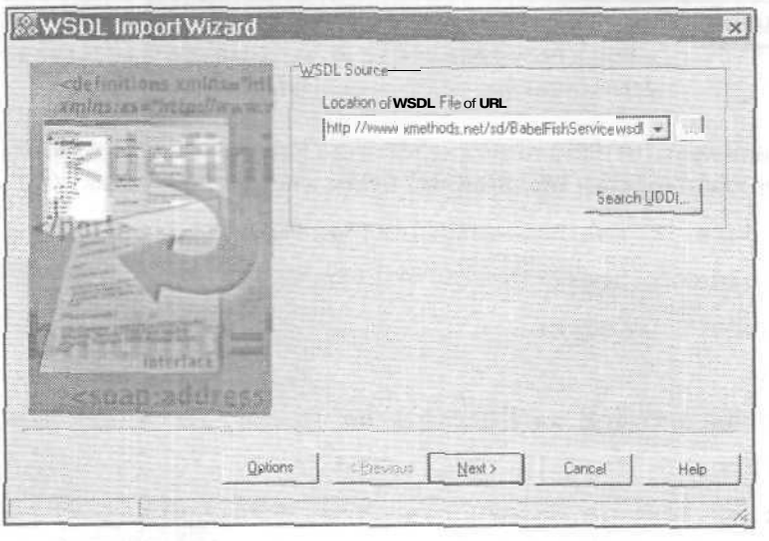


Рис. 11.16. *Мастер импорта интерфейса WebServices*

Будет сгенерирован новый модуль Unit3, описывающий интерфейс BabelFishPortType (этот модуль надо сделать доступным в главном модуле Unit1):

```
Unit Unit3;
Interface uses Types, XSBuiltIns;
type
   BabelFishPortType = interface(IInvokable)
     ['{E80150B7-9A27-11D5-A3F4-B6D59E41731B}']
      function BabelFish(const translationmode:
WideString;
         const sourcedata: WideString): WideString;
stdcall;
   end;

implementation uses InvokeRegistry;

initialization
  InvRegistry.RegisterInterface(TypeInfo(BabelFishPortType),
    '', '');
end.
```

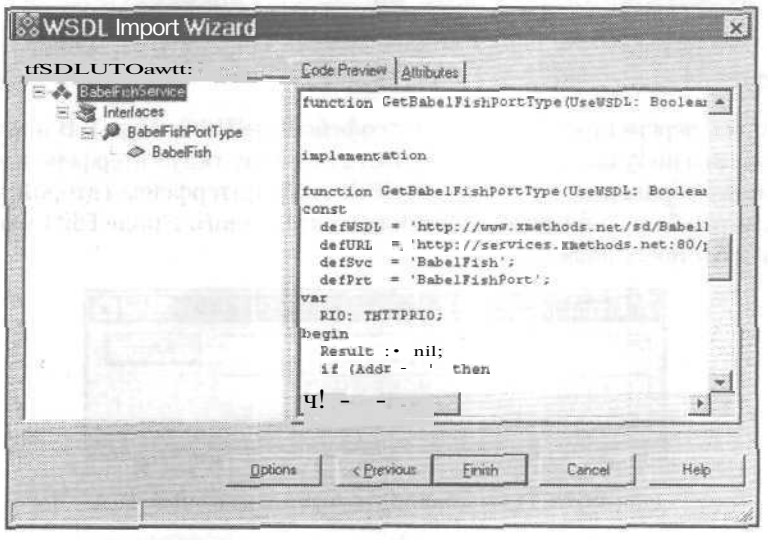


РИС. *11.17.* *Импорт интерфейса BabelFish*

Метод BabelFish выполняет перевод текста, заданного параметром sourcedata, и возвращает значение — переведенную строку в качестве результата. Этот метод может

выполнять перевод между разными языками и в разных направлениях. Для конкретизации языка и направления служит первый параметр translationmode. Первые два символа в нем определяют исходный язык (если это английский, то символы будут en), затем следует подчеркивание, и в заключение — целевой язык, на который происходит перевод. Немецкий язык обозначается так: de. Таким образом, первый параметр надо будет задавать следующим образом: 'en_de'.

Расположим на форме второй компонент, THTTPRIO, и настроим его свойство WSDLLocation, затем Service (доступным будет значение BabeLFish) и Port (значение BabelFishPort). Добавим третье поле, Edit3, для вывода результата перевода, а обработчик нажатия на клавишу перепишем так:

```
procedure TForm1.Button1Click(Sender: TObject);
var
   WordsForCheck: IWordsForCheck;
   Number: Integer;
   BabelFish: BabelFishPortType;
begin
   WordsForCheck := HTTPRI01 as IWordsForCheck;
   Number := StrToInt(Edit1.Text);
   Edit2.Text := WordsForCheck.GetWordsForCheck(Number);
   BabelFish := HTTPRIO2 as BabelFishPortType;
   Edit3.Text := BabelFish.BabelFish('en_de',Edit2.Text)
end;
```

Добавилась переменная BabelFish (интерфейс BabelFishPortType), В предпоследнем операторе она получает преобразованный к нужному типу интерфейс, а в последнем происходит обращение к методу BabelFish этого интерфейса (второй параметр - строка, автоматически сформированная из введенного в поле Edit1 числа с помощью службы WordsForCheck).

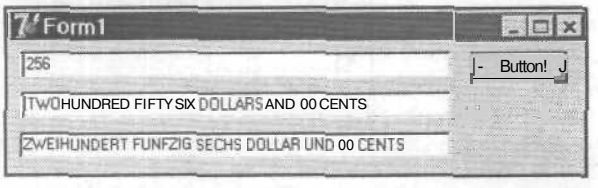


Рис. 11.18. *Результат обращения к сервису BabelFish*

Откомпилируем и запустим приложение, введем значение 256, и после нажатия на кнопку в поле Edit3 появится перевод:

```
ZWEIHUNDERT FUNFZIG SECHS DOLLAR UND 00 CENTS
```

Создание полноценного сервера Web Services

Вы можете создавать свои собственные службы *Web Services.* Их несомненное преимущество — простота создания (во многом благодаря *Delphi 7:* например, служба *WordsForChecнаписана* на *Delphi 7) и* перспективность. Предполагается, что в будущем все Интернет-приложения станут общаться друг с другом и предоставлять пользователям различные услуги именно через *Web Services* по протоколу *SOAP.*

В предыдущем разделе были рассмотрены базовые принципы создания серверов и клиентов *Web Services.* После того как определены вызываемый интерфейс и класс реализации этого интерфейса, не составляет труда на их основе создавать полноценные законченные серверные приложения *Web Services.*

Создадим сервер *Web Services,* предоставляющий интерфейс IMulValues клиентским программам, которые посылают к нему *SOAP*-запросы по *HTTP*-протоколу. Фактически это будет Wei-сервер, предназначенный для обработки *SOAP*-запросов.

Выполним команду File > New > Other (Файл ➤ Новый > Другое) и на закладке Web Services выберем значок Soap Server Application (*SOAP*-сервер), в окне Мастера укажем тип *Web*-серверного приложения (CGI Stand-alone executable) и нажмем кнопку OK. В окне *Web*-модуля автоматически размещаются три компонента: *SOAP*-диспетчер THTTPSoapDispatcher (он принимает *SOAP*-сообщения и передает их обработчику), компонент THTTPSoapPascalInvoker, интерпретирующий запрос и передающий вызовы затребованных в нем функций интерфейсов подходящим зарегистрированным в системе приложениям, а также компонент TWSDLHTMLPublish, который автоматически формирует описание *WSDL*-документа характеризующего конкретные услуги *Web Services,* предоставляемые данным сервером.

В принципе, никаких дополнительных настроек автоматически сгенерированный сервер не требует. Главное — добавить к проекту файл, в котором хранится описание интерфейса IMulValues, и скомпилировать полученное приложение. Далее это приложение (например, Project1.exe) копируется в подходящий каталог *Web-сервера,* допускающий запуск программ.

В свойстве PathInfo компонента TWSDLHTMLPublish указано значение wsdl*. Это значение определяет шаблон строки, задаваемой в адресе обращения к создаваемой службе *Web Services.* Если такая строка присутствует, то запрос будет автоматически обрабатываться данным компонентом, а не передаваться в THTTPSoapPascalInvoker. Как правило, подобные запросы со строкой wsdl (звездочка означает произвольное продолжение за wsdl) предназначены для получения информации об услугах, доступных на сервере.

Воспользуемся данной возможностью для подготовки *WSDL*-документа, описывающего интерфейс IMulValues. Обратимся из браузера к приложению, размещенному на *Web*-сервере, указав за названием программного модуля строку wdsl:

```
http://127.0.0.1/cgi-bin/Project1.exe/wsdl
```

В окне браузера появится картинка, представленная на рис. 11.19.

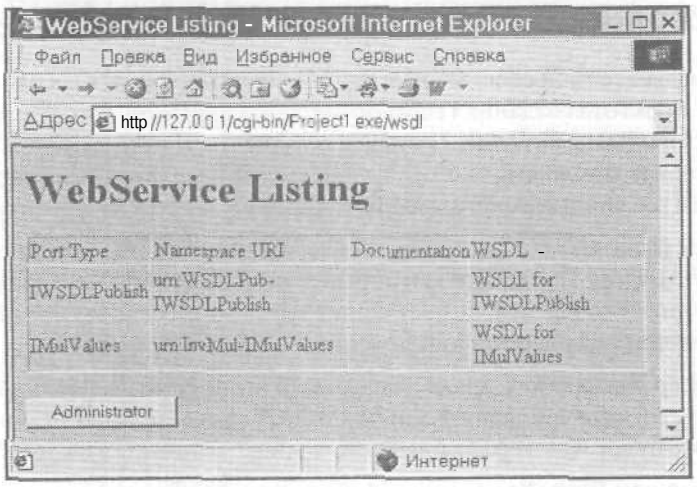


Рис. 11.19. *Доступ к WSDL-документу службы Web Services*

ПОДСКАЗКА Кнопка Administrator доступна, если в свойстве AdminEnabled компонента TWSDLHTMLPublish указано значение True. С помощью встроенного в Web-модуль администратора можно задавать несколько физических серверов (адрес и порт), поддерживающих соответствующую услугу, — это полезно в ситуациях, когда предполагается большое число пользователей услуги и желательно распределить обработку запросов между несколькими компьютерами.

После щелчка на ссылке WSDL for IMulValues на экране возникнет исходный текст нужного документа. Сохраним этот текст в файле IMulValues.xml — в каталоге wsdl *Web*-сервера. Теперь можно предоставлять описанный в нем интерфейс IMulValues всем внешним пользователям — достаточно сообщить им адрес этой услуги, например так:

```
http://MyWebServices.ru/cgi-bin/Project1.exe/wsdl/
  IMulValues
```

(сравните этот адрес с адресом услуги WordsForCheck).

Файлу Project1.exe можно дать более подходящее название, например MulValues.exe.

Клиентская программа создается описанным выше способом. Сформируем новое приложение, разместим на нем кнопку, два поля ввода и компонент THTTPRIO. В его свойстве WSDLLocation укажем данную строку:

```
http://MyWebServices.ru/cgi-bin/Project1.exe/wsdl
  /IMulValues
```

Для отладки на локальном компьютере можно ввести следующий адрес (рис. 11.20).

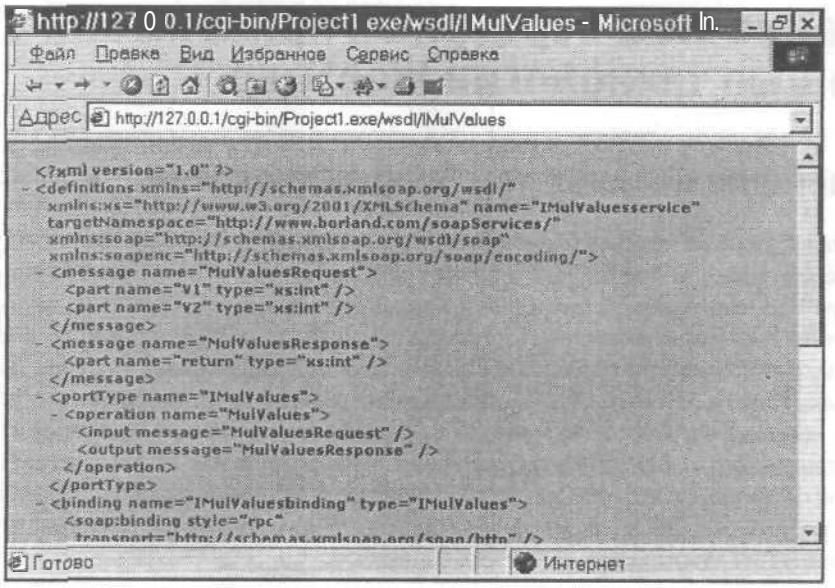


Рис. *11.20.* *Исходный текст WSDL-документав окне браузера*

```
http://127.0.0.1/cgi-bin/Project1.exe/wsdl/IMulValues
```

В раскрывающемся списке Service выберем единственное значение IMulValuesService, в свойстве Port выберем пункт IMulValuesPort. Обработчик нажатия на кнопку можно записать так:

```
procedure TForm1.Button1Click(Sender: TObject);
var
   MulValues: IMulValues;
begin
   MulValues := HTTPRIO1 as IMulValues;
   ShowMessage( IntToStr(
      MulValues.MulValues(StrToInt(Edit1.Text),
StrToInt(Edit2.Text)) ) );
end;
```

К этому модулю надо добавить ссылку на модуль InvMul с описанием интерфейса IMulValues. При нажатии на кнопку произойдет обращение к нашему *SOAP*-серверу и в окне-сообщении выведется результат умножения двух чисел.

Создание Web-серверных приложений с помощью технологии WebSnap

Важнейшие отличия WebSnap-технологии

В *Delphi 7* включен большой набор новых компонентов *WebSnap,* который значительно упрощает создание сложных сетевых приложений для Интернета. Архитектура *WebSnap* лучше всего подходит для быстрого и визуального создания динамических *Web*-приложений, работающих с базами данных. В отличие от подхода с использованием компонентов панели InternetExpress, рассмотренного ранее, *WebSnap* позволяет автоматически обрабатывать самые разные типы запросов, дает возможность использовать произвольные поставщики данных, содержит эффективные средства поддержки сессий и авторизации пользователей, а также обладает такой важной особенностью, как выполнение сценария на стороне сервера. Это позволяет очень гибко отделять логику приложения от логики, связанной с генерацией клиентских *HTML*-данных и клиентских сценариев.

В табл. 11.4 рассмотрены характерные особенности создания таких программ с помощью технологии InternetExpress и WebSnap.

Таблица 11.4. *Особенности технологий InternetExpress и WebSnap*

InternetExpress	WebSnap
Обеспечивает совместимость с исходным кодом программ, подготовленных в предыдущей версии Delphi	Позволяет использовать компоненты из прежних версий, но предусматривает работу в первую очередь с новыми компонентами
Позволяет создавать кросс-платформные приложения	Предназначена только для платформы Windows
Вся логика работы сосредоточена только в одном модуле	Допускается разбивать приложения на несколько Web-модулей, что дает возможность организовывать работу группы программистов над проектом
Разрешено применять только один Web-диспетчер	Для обработки различных типов запросов можно использовать различные специализированные диспетчеры
Для формирования содержимого Web-страницы требуется использовать специализированные компоненты — такие как поставщики страниц и компоненты с панели InternetExpress	Поддерживает все виды поставщиков информации, которые совместимы с Интернет-брокерами
Нет поддержки исполнения сценариев	Позволяет применять сценарии JScript или VBscript на стороне сервера, что очень полезно для разделения логики приложения
Нет встроенной поддержки именованных страниц	Именованные страницы могут формироваться серверными сценариями

Таблица 11.4. Особенности технологий InternetExpress и WebSnap (продолжение)

InternetExpress	WebSnap
Нет поддержки сессий	Реализована работа с сессиями, хранящими информацию о конечном клиенте, что полезно, например, при организации систем доступа и идентификации пользователя по его имени и паролю
Каждый запрос должен быть явно обработан в коде — например, с помощью объекта-действ ил	Компоненты обеспечивают автоматическую диспетчеризацию запросов в широком диапазоне
Лишь некоторые компоненты позволяют вести создание Web-модуля в визуальном режиме. Основная часть работы связана с ручным кодированием	Проектировщики WebSnap позволяют полностью визуализировать весь процесс создания Web-модуля и просматривать результаты его работы на этапе проектирования. Режимы предварительного просмотра доступны практически для всех компонентов, входящих в данный набор

Принципы работы приложения WebSnap

Адаптеры и Поставщики страниц

Для создания промежуточного независимого интерфейса между программным кодом и данными применяются компоненты-адаптеры. Они способны работать с разными типами данных, не обязательно хранящимися в базах данных. Это может быть любая информация, принимаемая от поставщиков данных.

Поставщики страниц, рассмотренные в предыдущих главах, расширены дополнительными возможностями. Они позволяют выполнять сценарные программы в зависимости от формы И типа запроса. Важно отметить, что такие сценарии выполняются на стороне сервера. Их работа обеспечивается технологией *Microsoft Active Scripting,*

Принципы функционирования WebSnap-приложения

Обработка данных (запросов) от *HTML-форм* контролируется компонентами-диспетчерами. Когда к *WebSnap*-модулю поступает запрос, автоматически создается объект TWebRequest, хранящий этот запрос, а также объект TWebResponse, который формирует итоговый ответ. В них в дальнейшем будет находиться вся информация, нужная для обработки конкретного запроса. На протяжении такой обработки эти объекты хранятся в глобальном системном объекте WebContext.

Объект TWebRequest разбивает *HTTP*-запрос на множество отдельных полей. Они требуются для корректной работы соответствующего действия-обработчика.

В зависимости от содержимого этого объекта диспетчер адаптеров TAdapterDispatcher передает запрос подходящему адаптеру для выполнения заданных действий по его обработке (в частности, для выполнения серверных сценариев).

Диспетчер страниц TPageDispatcher анализирует строку запроса — свойство PathInfo объекта TWebRequest, на ее основе выбирает подходящий зарегистрированный *Web-модуль* и передает ему управление. Диспетчер страниц работает совместно с обязательным для *Web-приложения Web-* диспетчер ом TWebDispatcher. Если диспетчер страниц не указан, то информация передается *Web*-модулю напрямую (PathInfo не разбирается). На этом этапе готовятся *HTML-формы,* содержащие необходимые клиентские сценарии.

Объект TWebResponse возвращает *Web*-клиенту через свойство Content образ *HTML*-данных, полученных от адаптера. Результирующий ответ генерируется заполнением соответствующих свойств или прямым вызовом подходящих методов TWebResponse. Все эти действия архитектура *WebSnap* выполняет автоматически, разработчику в большинстве случаев остается только настроить отдельные свойства ключевых компонентов *WebSnap.*

Быстрое создание WebSnap-приложения, работающего с базами данных

Рассмотрим конкретный пример создания полноценного *WebSnap*-приложения, работающего с базами данных (этот пример входит в документацию *Delphi 7).* Оно будет выполняться на *Web*-сервере и предназначено для воспроизведения в браузере таблиц базы данных, а также для выполнения различных действий над ними: редактирования отдельных полей, ввода новых записей или удаления существующих. Данный пример отличается от рассмотренных ранее тем, что в нем будут использованы два модуля (один для воспроизведения, а другой для редактирования информации, полученной из базы данных), причем синхронизация работы этих модулей и доступ к ним из браузера реализуется автоматически с помощью визуальных средств проектирования системы *Delphi.*

Создание такого приложения начинается с команды File ➤ New ➤ Other ➤ WebSnap (Файл > Создать >Другое > WebSnap), после чего в диалоговом окне выбирается зна-

чок WebSnap Application. На экране появится диалоговое окно, в котором определяются следующие параметры.

- Набор переключателей Server Type (Тип сервера). Рассматривался в предыдущих главах.
- Переключатель Application module components (Тип модуля). Значение Page Module позволяет создать модуль, который представляет собой поставщик *HTML*-страниц. Значение Data Module выбирается, когда требуется создать своеобразный единый контейнер для хранения компонентов, доступных в других модулях приложения.
- Кнопка Components позволяет определить, какие компоненты WebSnap будут исходно размещены в созданном модуле. После нажатия на нее открывается диалоговое окно, в котором с помощью флажков и раскрывающихся списков надо выбрать нужные для использования компоненты.

Рис. 11.21. *Выбор компонентов WebSnap*

Таблица 11.5. *Компоненты, выбираемые при генерации WebSnap-приложения*

Компонент	Назначение
Адаптер приложения (TApplicationAdapter)	Хранит информацию о текущем приложении
Адаптер конечного пользователя (TEndUserAdapter, TEndUserSessionAdapter)	Хранит информацию о пользователе: его имя, права доступа и другие сведения
Диспетчер страниц (TPageDispatcher)	Проверяет строку HTTP-запроса и вызывает соответствующий модуль
Диспетчер адаптеров (TAdapterDispatcher)	Распределяет работу между различными адаптерами в ходе формирования HTML-страницы
Диспетчируемое действие (TWebDispatcher)	Позволяет определить коллекцию объектов-действий для обработки поступающих запросов

продолжение

Таблица 11.5. *Компоненты, выбираемые при генерации WebSnap-приложения (продолжение)*

Компонент	Назначение
Служба размещения файлов (TLocateFileService)	Следит за загрузкой файлов-шаблонов и файлов сценариев в ходе работы приложения
Служба сессий (TSessionService)	Применяется для хранения информации о конечных пользователях, которая используется непродолжительное время, например в процессе отключения и подключения пользователя к системе
Служба списка пользователей (TWebUserList)	Отслеживает состояние подключенных к сети пользователей, их права и пароли

Для каждого из выбираемых объектов в раскрывающемся списке предлагается определенный компонент по умолчанию. При необходимости можно создавать свои собственные компоненты, предназначенные для обработки информации в архитектуре *WebSnap,*

В разделе Application module options в поле Page Name задастся название *WebSnap*-модуля. Кнопка Page Options позволяет задать дополнительные характеристики моду ля.

Рис. ***11.22. Настройка модуля WebSnap***

В частности, в разделе Producer выбирается один из множества доступных поставщиков информации, а также соответствующий ему сценарный язык. В зависимости от типа этого поставщика (от способа подготовки шаблонов в том или ином формате) следующий раздел будет называться HTML или XSL. Флажок New File устанавливают, если *Delphi* разрешается сгенерировать шаблон автоматически. В случае, если поставщик использует собственные шаблоны, задаваемые, как правило, в его внутренних свойствах HTMLDoc или HTMLFile, флажок надо сбросить.

В разделе Page задают имя модуля и его заголовок. Чтобы дать возможность модулю автоматически отвечать на запросы (при условии, что имя страницы Page Name СООТ-

ветствует строке запроса клиентской программы), надо включить флажок Published, Если требуется дополнительно вводить имя пользователя при обращении к данному модулю, надо установить флажок Login Required. Если создаваемый объект делается объектом по умолчанию (включается флажок Default), то данный модуль будет вызван для обработки клиентского запроса в случае, когда строка запроса пустая.

Вернемся в главное окно и в качестве типа сервера выберем значение Web App Debugger Executable (для упрощения отладки). В поле ввода CoClass Name укажем значение CountryTutorial — это значение будет совпадать с названием регистрируемого в *Windows COM*-сервера. Конечно, под таким именем никаких других серверов в системе быть не должно.

В качестве типа модуля выберем стандартное значение Page Module, что позволит сразу получить работоспособное *WebSnap*-приложение. В качестве имени модуля (поле ввода Page Name) зададим значение Home. По этому названию в дальнейшем можно отличить данный модуль от остальных. В завершение надо нажать кнопку OK. В результате система *Delphi* сгенерирует работоспособную заготовку.

Сгенерированную заготовку надо сохранить в отдельном каталоге. Поочередно сохраняются: модуль с названием Unit2 (по умолчанию) — в файле с более наглядным названием HomeU.pas; модуль Unit1 — в файле CountryU.pas, а файл проекта — в файле CountryTutorial.

Название приложения можно задать с помощью адаптера приложения (компонент TApplicationAdapter). В Инспекторе объектов выберем этот объект (он называется ApplicationAdapter и расположен в модуле Home) и в свойстве ApplicationTitle введем строку Тестовое приложение. Теперь можно переключиться в редактор и на вкладке Preview, расположенной в нижней части окна справа, посмотреть, как будет выглядеть начальная страница в клиентском браузере.

Чтобы организовать доступ к таблице баз данных, добавим в приложение новый модуль. Для этого дадим команду File > New > Other > WebSnap (Файл ➤ Создать > Другое > WebSnap) и выберем значок WebSnap Page Module. В появившемся окне надо задать параметры, специфичные для нового *Web*-модуля добавляемого к приложению.

В частности, чтобы готовить необходимые приложению данные на стороне сервера, потребуется компонент TAdapterPageProducer — он сформирует необходимую информацию, выполняя сценарии на сервере (эти сценарии написаны на языке *JScript*). Соответственно, в раскрывающемся списке Producer Type (тип поставщика) надо выбрать значение Adapter Page Producer. Назовем новый модуль CountryTable (раздел Page Name). Все остальные значения в окне оставим по умолчанию. В частности, в разделе *HTML* будет указано, что в качестве шаблона создаваемой этим модулем *HTML*-страницы используется автоматически сгенерированный *HTML*-файл. Он хранится в том же каталоге, что и все остальные файлы проекта. После нажатия на кнопку OK к проекту добавится новый *Web*-модуль. Проект надо сохранить, дав новому модулю название CountryTableU.pas.

Добавим в приложение компоненты, ответственные за работу с базами данных. К только что созданному модулю CountryTable добавим новый объект TTable с панели *BDE,* а также компонент TSession — архитектура *WebSnap* поддерживает сессии.

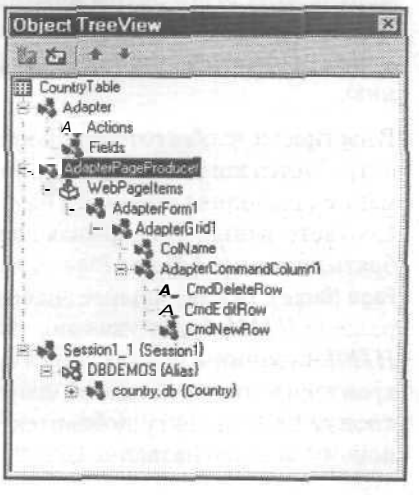
Эти два компонента надо правильно настроить. Так как всю работу с сессиями берут на себя компоненты WebSnap, то для объекта TSession достаточно в свойстве AutoSessionName с помощью Инспектора объектов установить значение True. Это означает, что для каждой вновь создаваемой сессии будет автоматически генерироваться уникальное название.

Компонент TTable надо настроить на таблицу country.db из базы данных *DBDEMOS.* Она входит в стандартную поставку *Delphi.* Сам объект (экземпляр TTable) назовем Country. Теперь его надо перевести в активное состояние, записав в свойство Active значение True. О том, как настраивать компоненты панели *BDE* на доступ к конкретным таблицам баз данных, рассказывалось ранее.

Для идентификации записи внутри таблицы необходимо указать ключевое поле этой таблицы, так как в приложении планируется выполнять такие действия над таблицей, как добавление или редактирование записей. Для задания ключевого поля надо вызвать редактор полей (команда Fields Editor контекстного меню табличного объекга Country) и в появившемся окне добавить поле Name. Подсвойство pfInKey свойства ProviderFlags этого поля надо установить равным True. Таким образом мы указали, что данное поле в приложении будет считаться ключевым.

Для непосредственного преобразования данных из таблицы (объект Country) в *HTML*-формат надо выполнить на сервере сценарий с помощью адаптера набора данных (компонент TDataSetAdapter). Этот компонент добавляется с панели WebSnap в текущий модуль. Для того чтобы привязать его к конкретному набору данных, надо в его свойстве DataSet выбрать объект-таблицу Country. Для наглядности назовем этот объект Adapter (свойсгво Name).

Теперь воспользуемся возможностями компонента TAdapterPageProducer для визуального проектирования *HTML*-страницы. Пользователю доступны богатые возможности по выбору и настройке различных полей отображения и редактирования данных в браузере.



Проще всего подобную настройку выполнять с помощью Просмотрщика дерева объектов. Для этого переключимся на модуль CountryTableU и вызовем Просмотрщик командой View ➤ Object TreeView.

Раскроем строку AdapterPageProducer, выберем единственный объект-наследник WebPageItems (он определяет набор визуальных элементов на *Web*-странице), с помощью контекстного меню дадим команду New Component (Новый компонент), в появившемся диалоговом окне выберем значение AdapterForm (Базовая форма

окна) и нажмем кнопку OK. У объекта WebPageItems появился новый наследник AdapterForm1, для которого теперь можно задавать конкретные визуальные *HTML*-компоненты. Выберем этот объект и дадим команду контекстного меню New Component.

Основной объект, занимающийся отображением информации из таблицы базы данных в окне браузера, — элемент AdapterGrid. Выберем его и нажмем кнопку OK. Для того чтобы связать данный визуальный объект с набором данных, в его свойстве Adapter (в раскрывающемся списке) надо выбрать значение Adapter — это ранее сформированный адаптер набора данных.

Теперь можно посмотреть, как выглядит результирующая страница в браузере. Для этого в окне редактора надо переключиться на закладку Preview. На *HTML*-странице отображается таблица со всеми значениями из соответствующей таблицы Country базы данных. Код сценария на языке *JScript* можно просмотреть и при необходимости отредактировать на вкладке окна редактора HTML Script.

Следующий шаг — добавление возможностей редактирования информации, доступной в браузере. Для примера насытим приложение такими возможностями, как добавление или уничтожение записей, редактирование существующих записей. Для этого в Просмотрщике дерева объектов выберем объект, который отвечает за отображение информации в окне браузера (AdapterGrid1) и дадим команду Add All Columns (Добавить все поля) контекстного меню. В результате в Просмотрщике появится единственный наследник данного объекта — объект ColName, который связан с единственным доступным полем Name объекта-таблицы Countries.

Теперь добавим объекту AdapterGrid1 новый объект AdapterCommandColumn, который описывает допустимые операции над связанным с данным адаптером набором данных. Выберем этот объект (он автоматически получит название AdapterCommandColumn1), дадим команду Add Commands (добавить действия) контекстного меню, и в появившемся окне выделим три строки: DeleteRow, EditRow и NewRow. Они обозначают соответственно разрешение на удаление, редактирование и добавление записи.

Посмотреть, как выглядит соответствующая форма в окне браузера, можно на вкладке предварительного просмотра Preview. Хорошо видны дополнительные кнопки работы с записями в каждой строке таблицы (рис. 11.23).

Рис. 11.23. *Предварительный просмотр экрана приложения*

Для того чтобы редактировать конкретную запись таблицы в окне браузера, требуется создать новую форму. Вся логика ее работы размещается в новом модуле (WebSnap Page Module). Он добавляется к приложению уже рассмотренным способом, а в качестве типа поставщика также выбирается значение TAdapterPageProducer. В качестве названия модуля в поле Page Name введем значение CountryForm и отключим флажок Published, так как данный модуль не предназначен для обработки *HTTP*-запросов. Он нужен только для редактирования отдельной записи в окне браузера. Далее этот модуль нужно сохранить с именем CountryFormU.pas.

Чтобы сделать доступными из данного *Web*-модуля объекты, расположенные в модуле CountryTable U, надо, сохраняя активным текущий модуль CountryForm, дать команду File ➤ Use Unit (Файл > Использовать модуль), выбрать модуль CountryTableU и нажать кнопку OK. Операцию можно проделать и вручную, явно указав название модуля в разделе реализации CountryFormU — после служебного слова uses. В визуальном режиме это делать, конечно, удобнее.

Теперь в данном модуле надо определить поля, которые предназначаются для редактирования значений записей в окне браузера. Рассмотренным выше способом добавим в Просмотрщике дерева объектов к компоненту AdapterPageProducer модуля CountryForm новый компонент AdapterForm1, а к этому компоненту (базовой форме) — новый объект AdapterFieldGroup. Он отвечает за группу полей редактирования. Переключимся на Инспектор объектов и для данного объекта AdapterFieldGroup1 в свойстве Adapter укажем объект CountryTable.Adapter — адаптер набора данных CountryTable.

Просмотреть создаваемую форму можно на вкладке Preview. Пока на ней видны лишь названия и значения полей, так как кнопки редактирования еще не определены. Это будет сделано на следующем шаге.

К объекту AdapterForm1 модуля CountryForm добавим новый компонент AdapterCommandGroup — список допустимых команд адаптера. Его надо связать с визуальными полями адаптера с помощью подсвойства Adapter свойства DisplayComponent. Значение CountryTable.Adapter задастся в этом подсвойстве в Инспекторе объектов.

Далее для объекта AdapterCommandGroup1 надо дать команду Add Commands (Добавить действия) контекстного меню, выбрать в диалоговом окне пункты Cancel, Apply и Refresh Row — они соответственно обозначают Прервать обновление, Внести изменения и Перезагрузить информацию из текущей записи.

На следующем шаге к добавленным на форму кнопкам требуется привязать соответствующие действия адаптера. Для этого в свойстве PageName кнопок CmdCancel и CmdApply надо ввести значение CountryTable — название модуля, который будет вызываться для формирования соответствующей *HTML*-страницы.

В заключение осталось указать, какая форма будет вызываться при нажатии на кнопки редактирования, добавления или удаления записей в модуле CountryTable. Переключимся на объект AdapterPageProducer этого модуля в Просмотрщике дерева объектов и в свойстве PageName (название соответствующего модуля) введем значение CountryForm для подобъектов CmdNewRow, CmdEditRow и CmdDeleteRow объекта AdapterCommandColumn1.

Создание приложения закончено. Мы не написали вручную ни одной строчки программного кода — вся работа велась исключительно с помощью визуальных средств, а созданное приложение обладает богатыми и гибкими возможностями по отображению и редактированию информации в браузерах. Приложение можно откомпилировать, запустить и проверить его в работе с помощью любого установленного в системе браузера.

Заключительный шаг — добавление в приложение средств, фиксирующих ошибки в ходе его работы и выводящих соответствующие предупреждающие сообщения в окнах браузеров. Подобные ошибки связаны, как правило, с попытками нескольких пользователей одновременно редактировать одну и ту же запись.

Добавим (через Просмотрщик дерева объектов) объекту AdapterForml модуля CountryTable новый компонент AdapterErrorList — адаптер, формирующий список ошибок. Для того чтобы возможные нарушения отлавливались раньше, чем они станут доступны средствам визуального отображения информации, этот объект надо сделать первым в списке подчиненных наследников объекта AdapterForml — просто переместите его вверх в списке нажатиями на кнопку «стрелка вверх» в Просмотрщике дерева объектов.

Чтобы привязать данный объект к адаптеру набора данных, в его свойстве Adapter (в раскрывающемся списке) надо выбрать значение Adapter.

Точно таким же способом добавляется адаптер, формирующий список ошибок, в модуль CountryForm. Его тоже надо расположить первым в списке наследников объекта AdapterForml, а в свойстве Adapter задать значение CountryTable.Adapter - ссылку на все тот же адаптер набора данных.

Как проверить механизм работы над ошибками? Заново перекомпилируем приложение и запустим его. Откроем два окна браузера, в одном из них нажмем кнопку Delete Row (Удалить запись). После этого, не выполняя обновление в другом окне браузера (такое обновление информации на экране выполняется стандартным образом — просто нажимается кнопка браузера Обновить экран, но сейчас этого делать не надо), попробуем удалить ту же самую (физически уже удаленную) запись нажатием этой же кнопки Delete Row. Программа среагирует на это генерацией прерывания и выводом в окне браузера над таблицей надписи Row not found in Country (Запись не найдена в таблице Country) (рис. \ **1.24).**

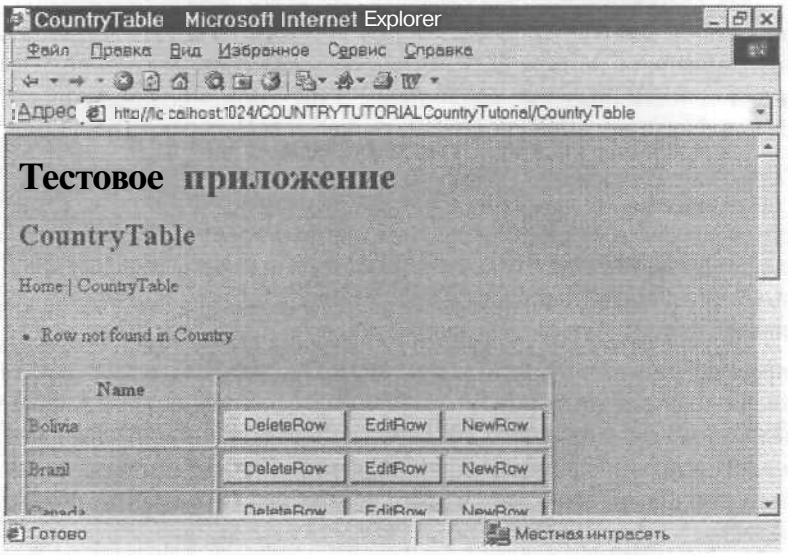


Рис. 11.24. *Браузер выводит сообщение об ошибке, полученное от модуля WebSnap*

Таблица 11.6. *Компоненты палитры WebSnap*

Компонент	Кнопка	Назначение
Адаптер TAdapter		Предназначен для выполнения на сервере сценария, с помощью которого формируются данные, в дальнейшем отображаемые в HTML-страницах и формах
Адаптер страниц TPageAdapter		Применяется, когда на клиентской стороне (в браузере) требуется отобразить большое количество информации. Данный компонент позволяет разбивать ее на несколько страниц и перемещаться между ними (для этого не требуется никакого дополнительного программирования). Число строк на экране в браузере задается в свойстве PageSize. Этот компонент фактически представляет собой расширение компонента TAdapter
Адаптер набора данных TDataSetAdapter		Предназначен для подготовки информации, хранящейся в базах данных. Для этого данный компонент связывается с другими компонентами — наборами данных. С помощью Адаптера набора данных можно дополнительно задавать допустимые операции над набором данных

Таблица 11.6. Компоненты палитры WebSnap (продолжение)

Компонент	Кнопка	Назначение
Адаптер формы доступа TLoginFormAdapter		Применяется для генерации форм, требующих ввода имени пользователя и пароля. Для обработки такой формы можно подготовить собственный обработчик события OnLogin. Он должен обращаться к свойствам рассматриваемых далее компонентов TWebUserList, TEndUserSessionAdapter и TSessionService, которые хранят всевозможную информацию о пользователях, подключенных в данный момент к системе
Список строковых значений TStringsValuesList		Представляет собой промежуточный адаптер, который можно использовать для формирования и поставки данных в виде списка строк «имя - значение»
Список значений набора данных TDataSetValuesList		Схож с компонентом TStringsValuesList, только значения списка не задаются в свойстве Strings напрямую, а берутся из набора данных
Хранитель компонентов TWebAppComponents		Предназначен для хранения компонентов, обеспечивающих базовую функциональность WebSnap-приложения
Адаптер приложения TApplicationAdapter		Может применяться для хранения списков действий (адаптеров), а также для хранения списка полей, в которых могут храниться пользовательские данные
Адаптер конечного пользователя TEndUserAdapter		Хранит информацию о конкретном пользователе, подключенном к системе (его имя и пароль). Дополнительная информация, в частности внутренний идентификатор пользователя, может применяться для определения прав доступа к различным сведениям в системе. Проверка этих прав выполняется в обработчике события OnHasRights
Адаптер сессии конечного пользователя TEndUserSessionAdapter		Хранит всю информацию об активности конкретного пользователя, подключенного к системе. Для его корректной работы требуется наличие в приложении компонента TSessionsService — службы сессий
Диспетчер страниц TPageDispatcher		Перенаправляет HTTP-запросы к соответствующему Web-модулю
Диспетчер адаптеров TAdapterDispatcher		Выполняет обработку информации, поступающей от HTML-форм, например когда пользователь ввел в поля формы в браузере различные данные и нажал на кнопку передачи. В зависимости от выполненного пользователем действия данный компонент выбирает подходящий обработчик подходящего адаптера. На данный компонент в обязательном порядке должно ссылаться свойство AdapterDispatcher компонента TWebAppComponents
Служба локальных файлов TLocateFileService		Этот компонент полезно применять, если требуется обращаться к информации, хранящейся в каталогах, отличных от применяемых по умолчанию. Дает возможность создавать собственные обработчики событий, возникающих при обращении других компонентов к файлам шаблонов и файлам сценариев

продолжение

Таблица 11.6. Компоненты палитры WebSnap (продолжение)

Компонент	Кнопка	Назначение
Служба сессий TSessionsService		Хранит информацию обо всех пользователях, подключенных в данный момент К системе, и автоматически проверяет их статус, выясняя, активны ли они. 6 свойстве DefaultTimeout задается время тайм-аута в минутах, по истечении которого соединение с конкретным пользователем считается прерванным. Этот компонент помогает очищать систему от неактивных или отключившихся клиентов
Список пользователей TWebUserList		Содержит список всех имен пользователей, их паролей и прав доступа. С помощью этого компонента выполняется проверка прав конкретного пользователя. Он позволяет создавать свои собственные проверки в моменты, предшествующие или последующие стандартным проверкам прав пользователя и корректности указанного им имени
Поставщик XSL-страниц TXSLPageProducer		Позволяет формировать содержимое Web-страниц путем преобразования данных, описанных на языке XML, на основе XSL-шаблонов. XML-данные указываются в свойстве XMLData
Поставщик страниц для адаптеров TAdapterPageProducer		Позволяет готовить HTML-данные, HTML-шаблоны и сценарии для адаптеров в визуальном редакторе

Сверхбыстрое создание Web-серверных приложений с помощью технологии IntraWeb

В седьмую версию *Delphi* вошел новый набор компонентов IntraWeb, сделавших процесс создания *Web*-приложении практически неотличимым от процесса создания обычного *Windows*-приложения. Подготовка *Web*-программы выполняется в визуальном проектировщике с помощью компонентов, внешне идентичных компонентам панели Standard. Отличить *Intra Web*-компонент можно по префиксу *IW*. Например, компонент-кнопка (TButton) в *Intra Web*-версии будет называться TIWButton и так далее.

Получающееся приложение можно быстро развертывать как на существующих *Web*-серверах, так и запускать в Интернет-среде вообще без *Web*-сервера.

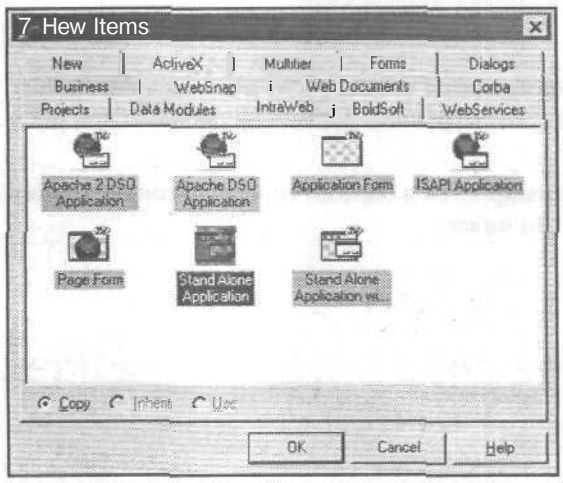
Intra Web-приложение автоматически распознает тип пользовательского браузера (поддерживаются *Internet Explorer* версий 4, 5 и 6, *Netscape* версий 4 и 6 и *Mozilla*) и динамически выполняет настройку своего интерфейса.

Сверхбыстрое создание Web-серверной игры «Камень-Ножницы-Бумага»

Б следующем примере рассмотрим разработку простой игровой Интернет-программы. Пользователь заходит (в браузере) на *Web*-страничкуигры, где ему предлагается начать новую игру и сделать выбор одного из трех вариантов «Камень-Ножницы-Бумага». Выбранный вариант отправляется на сервер, который случайно определяет ответ, вычисляет результат и показывает несложную статистику игры.

Несмотря на простые правила и полностью случайный исход, программа в среднем выигрывает немного чаще, потому что человеку сложнее сознательно выбирать совершенно случайные решения, не зависящие от истории игры.

Дадим команду File ► New ► Other (Файл > Новый х Другое) и на закладке IntraWeb выберем значок Stand Atone Application (Независимое приложение).



Рис, 11.25. *Создание IntraWeb-приложения*

Далее *Delphi* предложит сохранить создаваемый *IntraWeb*-проект в выбранном каталоге. В результате будет сгенерирован шаблон будущего приложения. Теперь надо перейги к форме приложения (например, командой View Form, SHIFT-F12 и выбором в списке названия FormMain. На экране появится обычная проектируемая форма.

Разместим на ней с панели IW Standard две кнопки TIWButton (Новая игра и Играть), группу переключателей TIWRadioGroup (создадим в пей три *переключателя* Камень, Ножницы и Бумага и укажем в свойстве ItemIndex значение -1, чтобы исходно ни один из переключателей не был выделен) и надпись TIWLabel, в которой будем показывать статистику игры (рис. 11.26).

Теперь запрограммируем логику работы программы. Нажатие кнопки Новая игра должно приводить к очистке результатов:

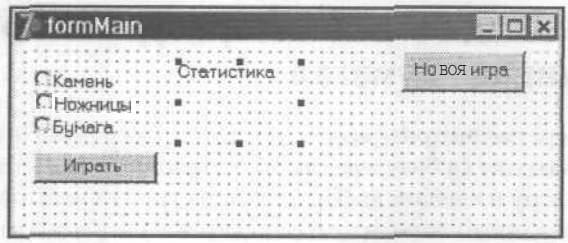


Рис. 11.26. Проект формы для игры

```
procedure TformMain.IWButton1Click(Sender; TObject);
begin
WinNum :=0;
TotalNum :=0;
ShowStat;
end;
```

В переменной WinNum хранится число побед человека, в RemiseNum — число ничьих, в **TotalNum** — общее число сыгранных партий. Процедура ShowStat выводит на экран статистику.

Логика игры программируется в обработчике нажатия на кнопку Играть. Полный текст модуля приведен ниже:

```
unit IWUnit1;
{PUBDIST}

interface

uses
    IWAppForm, IWApplication, IWTypes, IWCompLabel,
     IWExtCtrls, Classes, Controls, IWControl,
     IWCompButton;

type
    TformMain = class(TIWAppForm)
        IWButton1: TIWButton;
        IWButton2: TIWButton;
        IWRadioGroup1: TIWRadioGroup
        IWLabel1: TIWLabel,-
        procedure IWButton1Click(Sender; TObject);
        procedure IWButton2Click(Sender; TObject);
    public
```

```
    WinNum, remiseNum, TotalNum: Integer;

    procedure ShowStat;

    end;

implementation
    {SR *.dfm}

uses
    ServerController, SysUtils;

procedure TformMain.IWButton1Click(sender: TObject);
begin
WinNum :=0;
RemiseNum :=0;
TotalNum :=0;
ShowStat;
end;

procedure TformMain.ShowStat;
begin
IWLabel1.Caption :=
  'Сыграно' + IntToStr(TotalNum) + 'партий; Вы выиграли' +
  IntTo Dtr(WinNum) + 'партий; Вничью' +
  IntTo STR(RemiseNum);
end;

procedure TFormMain.IWButton2Click(Sender: TObject);
const Names: array[0..2] of String = ('Камень', 'Ножницы',
  'Бумага');
var A: Integer;
  S: String;
begin
if IWRadioGroup1.ItemIndex = -1 then
    WebApplication.ShowMessage('Сделайте выбор!')
else begin
```

продолжение

```
        inc(TotalNum);
        A:=random(3),-
        S :='Явыбираю' +Names[A] +' ; ';

        if A = IWRadioGroup1.ItemIndex then
        //варианты совпали
             begin
             WebApplication.ShowMessage(S+'Ничья!');
             inc(RemiseNum);
             end else

        if ((A+1) rood 3) = IWRadioGroup1.ItemIndex then
             begin
             webApplication.ShowMessage(S + 'Вы выиграли!');
             end
        else
             begin
             inc(WinNum);
             WebApplication.ShowMessage(S + 'Вы выиграли!');
             end;

        ShowStat;
        end;

end;
initialization
Randomise

end.
```

WebApplication — это объект класса TIWApplication, по характеристикам аналогичного стандартному классу TApplication.

Теперь можно откомпилировать и запустить приложение. Так как оно выполнено в Web-серверной архитектуре, то может быть развернуто либо на типовом *Web-сервере*, либо *запущено* с помощью сервера *Web-приложений IntraWebApplication Server*, входящего в поставку *Delphi*, *IntraWeb*-приложение Stand Alone Application будет выполняться под управлением именно последнего сервера, который заработает при запуске созданной программы из *Delphi*.

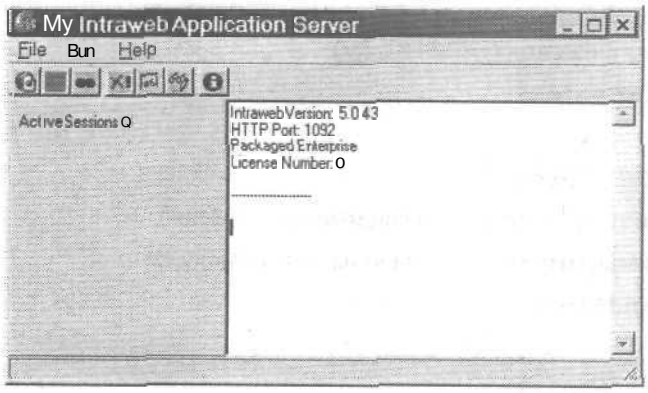


Рис. *11.27.* *Запуск сервера приложений IntraWeb*

Чтобы далее стартовать нашу программу, надо дать команду Run ➤ Execute (Запуск > Выполнить) этого сервера. В результате откроется браузер, в окне которого будет представлена спроектированная нами форма. С помощью кнопок управления можно начать игру с программой.

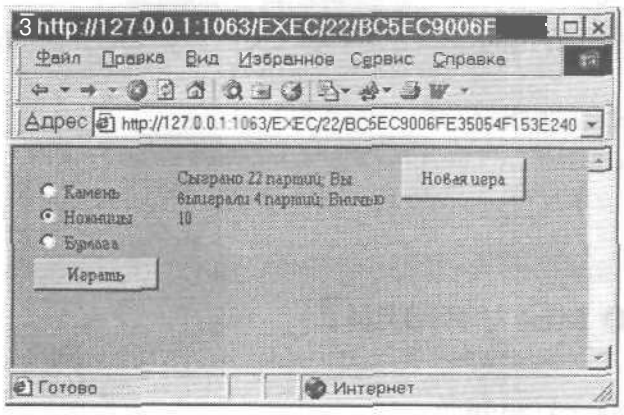


Рис. 11.28. *Игра с программой в браузере*

Необходимо отметить, что реальная работа программы происходит на сервере, а не на клиентском месте, Где запущен браузер (как бывает, если применяется технология активных Интернет-форм, см. следующую главу). При этом *IntraWeb*-программе вообще может быть не нужен стандартный *Web*-сервер типа *Apache* — достаточно использовать IntraWeb Application Server. Этот вариант также удобен, когда происходит разработка и отладка программы на обычном персональном компьютере.

Как перенести созданную нами программу на существующий *Web*-сервер? Для этого достаточно изменить всего несколько строчек кода, указав другие подключаемые библиотеки. Важно, что оригинальный исходный текст вообще никак не изменяется.

Например, для создания *DSO*-модуля для Apache 2 надо заменить первую строчку кода главного приложения (IWProject.dpr):

```
program IWProject;
```

на

```
library IWDSOProject;
```

(так как *DSP*-модули — это подключаемые библиотеки).

В списке подключаемых модулей надо изменить модуль

```
IWInitStandAlone
```

на

```
ApacheApp,IWInitApache
```

Б разделе реализации — указать расширение .so создаваемого файла:

```
{$R*.RES}
```

и добавить описание пространства *Apache-имен:*

```
exports
  apache_module Name 'IWDSOProject_module';
```

Для переноса в формат ISAPI/NSAPI(DLL) придется также изменить первую строчку на

```
library IWISAPIPriject;
```

и указать модуль IWInitISAPI вместо IWInitStandAlone в *uses-списке.*

Что нового мы узнали?

В этом уроке мы научились

- создавать модули для различных *Web*-серверов;
- отлаживать модули иа локальном компьютере;
- использовать *Web-формы;*
- публиковать данные на *Web*-сервере;
- проектировать клиенты и серверы *WebServices*;
- программировать сложные приложения *Web*-сервера с помощью технологии *Web Snap;*
- быстро создавать приложения *Web*-сервера с помощью технологии *IntraWeb.*

12 УРОК Дополнительные возможности системы Delphi

- Создание новых компонентов
- Создание апплетов Панели управления
- Взаимодействие с офисными приложениями
- Подготовка приложений к распространению
- Поддержка групповой работы
- Локализация приложений

Создание собственных компонентов

В этой главе рассматриваются способы создания компонентов *Delphi 7,* их включение в палитру компонентов, а также построение на их основе элементов *ActiveX.* Компоненты для системы *Delphi 7* разрабатываются сегодня во всем мире большим числом программистов, и такая деятельность часто полезна не только с профессиональной, но и с материальной точки зрения.

Создание компонентов Delphi 7

Рассмотрим основы технологии на примере разработки простейшего компонента, работающего как обычная надпись, но включающего дополнительные возможности. К ним относятся свойство FontColor для задания цвета шрифта, методы SetFontColor и GetFontColor для установки и получения цвета, а также событие OnFontColorChange, возникающее при изменении цвета шрифта.

Цвет шрифта проще устанавливать через характеристики свойства Font, поэтому наш пример служит только для иллюстрации процесса проектирования и программирования такого компонента.

\. Дайте команду File ➤ New ➤ Other (Файл ➤ Создать > Другое) и на вкладке New выберите значок Component (Компонент).

2. В открывшемся диалоговом окне в поле Ancestor type (Тип родителя) выбирается класс TCustomLabel. Мы создадим наш компонент на основе стандартной надписи. Обратите внимание, что используется название класса, начинающееся с префикса TCustom, который служит непосредственным предшественником всех пользовательских классов (рис. 12.1).

ЗАМЕЧАНИЕ Класс TCustomLabel (как и любой класс с именем, начинающимся с выражения TCustom) применяется, когда необходимо радикально изменить работу компонентов. Если же требуется лишь добавить новые возможности к уже существующим, обычно применяют класс TLabel или любой другой без префикса TCustom.

В поле Class Name (Имя класса) укажем название нового класса TMyLabel, в поле Palette Page (Страница палитры компонентов) — имя панели Samples (Образцы), в поле Unit file name (Имя файла модуля) — полный путь поиска и имя файла (например, C:\tmp\test\label\MyLabel). Если теперь щелкнуть на кнопке OK, система *Delphi 7* сгенерирует начальный текст будущего компонента. Пока что он полностью соответствует компоненту TLabel.

```
unit MyLabel;

interface
```

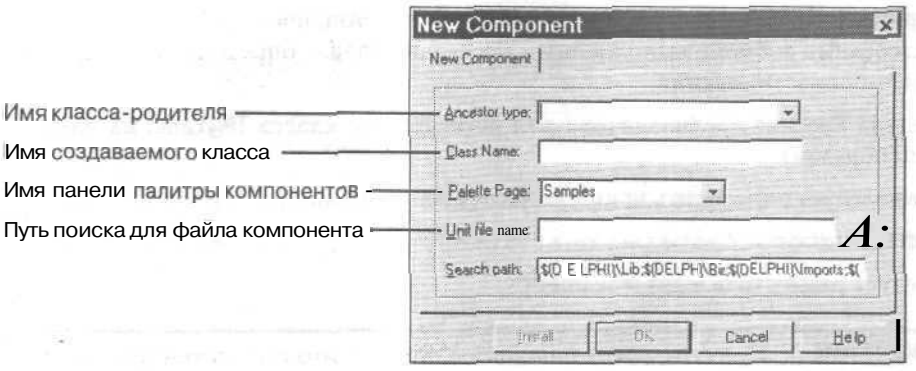


Рис. 12.1, *Настройка основных характеристик создаваемого компонента*

```
uses
  Windows, Messages, SysUtils, Classes, Graphics,
Controls, Forms, Dialogs,
  StdCtrls;

type
  TMyLabel = class(TCustomLabel)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure  Register;
begin
  RegisterComponents('Samples', [TMyLabel]);
end;

end.
```

Класс TMyLabel состоит из четырех разделов. После ключевого слова private обычно располагаются скрытые от посторонних внутренние детали реализации компонента,

после **protected** — определение интерфейса компонента, после **public** — определение интерфейса времени выполнения и после **published** — определение интерфейса времени проектирования.

Процедура Register предназначена для регистрации класса TMyLabel на панели Samples (Образцы).

3. Далее следует определить новый конструктор компонента.

```
constructor Create(AOwner: TComponent);
```

Его следует разместить в разделе **public**.

ЗАМЕЧАНИЕ При создании компонентов принято, что все параметры методов начинаются с буквы А.

Пока реализация его выглядит так:

```
constructor TMyLabel.Create(AOwner: TComponent);
begin
inherited Create(AOwner);
end;
```

Мы выполняем вызов конструктора родительского класса TCustomLabel.

Свойства класса TCustomLabel в компоненте TMyLabel не видны (скрыты в разделе **protected)** — их требуется вновь объявить. Каждое свойство начинается с ключевого слова **property.** Исключением являются самые общие базовые свойства, которые определяют положение компонента на форме, его размер и название.

Добавим свойство Caption, доступное на этапе проектирования, в раздел **published.**

```
property Caption;
```

Кроме того, нам потребуется свойство Font, на основе которого изменяется цвет надписи в поле:

```
property Font;
```

Теперь этот компонент в принципе можно использовать в реальной работе.

4. Если создается сравнительно большой компонент, желательно заняться его отладкой до того момента, когда он будет установлен на панели *Delphi 7.* Отлаживать компонент сам по себе нельзя, однако можно создать его копию в виде экземпляра класса в рамках обычного приложения. Для этого надо сформировать новый проект и включить в него модуль с описанием нашего компонента (рис. 12.2).

Сам компонент появляется в момент создания формы. При этом в параметре конструктора требуется указать значение Self (ссылку на главную форму), а свойству Parent (Родитель) компонента придется тоже присвоить значение Self.

```
unit Unit1;

interface
```

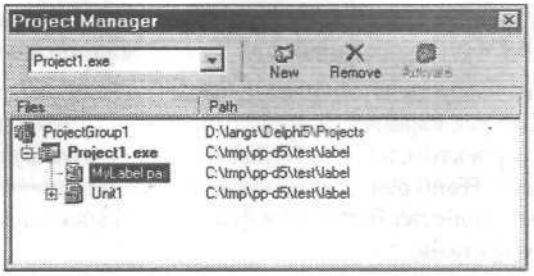


Рис. 12.2. *Создание тестовой программы, используемой для отладки нового компонента*

```
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs,
  MyLabel, StdCtrls;

type
  TForm1 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  L: TMyLabel;
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
procedure TForm1.FormCreate(Sender: TObject);
begin
L := TMyLabel.Create(Self);
L.Parent := Self;
L.Left := 100;
L.Top := 100;
L.Caption := 'hello!';
end;

end.
```

Запустив эту программу, мы увидим на форме текст в виде надписи. Пока что работа компонента TMyLabel ничем не отличается от работы компонента TLabel.

5. Следующий шаг — определение нового свойства FontColor, доступного на этапе проектирования. Так как параметры проектирования желательно сохранять между сеансами работы с проектом, в класс TMyLabel (раздел **private)** надо добавить внутреннюю переменную FFontColor (такие переменные начинаются с буквы F). Она сохраняет значение свойства FontColor в файле, описывающем форму с расположенными па ней объектами.

Само свойство FontColor записывается в раздел **published** с указанием типа, а также способа изменения его значения. Таких способов может быть два: либо простое присваивание (или считывание) значения внутренней переменной FFontColor, либо вызов специального метода, возвращающего значение (если происходит считывание свойства FontColor) или устанавливающего значение для этого свойства.

```
type
  TMyLabel = class(TCustomLabel)
  private
    { Private declarations }
  FFontColor: TColor;

  protected
    { Protected declarations }
  public
    { Public declarations }
    constructor Create (AOwner: TComponent);

  published
    { Published declarations }
    property Caption;
    property FontColor: TColor read FFontColor write
FFontColor;

end;
```

Ключевое слово **read** определяет способ считывания значения свойства FontColor, ключевое слово **write** — способ изменения его значения. Так как вслед за этими словами указано название внутренней переменной, то соответствующие действия выполняются путем обращения к переменной FFontColor. Можно также указать названия функций для выполнения более сложных действий над свойством, например так.

```
property FontColor: TColor read GetFontColor write
SetFontColor;
```

Названий этих функций должны начинаться с Get для считывания значения и с Set для его установки. Заканчивается имя метода названием свойства.

Сами методы помещаются к раздел **protected.**

```
procedure SetFontColor( AValue: TColor ); virtual;
function GetFontColor: TColor; virtual;
```

Их реализация в простейшем случае запишется следующим образом.

```
function TMyLabel.GetFontColor: TColor;
begin
Result ;= FFontColor
end;

procedure TMyLabel.SetFontColor(AValue: TColor);
begin
FFontColor : = AValue
end;
```

Метод SetFontColor (и любой другой метод, указанный после ключевого слова write) должен иметь один параметр, тип которого совпадает с типом свойства. Этот параметр является тем значением, которое требуется записать в данное свойство.

6. После того как базовая функциональность компонента определена, надо написать программный код, ответственный за вывод изображения на экран. Пока что этим занимается метод класса TLabel.

Переопределим метод Paint в разделе **protected,**

```
procedure Paint; override;
```

Его реализация может быть, например, такой.

```
procedure TMyLabel.Paint;
begin
Font.Color := FFontColor;
inherited;
end;
```

ВНИМАНИЕ Если вызывается переопределенный метод отрисовки объекта, то в нем обычно вызывается метод класса-родителя (ключевое слово **inherited),** как правило, в самом начале, чтобы отобразить основной вид компонента. В данном случае в переопределенном методе собственной работы с холстом не производится, а просто заранее меняется цвет шрифта, которым выводится надпись в методе класса-родителя.

7. Заключительный этап — добавление нового события. Это выполняется простым включением нового свойства и соответствующей ему внутренней переменной типа TNotifyEvent. Данный тип фактически является указателем на процедуру, обрабатывающую это событие.

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

Специально определять такой обработчик в компоненте не требуется, достаточно указать свойство подходящего типа (все события начинаются со слова On).

```
type
  TMyLabel = class(TCustomLabel)
  private
    { Private declarations }
  FFontColor: TColor;
  FOnFontChange: TNotifyEvent;

  protected
    { Protected declarations }
  procedure SetFontColor ( AValue: TColor ); virtual;
  function GetFontColor: TColor; virtual;
  procedure Paint; override;

  public
    { Public declarations }
    constructor Create(AOwner: TComponent) ;

  published
    { Published declarations }
    property Caption;
    property Font;

    property Fontcolor: TColor read GetFontColor write
      SetFontColor;
    property OnFontChange: TNotifyEvent read FOnFontChange
      write FOnFontChange;

  end;
```

В какой момент надо вызывать данное событие? Очевидно, когда изменяется значение свойства FontColor. Отследить изменение очень просто — оно происходит в методе SetFontColor.

```
procedure TMyLabel.SetFontColor(AValue: TColor);
begin
FFontColor := AValue;
if Assigned(FOnFontChange| then OnFontChange(Self);
end;
```

Однако это еще не все. Когда пользователь или программа изменяет цвет средствами Инспектора объектов, желательно сразу же поменять цвет надписи на новый. Для этого после оператора

```
FFontColor := AValue;
```

надо вызвать метод

```
Repaint;
```

Этот метод сразу же выполнит перерисовку внешнего вида объекта.

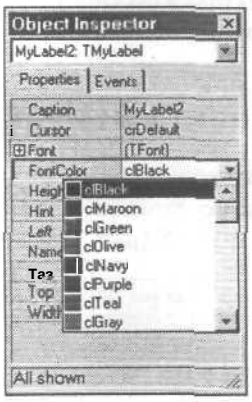
Установка и использование компонента

Итак, новый компонент создан. Осталось установить его на палитру *Delphi 7*.

Для этого выполняется команда Component > Install Component (Компонент ➤ Установить компонент). В появившемся диалоговом окне надо выбрать вкладку Into new package (В новый пакет). Компоненты хранятся в так называемых пакетах, специальных библиотеках *Delphi 7*. В поле Unit file name (Имя файла с модулем компонента) выбирается подготовленный модуль MyLabel.pas, в поле Package file name (Имя файла пакета) указывается произвольное имя пакета (например, LabelPack). В поле Package description (Описание пакета) обычно записывают название компонента в форме комментария (например, Cool Label), затем можно щелкнуть на кнопке OK. После компиляции модуля появится сообщение, что он успешно установлен на панель Samples. Если теперь взглянуть на эту панель, то компонент TMyLabel окажется самым последним в списке (рис. 12.3).

Рис, 12.3. *Новый компонент, представленный па панели Samples (Примеры)*

Далее можно создать новый проект и разместить на форме компонент TMyLabel. В Инспекторе объектов у него появится свойство FontColor (рис. 12.4).



РМС. ***12.4.*** *Изменение свойства нового компонента при помощи Инспектора объектов*

При его модификации сразу изменится и цвет надписи в поле. При желании можно стандартным способом задать обработчик события OnFontChange.

ЗАМЕЧАНИЕ Для удаления компонента надо дать команду Component > Install Packages (Компонент ➤ Установка пакетов). В диалоговом окне, содержащем список установленных пакетов, найдите пакет Cool Label, выделите его и щелкните на кнопке Remove (Удалить). Компонент будет удален.

Создание элементов ActiveX

Система *Delphi 7* обладает замечательной возможностью создавать компоненты *ActiveX на* основе имеющихся компонентов *Delphi 7.* Компоненты *ActiveX* являются фактическим стандартом применения компонентного подхода в среде *Windows,* их легко встраивать в любые системы программирования, а не только в *Delphi.* Поэтому найти людей, интересующихся хорошим компонентом *ActiveX,* обычно проще, чем тех, кто заинтересован в компоненте *Delphi* (см. www.activex.com).

Выполним команду File > New > Other (Файл ➤ Создать ➤ Другое) и на вкладке ActiveX выберем значок ActiveX Control (Элемент управления ActiveX).

В поле VCL Class Name (Имя класса VCL) укажем класс TButton, остальные поля заполняются автоматически. После щелчка на кнопке OK мы получим исходный текст приложения, описывающего класс TButtonX. Этот класс является наследником класса TButton (точнее, как объект *COM,* наследником его интерфейсов) и, с другой стороны, представляет собой элемент *ActiveX,* В сгенерированных файлах содержится подробное описание различных деталей функционирования этого класса, и при желании его можно легко расширить, например создать собственный редактор свойств объекта-кнопки.

После компиляции проекта в соответствующем каталоге появится файл ButtonXControl1.ocx — полноценный элемент *ActiveX.* Его можно на определенных условиях использовать в других проектах (предварительно ознакомившись с лицензионным соглашением корпорации *Borland).*

Подключение компонента ActiveX

Система *Delphi 7* способна не только экспортировать элементы *ActiveX,* но и включать их в состав своей палитры. Так, все компоненты *Delphi,* расположенные на панели ActiveX, являются элементами *ActiveX* (рис. 12.5).

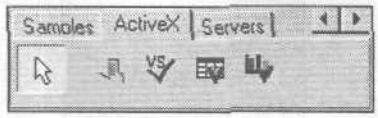


Рис. 12.5. Компоненты Delphi, представляющие собой элементы ActiveX

Допустим, нам требуется добавить в систему *Delphi 7* компонент OCX ButtonXControl1. Выполним команду Component ➤ Import ActiveX Control (Компонент ➤ Импорт элемента управления ActiveX). В открывшемся диалоговом окне щелкнем на кнопке Add (Добавить) и найдем файл ButtonXControll.ocx. Выбранную по умолчанию в поле Palette Page (Страница палитры) панель палитры ActiveX и другие настройки можно не менять. После щелчка на кнопке Install (Установить) потребуется указать произвольный пакет формата *Delphi 7,* в котором сохраняется оболочка элемента *ActiveX.* На панели ActiveX появится новый компонент TButtonX, функционально эквивалентный компоненту TButton.

Использование активных форм в Интернете

Что такое активные формы

Одной из разновидностей элементов *ActiveX* являются активные формы. Они готовятся на основе форм (используя базовые классы *Delphi 7),* что помогает создавать компоненты практически неограниченной сложности.

В этой главе рассказывается о разработке и способах применения активных форм, в частности встроенных в *Web*-страницы.

Создание активной формы

Активная форма создается командой File > New ➤ Other (Файл > Создать > Другое) и выбором значка ActiveForm (Активная форма) на вкладке ActiveX. В результате генерируется «пустая» (на самом деле, конечно, содержащая много программного кода) заготовка активной формы. Одновременно форма откроется в Проектировщике форм.

Сохраните проект в отдельном каталоге, а затем разместите на форме два текстовых поля, кнопку и надпись (рис. 12.6).

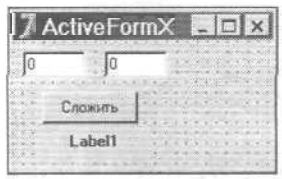


Рис. 12.6. Размещение элементов управления в рамках активной формы

После ввода в поля двух чисел и щелчка на кнопке их сумма выводится в виде надписи. Данный элемент *ActiveX* представляет собой небольшой калькулятор.

Вызовем Редактор библиотеки типов и добавим к интерфейсу IActiveFormX новый метод Compute. Он должен возвращать значение типа Integer на основе обработки двух своих параметров X и Y, тоже целого типа.

После щелчка на кнопке Refresh Implementation (Обновить реализацию) перейдем к модулю ActiveFormImpl1, описывающему реализацию работы элемента. Найдем в нем метод Compute и напишем, что он будет делать.

```
function TActiveFormX.Compute(X, Y: Integer) : Integer;
bagin
Result := X+Y
end;
```

Надо также определить, что произойдет по щелчку на кнопке Сложить. Создадим соответствующий обработчик.

```
procedure TActiveFormX.Button1Click(Sender: TObject);
begin
Label1.Caption := IntToStr(
   Compute( StrToInt(Edit1.Text),
            StrToInt(Edit2.Text) } );
end;
```

Выполним компиляцию проекта, и в каталоге появится файл ActiveFormProj1.ocx. Это и есть наш элемент *ActiveX.* Но регистрировать его в системе пока не следует. Этот элемент будет загружаться в самые разные браузеры пользователей, и его регистрация должна выполняться автоматически.

Включение активной формы в Web-страницу

Чтобы этот элемент OCX можно было распространять через Интернет, создадим небольшой файл *HTML* следующего содержания.

```
<HTML>
<Hl>Delphi 7 ActiveX Тест</H1xp>
<HR><center><P>

<OBJECT
 classid="clsid:15446ED4-AE1C-11D3-A3F4-F1483AC3561C"
 codebase="http://TestBed/ActiveFormProj1.ocx"
 width=134
 height=103
 align=center
 hspace=0
 vspace=0
>
</OBJECT>

</HTML>
```

Сам компонент описывается с помощью тега OBJECT. Значение его атрибута CLASSID должно равняться идентификатору *GUID* этого компонента. Значение *GUID* можно получить, обратившись, например, в Редакторе библиотеки типов к элементу ActiveFormX (рис. 12.7).

В атрибуте CODEBASE указывается местоположение данного компонента в Интернете, откуда он загружается в браузер пользователя. В нашем случае указан адрес локального *Web*-сервера http://TestBed/ActiveFormProj1.ocx.

Так как элементы *ActiveX в* отличие от *Web*-модулей выполняются не на *Web-сервере,* а на компьютере клиента, то любой программист может создавать такие элементы, хранить их на своей домашней странице (например, на сервере chat.ru, как рассказывалось выше) и вставлять в свои тексты *HTML.* Для этого надо указать в атрибуте CODEBASE полный путь поиска для элемента, например:

```
http://delphi5test.chat.ru/MyForm.OCX
```

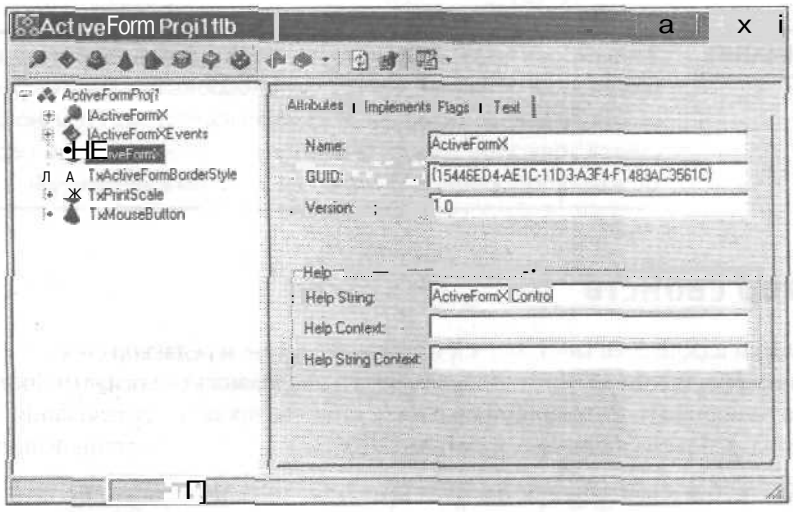


Рис. 12.7. Определение значения идентификатора GUID с помощью Редактора библиотеки типов

Что произойдет, если загрузить эту страничку в браузере? Осуществится автоматическая загрузка, подключение и запуск элемента *ActiveX* на компьютере пользователя (рис. 12.8).

***Рис. 12.8.** Работа созданного элемента ActiveX в рамках Web-страницы*

Не следует забывать, что элемент *ActiveX,* созданный с помощью системы *Delphi 7,* достаточно велик по объему и пользователь может просто не дождаться окончания его загрузки.

ВНИМАНИЕ В некоторых браузерах может быть установлен запрет на выполнение элементов ActiveX, так как они способны выполнять практически любые действия, что потенциально очень опасно. Обычно рекомендуется запускать только те элементы ActiveX, которые сертифицированы корпорацией Microsoft и заверены электронной подписью.

Редактор свойств

Практически каждый элемент *ActiveX* имеет собственный редактор свойств, который по назначению аналогичен Инспектору объектов и предназначен для настройки свойств элемента во время его функционирования, а также во время использования в других системах разработки (например, в системе *Delphi* 7) на этапе проектирования.

Допустим, требуется, чтобы наш элемент выполнял не только сложение, но и умножение двух чисел. Для этого добавим в раздел private класса TActiveFormX новую переменную FAdd.

```
private
    { Private declarations }
    FAdd: boolean;
```

Она принимает значение True, если необходимо выполнить сложение двух чисел. Тогда метод Compute перепишется следующим образом.

```
function TActiveFormX.Compute(X, Y: Integer): Integer;
begin
if FAdd then Result := X+Y
        else Result := X*Y
end;
```

Далее добавим компоненту новое свойство (в Редакторе библиотеки типов), назовем его Status. Пусть оно имеет тип WordBool (аналог типа Boolean в Паскале). Создадим реализацию методов Get_Status/Set_Status, предназначенных для получения и установки значения свойства:

```
function TActiveFormX.Get_Status : WordBool;
begin
Result := WordBool(FAdd)
end;

procedure TActiveFormX.Set_Status(Value: WordBool);
begin
FAdd := boolean(Value);
if FAdd then Button1.Caption := 'Сложить'
        else Button1.Caption := 'Умножить'
end;
```

При установке значения изменяется также и надпись на кнопке.

Теперь можно добавить к проекту заготовку редактора свойств нашего элемента. Для этого выполняется команда File > New > Other (Файл ➤ Создать ➤ Другое) и на вкладке ActiveX выбирается значок Property Page (Страница свойств).

Появится новая пустая форма, па которой надо разместить элементы управления для задания значения свойству Status нашего компонента. Разместим на ней один компонент TRadioGroup с двумя переключателями. В программном коде соответствующего модуля присутствуют две процедуры.

- Процедура UpdatePropertyPage предназначена для установки состояния элементов управления на форме в зависимости от значения свойств компонента.
- Процедура UpdateObject предназначена для задания значения свойств компонента в зависимости от состояния элементов управления.

Сам компонент доступен в программном коде формы через переменную OleObject.

```
procedure TPropertyPage1.UpdatePropertyPage;
begin
  { Update your controls from OleObject }
  RadioGroup1.ItemIndex := OleObject.Status;
end;

procedure TPropertyPage1.UpdateObject;
begin
  { Update OleObject from your controls }
  OleObject.Status := RadioGroup1.ItemIndex;
end;
```

Теперь надо подключить этот редактор свойств к компоненту. Для этого вернемся в модуль ActiveFormImpLl и найдем в его начале процедуру DefineProperty Pages. В нее надо включить вызов конкретной настроечной формы. Для этого используется метод DefineProperty Page, в качестве параметра которого указывается идентификатор *GUID* класса формы свойств. Соответствующая константа генерируется системой *Delphi 7* автоматически. Ее можно найти сразу после описания класса TPropertyPage в модуле Unit1.

```
const
  Class_PropertyPage1: TGUID =
  '{15446EDC-AE1C-11D3-A3F4-F1483AC3561C}';
```

Тогда вызов формы свойств запишется следующим образом.

```
procedure
TActiveFormX.DefinePropertyPages(DefinePropertyPage:
TDefinePropertyPage) ;
```

продолжение

```
begin
DefinePropertyPage(Class_PropertyPage1);
end;
```

Теперь выполним компиляцию всего проекта, и на этом создание активной формы с редактором свойств будем считать законченным. Объект ActiveFormProj1.ocx можно, например, добавить на панель компонентов *Delphi 7,* выполнив команду Component ➤ Import ActiveX Object (Компонент ➤ Импорт объекта ActiveX) и включив наш компонент *ActiveX* в список компонентов Delphi (он появится на панели ActiveX). Далее его можно установить на форму. При двойном щелчке на нем откроется редактор свойств (рис. 12.9).

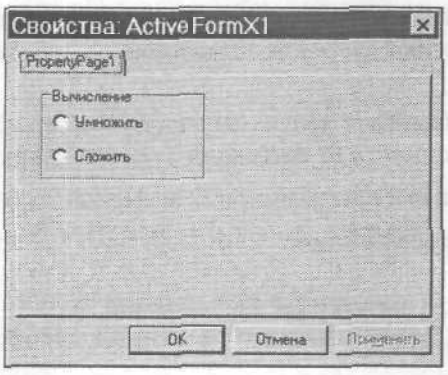


***Рис. 12.9.** Редактирование свойств компонента ActiveX в ходе работы программы*

Доступ к интерфейсу компонента из **кода** HTML

Добавив к интерфейсу IActiveFormX новый метод Compute и новое свойство Status, мы сделали их доступными для использования любыми другими программными системами, поддерживающими технологию *COM.*

В частности, управлять работой компонентов *ActiveX* может любой браузер, поддерживающий их выполнение. Для этого требуется включить в код страницы *HTML* внутрь элемента <OBJECT> (но не в сам тег!) дополнительные теги, описывающие параметры запуска элемента *ActiveX.* В **этих** параметрах можно указать начальные значения доступных свойств. В нашем случае устанавливается начальное значение свойства Status, для определения действия выполняемого элементом (сложение или умножение).

Расширенный текст тега **<OBJECT>** выглядит следующим образом.

```
<OBJECT
 classid="clsid:15446ED4-AE1C-11D3-A3F4-F1483AC3561C"
 codebase="http://TestBed/ActiveFormProj1.ocx"
 width=134
 height=103
 align=center
```

```
    hspace=0
    vspace=0
  >

  <PARAM NAME="Status" VALUE="0">

  </OBJECT>
```

Новый тег PARAM *имеет два атрибута:* NAME, значение *которого равно названию* свойства компонента, и VALUE, значение которого равно значению этого свойства. В нашем случае 0 эквивалентен значению False, а 1 — значению True.

Если открыть созданную страницу *HTML в* браузере, она получит следующий вид (рис. 12.10).

Рис. 12.10. *Элемент ActiveX, настраиваемый путем задания параметров на странице HTML*

ЗАМЕЧАНИЕ Обращаться к методам компонентов ActiveX можно также из языков сценариев JavaScript и VBScript, активно встраиваемых сегодня во многие страницы HTML

Работа с Панелью управления Windows

Принцип работы с Панелью управления

На Панели управления *Windows* (она вызывается командой Пуск > Настройка > Панель управления) расположены значки объектов (так называемых *апплетов*), которые

применяются для настройки различных системных программ и драйверов *Windows*, а также принтеров, другого оборудования, сети и так далее.

На самом деле все эти утилиты и программы представляют собой обычные библиотеки .DLL, переименованные в файлы с расширением .CPL *(Control Panel Library* — библиотека Панели управления). Чтобы добавить новый элемент на эту Панель, надо просто скопировать файл .DLL (выполненный в соответствии с определенными требованиями на экспорт функции CPlApplet, фактически выполняющей всю работу по нужным настройкам) в каталог \Windows\System, изменив его расширение.

На Панель управления, как правило, устанавливаются значки программ, ответственных за различные системные процедуры. Такие программы обычно хранят свои настройки в системном реестре и в каталоге *Windows* в файлах с расширением .INI, разбитых на разделы, названия которых взяты в квадратные скобки. Структура файлов .INI стандартна:

```
название = значение
```

Например, программа-администратор драйверов *ODBC* (odbccp32.cpl) хранит свои настройки в файлах ODBC.INI и ODBCINST.INI.

Требования к хранению настроечной информации в системном реестре или в каталоге *Windows* именно в таком формате необязательны. Программа, вызываемая щелчком на значке Панели управления, может выполнять самые разные действия, но желательно все же придерживаться стандартных требований *Windows* к организации работы подобных приложений.

Создание заготовки апплета

В системе *Delphi 7* имеется Мастер создания библиотеки .CPL, в которую может входить несколько апплетов. Этот Мастер вызывается командой File ➤ New > Control Panel Application (Файл > Создать ➤ Приложение Панели управления).

Он создает пустую заготовку библиотеки .CPL с одним, также пустым апплетом, функция CPlApplet для которого экспортируется автоматически.

В дальнейшем новые апплеты добавляют в библиотеку командой File ➤ New ➤ Control Panel Module (Файл > Создать > Модуль Панели управления).

Создадим таким способом главный модуль и добавим в него два новых апплета. На вкладке Components (Компоненты) каждого апплета можно размещать различные невизуальные компоненты *Delphi 7*, например для доступа к базе данных.

Значок, который отображается на Панели управления для конкретного апплета, задается в свойстве AppletIcon, а подпись под ним указывается в свойстве Caption. От апплетов обычно требуется обработка некоторых стандартных сообщений *Windows* (если апплет выделен или требуется перерисовать Панель управления). К таковым относятся запросы на получение от апплета информации о его значке и названии, а также дополнительных сведений. По умолчанию весь программный код, подставляющий стандартные значения в ответ на такие запросы (это, в частности, значения свойств AppletIcon и Caption), генерируется автоматически. Программисту остается

только определить реакцию на событие OnActivate, то есть указать, что произойдет, когда пользователь вызовет апплет щелчком на соответствующем значке Панели управления.

ЗАМЕЧАНИЕ Когда Панель управления закрывается, каждый апплет получает сообщение OnStop,

Пример

Создадим библиотеку .CPL и добавим в нее два апплета. Настроим каждый из них: зададим оригинальные значения свойствам Caption и AppletIcon. При вызове апплеты должны выдавать информационное сообщение.

```
procedure TAppletModule1.AppletModuleActivate(Sender:
TObj ect;
  Data: Integer);
begin
// здесь вставляется оригинальный код

ShowMessage('апплет 1');
end;

...

procedure TAppletModule2.AppletModuleActivate(Sender:
TObject;
  Data: Integer);
begin
// здесь вставляется оригинальный код

ShowMessage('апплет 2');
end;
```

Регистрация и отладка библиотек CPL

Теперь приложение можно откомпилировать. Б том каталоге, где был сохранен соответствующий проект, появится файл Project1.cpl. Его надо зарегистрировать в системе. Это можно сделать двумя способами.

1. Файл Projectl.cpl копируется в каталог \Windows\System, после чего производится перезагрузка *Windows*.Теперь при открытии Панели управления на ней появятся два новых значка с подписями из свойств Caption. При щелчке на любом из них откроется диалоговое окно с коротким сообщением.

   Это стандартный подход. Когда библиотека .CPL включается в состав широко распространяемого приложения, го обычно копирование соответствующего

файла в каталог \Windows\System выполняется в процессе установки с помощью стандартной программы. Можно также копировать этот файл программно (например, с помощью метода CopyFrom), а затем перезагружать систему.

2- Во время разработки сложного апплета его приходится часто регистрировать и затем удалять. Каждый раз перезагружать *Windows* неудобно и долго, поэтому в системе *Delphi 7* имеется возможность быстрой установки апплетов и столь же быстрого их удаления.

Для этого надо перейти к модулю соответствующего апплета в Проектировщике и вызвать контекстное меню. В его верхней части располагаются три пункта.

- Пункт Install Control Panel Applet (Установить апплет Панели управления) осуществляет установку апплета на Панель управления. Реально происходит добавление всех апплетов, входящих в соответствующую библиотеку.
- Пункт Uninstall Control Panel Applet (Удалить апплет Панели управления) позволяет удалить апплет с Панели управления. Реально происходит удаление всех апплетов, входящих в соответствующую библиотеку.
- Пункт Launch Control Panel (Запустить Панель управления) позволяет открыть Панель управления.

Для отладки апплета его надо запустить с помощью стандартной программы Rundll32.exe. Для этого выбирается пункт строки меню Run ➤ Parameters (Запуск ➤ С параметрами), в поле Host Application (Приложение, запускающее модуль) надо указать строку c:\windows\rundll32.exe или c:\winnt\system32\rundll32.exe, в зависимости от того, в каком каталоге хранится программа rundll32.exe. В поле Parameters (Параметры) надо задать строку shell32.dll,Control_RunDLL "AppletName", где AppletName - полный путь поиска для библиотеки .CPL, например:

```
shell32.dll,Control_RunDLL "c:\tmp\test.cpl"
```

Теперь после выполнения команды Run (Запуск) или нажатия клавиши F9 сразу запускается апплет из этой библиотеки.

Управление работой офисных приложений

Офисные программы как серверы автоматизации COM

При создании объектов *COM* упоминались такие расширения технологии *COM,* как серверы автоматизации *COM* — программы, функциями которых можно управлять через интерфейс *COM*. Все приложения, входящие в состав последних версий *Microsoft Office,* являются такими серверами. Например, можно программно вызвать редактор *Word,* загрузить в него нужный документ, выполнить проверку орфографии или заполнить поля документа значениями из базы данных, а потом выве-

сти документ на печать, после чего закрыть редактор. Аналогичным способом организуется и работа с электронной таблицей *Excel,* с базой данных *Access, с* почтовой программой *Outlook* и т. д.

Возможность обращения к серверам автоматизации в своих программах существовала и в прежних версиях системы *Delphi,* однако там она требовала от разработчика специфических знаний нюансов автоматизации *COM.* В версии *Delphi 7* появилась целая панель Services (Службы), на которой расположены готовые компоненты, позволяющие быстро и легко обращаться к наиболее популярным офисным приложениям *Microsoft,* а также к созданным с их помощью документам (что очень важно!) как к серверам автоматизации *COM.*

Все эти компоненты являются наследниками класса TOleServer:

```
type TOleServer = class(TComponent, IUnknown);
```

Он сочетает в себе свойства обычного компонента *Delphi* и возможность организации доступа через интерфейс IUnknown. Благодаря этому разработчик получает возможность использовать так называемое *раннее связывание* (на этапе компиляции) методов и свойств с сервером *COM.* Когда конкретные типы используемых методов и списки их аргументов известны заранее, возникает возможность избежать ошибок преобразования типов и множества других неточностей.

Раннее связывание выполняется прямым обращением к таблицам виртуальных функций (VTable). Эта технология создавалась специально для использования разработчиками встроенного во все продукты *MS Office* языка программирования *Visual Basic for Applications.* Она реализована, начиная с 5-й версии системы *Delphi.*

В документации корпорации *Microsoft* подробно описаны все свойства и методы соответствующих серверов *COM* (редактора и документа *Word,* программы работы с электронными таблицами *Excel* и пр.). Для работы с компонентами панели Servers необходимо воспользоваться этой документацией.

Справочная система по всем методам и свойствам объектов *MS Office* входит в стандартную поставку этого продукта, хотя и не инсталлируется по умолчанию. В каталоге, где установлены программы *MS Office,* надо поискать файлы .HLP, начинающиеся с префикса *VBA.* Если их нет, придется выполнить пользовательскую установку пакета *MS Office.* Например, описание объекта COM Microsoft Excel 97 хранится в файле vbaxl8.hlp.

С помощью Редактора библиотеки интерфейсов можно получить описание всех свойств и методов соответствующих объектов на языке Паскаль. В состав системы *Delphi 7* входят 20 файлов, сгенерированных автоматически и содержащих подробные описания всех интерфейсов для продуктов *MS Office 97/2000.* Эти файлы расположены в каталоге \Delphi 7\Ocx\Servers.

Пример автоматической загрузки редактора Word

В качестве примера рассмотрим такую задачу: надо загрузить редактор *MS Word,* открыть документ c:\doc\MyDoc.doc, ввести в него строку Документ для печати, затем распечатать документ, сохранить его под именем dd.doc и закрыть редактор.

Создадим новый проект, разместим на нем компонент TWordApplication, ответственный за запуск редактора, и компонент TWordDocument, ответственный за обработку конкретного документа. Добавим также кнопку, по щелчку на которой и будут выполнены описанные действия.

Свойствам AutoConnect (автоматический запуск сервера *COM,* в нашем случае редактора *Word)* и AutoQuit (автоматическое завершение работы сервера *COM)* компонента TWordApplication надо присвоить значение True. В свойстве ConnectKind (вид связи с сервером) остается значение по умолчанию ckRunningOrNew. Оно означает, что используется уже запущенный сервер *COM* (копия редактора). В противном случае автоматически запускается новый:

```
procedure TForm1.Button1Click(Sender: TObject);
var n: OleVariant;
begin
   n := 'c:\doc\MyDoc.doc';
   WordApplication1.Documents.Open(n,EmptyParam,EmptyParam,
      EmptyParam,EmptyParam,EmptyParam,EmptyParam,
      EmptyParam,EmptyParam,EmptyParam);
  WordDocument1.ConnectTo(WordApplication1.ActiveDocument);
   WordApplication1.Selection.TypeText ('Документ для
    печати');
   WordDocument1.PrintOut;
   n := 'dd.doc';
   WordDocument1.SaveAs(n);
   WordDocument1.Close;
end;
```

Название файла документа заносится в переменную п типа OleVariant. Это требование интерфейса Vtable: большинство параметров различных функций должны отвечать формату языка *Visual Basic.* Затем происходит обращение к свойству Documents (массиву открытых документов) объекта-редактора WordApplicationl и выполняется метод открытия документа Open. И свойство Documents, и метод Open описаны в упоминавшихся файлах документации по *MS Office.* У метода Open много параметров, но нам нужен только первый — название документа. Для неиспользуемых параметров можно указать значение EmptyParam (это стандартная константа *Delphi 7).*

Затем выполняется очень важное действие: объект WordDocumentl с помощью метода ConnectTo связывается с уже существующим интерфейсом текущего активного документа (открытого файла MyDoc.doc). Теперь к данному объекту можно обращаться, как к соответствующему активному документу редактора. Чтобы это действие выполнялось корректно, на этапе проектирования для объекта WordDocumentl в свойстве ConnectKind надо указать значение ckAttachToInterface (подключение к существующему интерфейсу).

Далее выполняется ввод строки (с помощью метода TypeText свойства Selection объекта WordApplicationl), затем печать (метод Printout). Послеэтогодокументсохра-

няется под новым именем (dd.doc) и закрывается. На протяжении всего процесса редактор *Word* работает в фоновом режиме и на экране не отображается. Запущено только ядро сервера *COM,* ответственное за выполнение соответствующих методов,

Пример автоматической загрузки электронной таблицы Excel

Теперь рассмотрим сходный пример работы с электронной таблицей *Excel 97.* Точно так же на форме размещаются компоненты TExcelApplication (свойствам AutoConnect и AutoQuit присваивается значение True) и TExcelWorkbook (Книга Excel), для которого в свойстве ConnectKind указывается значение ckAttachToInterface.

Требуется открыть существующий файл k1.xls, в ячейку (1,1) записать текст для печати, распечатать документ и закрыть файл. Технология работы с компонентами такая же, как в предыдущем примере.

```
procedure TForm1.Button1Click(Sender: TObject);
var n: OleVariant;
begin
   n := 'c:\k1.xls';
   ExcelApplication1.Workbooks.Add(n,0);
  ExcelWorkbook1.ConnectTo(ExcelApplication1.ActiveWorkbook);
   ExcelApplication1.Cells.Item[1,1].Value := 'Для печати';
   ExcelWorkbook1.PrintOut;
   ExcelWorkbook1.Save;
   ExcelWorkbook1.Close;
end;
```

В примере опять использован метод Add, а также свойство Cells (ячейки таблицы), описанное в документации корпорации *Microsoft* по *Excel 97* (файл vbaxl8.hlp).

Пример доступа к базе данных Access 97

По сравнению с другими компонентами панели Services продукт *MS Access 97* является исключением, потому что для него не существует компонента TAccessApplication. Это связано с тем, что корпорация *Microsoft* скрыла интерфейс этого сервера *COM,* сделав его по каким-то собственным соображениям недоступным. Поэтому использовать вышеописанный подход не удается. Для получения доступа к серверу *COM Access* мы прибегнем к более сложному методу, который применялся в старых версиях системы *Delphi.*

Такой метод относится к технологии *позднего связывания,* когда доступ к интерфейсу формируется во время работы программы. При этом не удается контролировать корректность типов параметров вызываемых методов сервера *COM.* Кроме того, позднее связывание работает менее эффективно, чем раннее.

Следующий код динамически создает объект *OLE,* представляющий собой сервер *COM Access,* который открывает базу данных C:\bs.mdb и выполняет связывание с

компонентом TDoCmd (он должен быть размещен на форме), ответственным за выполнение команд *Access*. После этого имеющаяся в базе данных таблица MyTabLe открывается для просмотра в режиме «только чтение».

```
var R: IDispatch;
    a: Access97 .Application;
begin
   R := CreateOleObject('Access.Application');
   OleCheck(R.QueryInterface(_Application, a) | ;
   a.OpenCurrentDatabase('C:\BS.MDB',False);
   DoCmd1.ConnectTo(a.DoCmd);
   DoCmd1.OpenTable('MyTable',acViewNormal,acReadOnly);
...
```

Функция CreateOleObject возвращает интерфейс IDispatch (наследник интерфейса IUnknown, ориентированный на применение в серверах *COM*). Далее с помощью метода QueryInterface происходит связывание интерфейса _Application (стандартного интерфейса для компонентов типа TxxxApplication) с неременной a. Теперь к этой переменной можно обращаться практически так же, как к несуществующему компоненту TAccessApplication (по аналогии с TWordAplication, TExcelAplication).

Заключение

На панели Servers имеется еще ряд компонентов, обеспечивающих доступ к приложениям *PowerPoint* к *Outlook*, а также к документам соответствующих форматов и различным элементам оформления (например, к шрифту редактора *Word)*, которые реализованы в виде серверов автоматизации *COM*. Для их использования надо обратиться к справочным руководствам корпорации *Microsoft* по языку *Visual Basic for Application*.

Установка и развертывание приложений

Общие принципы

Процесс создания готового продукта не завершается на этапе написания работоспособной программы. Приложение, как правило, должно быть установлено у пользователя или группы пользователей, в локальной сети или Интернете. Надо учитывать, что обычный пользователь и не подозревает о таких понятиях, как библиотека .DLL, элемент *ActiveX,* сервер *COM* или *BDE*. Многие не знают о понятиях «файл» или «программа» и работают с *Windows* на уровне значков и папок. Для таких категорий пользователей в *Windows 9x* появился неофициальный стандарт на установку новых приложений, например с компакт-дисков. Для этого использу-

ются инсталляционные программы (обычно SETUP.EXE), которые работают аналогично Мастерам *Delphi 7.* Они задают в процессе установки несколько уточняющих вопросов (например, о каталоге, в который должна быть помещена программа) и затем выполняют все действия по инсталляции приложения. В стандартную поставку системы *Delphi 7* входит программа создания инсталляционных копий InstallShield Express Limited Edition 3.03 for Delphi 7. Прежде чем приступить к ее рассмотрению, остановимся на работе Реестра *Windows.*

Работа с Реестром

Файлы .INI

Рассматривая создание компонентов для Панели управления, мы упоминали про способ сохранения настроек программ в виде текстовых файлов .INI. Они разбиты на разделы, названия которых заключены в квадратные скобки, а следом идут конкретные значения настроек в определенном виде:

```
название-параметра-настройки=значение
```

ЗАМЕЧАНИЕ Названия настроек называются *ключами.*

Хотя этот способ устаревший, многие самые современные приложения пользуются им по сей день. Для его применения в системе *Delphi 7* имеется класс TIniFile, который обладает небольшим набором свойств и методов для гибкой обработки файлов .INI. Свойство у этого класса только одно — FileName, имя файла .INI. Методы класса TIniFile приведены в табл. 12.1.

Таблица 12.1. *Методы класса TIniFile*

Метод	Назначение
procedure DeleteKey (const Section, Ident: string);	Удаление ключа и связанного с ним значения
procedure EraseSection(const Section: string);	Удаление целого раздела
procedure ReadSection (const Section: string; Strings: TStrings);	Считывание названий всех ключей в разделе
procedure ReadSections(Strings:TStrings);	Считывание названий всех разделов
procedure ReadSectionValues(const Section: string; Strings: TStrings);	Считывание значений всех ключей в разделе
function ReadString (const Section, Ident Default: string): string;	Считывание всех строк раздела, которые представляют собой пары «ключ = значение»
function UpdateFile;	Запись данных из буфера в файл .INI
procedureWriteString { const Section, Ident, Value: string);	Запись строки в файл .INI
constructor Create(const FileName String);	Создание файла .INI

продолжение

Таблица 12.1. Методы класса TIniFile (продолжение)

Метод	Назначение
function ReadBool (const Section, Ident: String; Default: Boolean): Boolean; procedure WriteBool(const Section, Ident: String; Value: Boolean);	Считывание/запись значения ключа в формате Boolean
function ReadDate (const Section, Ident; string; Default: TDateTime): TDateTime; procedure WriteDate(const Section, Ident: String; Value: TDateTime);	Считывание/запись значения ключа а формате даты (TDateTime)
function ReadDateTime (const Section, Ident: String; Default: TDateTime): TDateTime; procedure WriteDateTime(const Section, Ident: String; Value: TDateTime);	Считывание/запись значения ключа в формате даты/времени (TDateTime)
function ReadFloat (const Section, Ident: String; Default: Double): Double; procedure WriteFloat (const Section, Ident: String; Value: Double);	Считывание/запись значения ключа в формате Double
function ReadInteger(const Section, Ident: String; Default: Longint): Longint; procedure WriteInteger(const Section, Ident: String; Value: Longint);	Считывание/запись значения ключа в формате Longint
function ReadTime (const Section, Ident: String; Default: TDateTime): TDateTime; procedure WriteTime(const Section, Ident: String; Value: TDateTime);	Считывание/запись значения ключа в формате времени (TDateTime)
function SectionExists (const Section: String): Boolean;	Возвращает значение True, если указанный раздел существует

Следующий пример демонстрирует загрузку названий всех разделов системного файла WIN.INI в список ListBox1. При этом в список ListBox2 выводятся все ключи раздела Ports, а в список ListBox3 — соответствующие им значения:

```
var AppIni: TIniFile;
begin
   Applni := TIniFile.Create('WIN.INI');
   Applni.ReadSections(ListBox1.Items);
   Applni.ReadSection('Ports',Listbox2.Items);
   AppIni.ReadSectionValues('Ports',ListBox3.Items);
   AppIni.Free;
end;
```

Повторим, что этот способ официально признан устаревшим и гарантировать его поддержку в будущих версиях *Windows* нельзя. В то же время в *Linux* системного Реестра нет, а так как исходные тексты *Delphi 7* в определенных ситуациях можно переносить в *Kylix*, то поддержка в *Delphi .INI*-файлов, популярных в *Linux* (где они используются для настройки приложений), обретает второе дыхание.

Системный Реестр

В последних версиях *Windows* вместо файлов .INI применяется системный Реестр *(Registry)* — база данных, в которой данные организованы в виде сложной иерархической структуры. То есть, если исходить из аналогии с файлами .INI, у любого раздела может быть неограниченное число вложенных подразделов на произвольную глубину. Файлы Реестра имеют не текстовую, а двоичную структуру, и вручную редактировать их нельзя. Для этого применяют стандартную *Windows*-утилиту RegEdit. Для каждого ключа может быть создан набор пар «имя — значение», а также вложенные ключи. На самом верхнем уровне в Реестре имеется шесть ключей. Четыре первых ключа относятся к настройкам компьютера. Остальные предназначены для описания пользователей, так как *Windows*— многопользовательская система.

Таблица 12.2. *Корневые ключи Реестра Windows*

Ключ	Назначение
HKEY_CLASSES_ROOT	Настройки пользовательского интерфейса, расширений, технологии COM
HKEY_LOCAL_MACHINE	Настройки компьютера и системных программ
HKEY_CURRENT_CONFIG	Текущие настройки аппаратной конфигурации
HKEY_DYN_DATA	Настройки подключаемых устройств
HKEY_USERS	Пользователи и настройки Windows
HKEY_CURRENT_USER	Информация о текущем пользователе и настройках его интерфейса Windows

Доступ к значениям Реестра происходит примерно так же, как и к разделам файлов .INI. Для этого используется класс TRegistry (модуль Registry). Как и другие классы, он имеет свойства (табл. 12.3) и методы (табл. 12.4).

Таблица 12.3. *Свойства класса TRegistry*

Свойство	Назначение
Access	Уровень доступа к Реестру
CurrentKey	Текущий открытый ключ Реестра
CurrentPath	Путь в Реестре к текущему ключу
LazyWrite	Имеет значение True, если программа работает только с открытыми ключами. В противном случае производительность программы может резко снизиться
RootKey	Базовый ключ регистра

Таблица 12.4. *Методы класса* ***TRegistry***

Метод	Назначение
procedure CloseKey;	Записать значение текущего ключа в Реестр и закрыть ключ
function CreateKey(const Key: String): Boolean;	Создать новый ключ

продолжение &

Таблица 12.4. *Методы класса TRegistry(продолжение)*

Метод	Назначение
function DeleteKey(const Key: String): Boolean;	Удалить указанный ключ И все связанные с ним значения
function DeleteValue (const Name: String): Boolean;	Удалить все значения, связанные с текущим ключом
function GetDataInfo(const ValueName: String; var Value: TRegDataInfo): Boolean;	Получить информацию о типе и размере значения текущего ключа
function GetDataSize(const ValueName: String): Integer;	Получить размер значения текущего ключа в байтах
function GetDataType(const ValueName: String): TRegDataType;	Получить информацию о типе значения текущего ключа
function GetKeyInfo(var Value: TRegKeyInfo): Boolean;	Получить подробную информацию о значении текущего ключа
procedure GetKeyNames(Strings: TStrings);	Получить список названий всех подключеи текущего ключа
procedure GetValueNames(Strings: TStrings);	Получить список названий всех значений текущего ключа
function HasSubKeys: Boolean;	Возвращает значение True, если ключ имеет вложенные подключи
function KeyExists(const Key: String): Boolean;	Возвращает значение True, если указанный ключ существует
function LoadKey (const Key, FileName: String): Boolean;	Создать новый подключ базового ключа и загрузить его значение из файла
procedure MoveKey(const OldName, NewName: String; Delete: Boolean);	Переместить ключ вместе со всеми его значениями и вложенными подключами на новое место
function OpenKey(const Key: String; CanCreate: Boolean): Boolean;	Открыть ключ
function OpenKeyReadOnly(const Key: String): Boolean;	Открыть ключ в режиме «только чтение»
function Read BinaryData (const Name: String; var Buffer; BufSize: Integer): Integer; procedure WriteBinaryData(const Name: String; var Buffer; BufSize: Integer);	Считать/записать значение ключа в двоичном виде
function ReadBool(const Name: String): Boolean; procedure WriteBool(const Name: String; Value: Boolean);	Считать/записать значение ключа в виде значения Boolean
function ReadCurrency(const Name: String): Currency; procedure WriteCurrency(const Name: String; Value: Currency);	Считать/записать значение ключа в виде значения Currency
function ReadDate(const Name: String): TDateTime; procedure WriteDate(const Name: String; Value: TDateTime);	Считать/записать значение ключа в виде значения даты (TDateTime)

Таблица 12.4. Методы класса TRegistry (продолжение)

Метод	Назначение
function ReadDateTime(const Name: String): TDateTime; procedure WriteDateTime(const Name: String; Value: TDateTime);	Считать/записать значение ключа в виде значения даты/времени (TDateTime)
function ReadFloat(const Name: String): Double; procedure WriteFloat(const Name: String; Value: Double);	Считать/записать значение ключа в виде значения Double
function ReadInteger(const Name: String): Integer; procedure WriteInteger(const Name: string; Value:Integer);	Считать/записать значение ключа в виде значения Integer
function ReadString(const Name: String): String; procedure WriteString(const Name, Value: String);	Считать/записать значение ключа в виде значения String
function ReadTime(const Name: String): TDateTime; procedure WriteTime(const Name: String; Value: TDateTime);	Считать/записать значение ключа в виде значения времени (TDateTime)
function RegistryConnect(const UNCName: String): Boolean;	Возвращает значение True, если удалось установить связь с Реестром на другом компьютере
procedure RenameValue(const OldName, NewName: String);	Изменить название значения, связанного с текущим ключом
function ReplaceKey(const Key, FileName, BackUpFileName: String): Boolean;	Заменить всю информацию об указанном ключе и вложенных подключах на информацию из файла FileName. Старая информация сохранится в файле BackUpFileName
function RestoreKey(const Key, FileName: String): Boolean;	Восстановить всю информацию об указанном ключе и его вложенных подключах из файла FileName
function SaveKey(const Key, FileName: String): Boolean;	Сохранить всю информацию об указанном ключе и его вложенных подключах в файле FileName
function UnLoadKey(const Key: String): Boolean;	Удалить ключ из Реестра
function ValueExists(const Name: string): Boolean;	Возвращает значение True, если указанное значение существует для текущего ключа

Следующий код поочередно отображает в диалоговом окне имена значений в ключе (сам ключ указывается в поле Edit1) и запрашивает у пользователя разрешение на удаление текущего значения:

```
var Reg: TRegistry;
    Val: TStringList;
    I: Integer;
begin
```

продолжение

```
    Reg:=TRegistry.Create;
    try
      Val:=TStringList.Create;
      try
        Reg.RootKey:=HKey_Local_Machine;
        if not Reg.OpenKey(Edit1.Text,False) then
          ShowMessage('Не удается открыть ключ')
        else begin
          Reg.GetValueNames(Val);
          for I:=0 to Val.Count-1 do
            ShowMessage(Edit1.Text + ': ' +
              Val.Strings[I] + '=' +
              Reg.ReadString(Val.Strings[I]));
            if MessageDlg('Удалить ключ?',
              Mtinformation,[mbYes,mbNo],0) = mrYes
              then Reg.DeleteKey(Edit1.Text);
        end;
      finally
        Val.Free;
      end;
    finally
      Reg.Free;
    end;
  end;
```

В системе *Delphi 7* имеется также класс TRegIniFile, который позволяет без больших усилий переводить приложения, использующие сохранение настроек в файлах .INI, на работу с системным Реестром. Этот класс, наследник класса TRegistry, позволяет работать с Реестром с помощью тех же методов, что и класс TIniFile, только в свойстве FileName указывается не имя файла, а ключ Реестра.

Настройка коммерческой версии приложения

Перед созданием инсталляционной копии программы необходимо сформировать версию, ориентированную на массовое использование. Для этого надо дать команду Project ➤ Options (Проект > Параметры) и выполнить ряд настроек.

Вкладка Application (Приложение)

В поле Title (Заголовок) вводится заголовок приложения, привязываемый к его значку. После установки приложения он отображается в папках *Windows, в* Главном меню, при переключении между запущенными программами с помощью комбинации ALT+TAB и других местах. В поле Help file (Файл справки) указывается имя файла справки, в раздел Icon (Значок) с помощью кнопки Load Icon (Загрузить значок) загружается значок приложения.

Вкладка Compiler (Компилятор)

С помощью этой вкладки обеспечивается максимальное быстродействие программного кода; отладочная информация не генерируется. Необходимо установить флажки, как показано на рис. 12.11.

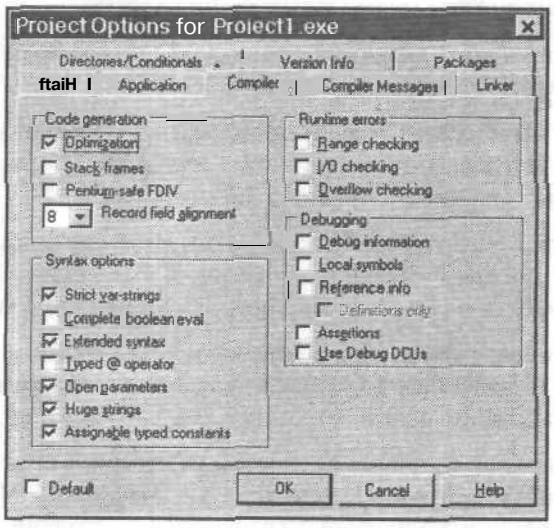


Рис. 12.11. Установка флажков, обеспечивающих максимальное быстродействие программы

Все флажки на панелях Runtime errors (Ошибки времени выполнения) и Debugging (Отладка) должны быть сброшены. При этом отключается контроль ошибок времени выполнения: выхода индекса за границы массива, переполнения, ошибок ввода/вывода и др.

Вкладка Packages (Пакеты)

Очень важно правильно настроить элементы управления этой вкладки. Если посмотреть на размер любого автономного приложения, созданного с помощью системы *Delphi 7,* то он, скорее всего, составит не менее 300-400 Кбайт. Это связано с тем, что к программе подключены различные пакеты системы *Delphi 7:* библиотеки стандартных функций, классов, визуальных компонентов, модулей, ответственных за работу *BDE и* серверов *COM* и др. Обойтись без них практически никогда не удается, однако если определенные пакеты уже установлены у пользователя, то повторно включать их в программу установки уже не требуется и размер *EXE*-файла может уменьшиться в десять и более раз.

Чтобы создать программу, не включающую содержимое пакетов *Delphi 7,* надо установить флажок Build with runtime packages (Создать программу, использующую пакеты времени выполнения). Создать программу в таком режиме можно, только если разработчик уверен, что у пользователя эти пакеты уже установлены, или если они включаются в состав устанавливаемого пакета.

Вкладка Version Info (Информация о версии)

Если установлен флажок Include version information in project (Включить в проект сведения о версии), то в программу помещается информация о текущей версии. Она доступна, если пользователь выберет значок программы, установленной в системе *Windows,* и в контекстном меню укажет пункт Свойства.

В случае установки этого флажка доступны следующие поля. Каждый из номеров может принимать значения в диапазоне от 0 до 65535.

- Major version number (Главный номер версии). Обычно меняется, когда программа переделывается полностью или в нее вносятся принципиальные улучшения (например, версии 5 и 6 системы *Delphi*).
- Minor version number (Вспомогательный номер версии). Обычно определяет внесение в программу менее существенных изменений, связанных с повышением эффективности или расширением функциональных возможностей.
- Release version number (Номер версии выпуска). Обычно фиксирует незначительные изменения в программе: исправление обнаруженных ошибок, усовершенствование пользовательского интерфейса и пр.
- Build version number (Номер версии сборки). Обычно определяет число полных компиляций программы, выполненных командой Project > Build (Проект > Построить). Контроль сборки требуется при создании крупных проектов, когда *параллельно* работают независимые группы *программистов* и требуется согласовывать вносимые изменения. Данный номер увеличивается автоматически, если включить флажок Auto-increment build number (Автоматическое увеличение номера сборки),

В разделе Module attributes (Атрибуты модуля) задаются значения флажков. Эти флажки используются только для хранения установленных значений и никак не связаны с реальными настройками программы. За правильностью таких установок разработчику надо следить самому.

Таблица 12.5. *Атрибуты версии программы*

Флажок	Назначение
Debug build	Отладочная версия
Pre-release	Предварительная некоммерческая версия
DLL	Динамически загружаемая библиотека
Special build	Специальный вариант стандартной версии
Private build	Версия, не предназначенная для распространения

В раскрывающемся списке Language (Язык) выбирается язык, на котором оформлен пользовательский интерфейс, в списке Key/Value list box (Список ключей и значений) — различная дополнительная информация о проекте (название продукта, компании, авторские и имущественные права и прочие сведения).

Вкладка Directories/Conditionals (Каталоги и настройки компиляции)

Здесь задаются каталоги для компиляции, сборки и получения готовых версий программы, а также дополнительные директивы компилятора. Если в соответствующем окне установлен флажок Default (По умолчанию), то выбранные настройки используются как принятые по умолчанию. При создании нового проекта он получает именно такие настройки.

В завершение надо закрыть окно Project Options (Параметры проекта) щелчком на кнопке OK и выполнить полную перестройку проекта командой Project > Build (Проект ➤ Построить).

Приложение InstallShield

Сравнительно крупные программы хранят в Реестре (или в файлах .INI) большое число собственных настроек. Работать с Реестром вручную, выполнять множество рутинных действий при установке и удалении приложения весьма трудоемко. Для создания инсталляционной копии программы, которая автоматически внесет в Реестр все настройки, а при удалении ликвидирует их, в поставку *Delphi 7* включена программа *InstallShield.* Она обладает рядом ограничений, связанных с лицензионными требованиями к ядру баз данных *BDE,* СУБД *InterBase* и т. д.

Таблица 12.6. *Разделы настройки процесса инсталляции*

Раздел	Содержание
General Information	Общая информация о продукте: его название, название компании, различная контактная информация, значок программы, версия продукта и т. д.
Features	В данном пункте задаются специфические характеристики процесса установки. С их помощью можно определить, какие части приложения будут устанавливаться в зависимости от выбора пользователя. Имеется возможность организации такого выбора в иерархическом виде
Setup Types	Тип установки, позволяющий устанавливать программу в типичной поставке, в минимальной поставке или в зависимости от настроек пользователя
Upgrade Paths	С помощью данного пункта можно задать способ обновления ранее установленных версий данного продукта
Specify Application Data	В данном разделе выбираются файлы, входящие в приложение (Files). В подразделе Files and Features задаются дополнительные характеристики этих файлов. В пункте Objects/Merge Modules можно добавлять к проекту всевозможные стандартные объекты и библиотеки, входящие в поставку Delphi и Windows. Это могут быть, например, библиотеки времени выполнения для средств разработки типа Microsoft Visual C++. Наконец, в пункте Dependencies выполняется полный анализ взаимосвязей между всеми входящими в проект файлами на наличие повторов, противоречий, отсутствия необходимых объектов и т. д.

продолжение

Таблица 12.6. Разделы настройки процесса инсталляции (продолжение)

Раздел	Содержание
Configure the Target System	Задаются каталоги (Shortcuts/Folders) целевой системы, в которой будет устанавливаться приложение. Пункт Registry позволяет с помощью визуального редактора определить, какие изменения будут вноситься в системный Реестр, пункт ODBC Resources с помощью подобного древовидного редактора позволяет легко настроить драйверы ODBC. В пункте INI File Changes задаются изменения, вносимые в файлы инициализации, как стандартные, так и используемые самой программой. С помощью пункта File Extensions можно сформировать взаимосвязи между расширениями устанавливаемых в операционной системе файлов и приложением, которое будет обрабатывать эти файлы по умолчанию. Пункт Environment Variables позволяет дополнить список переменных среды окружения — это требуется некоторым программам для корректного функционирования
Customize the Setup	На шаге Dialogs определяется, какие диалоговые окна отображаются в ходе инсталляции и какую информацию они выдают пользователю. В пункте Billboards можно задать всевозможные экранные заставки, сопровождающие процесс инсталляции. Пункт Text and Messages позволяет вынести текстовые сообщения в отдельный модуль, чтобы не привязываться к конкретному функциональному языку и в дальнейшем иметь возможность быстро настраивать процесс инсталляции на конкретную локализованную версию
Enable Automatic Updates	Услуга InstallShield Update Service позволяет создавать самообновляющиеся приложения. При наличии Интернет-соединения такое приложение само будет периодически обращаться на Web-сервер с поисках обновлений. Пользователь при условии выполнения регистрации будет автоматически уведомляться о наличии новых программных улучшений
Define Setup Requirements and Actions	В разделе Requirements можно определить конкретные требования, которые устанавливаемая программа предъявляет к операционной системе. Это может быть тип этой системы, ее версия, процессор, объем оперативной памяти, разрешение экрана и т. п. В некоторых случаях возможностей данной инсталляционной программы недостаточно для установки сложных приложений. Тогда можно воспользоваться разделом Custom Actions. Он определяет дополнительные пользовательские действия, точнее, программы, которые выполняют специфические действия по настройке конкретной целевой платформы. Таким программам могут потребоваться собственные файлы, которые имеет смысл устанавливать на компьютер пользователя только на время работы инсталляционной программы. Эти временные файлы можно определить в пункте Support Files
Prepare for Release	В пункте Build Your Release задается конкретный физический носитель, на котором планируется разместить инсталляционную программу. В зависимости от типа такого носителя можно дополнительно указать требуемый объем, способ сжатия, стандартные средства для синхронизации программ установки, входящих в ядро Windows, и тому подобные характеристики. Тестирование созданной инсталляционной копии выполняется на шаге Test Your Release в полном соответствии с выбранным типом целевого физического носителя. На последнем этапе Distribute Your Release созданный ранее образ инсталляционного приложения переносится на выбранный физический носитель. Очень удобно, что в данной версии программы поддерживаются такие носители, как CD-ROM, DVD

Программа довольно проста в использовании. После ее запуска выбирается либо пункт Создание нового проекта (Create a new project), либо пункт Открытие существующего проекта (Open a project), а затем появляется окно настройки параметров.

Процесс создания инсталляционной копии состоит из шести этапов. На первом задаются общие настройки инсталляционного проекта, на втором перечисляются файлы, входящие в копию, на третьем определяются настройки на целевую платформу, где будет работать приложение, на четвертом описывается сам процесс инсталляции и сопроводительная информация, на пятом — требования к самой программе инсталляции и на шестом выполняется создание и тестирование инсталляционной программы.

При распространении приложений, созданных с помощью системы *Delphi 7,* лучше всего применять приложение *InstallShield.* Это связано с тем, что оно входит в число немногих программ, официально сертифицированных корпорацией *Borland* для создания инсталляционных копий, способных автоматически устанавливать на компьютеры пользователей механизм *BDE* и другие важнейшие части приложений, правильно вносить регистрационную информацию в Реестр. Корпорация *Borland* рекомендует применять программу *InstallShield* для развертывания приложений, созданных с помощью системы *Delphi 7.*

Отметим главные положительные особенности этой программы:

- корректная регистрация в системе не только *EXE*-файлов, но и вспомогательных библиотек .DLL, файлов справки, пакетов *Delphi,* компонентов *ActiveX;*
- автоматическая установка драйверов *BDE;*
- автоматическая инсталляция модулей и драйверов *DataSnap.*

Создание справочной системы

Использование справочной системы в программах

Любая программа, предназначенная для массового распространения, снабжается электронной справочной системой (файлы с расширением .HLP), вызываемой, как правило, нажатием клавиши F1. При этом отображается именно тот раздел системы, который относится к текущей области действий пользователя. Например, если нажать клавишу F1, находясь в Проектировщике форм системы *Delphi 7,* то откроется раздел справочной системы, описывающий текущий выделенный компонент на форме.

Справочная система по принципу создания напоминает составление программы на Паскале. Сначала готовится исходный текст, который содержит специальные

управляющие символы. Затем этот текст включается в проект, который компилируется — переводится в файл в формате .HLP. Его можно стандартным образом просматривать в системе *Windows,* а также включать в свои программы.

Как создать простой раздел справочной системы

1. Введите текст раздела в редакторе, который способен сохранять файлы в формате .RTF. Удобнее всего использовать для этого редактор *Word 97.*
2. Добавьте идентификационное обозначение раздела, которое в дальнейшем будет использоваться для обращения к нему.

   Такой идентификатор записывается в виде IDH_<произвольный-текст>, а перед ним предварительно вставляется символ # в режиме сноски (команда Вставка ➤ Сноска). Пример вставки такого идентификатора показан на рис. 12.12.

ЗАМЕЧАНИЕ Названия идентификаторов разделов должны начинаться с префикса IDH_ (хотя это и не обязательно), для того чтобы иметь возможность контекстного вызова конкретного раздела справочной системы из программы при нажатии клавиши F1.

3. В начало раздела надо в режиме сноски добавить заголовок, которому должен предшествовать символ $ (рис. 12.12).

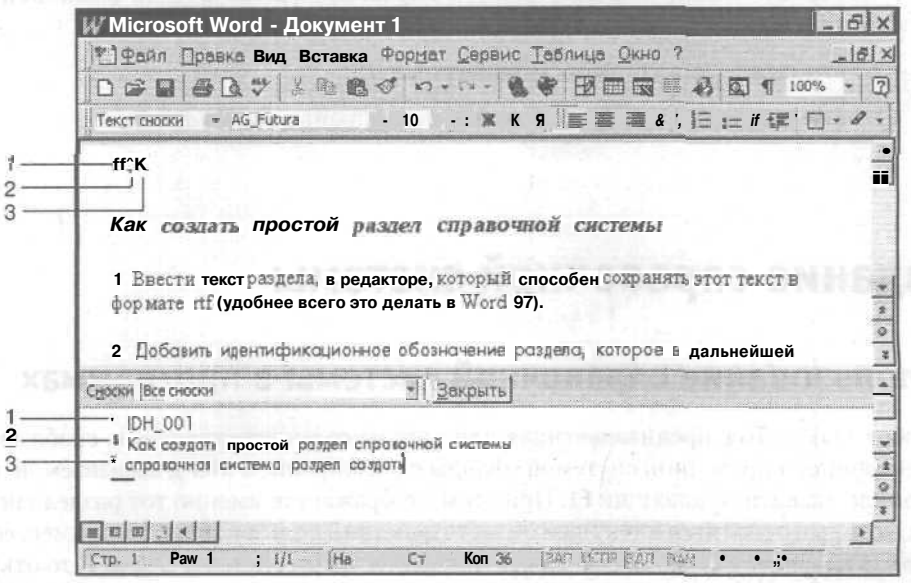


Рис. 12.12. *Разметка документа, содержащего раздел справочной системы: 1 — идентификация раздела; 2 — заголовок раздела; 3 — указатель ключевых слов*

4. При необходимости можно добавить дополнительные управляющие последовательности символов, предназначенные, например, для размещения ключевых слов, положения раздела в окне выбора, определения вида окна справочной системы и т. д. Эти последовательности описаны в справочном файле HCW.HLP, расположенном в каталоге Delphi 7\Help\Tools.

   Например, введем несколько ключевых слов для отображения на вкладке Указатель диалогового окна справочной системы. Для этого в режиме сноски добавим в начало раздела латинскую букву K, за которой должно идти ключевое слово или слова (рис. 12.12).

5. В один файл можно добавить любое число разделов. Каждый раздел отделяется от других разрывом страницы.

6. Файл сохраняется в формате .RTF.

Как указать ссылку на раздел

Справочная система организована по принципу гиперссылок. Выделенные слова (по умолчанию используется зеленый цвет) являются ссылками, и при щелчке на них выполняется переход к другому разделу системы.

Допустим, в файле имеются два раздела с идентификаторами IDH_TOPIC001 и IDH_TOPIC002. Чтобы сделать фрагмент текста гиперссылкой, используется следующий порядок действий.

1. После окончания фрагмента надо немедленно (без пробела) ввести идентификатор раздела, на который должен быть совершен переход:

   Здесь переходIDH_TOPIC002

2. Необходимо выделить слово (или слова), входящие в ссылку, и задать режим подчеркивания шрифта. Двойное подчеркивание означает, что после перехода новое содержимое надо открыть в текущем окне справочной системы:

   Здесь переходIDH_TOPIC002

   В случае одинарного подчеркивания текст раздела отображается во всплывающем окне:

   Здесь переходIDH_TOPIC002

3. Надо выделить идентификатор (IDH_TOPIC002) и сделать соответствующий шрифт скрытым:

   Здесь переходIDH_TOPIC002

Создание файла проекта

1. Для создания файла справочной системы необходимо вызвать утилиту *Microsoft Help Workshop*. Она расположена в каталоге Delphi 7\Help\Tools и называется hcw.exe (рис. 12.13).

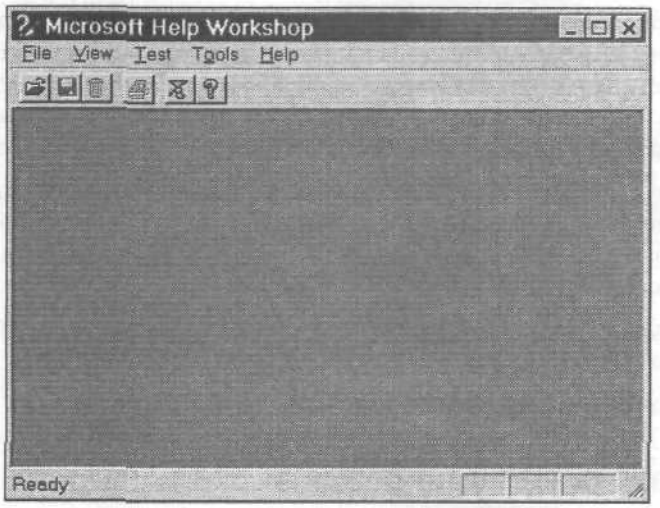


Рис. 12.13. Окно утилиты для работы с файлами справочной системы

2. Выполните команду File ➤ New (Файл ➤ Создать). В открывшемся диалоговом окне выберите пункт Help Project (Проект справочной системы) и щелкните на кнопке OK (рис. 12.14).

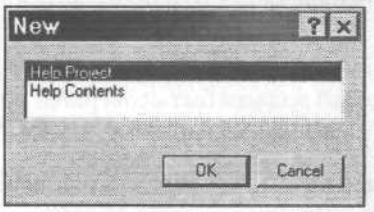


Рис. 12,14. Создание проекта справочной системы

3. В окне выбора файлов указывается (или создается) каталог и задается имя файла проекта с расширением .HPJ (например, primer.hpj). Щелкните на кнопке OK. С помощью кнопки Options (Параметры) можно ввести заголовок файла подсказки и указать идентификатор раздела, который будет показываться по умолчанию, если не найден раздел, запрошенный пользователем. На вкладке Sorting (Сортировка) в раскрывающемся списке надо указать язык справочной системы (Русский).

4. На вкладке Files (Файлы) указываются следующие сведения:

   - в поле Help File (Файл справки) задается имя будущего файла справки;
   - в поле RTF Files (Файлы RTF) вводятся имена файлов .RTF с описанием структуры справочной системы.

5. Окно Options (Параметры) можно закрыть. Теперь надо щелкнуть на кнопке Map (Соответствие), чтобы открыть одноименное окно. В нем с помощью кнопки Add

(Добавить) задаются соответствия между идентификатором раздела и его числовым эквивалентом (целым числом, начиная с единицы). Это требуется для обращения к разделу по его номеру, а не по идентификатору. Следует указать соответствие для каждого идентификатора, который используется в программе для контекстного вызова справочной системы. В разделе Map (Соответствие) проекта появится список соответствий (рис. 12.15).

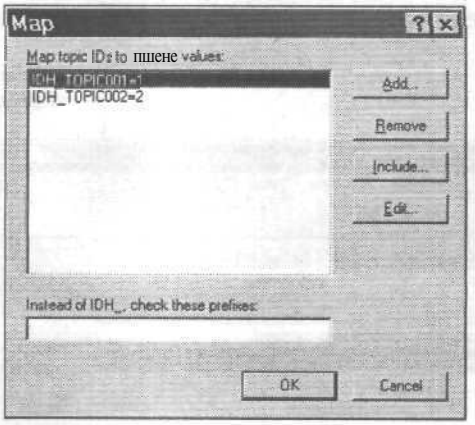


Рис. 12.15. *Список соответствий между идентификаторами разделов и их номерами, заданных в справочной системе*

6. Сохраните и закройте файл проекта.

Как подготовить содержимое справочной системы

Проект справочной системы создан, теперь надо подготовить список входящих в него разделов. Это выполняется командой File > New > Help Contents (Файл ➤ Создать > Содержание справочной системы).

В поле Default filename (Имя файла по умолчанию) указывается имя файла .HLP, используемое по умолчанию. В иоле Default title (Заголовок по умолчанию) задается заголовок первого диалогового окна справочной системы. Чтобы добавить новый раздел в содержание, надо щелкнуть на кнопке Add Above или Add Below (Добавить выше/ниже текущего раздела), после чего в диалоговом окне Edit Contents (Изменить содержание) следует установить переключатель Topic (Раздел).

При этом в поле Title (Заголовок) указывается имя раздела, под которым он фигурирует в справочной системе, в поле Topic ID (Идентификатор темы) — идентификатор раздела, в поле Help file (Файл справки) — имя файла .HLP, в котором хранится этот раздел.

Процесс добавления раздела должен быть согласован с процессом записи заголовков — аналогов папок, которые обычно представлены значком в виде закрытой книги. Эти заголовки разворачиваются по двойному щелчку и показывают вложенный список разделов. Может быть до девяти уровней вложенности.

Заголовок добавляется с помощью тех же кнопок, только устанавливается переключатель Heading (Заголовок), а другие поля не заполняются. В режиме формирования раздела такой заголовок представляется в виде раскрытой книги.

Чтобы добавить раздел «внутрь» заголовка (папки), используются кнопки Move Left (Влево) и Move Right (Вправо). Они перемещают текущий раздел вместе с имеющимися подразделами на один уровень вложенности влево или вправо (рис. 12.16). Созданная структура содержания сохраняется в файле с расширением .CNT.

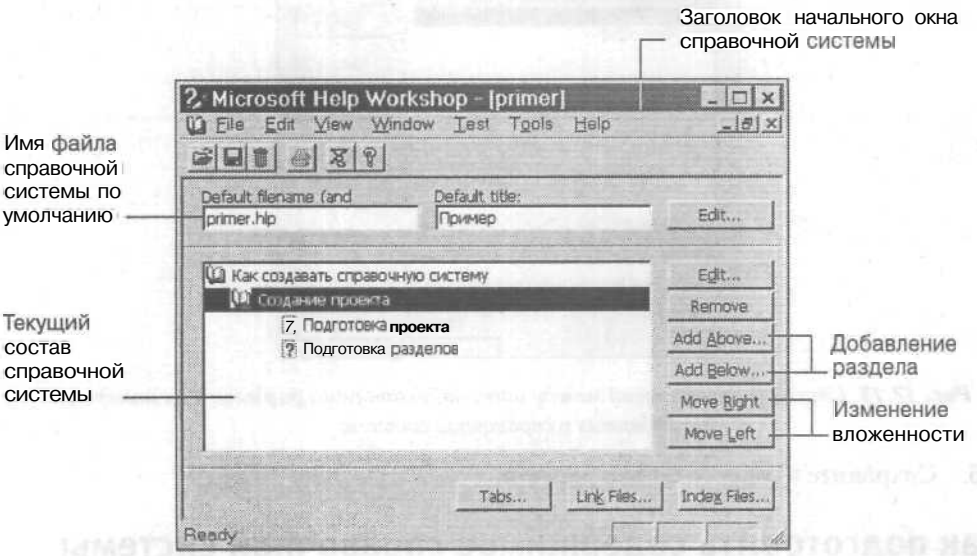


Рис. 12.16. *Формирование содержания справочной системы*

Создание справочного файла

В заключение загружается файл проекта и на вкладке Files (Файлы) в окне Options (Параметры) файл .CNT, созданный на предыдущем этапе, указывается в поле Contents file (Файл описания содержания). Далее надо закрыть окно настроек и щелкнуть на кнопке Compile (Компиляция). Откроется диалоговое окно настроек режима компиляции. Здесь надо обязательно установить флажок Include .rtf filename and topic ID in Help file (Включить имя файла RTF и номера разделов в файл справки), который потребует от компилятора включения в создаваемый файл .HLP не только идентификаторов IDH_, но и соответствующих им номеров разделов.

После компиляции выводится информация о числе разделов, гиперссылок, ключевых слов и добавленных в файл справки рисунков. В каталоге, где расположен файл проекта .HPJ, появится новый файл .HLP, который теперь можно использовать в своих программах. Желательно, чтобы в строке, где указывается число замечаний *(notes)* и предупреждений *(warnings),* стояли нули. В противном случае корректная работа файла-справки не гарантируется.

Как добавить справочный файл в программу

Созданный файл .HLP надо скопировать в каталог, в котором хранится проект разрабатываемого приложения. Затем, загрузив этот проект в систему *Delphi 7,* надо дать команду Project ➤ Options (Проект > Параметры) и на вкладке Application (Приложение) выбрать этот файл в поле Help file (Файл справки) с помощью кнопки Browse (Обзор).

Далее на форме можно разместить произвольные компоненты, а в их свойствах HelpContext указать подходящие номера разделов, связанные с их идентификаторами. Эти номера должны быть больше нуля. Если запустить программу и выбрать любой элемент управления, у которого имеется такая ссылка на раздел файла-справки, то по нажатию клавиши F1 отображается нужный раздел.

ЗАМЕЧАНИЕ Отдельные файлы справки могут быть созданы для каждой из форм.

Перспективы развития справочной системы

Рассмотренная нами справочная система *Windows* уже давно стала стандартом, однако с активным распространением Интернета и появлением языка описания гипертекста *HTML* многие компании распространяют справочные системы в новом формате, позволяющем просматривать информацию из браузера. Уже сегодня многие продукты *Microsoft* имеют справочную систему, написанную именно на *HTML,* и можно предположить, что в будущих версиях системы *Delphi* появится возможность автоматического вызова браузера и загрузки нужной страницы *HTML.* А пока такой режим работы несложно реализовать самостоятельно.

Поддержка групповой работы. Система TeamSource

Принципы организации групп программистов

Когда над проектом трудятся несколько человек, необходимо обеспечить работу каждого из них над корректным вариантом исходного текста. Ведь не исключено, что два программиста возьмутся править один и тот же файл с исходными текстами или перепроектировать одну и ту же форму. Результат непредсказуем, а ошибки возникнут наверняка. Поэтому при групповой работе необходимо применять средства контроля версий кода, которые автоматически фиксируют все изменения в исходных текстах.

Модель Team Source

Модель *Team Source* — это средство управления потоками информации между разработчиками, участвующими в проекте по созданию программной системы на

основе *Delphi 7* (или других продуктов *Borland,* например *C++Builder*). Оно входит в стандартную поставку *Delphi 7.*

Механизм *Team Source* поддерживает систему контроля версий кода (она позволяет работать с известным продуктом *PVCS* фирмы *Intersolv* и собственной более простой системой *ZLib).* Однако механизм *Team Source* обладает более широкими возможностями, чем простой контроль версий файлов. Он базируется на *параллельной модели* управления исходными версиями проектов. Идея ее в том, чтобы в любой момент времени каждый из разработчиков мог работать с любым файлом проекта (точнее, с его локальной копией) и вносить в него произвольные изменения.

Когда редактирование локальной копии заканчивается, изменения должны быть согласованы с основной *мастер-версией* файла. Эти изменения должны быть корректно в нее внесены, а в локальную версию программиста в свою очередь добавляются изменения, которые обнаружены в мастер-версии со времени последнего обращения к ней данного программиста. Процесс синхронизации локальной и мастер-копий полностью контролируется разработчиком.

ЗАМЕЧАНИЕ Помимо параллельной, существует еще *последовательная* модель управления, которая отличается тем, что в любой момент времени с каждым файлом проекта может работать только один человек. Преимущество этой модели в том, что нет необходимости выполнять достаточно сложный процесс согласования двух копий измененного файла, недостатки — в том, что редактируемый одним разработчиком файл становится недоступным всем другим участникам проекта.

Задачи, решаемые системой *Team Source,* делятся на две большие группы. Первая группа — это пользовательские задачи, вторая — задачи администрирования.

Пользовательские задачи Team Source

В проект *Team Source* может входить несколько групповых проектов *Delphi 7.* Для него определяются группы пользователей с тремя видами доступа к исходным файлам: только на просмотр, на просмотр и редактирование, административный. Расширение имени файла такого проекта — `.CPJ`.

Проект *Team Source* обычно записывается в сетевой каталог, доступный каждому разработчику. Конечно, систему *Team Source* можно и нужно применять и при индивидуальной разработке. Возможность контроля своей работы и анализа выполненных объемов программирования очень полезна при оценке объемов новых проектов, а такая потребность у профессиональных разработчиков возникает постоянно.

Каталог системы содержит три вложенных каталога.

- В каталоге `\Admin` расположены все архивные файлы, связанные с контролем версий, а также файл проекта `.CPJ`.

- В каталоге \Source хранятся версии наиболее часто используемых файлов в режиме «только чтение».
- В каталоге \History находятся протоколы выполненных синхронизации файлов.

Запуск системы Team Source

Запуск системы *Team Source* выполняется командой Team Source из главного меню системы *Windows*. При первом запуске запрашивается служебный идентификатор пользователя (по умолчанию Guest), полное имя пользователя и его электронный адрес. После этого система предлагает либо создать новый проект (тогда пользователь автоматически становится его администратором), либо подключиться к уже существующему проекту. В последнем случае администратор существующего проекта должен заранее сформировать права доступа для нового пользователя (рис. 12.17).

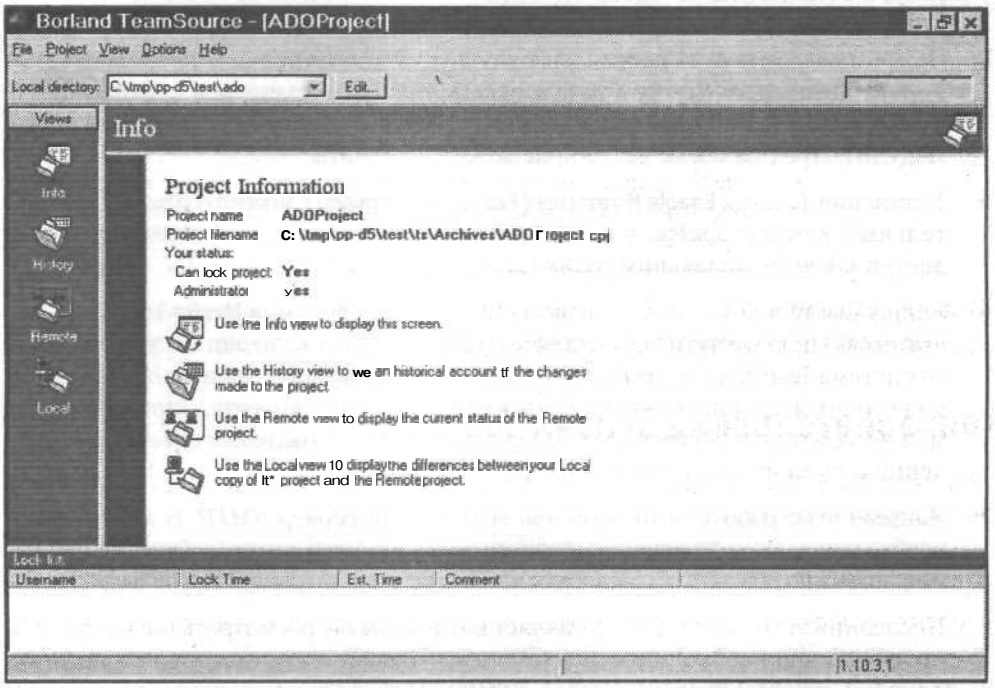


Рис. 12.17. *Информация о групповом проекте в системе TeamSource*

ЗАМЕЧАНИЕ Здесь и далее под понятием «проект» подразумевается проект Team Source, если особо не оговорено что-либо иное.

Создание нового проекта

После выбора пункта Create new project from scratch (Пошаговое создание нового проекта) щелкните на кнопке OK.

1. Появляется диалоговое окно первого (из восьми) этапов работы Мастера создания проекта. В первом текстовом поле указывается название будущего проекта, например ADOProject. В нижнем поле появится то же название с расширением .CPJ. Менять его не следует. Щелкните на кнопке Next (Далее).
2. Если несколько проектов используют общий сетевой каталог, то надо указать его имя. По умолчанию можно оставить имя root. В раскрывающемся списке Version controller for project files (Контроль версий файлов проекта) выбирается система контроля версий исходных тестов. Система *Team Source* способна работать с двумя такими системами: *PVCS* (она поставляется отдельно от системы *Delphi 7* и доступна, если установлена на компьютере) и *Borland.ZLib*.
3. Укажите полный путь поиска для каталога, в котором будет храниться проект. Выбрать его можно с помощью кнопки обзора.
4. Далее определите пути поиска для будущих каталогов проекта: корневого каталога Root, каталога History и каталога Lock, где будет храниться информация о выполненных блокировках проекта (на время внесения изменений в его модули). Предлагаемые настройки можно не менять.
5. Установив флажок Enable Hirrortree (Разрешить зеркало), можно создать дополнительный каталог Source, в котором будет храниться копия дерева проекта, доступная всем желающим только для просмотра.
6. Если указать в полях Master Summary file (Файл сводки) или Master Log file (Файл протокола) полные пути поиска для этих файлов (даже если они не существуют), то система Team Source создаст эти файлы, которые будут хранить в текстовом виде полный протокол своей работы в рамках данного проекта. Дополнительно можно задать адрес сервера *SMTP* для отправки пользователям проекта уведомлений о внесенных в него изменениях.
7. Данный этап работы выполняется, если указан сервер *SMTP*. В этом случае необходимо задать электронные адреса пользователей, которые будут получать информацию обо всех изменениях в проекте.
8. Последний этап работы предоставляет возможность просмотреть все настройки проекта и при необходимости вернуться к одному из предыдущих пунктов с помощью кнопки Previous (Назад). Завершается создание проекта щелчком на кнопке Finish (Готово).

Первое, на что сразу же укажет система, — это отсутствие локальных каталогов и необходимость их создания. На заданный вопрос необходимо ответить положительно, потому что без локальных каталогов, содержащих пользовательские копии всех рабочих файлов проекта *Delphi,* функционирование системы *Team Source* не

имеет смысла. Локальный каталог добавляется щелчком на кнопке Add (Добавить) в окне редактирования локальных каталогов, которое открывается по щелчку на кнопке Edit (Изменить) рядом со списком Local directory (Локальный каталог) слева в верхней части окна.

Рассмотрим пример подключения к системе *Team Source* проекта *Delphi 7* по работе с компонентами ADO, На первом этапе надо указать каталог, в котором хранится соответствующий проект *Delphi* и все принадлежащие ему файлы (исходные тексты, файлы форм и пр.).

На основании результатов просмотра содержимого этого каталога система *Team Source* запросит разрешение на автоматический анализ структуры проекта. Ответить надо утвердительно — Yes (Да), после чего потребуется утвердить предлагаемый набор шаблонов файлов, которые подвергнутся контролю (рис. 12.18).

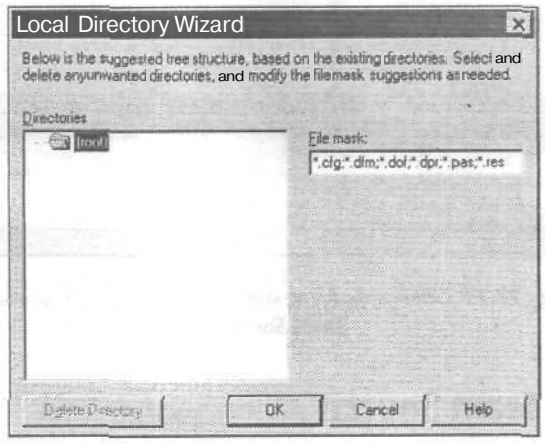


Рис. 12.18. *Выбор файлов и каталогов, контролируемых системой Team Source*

В результате будет сформирован новый проект *Team Source.* Теперь надо щелкнуть накнопкеLocal(Локальный),чтобыперейтивлокальныйкаталогпроекта.Приэтом система *Team Source* автоматически просканирует содержимое каталога с проектом *Delphi* и отобразит наэкране список всех файлов, подлежащих контролю (рис. 12.19).

Создание копии проекта

Когда к проекту подключаются новые пользователи, для них требуется создать новые локальные копии проекта. Это выполняется за один шаг с помощью команды Project x Pull to (Проект ➤ Передать новому пользователю). Предварительно надо создать новый локальный каталог с помощью кнопки Edit (Изменить) в главном окне *Team Source* и при выполнении команды Pull to (Передать новому пользователю) выбрать нужный каталог в раскрывающемся списке.

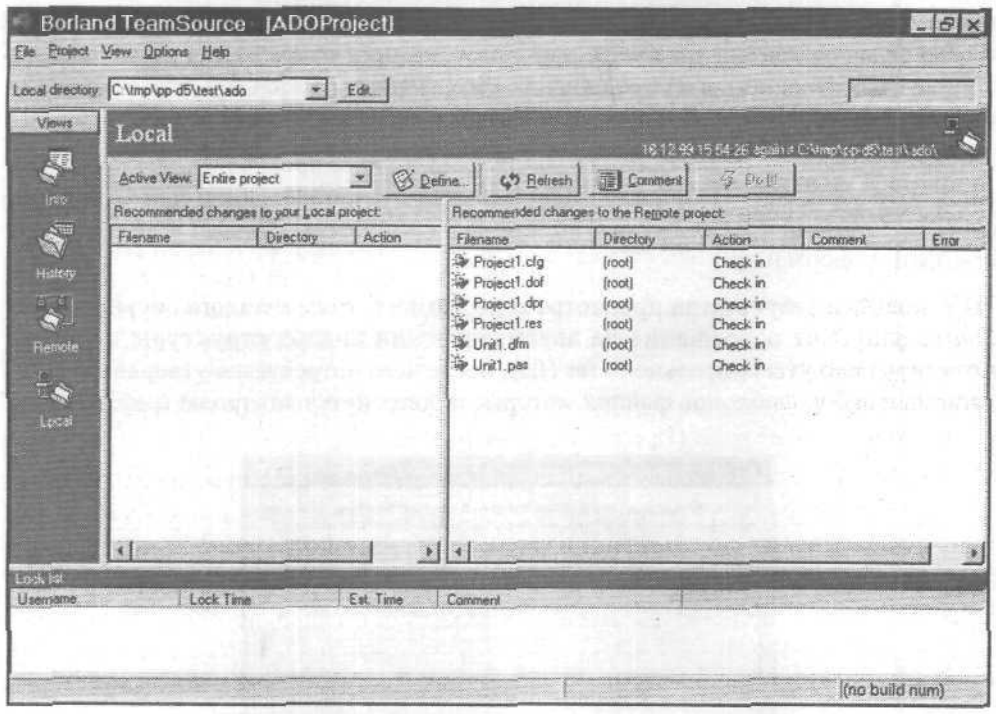


Рис. *12.19.* *Список файлов, контролируемых системой Team Source*

Главное окно Team Source

Локальные каталоги

В каждый проект *Delphi* может входить несколько физических каталогов, хранящих различные части проекта. Такие каталоги в системе *Team Source* называются *локальными*. Текущий локальный каталог задается в раскрывающемся списке Local directory (Локальный каталог) главного окна.

В этот список можно добавить новые каталоги, изменить или удалить существующие с помощью кнопки Edit (Изменить).

В левой части окна расположены четыре кнопки.

- Кнопка Info (Сведения) позволяет получить полную информацию о текущем проекте и статусе текущего пользователя.
- Кнопка History (Журнал) отображает журнал редактирования проекта. В левой части центрального окна отображается дата и время внесения очередного изменения, имя лица, вносившего правку, а при выборе одной из записей детальное описание модификации выводится в правой части окна.

- Кнопка Remote (Сетевые сведения) позволяет увидеть структуру проекта с учетом расположения различных каталогов в сети. Предоставляется возможность навигации по структуре сетевых каталогов проекта с отображением файлов, хранящихся в соответствующих папках. В столбце Version (Версия) указывается последняя версия данного файла. По умолчанию нумерация версий в системе *Team Source* начинается с версии 1.0, а затем вторая цифра увеличивается на единицу после каждого обновления мастер-версии любым пользователем. В столбце Date (Дата) указывается дата последнего изменения. Для просмотра полного списка всех изменений (всех версий файла) выбирают пункт контекстного меню View Any Revision (Просмотр всех редакций). При желании можно вернуться к любой из ранних версий, сравнить две версии с помощью пункта Compare Revisions (Сравнить редакции) или просмотреть отчет о модификациях файла (кто, когда, зачем) с помощью пункта View Archive Report (Просмотр архивного отчета).
- Кнопка Local (Локальные данные) отображает информацию о локальных версиях файлов конкретного пользователя и рекомендации о необходимости выполнения изменений.

В нижней части окна Team Source выводится информация о текущей активности других участников проекта. При внесении изменений в проект *Team Source* определенные файлы проекта *блокируются,* что отображается в списке значками.

Таблица 12.7. *Состояние контролируемых файлов проекта*

Значок	Состояние
	Пользователь ожидает возможности блокирования проекта для внесения в него изменений
	Пользователь заблокировал часть проекта
	Пользователь подтверждает блокирование проекта

Запрос на блокировку проекта

Абсолютное большинство вносимых в проект изменений требует его предварительной блокировки, чтобы запретить доступ другим пользователям. Запрос на блокировку формируется следующим образом.

Щелкните правой кнопкой мыши в нижней части окна Team Source — на панели Lock list (Список блокировок) — и выберите в контекстном меню пункт Requiest Lock (Запрос на блокировку). В открывшемся диалоговом окне в поле Lock Comment (Причина блокировки) введите краткое описание причины запроса на блокировку (например, Добавление нового пользователя). В поле Estimated time you will need the lock (Примерное время блокировки) укажите примерное время, которое потребуется на эту блокировку (можно указать 5 минут), и установите флажок Lock as Administrator Lock (Административная блокировка) (рис. 12.20). В списке блокировок появится новая строка (рис. 12.21).

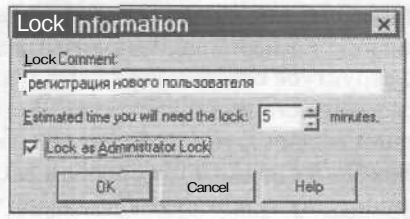


Рис. 12.20. Формированиезапроса на блокировку

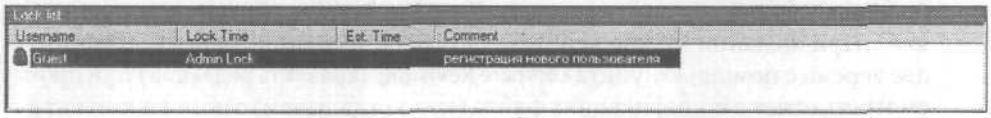


Рис. 12.21. Список действующих блокировок

Для пользователя, сформировавшего запрос с правами администратора, запрошенное время в графе Est.Time (Примерное время) не отображается. Чтобы указать конкретный период времени, надо выделить строку с запросом и в контекстном меню выбрать пункт Extend Lock (Продлить блокировку), с помощью которого можно задать определенный промежуток времени. Для обычного пользователя обратный отсчет запрошенного времени начнется автоматически.

Чтобы снять блокировку с проекта, надо выбрать в контекстном меню пункт Clear lock (Снять блокировку).

Регистрация нового пользователя

Перед регистрацией нового пользователя необходимо сформировать запрос на блокировку, в противном случае соответствующие возможности системы будут доступны в режиме «только чтение».

Выполните команду Project > Options (Проект ➤ Параметры) и в открывшемся диалоговом окне на вкладке Users (Пользователи) щелкните на кнопке Add User (Добавить пользователя). В открывшемся окне введите имя нового пользователя и укажите его права: либо только просмотр проекта — переключатель Read-Only (Только чтение), либо просмотр и редактирование — переключатель Read-Write (Чтение и запись). В последнем случае дополнительно предоставить пользователю права администратора позволяет флажок User is an Administrator (Пользователь-администратор), который по умолчанию сброшен. В дальнейшем с помощью кнопки Edit User (Изменить пользователя) можно изменить права любого пользователя, а с помощью кнопки Delete User (Удалить пользователя) — удалить любого пользователя (за исключением себя самого).

Просмотр файлов, нуждающихся в проверке

После того как проект создан, в окне Local (Локальные) отображаются версии файлов, нуждающихся в процедуре синхронизации с мастер-версиями, расположен-

ными в общем сетевом каталоге \Root. Первоначально все файлы, включенные в проект, отображаются в правой части окна Local. Это означает, что в данных файлах система *Team Source* зарегистрировала изменения и теперь рекомендуется внести их и в мастер-копии (поначалу в каталоге \Root никаких файлов нет).

ЗАМЕЧАНИЕ Процедура сверки локальных версий с мастер-версиями и внесения изменений называется регистрацией (Check In). Эта процедура, как правило, имеет одинаковое название в большинстве систем контроля версий, в том числе и в системе PVCS.

Локальные версии файлов выделяются. Их можно выделить сразу все с помощью команды контекстного меню SelectAll (Выбрать все). Затем для выделенных файлов выполняется процедура Check In (Регистрация) путем выбора одноименного пункта контекстного меню. Перед данной операцией необходимо сформировать запрос на блокировку.

Далее система запросит дополнительное подтверждение выполнения операции сверки для выделенных файлов. Надо ответить Yes (Да). По окончании операции сообщается число реально обработанных файлов и количество ошибок, выявленных при синхронизации информации. Если теперь перейти в окно Remote (Сетевые сведения), то в каталоге Root обнаружится полный список общедоступных файлов.

Как работает система Team Source

Работа системы *Team Source* основана на синхронизации версий файлов, расположенных в разных локальных каталогах. Когда разработчик вносит изменения в свои локальные файлы, наличие изменений отображается в правой части окна Local (Локальные). Выполнить синхронизацию можно с помощью команды Check In (Регистрация) контекстного меню. Это допускается, если файлы в удаленном каталоге \Root можно заменить новыми версиями.

В локальные копии одного и того же файла разные программисты могут вносить несогласованные изменения. Один человек может исправить некоторые функции в исходном тексте, а другой — удалить их. Пусть один из разработчиков выполнил операцию Check In (Регистрация). Когда второй также решит синхронизировать свои файлы с мастер-версиям и, ему для получения списка локальных файлов, нуждающихся в синхронизации, надо щелкнуть на кнопке Refresh (Обновить) в окне Local (Локальные). При этом окажется, что в мастер-версии внесены такие изменения, которые не допускают простого (автоматического) обновления.

Предположим, что программист пытается синхронизировать описание некоторой процедуры с мастер-описанием, в котором эта процедура вообще отсутствует или подверглась серьезным изменениям. Тогда список файлов появится не в правой, а в левой части окна. Это означает, что в данный момент внести корректировку в мастер-версию невозможно и надо прежде всего выполнить синхронизацию мастер-кода с собственной локальной версией. В зависимости от значимости сделанных изменений синхронизация может выполняться разными путями, например прос-

тым копированием мастер-копии или автоматическим слиянием (такая возможность, называемая обычно Check Out, в системе контроля версий *Borland.ZLib* отсутствует), ручной корректировкой. О последнем способе сигнализирует строка Corect by hand (Исправить вручную) в столбце Action (Действие) (рис. 12.22).

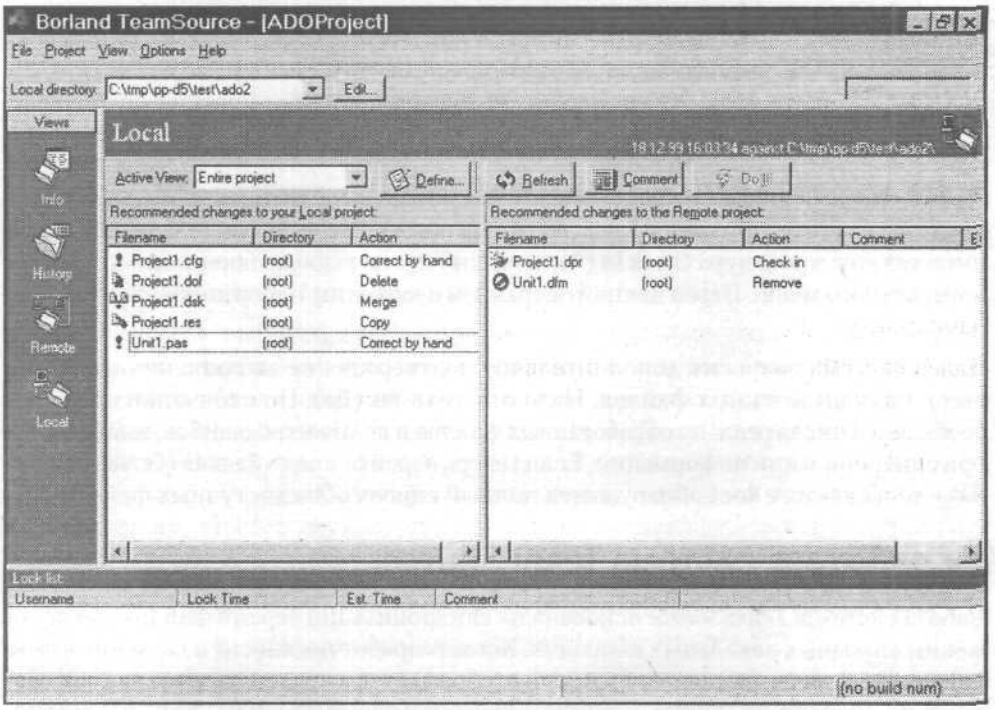


Рис. 12.22. *Сведения о нарушении синхронизации версий и рекомендации по ее восстановлению*

Выяснить, какие именно отличия внесены в файл, можно, выбрав в контекстном меню мастер-версии пункт View Remote Changes (Просмотреть изменения в удаленном файле). При этом различия между файлами отображаются в новом окне. Желтым цветом и символами «+» выделяются новые, добавленные строки. Удаленные строки помечаются красным цветом и символом «–».

ЗАМЕЧАНИЕ

С помощью пункта меню View Local Changes (Просмотреть изменения в локальном файле) можно узнать, какие изменения внесены в локальную версию по сравнению с версией, которая в последний раз отправлялась в каталог \Root с помощью процедуры Check In (Регистрация).

Для просмотра текста, записанного в мастер-версию, надо перейти в окно Remote (Сетевые сведения), выбрать в списке нужный файл и в контекстном меню указать пункт View Tip Revision (Просмотр последней версии). Подобный процесс синхрони-

зации локальных версий с мастер-версиями работает в обе стороны и поддерживается системой *Team Source.*

Примечания к изменениям

Чтобы в процессе работы над проектом *Delphi* можно было в любой момент детально проанализировать эффективность труда каждого программиста, определить наиболее узкие участки в работе, все вносимые в мастер-версии изменения надо обязательно комментировать, причем чем подробнее, тем лучше. Содержание примечаний обычно определяется руководителем проекта в организационном порядке, но чтобы разработчики не забывали их вводить, администратору лучше явно задать необходимость ввода примечаний. Это делается обращением к окну задания свойств проекта командой Project ➤ Options (Проект ➤ Параметры) и выбором вкладки General (Общие). Предварительно надо сформировать запрос на блокировку.

На соответствующей панели устанавливаются флажки Require summary comments (Требовать сводные примечания) и Require file comments (Требовать примечания к файлам). Первый из них обязывает указывать общее примечание к каждой процедуре Check In (Регистрация) с помощью кнопки Comment (Примечание). Второй флажок требует давать примечания к локальному файлу выбором пункта Edit File Comment (Изменить примечание к файлу) в контекстном меню. Теперь без предварительного указания общих и «локальных» примечаний процедура Check In (Регистрация) не выполняется (рис. 12.23).

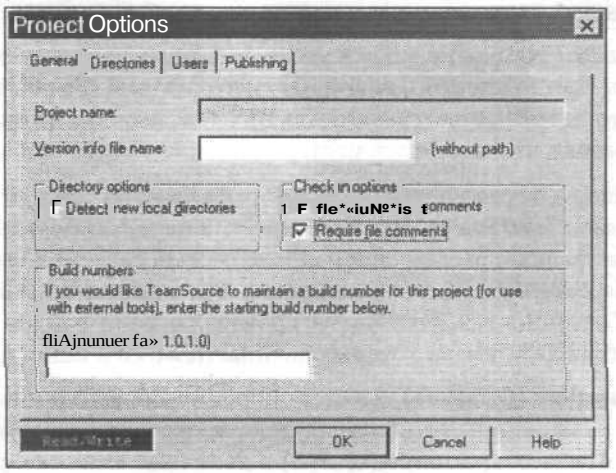


Рис. 12.23. *Настройка правил задания примечаний для процедуры регистрации изменений*

Что включать в анализ

Подвергать контролю все файлы, входящие в проект *Delphi,* не имеет смысла. Среди них имеются файлы локальных настроек среды *Delphi,* локальные версии проектов

Delphi и др. (например, двоичные файлы ресурсов .RES), не нуждающиеся в контроле. Администратор проекта может удалить любой мастер-файл в окне Remote (Сетевые сведения), выбрав в контекстном меню пункт Remove from project (Удалить из проекта). Наиболее целесообразно подвергать контролю файлы исходных текстов. В обычном текстовом формате в проекте *Delphi* хранятся файлы со следующими расширениями:

- .PAS — исходный текст программы на Паскале;
- .DFM — описание формы;
- .DPR — исходный текст главного файла проекта (на Паскале);
- .DOF — основные настройки системы *Delphi* при создании приложения;
- .CFG — настройки компилятора и набор конфигурационных параметров.

Их и следует контролировать с помощью системы *Team Source*.

Однако, хотя подобным образом можно контролировать все исходные тексты, результирующий исполнимый *EXE*-файл не будет сформирован или останется прежним, если программист после корректировки текстов не выполнил компиляцию или в ней возникли ошибки. Поэтому система *Team Source* позволяет указывать связи между файлами проекта, на основании которых создаются другие файлы. Например, на основании файлов .PAS создаются двоичные модули .DCU. Для этого в окне Remote (Сетевые сведения) в контекстном меню для каталога \Root надо выбрать пункт Properties (Свойства).

В открывшемся диалоговом окне указываются следующие данные.

- В поле Includes (Включая) задаются маски файлов, контролируемых системой Team Source. При добавлении новой маски в окне Local (Локальные) появятся файлы, которые ей соответствуют. Для них надо выполнить операцию Check In (Регистрация), чтобы создать мастер-версии.
- В поле Excludes (Исключая) задаются маски файлов, которые не контролируются системой Team Source. Это полезно, если надо освободить от контроля конкретные файлы с расширением, проверяемым системой Team Source. Например, для контролируемого расширения .DCU в данном поле можно указать test*.dcu. Тогда система *Team Source,* проверяя версии файлов .DCU, пропустит все файлы .DCU, имена которых начинаются с сочетания *test*.
- В поле Productions (Создание) задается список преобразований одних форматов в другие. Например, для контроля файлов, преобразованных из исходных текстов на Паскале (расширение .PAS) в двоичные модули с расширением .DCU, надо указать строку

  ```
  .pas->.dcu;
  ```

  Здесь комбинация символов -> обозначает соответствующее преобразование.

Чтобы изменить статус файлов, которые требуют модификации в соответствии с измененными мастер-версиями (правая панель) надо выделить соответствующие

файлы и выбрать в локальном меню пункт Change File Status (Изменить статус файлов). **В диалоговом** окне появится следующий набор состояний (статусов), которые можно присвоить выделенным файлам.

- Copy (Remote -> Local). Копировать мастер-версию в локальный каталог.
- Check In (Local -> Remote). Выполнить процедуру Check In (Регистрация).
- Delete (from Local). Удалить локальную версию файла.
- Merge. Выполнить слияние файлов в соответствии с возможностями текущей системы контроля версий.
- Remove (from project). Удалить файл из проекта.

Эти состояния не приводят к реальному выполнению соответствующих действий. Они только изменяют статус соответствующего файла, чтобы пользователь мог визуально определить, что с ним надо сделать. А конкретные действия выполняются либо двойным щелчком на имени выбранного файла, либо выбором подходящей операции из контекстного меню.

Закладки

Системы конфигурационного управления обычно позволяют создавать так называемые *слепки* проекта. В терминологии *Team Source* они называются *закладками* (Bookmarks). Сформировав слепок проекта (его статический образ на определенный момент времени), в дальнейшем можно вернуться к состоянию на момент снятия слепка.

Зачем это надо? Во время работы над крупным проектом нередки случаи, когда работа заходит «не туда». Может оказаться, что последние, возможно весьма объемные и существенные, изменения не только не улучшили работу приложения, а даже ухудшили ее и привели к возникновению неожиданных ошибок. В такой ситуации правильнее всего вернуться к тому этапу работы над проектом, когда эти изменения еще не были внесены.

Создавать слепки проекта можно, например, для очередной альфа- или бета-версии продукта. Но ведь для того, чтобы вернуться в прошлое, требуется восстановить множество исходных файлов, локальных и мастер-версий. Автоматизировать этот процесс и помогает технология закладок *Team Source*.

Чтобы создать слепок текущего проекта, надо выполнить следующие действия.

1. Сформируйте запрос на блокировку.
2. Дайте команду Project ➤ Bookmarks (Проект > Закладки).
3. В диалоговом окне Bookmarks (Закладки) щелкните на кнопке Add (Добавить).
4. В поле Name (Имя) введите имя закладки, например Бета 0.99.
5. В списке Date (Дата) выберите любую из доступных дат выполнения процедуры синхронизации мастер-версии Check In (Регистрация).

6. С помощью переключателя Scope (Охват) укажите режим создания закладки: Local (Локальная закладка) или Global (Глобальная закладка). Последний вариант может создаваться только администратором.
7. Щелкните на кнопке OK.

В дальнейшем, обратившись к диалоговому окну Bookmarks (Закладки), можно редактировать или удалять существующие закладки. Командой Pull to (Передать) в списке Bookmark (Закладка) проект возвращается в состояние, соответствующее избранной закладке.

Заключение

Реализация в рамках системы *Delphi 7* механизма управления групповой работой и версиями кода — очень важное и полезное решение, без которого немыслимо эффективное создание крупномасштабных приложений. Кроме того, научившись работать с такой системой, любой программист приобретет очень ценный опыт контроля производительности собственного труда и планирования своей работы.

Локализация приложений

Общие принципы локализации

Если создается программа, которая должна поддерживать пользовательский интерфейс на нескольких языках (например, на русском и английском), то при ее написании надо придерживаться определенных правил. Эти правила позволят в дальнейшем легко перенести интерфейс на любую языковую платформу. При создании такого приложения надо обязательно проверить его работу во всех версиях *Windows*, поддерживающих соответствующие национальные языки (в нашем случае — в панъевропейской и русифицированной версиях *Windows)*.

В некоторых случаях требования к переносу программы на язык конкретной страны не сводятся только к переводу сообщений и названий. Может потребоваться также активное использование свойства BiDiMode, которое определяет направление вывода текста (слева направо или справа налево), а также внесение изменений в функциональные блоки программы, что характерно для всевозможных финансовых приложений. Такие изменения могут быть мелкими, вроде выбора способа представления десятичной запятой, или крупными, типа адаптации под требования национального законодательства. Эти вопросы в данной книге не рассматриваются.

Главная задача локализации приложения — выделение и перевод всех его текстовых ресурсов. В прежних версиях *Delphi* все строки обычно описывались с помощью ключевого слова resourcestring:

```
resourcestring
   STR_HELLO = 'Привет!' ;
   STR_ASK =  'Закрыть?';
```

Затем они вместе с другими текстовыми ресурсами помещались в отдельную библиотеку .DLL, которая динамически загружалась во время работы программы. Достаточно было только заменить эту библиотеку, чтобы изменить язык интерфейса программы, однако такая работа довольно трудоемка.

В системе *Delphi 7* имеется встроенная возможность создания программ, поддерживающих разные национальные языки. При этом от разработчика не требуется писать ни одной строчки кода.

Локализация в Delphi 7

Подготовка пример**а**

В системе *Delphi 7* имеется так называемая Интегрированная оболочка перевода *(Integrated Translation Environment, ITE)*, которая автоматически создает нужные библиотеки .DLL с ресурсами. Рассмотрим ее работу на следующем примере. Пусть имеется программа, которая по щелчку на кнопке выводит стандартное диалоговое окно со строкой, поясняющей, сколько байтов памяти эта программа использует.

К текстовым ресурсам надо отнести заголовок программы (Сколько памяти), название кнопки (Узнать память), а также внутреннюю константу Использовано памяти (рис. 12.24).

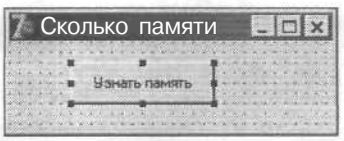


***Рис. 12.24.** Форма для программы, отображающей сведения об использовании памяти*

Исходный текст программы будет таким:

```
unit Unit1;
interface
uses
   Windows, Messages, SysUtils, Classes, Graphics,
Controls, Forms, Dialogs,
   StdCtrls;
const STR_ASK = 'Использовано памяти: ' ;
type
   TForm1 = class (TForm)
```

продолжение

```
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage(STR_ASK+IntToStr(AllocMemSize));
end;
end
```

Выбор языка

Перед тем как приступить к переводу текста, надо создать в проекте список поддерживаемых языков. Для этого проект надо сначала откомпилировать командой Build (Построить) и сохранить, а затем выполнить команду добавления нового языка Project ➤ Languages > Add (Проект ➤ Языки > Добавить). На экране появится диалоговое окно Мастера добавления национального языка (рис. 12.25).

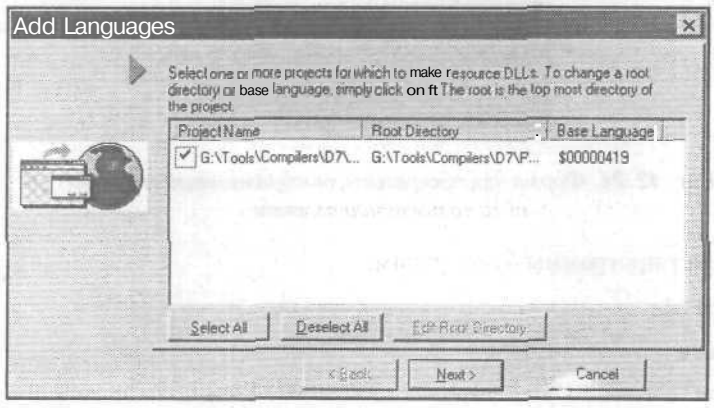


РИС. *12.25. Сведения о поддерживаемых языках в титульном окне Мастера добавления языков*

Исходно поддерживается только один язык с шестнадцатеричным кодом 419 (русский). На следующем этапе работы Мастера надо выбрать английский язык (шестнадцатеричный код 809). Русский язык, как базовый, помечать не надо. Далее на очередном этапе можно изменить путь поиска для каталога, в который записыва-

ется библиотека .DLL с ресурсами, и таким образом дойти до последнего этапа работы Мастера. По завершении работы он создает в каталоге проекта новый подкаталог епд, в котором сохраняются исходные тексты библиотеки ресурсов на английском языке.

После того как Мастер выполняет генерацию кода, он выводит окно с сообщением о процессе генерации и вызывает Менеджер перевода (Translation Manager).

Перевод текстов

На левой панели Менеджера перевода указывается иерархическая структура проекта с узлами, в которых находятся объекты со строками, подлежащими переводу. На правой панели указан список ресурсов текущего объекта, которые нуждаются в переводе или уже переведены.

В столбце Русский указывается оригинальное содержание константы, в столбце Английский — текст на английском языке (который пока отсутствует). Столбцы, заголовок которых начинается со слова Previous (Предыдущий), позволяют контролировать последние изменения, внесенные в перевод. Следующие столбцы Created (Создан) и Modified (Изменен) отображают даты создания и модификации переведенного содержимого. В последнем столбце Comment (Примечание) можно записать примечание к соответствующему ресурсу.

Переведем на английский язык две строки: 'Сколько памяти' и 'Узнать память', — после чего щелкнем на кнопке Save (Сохранить). При этом в столбце Status (Состояние) для этих строк сообщение Untranslated (Не переведено) изменится на Translated (Переведено).

Хранение переведенных строк

При создании программных проектов, поддерживающих несколько языков, довольно часто встречаются одни и те же выражения, которые желательно переводить единообразно. В этом поможет Склад переводов (Translation Repository), который вызывается командой Tools ➤ Translation Repository (Сервис > Склад переводов). Он способен хранить ранее созданные в Менеджере переводов варианты перевода и в дальнейшем автоматически подставлять соответствующие значения, если в ресурсах обнаруживается строка, обработанная ранее. Эти варианты перевода добавляются из Менеджера перевода выбором в контекстном меню переведенной строки пунктов Repository > Add strings to repository (Склад ➤ Добавить строки на склад). Если теперь щелкнуть на кнопке Launch repository (Запуск склада), то в списке вариантов перевода появится новая строка (ряс. 12.26).

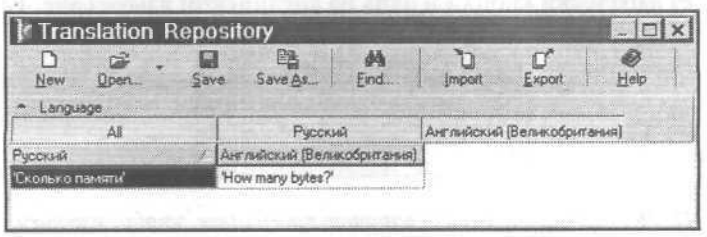


***Рис. 12.26.** Добавление стандартных вариантов перевода строк па склад переводов*

Чтобы такие пары значений можно было использовать для автоматического перевода (при каждом внесении изменений в библиотеку ресурсов), надо дать команду Tools ➤ Translation Tools Options (Сервис ➤ Настройка средств перевода) и на вкладке Preferences (Свойства) установить флажок Automatic Repository Query (Автоматический опрос склада).

Как переключать текущий язык приложения

Чтобы отлаживать программу, интерфейс которой выполнен на одном из языков локализации, надо выполнить следующие действия.

1. Дайте команду Project ➤ Languages > Set Active (Проект ➤ Языки > Установить активный язык). В списке первоначально появятся два названия языка (рис. 12.27).

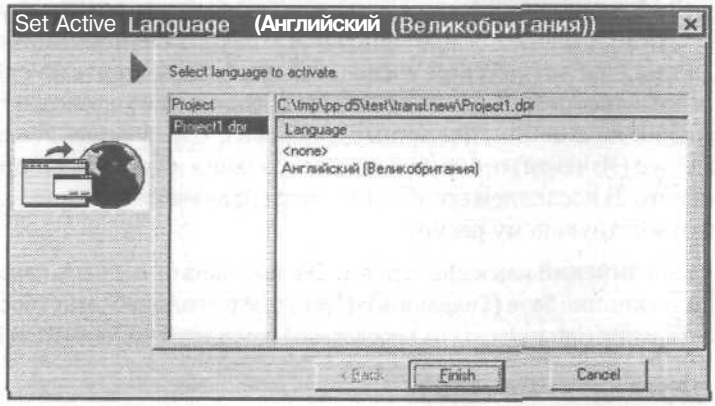


Рис. ***12.27.*** *Выбор текущего языка для отладки локализуемого приложения*

Пункт <none> (Нет) обозначает базовый язык приложения, в нашем случае русский. Для него создавать библиотеку ресурсов не требуется. Второй пункт — Английский (Великобритания). Его и надо выделить, после чего щелкнуть на кнопке Finish (Готово).

2. Вызовите Менеджер проектов командой View > Project Manager (Вид ➤ Менеджер проектов). В текущей группе проектов окажутся два проекта: Project1.exe, исполнимое приложение, и Project1.eng, библиотека английских ресурсов. Сделайте активным проект Project1.eng (библиотеку .DLL), выполните его компиляцию, затем снова переключитесь на проект Project1.exe и запустите его. Он выведет заголовки кнопки и окна на английском языке (рис. 12.28).

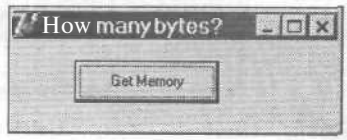


Рис. 12.28. «Локализованный» вариант программы, отображающей сведения об использовании памяти

В дальнейшем надо просто сменить активный язык на тот, который нужен в данный момент. Библиотеку ресурсов можно загрузить как обычную библиотеку .DLL или просто скопировать описания форм и других ресурсов **из** каталога eng.

Как использовать строковые константы внутри программы

Хотя значения текстовых свойств переведены, при щелчке на кнопке сообщение по-прежнему отображается на русском языке. Это связано с тем, что строковые константы из исходных текстов на Паскале, к сожалению, не включаются в Менеджер перевода. Наиболее удобный выход из такого положения — создать на одной из форм невидимый список, у которого значение свойства Visible равно False, и заполнить его содержимое (свойство Items) строками-константами. А в программе в качестве констант применить не строки, а номера этих строк в списке ListBoxl. Можно для удобства создать дополнительную функцию GetStr, которая будет возвращать строку по заданному номеру.

Полный текст примера приведен ниже:

```
unit Unit1;

Interface

uses
   Windows, Messages, SysUtils, Classes, Graphics,
Controls, Forms, Dialogs,
   StdCtrls;

const STR_ASK = 0;

type
   TForm1 = class(TForm)
      Button1: TButton;
      ListBox1: TListBox;
      procedure Button1Click(Sender: TObject);
   private
      { Private declarations }
   public
      { Public declarations }
      function GetStr(StrNum: integer) : string;
   end;
var
   Form1: TForm1;

implementation
```

продолжение

```
{$R *.DFM}

function TForm1.GetStr(StrNum: integer) : string;
begin
   Result := ListBox1.Items[StrNum]
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
   ShowMessage(GetStr(STR_ASK)+IntToStr(AllocMemSize));
end;

end.
```

Что нового мы узнали?

В этом уроке мы научились

- добавлять новые компоненты;
- готовить активные элементы *Web*-страниц;
- конструировать элементы Панели управления;
- программно управлять работой офисных приложений;
- формировать программы установки приложений;
- создавать справочную систему для программы;
- выполнять локализацию приложений;
- организовывать групповую работу коллектива профаммистов.

УРОК 13 Система UML-моделирования ModelMart

- ModelMart как CASE-система
- Диаграммы UML
- ModelMart: быстрый старт
- Документирование проекта
- Работа с диаграммами
- Настройка стилей представления

Проектирование приложений на языке UML

ModelMart— это система визуального проектирования структуры приложения для *Delphi*. Она основывается на языке визуального моделирования *UML*. С помощью системы *ModelMart* можно быстро создавать иерархию классов, переводить их Б код на Паскале и затем безошибочно компилировать. Возможен и обратный путь — существующий *Delphi-проект* может быть легко и автоматически перенесен в *ModelMart* для сопровождения в удобной среде построения визуальной модели проекта. *ModelMart* фактически открывает новый путь создания программ *на Delphi,* предлагая разработчику мыслить в терминах не программного кода, а высокоуровневыми понятиями и концепциями.

В дополнение к возможностям работы с *UML*-диаграммами в *ModelMart* реализовано немало расширений, делающих эту систему тесно интегрированной с *Delphi* и по своему уникальной. Так, предлагаются гибкие средства автоматической подготовки документации. Кроме того, в *ModelMart* реализована поддержка технологии шаблонов проектирования.

ModelMart как CASE-система

Система *ModelMart* по праву считается так называемой *CASE*-системой *(computer-aided system/software engineering),* хотя ее возможности гораздо шире. *CASE*-подход начал активно развиваться примерно с 80-х годов прошлого века. Он возник, когда разработчики крупных и сложных программных систем (прежде всего военные) осознали необходимость инструментария, позволяющего формализовать процесс общения заказчика с программистами и перевести его на уровень, не привязанный к программному коду. Одной из первых была переведена на компьютеры *SADT* технология структурного анализа и проектирования *SADT (Structured Analysis and Design Technique),* придуманная в военно-воздушных силах США. В дальнейшем она выросла в промышленный стандарт *IDEF,* до появления *UML* реализовывавшийся практически во всех *CASE*-системах. Однако по мере развития информационных технологий минусы *IDEF,* связанные прежде всего с отсутствием в этом стандарте поддержки объектных подходов, стали очень серьезным препятствием к его дальнейшим программным реализациям. Поэтому язык *UML,* тоже не лишенный недостатков, однако ориентированный на объектную разработку, быстро завоевал популярность, и сегодня успех *CASE*-системы на рынке немыслим, если в ней нет поддержки *UML*.

ЗАМЕЧАНИЕ

CASE-системы типа ModelMart часто используются в крупных проектах. Поэтому одновременно с системами моделирования нередко применяются и средства поддержки групповой работы. ModelMart, как и Delphi, допускает интеграцию с такими продуктами версионного контроля, как Microsoft Visual Source Safe 6 и QSC Team Coherence.

UML — универсальный язык программирования

Первая версия универсального языка моделирования *Unified Modelling Language (UML)* была опубликована в начале 1997 года. *UML* несколько лет создавался ведущими специалистами в области объектно-ориентированного анализа и проектирования программных систем Гради Бучем, Джеймсом Румбахом и Айраном Джекобсоном из известной корпорации *Rational Software*. Последняя версия *UML 1.3* признана в качестве стандарта независимым консорциумом *OMG (Object Management Group)*, занимающимся стандартизацией объектных технологий. Она используется в ходе разработки программ самой разной сложности сотнями крупнейших и тысячами средних и мелких компаний во всем мире. Сегодня всеми вопросами развития языка *UML* занимаются специалисты *OMG*.

Язык *UML* предназначен для визуального построения моделей программных систем. На его основе выпускаются продукты, позволяющие переводить графические *UML*-модели в программный код конкретной среды разработки. Язык *UML* позволяет отойти от низкоуровневого кодирования к проектированию приложения на уровне достаточно абстрактных сущностей, не погружаясь с самого начала в детали реализации программы.

Диаграммы UML

Основным понятием языка *UML* является диаграмма, графически отображающая некую сущность или понятие системы, а также связи между понятиями. Это может быть, например, класс, объект, пользователь, сопроводительная информация и т. д. В версии *UML 1.3* принято восемь типов диаграмм.

- Диаграмма прецедентов или вариантов использования (Use Case Diagram) предназначена для предварительного определения наиболее важных сущностей системы на уровне общения с пользователями. Диаграмма прецедентов обычно не привязывается к конкретным аспектам реализации системы. Она активно применяется прежде всего для формализации выдвигаемых заказчиком требований и синхронизации его взгляда на систему с взглядом исполнителя. Разработка сложного приложения с нуля обычно всегда начинается с построения диаграмм прецедентов.
- Диаграмма видов деятельности (Activity Diagram). Программа представляет собой последовательность операторов, анализирующих либо изменяющих данные. Операторы обычно объединяются в небольшие логические блоки (процедуры), выполняющие конкретное смысловое действие в контексте программного окружения. В ходе проектирования приложения очень полезно создавать не только диаграммы, описывающие взаимосвязи и структуру абстрактных или конкретных элементов, но и диаграммы, определяющие последовательность работы приложения — ход выполнения развернутых во времени процессов. Диаграммы видов деятельности позволяют в наглядном виде представлять на экране любые последовательности операций и редактировать их. Они немного напоминают блок-схемы алгоритмов.

- Диаграмма взаимодействия (Interaction Diagram). Как следует из названия, предназначена для описания способов взаимодействия между объектами программы. На практике обычно используется тип этих диаграмм, называемый диаграммами последовательностей (Sequence Diagram). Эти диаграммы дополняют диаграммы видов деятельности и поясняют, как в системе происходит обмен сообщениями между различными классами и объектами.
- Диаграмма классов (Class Diagram). Основной тип диаграмм, описывающий классы программы и взаимосвязи между ними. Активно используется *CASE*-системами для автоматической генерации исходных текстов приложений.
- Диаграмма состояний (State Diagram) уточняет принципы реализации классов системы, определяя, какие состояния они могут принимать в ходе работы программы, а также формализует переходы между такими состояниями.
- Диаграмма кооперации (Collaboration Diagram) объясняет, каким способом разные классы модели взаимодействуют друг с другом. Данная диаграмма не показывает временные аспекты модели — она только определяет, какие составляющие каждого класса, участвующего во взаимодействии, будут задействованы в этом взаимодействии.
- Диаграмма компонентов (Component Diagram) определяет, как будет реализовываться модель в конкретной системе разработки. Диаграммы компонентов позволяют уточнить конкретные особенности реализации, специфичные, например, для определенного языка программирования или конкретной компонентной технологии.
- Диаграмма развертывания (Delpoyment Diagram) дополняет диаграммы компонентов и позволяет фиксировать техническую структуру, на которой будет разворачиваться создаваемая программная система (серверы, сети) и сформировать накладываемые на нее ограничения.

В книге будет рассмотрен тип диаграмм классов, удобный для быстрого освоения и использования в разработке программ любого размера.

ЗАМЕЧАНИЕ Важнейшее отличие системы ModelMart от других CASE-систем заключается в том, что ModelMart тесно интегрирована с Delphi и с языком Delphi [бывший Object Pascal]. Поэтому система ModelMart использует структуру классов, реализованную в Delphi, и основывается на структуре исходного текста, принятого в Delphi. ModelMart реально обеспечивает работу с набором диаграмм, соответствующем последней спецификации UML 1.3.

Шаблоны проектирования

Шаблон проектирования — это объектный способ представления в общем виде условий задачи и правильных подходов к ее решению. Шаблоны позволяют, в частности, проектировать сложные системы, не создавая громоздкой иерархии насле-

дования, и в то же время дают возможность быстро добавлять в систему новые сущности.

Число шаблонов проектирования ограничено и насчитывает несколько десятков. Появлению нового шаблона предшествует достаточно большой объем анализа практического опыта и длительное его тестирование в прикладных проектах. Вот примеры популярных шаблонов:

- Facade — создание единого интерфейса для всех интерфейсов системы;
- Adapter — создание нового интерфейса для класса, интерфейс которого неудобен для решения текущей задачи;
- Bridge — разделение интерфейса от реализации в ситуациях, когда имеются вариации как в абстрактном представлении некоторой концепции, так и в ее реализации;
- Abstract Factory — организация интерфейса между семействами связанных между собой объектов без привязки к конкретным классам;
- Strategy — обеспечение независимости от алгоритма реализации с возможностью динамического выбора последнего;
- Decorator — динамическое расширение функциональных возможностей объекта;
- Singleton — обеспечение единственности созданного экземпляра класса;
- Observer — организация отношения «один-ко-многим» между объектами и автоматизация их обновления при изменении состояния основного объекта;
- Template Method — передача части алгоритма реализации в подклассы без изменения структуры родительского класса;
- Factory Method — передача функции создания объекта конкретного типа от абстрактного класса к производным классам.

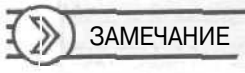
ModelMart: **быстрый** старт

Система *ModelMart* обладает очень богатой функциональностью. Одно ее сокращенное описание может занять сотни страниц. Однако в реальной работе программист обычно использует небольшой процент всех возможностей системы. Поэтому в данном разделе мы познакомимся с системой *ModelMart,* реализовав в *Delphi* небольшой конкретный проект.

Запуск системы ModelMart

Система *ModelMart* может быть запущена разными способами. Это можно сделать через стандартное меню *Windows* - система *ModelMart* поставляется как самостоя-

тельное решение. Однако наиболее удобный и эффективный способ эксплуатации *ModelMart* состоит в запуске этой *UML*-системы непосредственно из *Delphi*. Это выполняется командой ModelMaker ➤ Run ModelMaker главного меню *Delphi*.



ЗАМЕЧАНИЕ

Запущенная связка Delphi + ModelMart весьма требовательно к ресурсам компьютера. Минимальные требования находятся примерно на уровне Pentium III / 500 МГц и 256 Мб ОЗУ.

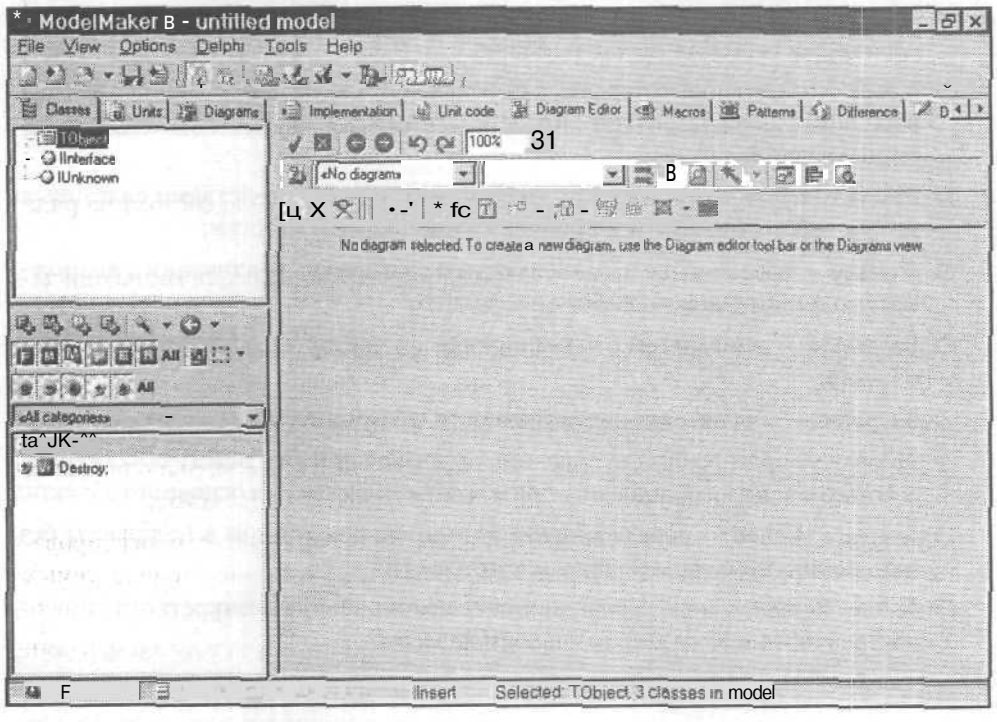
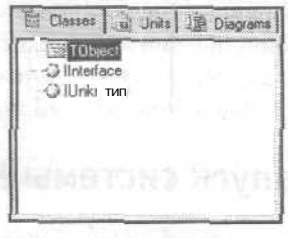


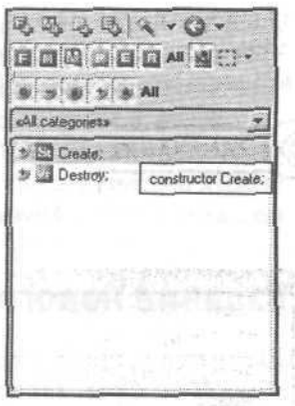
Рис. 13.1. *Основной экран системы ModelMart после запуска*

Основной экран системы ModelMart



Основной экран *ModelMart* разбит на несколько основных разделов. Верхняя часть традиционно отводится под главное меню и панель быстрых кнопок.

В левом верхнем углу располагается Просмотрщик классов (Classes view). Он позволяет быстро переключаться между описаниями классов и интерфейсов (панель Classes), связанных с ними модулей *Delphi* (панель Units) и диаграмм (панель Diagrams).



Обработка всех доступных элементов в данном Просмотрщике может вестись в двух визуальных режимах. Первый представляет структуру элементов в виде дерева (команда локального меню Display as tree (Представить в виде дерева)), а второй — в виде списка (команда Display as list (Представить в виде списка)). В большинстве случаев первый вариант предпочтительнее, и в дальнейшем подразумевается, что используется именно режим представления в виде дерева.

Ниже этого Просмотрщика расположен Просмотрщик интерфейса классов.

Он дает возможность определять и редактировать новые свойства, методы и события класса, настраивая средства их просмотра.

В правой части экрана находится главное окно программы. Оно состоит из ряда Панелей:

- Implementation (Реализация) — содержит программную реализацию описываемых классов;
- Unit Code (Код модуля) — позволяет выполнять навигацию по модулю *Delphi* и просматривать доступные процедуры;
- Diagram Editor (Редактор диаграмм) — визуальный построитель *UML*-диаграмм;
- Macros (Макросы) — предназначен для описания макросов, выполняющих автоматическое документирование проекта и исходных текстов;
- Patterns (Шаблоны) — дает возможность работать с шаблонами проектирования — набором готовых функциональных решений, широко используемых в ряде методологий разработки программного обеспечения;
- Difference (Различия) — формирует подробные различия между элементами проекта (модулями кода);
- Documentation (Документация) — представляет документацию проекта, как сгенерированную автоматически, так и подготовленную вручную;
- Events (События) — позволяет хранить и редактировать события проекта.

Модель ModelMart

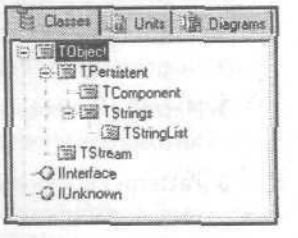
Система *ModelMart* реально не предоставляет возможность изменения исходного текста программы напрямую. Она хранит модель проекта в едином внутреннем формате. Возможности, предоставляемые визуальной оболочкой *ModelMart* и различными Просмотрщиками, просто представляют различные аспекты этой модели. Пользователь *ModelMart* работает с файлом с расширением .mpb (*ModelMart Project Bundles*), который хранит всю структуру модели системы. При этом файл *не* содержит исходный код приложения, а только высокоуровневое представление систе-

мы, преимущественно в виде *UML*-диаграмм. Генерация исходного кода на основе этой модели может происходить как автоматически (с заданной пользователем периодичностью,) так и вручную, при выполнении команды ModelMart Generate. Такой исходный код в свою очередь не содержит данных о связанных с ним конкретных моделях *ModelMart* (хотя их можно получить на основе этого кода автоматически), поэтому разработчику нужно самостоятельно следить за сопровождением проектов *Delphi* и *ModelMart*.

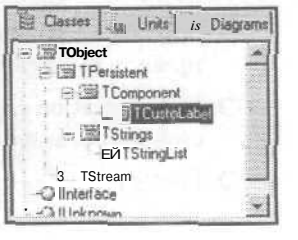
Создание нового проекта ModelMart

Рассмотрим пример создания нового компонента с использованием системы *ModelMart*. Сравнение двух подходов — с помощью стандартных возможностей *Delphi* (см. главу 12) и в визуальном построителе *ModelMart* — наглядно покажет преимущество нового подхода.

Работа с системой *ModelMart* начинается с формирования нового проекта. Для этого надо дать команду на его создание. Таких команд в *ModelMart* несколько. Команда File ➤ New (Файл ➤ Новый) создает пустой проект, который позволяет добавлять новые классы, основываясь на базовом классе Delphi TObject. Команда File ➤ New From Default (Файл ➤ Новый по умолчанию) дополнительно добавляет к проекту стандартные классы TPersistent, TObject, TStream и ряд других. Команда File ➤ New From Template (Файл > Новый на основе шаблона) создает проект, в котором иерархии классов берутся из ранее созданного *ModelMart*-проекта.



В нашем случае дадим команду File > New From Default. В Просмотрщике классов будет доступна иерархия, представленная на рисунке.



Добавим в дерево пока неописанный в нашем проекте, но существующий в *Delphi* класс TCustomLabel. Он является наследником класса TComponent, поэтому выделим в диаграмме класс TComponent и добавим ему нового наследника. Это можно сделать командой Add Descendant (Добавление наследника) локального меню или нажатием клавиши INS. Назовем новый класс TCustomLabel.

Иерархия классов *Delphi* довольно сложна, и не всегда удается быстро и корректно добавить в дерево классов новый класс с учетом правильной цепочки наследования. Если в описании иерархии используются существующие классы *Delphi* (не важно, стандартные или пользовательские), описание и реализацию которых в рамках данной модели вводить не надо, *ModelMart* допускает их наличие в проекте в виде реально не описываемых, так называемых «замороженных позиций» *(placeholder)*. Чтобы сделать TCustomLabel замороженным, надо дать команду локального меню Edit Class на редактирование класса или дважды щелкнуть на его названии кнопкой мыши. В диалоговом окне надо включить флажок placeholder:

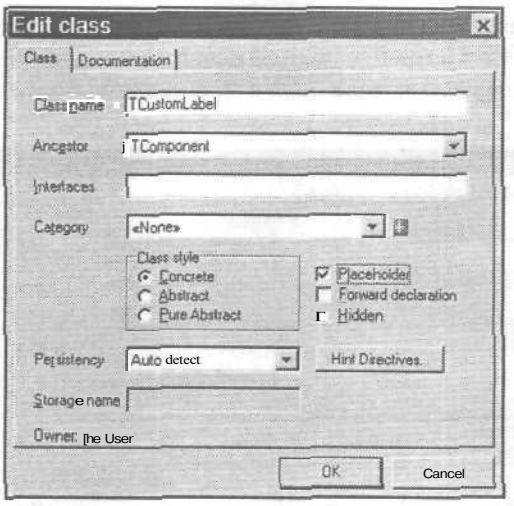
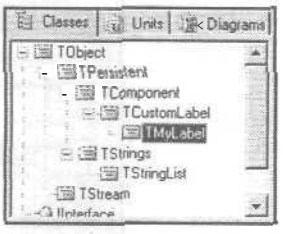


Рис. /3.2. *Редактирование описания класса*

Значок слева от названия замороженных классов обводится пунктиром. На данный момент все классы нашего проекта заморожены.



Теперь таким же способом добавим к TCustomLabel описание нового, создаваемого класса TMyLabel, наследника TCustomLabeL Только, конечно, не надо помечать его как замороженный, так как именно этот класс мы будем далее определять.

Как мы помним из первоначального примера создания TMyLabel вручную, новому классу надо добавить новые свойства и методы для считывания значений этих свойств и указания новых значений. Работа с интерфейсом класса выполняется в Просмотрщике интерфейса классов.

Выполним добавление свойства Caption. Для этого надо нажать кнопку Add Property (Добавить свойство) Просмотрщика интерфейса. В диалоговом окне укажем:

- в названии свойства (Name) — Caption;
- в названии типа (Data Type Name) — TCaption. В нашей иерархии данного типа нет, поэтому предварительно в группе переключателей типов Data type выберем значение User Defined (определяемое пользователем). При этом станет доступной нижняя часть окна. В поле Data Type Name введем строку TCaption;
- видимость (Visibility) — published.

Так как данное свойство Caption наследуется от класса TCustom Label (там оно скрыто), то функции считывания и задания его значения можно не создавать — достаточно указать его видимость в нашем дочернем классе. Поэтому в группах Read Access (способ доступа к значению) и Write Access (способ записи значения) выберем переключатели None (Нет). Перед этим обязательно включим флажок Property Override

(Перезаписать родительское свойство), потому что если наше свойство не является наследуемым, *ModelMart* потребует явного задания способов считывания и изменения его значений — переключатели None в группах Read Access и Write Access будут недоступны.

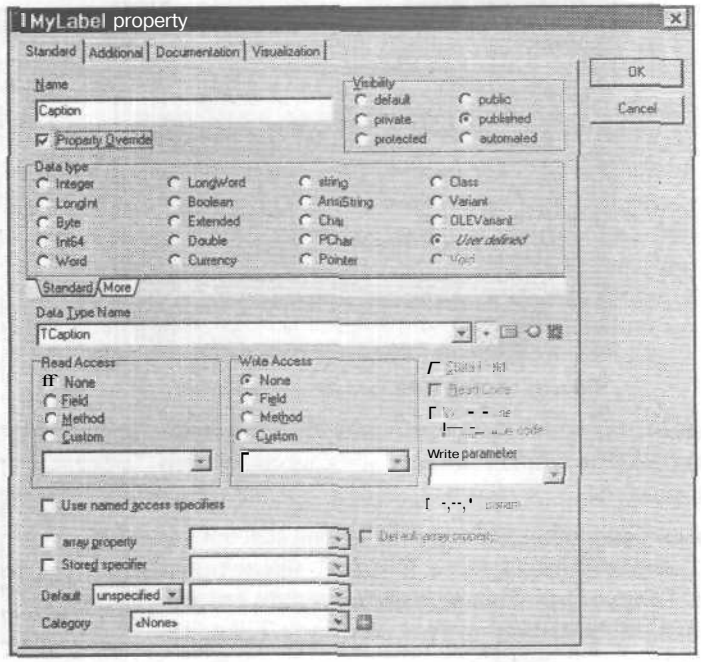
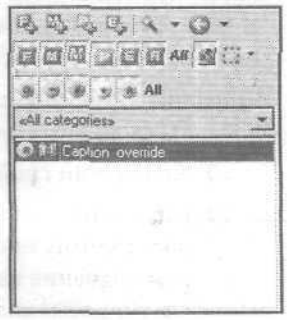


Рис. 13.3. *Редактирование интерфейса класса TMyLabel*



Нажмем кнопку OK. В Просмотрщике интерфейса появится свойство Caption.

Теперь добавим новое свойство FontColor. Оно также будет иметь видимость published и, конечно, должно описываться с выключенным флажком Property Override, так как является оригинальным. Тип данного свойства — определяемое пользователем значение TColor,

В группе Read Access вместо выставленного для свойства Caption по умолчанию значения Field (поле, подразумевающее доступ к значению через *private*-свойство с префиксом F) укажем Method (Метод). ModelMart в соответствии с требованиями Delphi автоматически сгенерирует название такого метода (GetFontColor). Выполним аналогичные действия для группы Write Access (Метод записи), описав метод SetFontColor. Параметр последнего метода задается в списке Write Parametr (Параметр для метода изменения значения) в левой нижней части окна. По умолчанию здесь задано Value, изменим его на AValue.

Кроме того, мы хотим получить доступ к данному полю внутри реализации класса также через *private*-свойство FFontColor. Для того чтобы оно было добавлено к уже сформированным методам считывания и изменения значения, включим флажок State Field (Статическое поле).

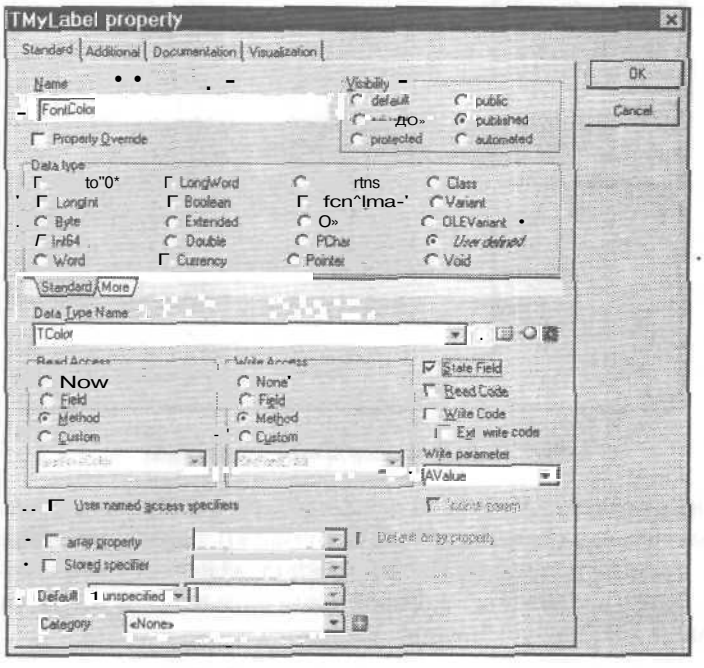
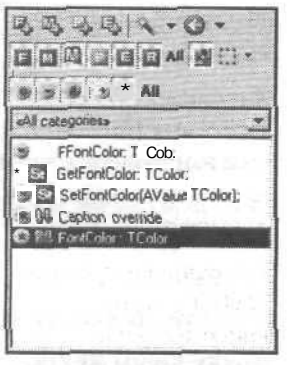


Рис. 13.4. Редактирование свойства FontColor

Нажмем кнопку OK. В Просмотрщике интерфейса появится новое свойство FontColor с зеленым значком, и *private*-свойство FFontColor с синим значком.



На данном этапе можно сохранить проект, выполнив команду File ➤ Save (Файл > Сохранить) и указав каталог.

Теперь требуется подготовить программную реализацию двух данных методов. Для этого выберем в Просмотрщике интерфейса метод GetFontColor и дадим команду View ➤ Implementation (Просмотр > Реализация) главного меню или перейдем с помощью мышки на вкладку Implementation главного окна *ModelMart* (рис. 13.5).

Подготовка программной реализации некоего метода в *ModelMart* существенно отличается от приемов, используемых в редакторе *Delphi*. Это связано с тем, что *ModelMart* накладывает очень строгие формальные требования к описанию исходного кода, так как возможные человеческие ошибки могут вызвать внутренние противоречия в описываемой системе и заведут *ModelMart* в тупик.

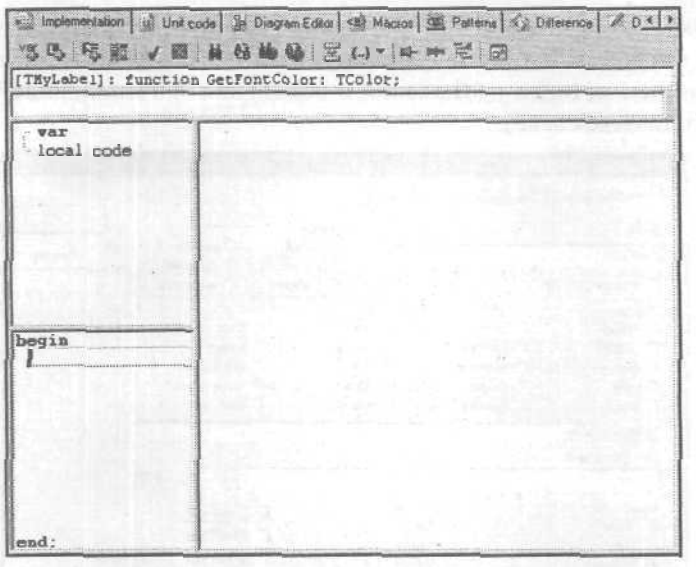


Рис. *13.5. Редактор программного кода*

Ввод команд в редактор *ModelMart* осуществляется построчно. Сначала разработчик выбирает строку (новую или существующую) текущей реализации (небольшое окно с полным текстом на Паскале расположено в нижней левой части главного окна). Затем вводит текст в правой части окна и нажимает кнопку Save code (Сохранить код) панели быстрых кнопок.

ЗАМЕЧАНИЕ Набор последовательных строк данного редактора, объединенных одинаковым значком слева, называется разделом. Разделы, созданные программистом, подсвечиваются зелеными значками; разделы, сгенерированные ModelMart, — красными.

При этом *ModelMart* исходно создает все описание текущего реализуемого метода и автоматически добавляет начальное и конечное ключевые слова begin/end.

Б нашем случае достаточно ввести одну секцию, состоящую из одного оператора (рис. 13.6):

```
Result := FFontColor
```

и нажать на кнопку Save code.

Точно таким же способом зададим реализацию метода SetFontColor:

```
FFontColor := AValue
```

В заключение добавим компоненту TMyLabel переопределяемый метод Paint (кнопка Add Method (Добавить метод)). Его надо пометить как protected, установить переключатель Override в группе Method binding kind (Метод связывания) и включить флажок Call inherited, чтобы вызвать метод перерисовки компонента-родителя. Этот

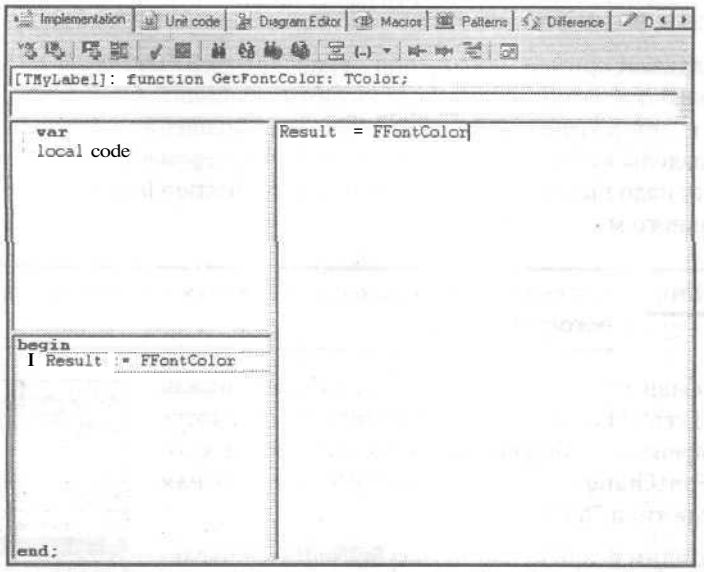


Рис. 13.6. Реализация функции GetFontColor

вызов будет помечен в окне исходного текста в виде красно-белого значка (в отличие от зеленого значка пользовательского кода).

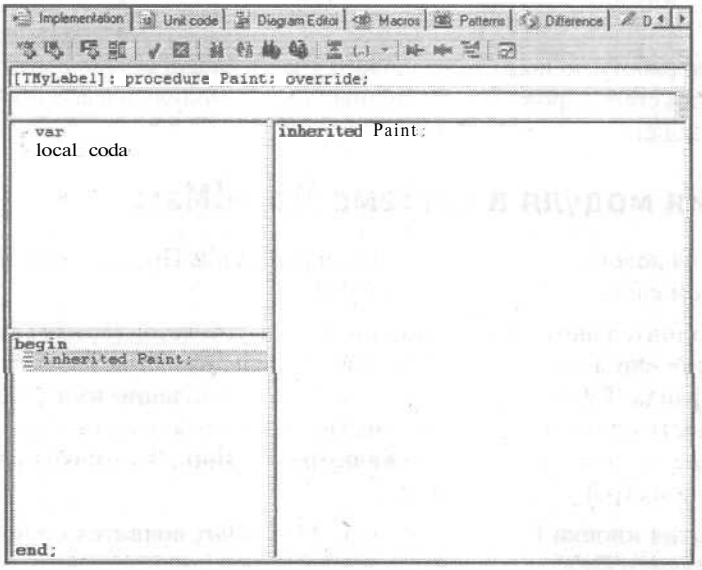


Рис. 13.7. *Реализация процедуры Paint*

Следом за этим вызовом добавим оператор задания нового цвета. Для этого нажмем кнопку Add section (Добавить секцию) и введем следующий код:

```
Font.Color := FFontColor;
```

В первоначальном примере обращение к родительскому методу Paint выполнялось после данного оператора. Однако *ModelMart* позволит вводить новые разделы только после автоматически сгенерированного обращения inherited Paint;. Чтобы поменять разделы местами (в нашем случае, чтобы переместить красно-белый раздел вниз), их надо выделить и дать команду Move Section Down (Переместить раздел вниз) локального меню.

ЗАМЕЧАНИЕ Для перемещения раздела вверх служит команда Move Seclion Up локального меню.

Наконец добавим классу TMyLabel новое событие, нажав кнопку Add Event. Назовем его OnFontChange. В Просмотрщике интерфейса можно увидеть, как в дополнение к событию OnFontChange появилась *private*-переменная FOnFontChange типа TNotifyEvent.

Теперь дополним реализацию мегода SetFontColor до следующего текста:

```
FFonCColor := AValue;
Repaint;
if Assigned(FOnFontChange) then
 OnFontChange(Self);
```

На этом всю работу по подготовке нового класса *Delphi* можно считать законченной. Осталось сгенерировать завершенный *Delphi*-модуль и выполнить установку нового компонента.

Генерация модуля в системе ModelMart

Для создания нового модуля перейдем на панель Units Просмотрщика классов и нажмем кнопку Add (Добавить модуль).

В поле дополнительного имени каталога исходных текстов (Source path Alias) оставим значение <No alias> (Без дополнений), В поле, где задастся местоположение исходного файла (Relative Unit file name), укажем подходящие имя файла и каталог. В нижней части окна (рис. 13.8) переместим с помощью кнопок-стрелок в правую часть название класса TMylabel, подлежащее реализации. В крайней правой колонке VCL Page введем строку ModelMart Test.

После нажатия кнопки OK в правом окне *ModelMart* появится сгенерированный шаблон будущего *Delphi-модуля:*

```
unit <!UnitName!>;

interface
```

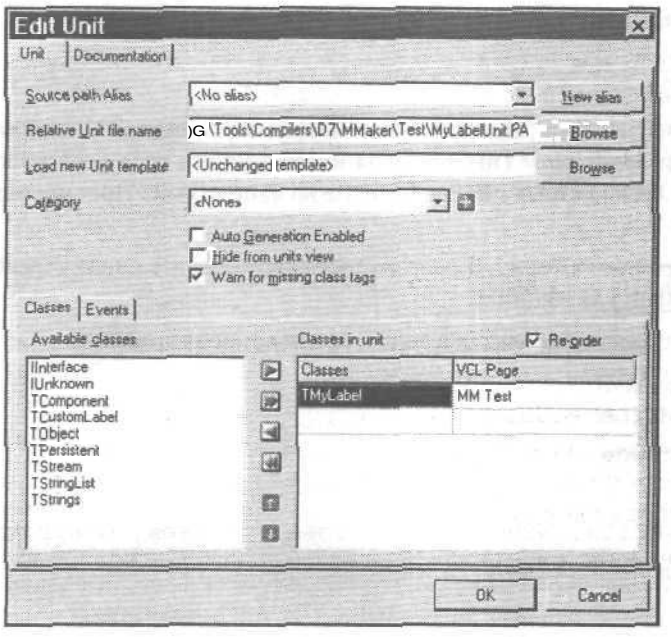


Рис. 13.8. Генерация нового модуля

```
uses
  SysUtils, Windows, Messages, Classes, Graphics,
Controls, Forms, Dialogs;

type
MMWIN:STARTINTERFACE
MMWIN:CLASSINTERFACE TMyLabel; ID=542;

procedure Register;

implementation

procedure Register;
begin
MMWIN:CLASSREGISTRATION TMyLabel; ID=542;
Page='MODELMARTTest';
end;

MMWIN:STARTIMPLEMENTATION
```

продолжение

```
MMWIN:CLASSIMPLEMENTATION TMyLabel; ID=542;

end
```

Чтобы перейти к генерации конечного *Delphi*-кода, надо разблокировать генератор кода *ModelMart* (щелчком на кнопке Unlock code generation главной панели) и затем нажать кнопку Generate на панели Units Просмотрщика классов.

Теперь перейдем в *Delphi*. Для этого достаточно нажать кнопку Locate In Delphi главного меню *ModelMart*.

В *Delphi* откроется проект, и в редакторе появится готовый модуль. Его текст будет примерно таким:

```
unit MyLabelUnit;
Interface
uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;
type
  TMyLabel = class (TCustomLabel)
  private
    FFontColor: TColor;
    FOnFontChange: TNotifyEvent;
    function GetFontColor: TColor;
    procedure SetFontColor(AValue: TColor);
  protected
    procedure Paint; override;
  published
    property Caption;
    property Fontcolor; TColor read GetFontColor write
     SetFontColor;
    property OnFontChange: TNotifyEvent read FOnFontChange
     write FOnFontChange;
  end;
procedure Register;
implementation
procedure Register;
begin
  RegisterComponents("MODELMARTTest", [TMyLabel]);
end;
```

```
{
******************** TMyLabel ********************
}
function TMyLabel.GetFontColor: TColor;
begin
  Result := FFontColor
end;

procedure TMyLabel.Paint;
begin
  Font.Color := FFontColor;
  inherited Paint;
end;

procedure TMyLabel.SetFontColor(AValue: TColor);
begin
  FFontColor := AValue;
  Repaint;
  if Assigned(FOnFontChange) then OnFontChange(Self);
end;
end
```

Добавим данный модуль к стандартному *Application*-проекту и выполним компиляцию, чтобы убедиться, что исходный текст не содержит ошибок или пропущенных определений (например, недостающих стандартных модулей). Однако в нашем случае произойдет ошибка. Компилятор укажет на строчку

```
TMyLabel = class (TCustomLabel)
```

и сообщит, что класс TCustomLabel в программе не объявлен. Действительно, его описание хранится в модуле StdCtrls, который не указан в списке подключаемых модулей, сгенерированных ModelMart (система не знает о местоположении его описания).

Данное изменение можно внести в редакторе *Delphi,* однако это будет неправильно с методологической точки зрения. Поэтому вернемся в *ModelMart* и на панели Unit code главного окна добавим в список подключаемых модулей название StdCtrls, выполним команду Save code, после чего сохраним проект и повторно выполним кодогенерацию.

ЗАМЕЧАНИЕ Наилучшим решением в данной ситуации будет внесение изменений [добавление нового подключаемого модуля) в шаблон ModelMart, который используется при создании проекта командой File ➤ New From Default. Эта возможность описана в документации по ModelMart.

На этом создание нового класса-компонента можно считать завершенным Его установка в *Delphi* выполняется обычным способом, как описано в исходном примере.

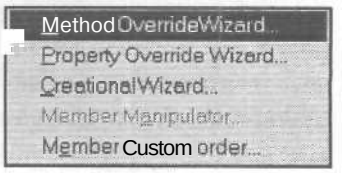
Внесение изменений в существующий проект

Допустим, мы хотим дополнить наш класс новыми возможностями. Например, желательно, чтобы исходно значение заголовка содержало строку Это объект класса TMyLabel, а ее начальный цвет всегда был зеленым. По умолчанию же содержимое заголовка соответствует значению свойства Name (Название) объекта-надписи, *а* цвет обычно устанавливается черным.

Вернемся в *ModelMart,* В принципе мы можем внести все изменения в *Delphi,* но тогда эти изменения не будут видны в системе моделирования. Кроме того, *ModelMart* в процессе генерации кода перезаписывает существующий файл кода на языке Паскаль. Поэтому если будет продолжено использование *ModelMart,* то все изменения, сделанные нами в *Delphi,* потеряются. Некоторые операции по модификации проекта действительно легче и быстрее выполнять в *Delphi,* но выигрыш, получаемый от ведения проекта в *ModelMart* на значительно более высоком концептуальном уровне, превосходит все локальные проигрыши.

Теперь надо добавить нашему классу TMyLabel новый метод. Назовем его Create - это будет конструктор класса, в котором мы станем задавать начальное значение свойства Caption.

У конструктора имеется типовой параметр AOwner (тип TComponent). Этот конструктор можно добавить в Просмотрщике интерфейса, выполнив команду Add Method и задав все настройки вручную. Однако, в отличие от ранее созданного и перезаписанного метода Paint, на этот раз мы воспользуемся Мастером создания перезаписываемых методов, позволяющим быстро подготовить описание соответствующего метода, если он имеется в исходной шаблонной модели *ModelMart,* которая применялась при подготовке текущего проекта. Конструктор Create имеется у класса TComponent.



Дадим команду Wizards ➤ Method Override Wizard (Мастеры ➤ Мастер создания перезаписываемых методов) Просмотрщика интерфейса.

В открывшемся диалоговом окне в списке доступных для перезаписи методов выберем конструктор Create. Убедимся, что флажок Call Inherited method в нижней части диалогового окна включен — это необходимо для добавления вызова наследуемого конструктора родительского класса. Щелкнем на кнопке OK — на этом создание конструктора завершится. Подробное описание конструктора можно получить, выполнив двойной шелчок на названии Create в Просмотрщике интерфейсов.

Теперь перейдем на вкладку Implementation главного окна в редактор исходных текстов. Следом за красно-белым наследуемым вызовом Create добавим наш код задания начального значения заголовка и цвета, нажав кнопку Add section (Добавить раздел):

```
Caption := "Это объект класса TMyLabel",-
FFontColor := clLime;
```

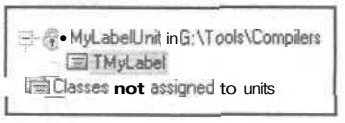
Теперь можно выполнить перегенерацию исходного кода (кнопка Generate Просмотрщика классов).

В ходе интенсивной модификации проекта *ModelMart* не всегда бывает удобно выполнять данную операцию вручную. Возможны, например, варианты, когда разработчик просто забудет выполнить перегенерацию кода и в *Delphi* будет компилироваться старый проект.

Чтобы установить режим автоматической кодогенерации при внесении в проект изменений, надо нажать кнопку Enable Auto Generation. Она расположена справа от кнопки Generate.

Вернуться в режим автоматической кодогенерации можно, нажав кнопку Disable Auto Generation, отменяющую данный режим.

Немаловажно, что режим автоматического переформирования кода отдает *Delphi* распоряжение перезагрузить модифицированные файлы на языке Паскаль, если они открыты в редакторе *Delphi*. Это происходит незаметно для разработчика. При этом, конечно, надо проследить, чтобы режим кодогенерации был разблокирован (была нажага кнопка Unlock code generation главной панели *ModelMart)*. Если она отжата (что в данном случае неправильно), то в Просмотрщике классов слева от названия текущего редактируемого модуля будет виден значок замка.



ЗАМЕЧАНИЕ Просмотрщик классов обладает удобным пользовательским интерфейсом и поддерживает технологию «перетащи и оставь». Если в данный момент удалить наш класс TMyLabel из текущего указанного для генерации модуля (MyLabelUnit.pas) — для этого служит клавиша DEL — то он не исчезнет из проекта, а переместится вниз, в раздел Classes not assigned to units (Классы, не распределенные по модулям). В дальнейшем класс можно перетащить мышкой обратно в любой доступный модуль. Дерево модулей при этом перестроится автоматически.

Переустановим наш компонент в *Delphi* и убедимся, что он работает в соответствии с новыми дополнениями — принимает на форме зеленый цвет и исходно содержит явно задаваемую в конструктор строку.

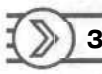
Интеграция Delphi с системой ModelMart

Delphi тесно интегрирован с системой *ModelMart.* В любой момент в *ModelMart* можно передать текущий модуль с хранящимися в нем описаниями классов и их реализацией. Для этого достаточно дать команду *Delphi ModelMart* — Add to Model (Прибавить к текущей модели). При этом в текущий проект *ModelMart* будет передан

текущий модуль *Delphi*. Он появится в Просмотрщике классов на вкладке Units. Если требуется создать новый проект *ModelMart* на базе существующего модуля *Delphi*, надо дать команду ModelMart > Convert to Model (Преобразовать модуль в модель ModelMart). Допускается также сразу создать готовый проект *ModelMart* на основе всего проекта *Delphi* и сохранить его в файле. Для этого служит команда ModelMart ➤ Convert project to Model (Преобразовать проект в модель ModelMart).

При желании можно настроить запущенные *ModelMart* и *Delphi* так, чтобы любые вносимые в проекты изменения делались одновременно доступными в обеих оболочках. Синхронизация выполняется, когда либо проект сохраняется, либо выполняется компиляция.

Для включения данной возможности надо вызвать диалоговое окно *ModelMart* Intergation options (Настройки объединения ModelMart) командой *Delphi* ModelMart ➤ Intergation Options и включить флажок Enable Auto Refresh (Разрешить автоматическое обновление).



ЗАМЕЧАНИЕ Такая возможность обладает определенным недостатком, и использовать ее при работе в Delphi надо осторожно. Если, например, сохранить проект в момент, когда исходный код содержит незаконченные элементы (незавершенные логические блоки или реализации методов), *ModelMart* начнет *автоматическое* создание совместимой с проектом Delphi и логически целостной модели, исключая неверные или ошибочные части кода из проекта. В результате значительная часть работы может быть утеряна.

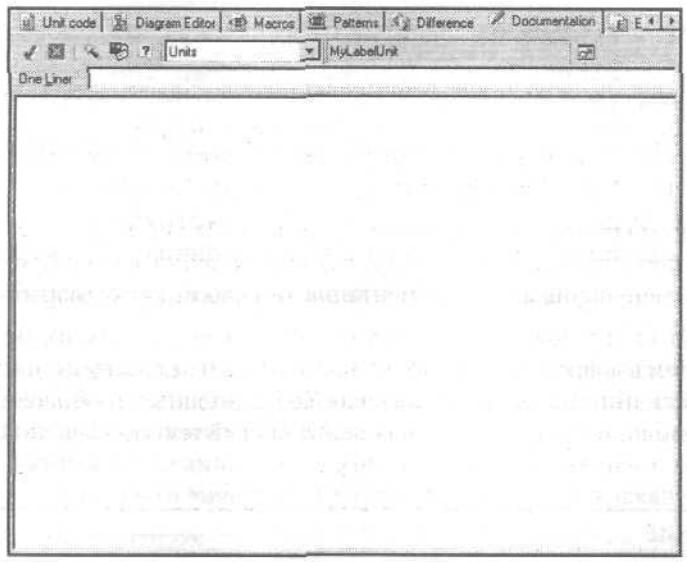
Документирование работы

Средства автоматического документирования текущего проекта *ModelMart весьма* мощны. Они поддерживаются набором Мастеров и позволяют формировать стандартные комментарии для различных элементов интерфейсов классов, других частей исходных текстов, а также генерировать файлы справки. Эти возможности основываются на средствах визуального построителя диаграмм *ModelMart*.

Добавление документации к проекту

Большинство элементов диаграмм *ModelMart* может быть сопровождено как коротким однострочным комментарием (One Liner), так и более подробной многострочной документацией (Documentation). Поддержка режима документирования происходит на панели Documentation главного окна.

Сначала приступим к сопровождению нашего класса TMyLabel с помощью Мастера. Он вызывается нажатием кнопки Wizard. Первоначально Мастер спросит, создавать ли стандартную документацию для класса TMyLabeL Надо ответить OK.



Рис, 13.9- Окно документирования

В раскрывающемся списке в центре верхней части окна документирования (рис. 13.10) выберем значение Class Members (Члены класса).

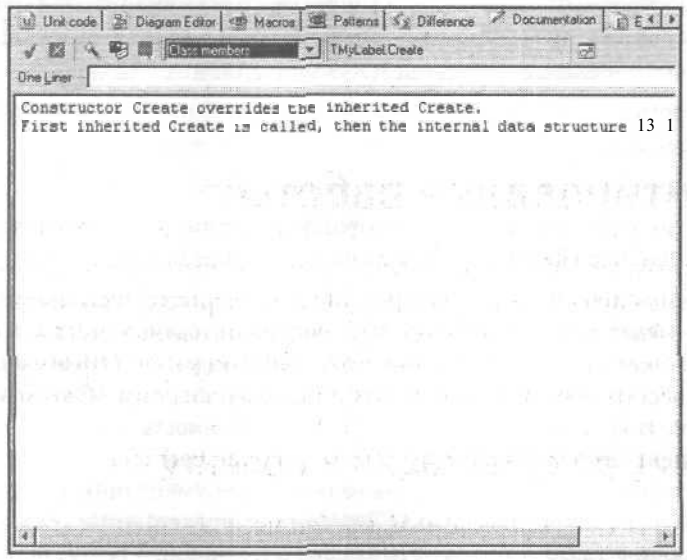


Рис. 13.10. Ввод комментариев

Теперь перейдем в Просмотрщик интерфейса и выберем там метод (конструктор) Create. В окне документирования появятся две строчки комментариев, которые были сгенерированы для данного метода автоматически.

```
Constructor Create overrides the inherited Create.
First inherited Create is called, then the internal data
structure is initialized
(Конструктор Create перезаписывает родительский метод
Create.
Сначала будет вызван наследуемый конструктор Create, затем
будет выполнена инициализация внутренних структур данных)
```

Как видим, сгенерированная документация хорошо понятна разработчику.

Обратим внимание, какой текст был сгенерирован для других методов класса TMyLabel. Затем выберем метод Paint — в нем мы ввели не совсем стандартную последовательность инициализации внутренних переменных и обращения к методу класса-родителя. В связи с этим сопроводительный текст на панели Documentation желательно изменить вручную. Добавим к нему приведенную ниже строку, не забыв в конце нажать кнопку Save (Сохранить внесенные изменения):

```
Но мы не выполняем никакой работы с холстом, а просто
меняем цвет шрифта, а потом вызываем родительский метод
отрисовки.
```

Подобным образом добавляется документация и для классов — выбором в раскрывающемся списке панели Documentation значения Classes. Зададим для нашего класса TMyLabel однострочный комментарий (поле One Liner) в виде строки Пример создания класса в ModelMaker, продублировав его в основном окне комментариев. Модуль MyLabelUnit сопроводим двустрочным комментарием:

```
Это модуль для работы с классом TMyLabel, расширяющим
возможность стандартного класса TLabel.
Он создан автоматически из ModelMaker.
```

Теперь можно переходить к созданию файла помощи. Для этого надо щелкнуть на кнопке Help File (Файл помощи) па панели Document

В открывшемся диалоговом окне выполним настройки создаваемого файла. Поле Help RTF File (Файл помощи в формате RTF) содержит полный путь к *RTF-файлу,* содержащему текстовое представление структуры подсказки (файлы помощи стандартным способом создаются из *RTF*-файлов; подробнее см. об этом в соответствующей главе). В группе Class member Visibilities (Видимость элементов класса) можно указать степень детализации подсказок. Значение User (Пользователь) указывает, что в файл подсказки будет включена только та документация, которая привязана к общедоступным элементам класса. Значение Component writer (Разработчик компонента) добавляет также описания *protected*-элементов. Третье значение Designer (Проектировщик) означает, что в файл помощи будет включена вся документация. После щелчка на кнопке OK в указанном каталоге для *RTF*-файла появится также файл с расширением .HLJ (файл проекта помощи).

Сборка результирующего файла помощи (расширение .HLP) описана в соответствующем разделе книги.

Редактирование диаграмм класса TMyLabel

Система *ModelMart* предназначается прежде всего для работы с диаграммными описаниями. Это одна из наиболее сильных его сторон. Пока мы не работали с нашим классом непосредственно в редакторе диаграмм, потому что графическое древовидное представление нашего класса пока не сформировано. Делается это так.

- Дать команду View > Diagrams (Просмотр > Диаграммы) главного меню или выбрать закладку Diagrams Просмотрщика классов.
- Дать команду View ➤ Diagram Editor (Просмотр > Редактор диаграмм) главного меню или выбрать закладку Diagram Editor в главном окне. Будет показана уже известная нам с момента запуска *ModelMart* диаграмма базовых классов *Delphi*.

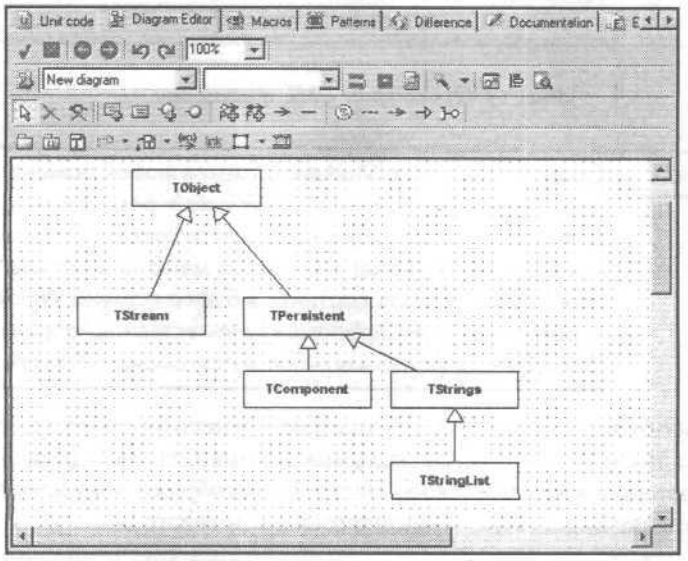


Рис. 13.11. *Редактор UML-диаграмм*

- Создать новую диаграмму классов, нажав кнопку Add Diagram и указав в открывшемся диалоговом окне:
  - в поле Name — название новой диаграммы (например, TMyLabel Diagram);
  - в поле Type — тип *UML*-диаграммы (в нашем случае Class Diagram, диаграмма классов);
  - в поле Parent — значение <no parent diagram> (нет родительской диаграммы). В этом поле можно указать существующие в текущем проекте диаграммы, с которыми будет связан описываемый на новой диаграмме класс. В нашем случае это может быть текущая диаграмма с базовыми классами. Однако

сейчас мы хотим сосредоточиться на описании только одного нашего класса TMyLabel, поэтому оставим в данном поле оригинальное значение.

Щелкнем на кнопке OK — экран редактора диаграмм очистится. Теперь перейдем в Просмотрщик классов (на вкладку Classes) и перетащим мышью на экран редактора диаграмм сначала класс TMyLabel, а затем родительский класс TCustomLabel. Между ними будет автоматически сформирована соответствующая *UML*-связь.

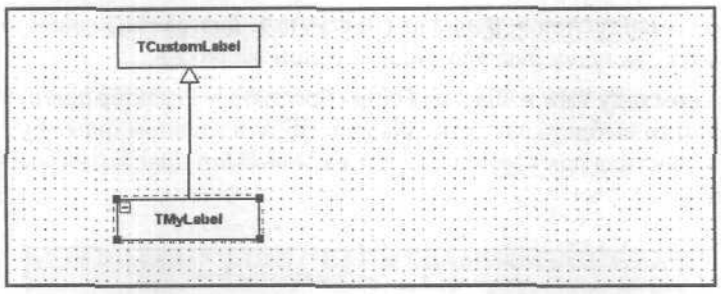


Рис. 13.12. *Создание UML-диаграммы*

ЗАМЕЧАНИЕ Созданную в ModelMart диаграмму в любой момент можно сохранить во внешнем файле в виде рисунка (в формате .bmp, .jpg и других). Для этого надо дать команду Export as image ➤ File (Экспортировать как картинку ➤ Файл) локального меню диаграммы. Точно так же можно скопировать картинку в буфер обмена Windows (команда Export as image ➤ Clipboard) и затем вставить, например, э редактор Word. Печатьдиаграммы выполняется командой Printлокального меню.

Каждый класс будет отображаться на диаграмме в некотором стиле, определенном настройками *ModelMart*. Текущий стиль можно изменить, дважды щелкнув мышью на прямоугольнике класса TMyLabeL Настройка отображаемых элементов выполняется на вкладке Symbol style открывшегося диалогового окна.

Прежде всего необходимо изменить значение Diagram Auto Member list раскрывающегося списка Member List style (Список стиля элементов) — оно определяет, какие элементы класса будут показываться — на значение Auto Member list, что сделает доступными дополнительные флажки фильтрации этих элементов в нижней части окна. Включим флажки Properties (Свойства) и Methods (Методы). В раскрывающемся списке Member visibility filter (Фильтр видимости элементов класса) выберем значение public, published (на диаграмме будут показываться только *public*- и *published*-элементы.)

В разделе Member display options (Настройки отображения элементов класса) задаются характеристики вывода на экран выбранных элементов. Неактивные состояния флажков означают, что при включении соответствующего флажка будут использованы настройки родительской диаграммы, на основе которой создавалась текущая диаграмма.

Установим флажки Show data type (Показывать тип данных) и Events in new compartment (События в новом разделе). В правой нижней части окна на панели Show display options (Показывать настройки отображения) включим флажок Show module (unit) name (Показывать название модуля). Высоту и ширину диафаммы можно либо формировать автоматически, либо задавать вручную (для этого надо установить флажки Auto size width/Auto size height).

Рис. 13.13. Настройка стиля отображения

Щелкнем на кнопке OK. Диаграмма примет вид, показанный на рис. 13.14.

В верхней части перед названием класса следует название модуля, отделенного двумя двоеточиями. Далее следуют *public/published*-свойства (Caption и FontColor) и метод Create.

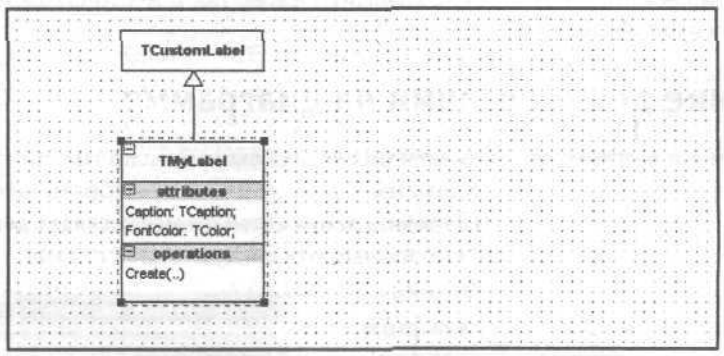


Рис. 13.14. Детализация элементов диаграммы

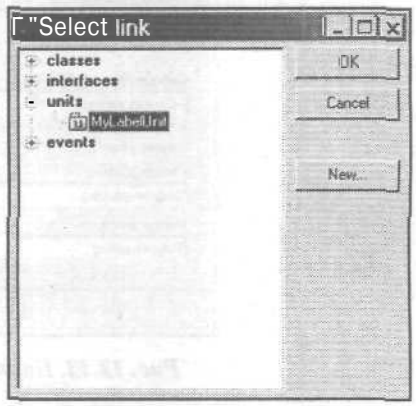
ЗАМЕЧАНИЕ Каждую секцию диаграммы можно свернуть и развернуть, щелкнув соответственно на знаках минус или плюс в левом верхнем углу секции.

Отображение модулей на диаграмме

Система *ModelMart* ориентирована на решение прикладных задач и нередко используется не только как *CASE*-инстумент, но и как среда для ускорения процесса создания программ на *Delphi*. Поэтому возможности *ModelMart* расширены набором возможностей, ориентированных на программиста. Так, на одной диаграмме с описанием классов полезно иметь также информацию о модулях *Delphi,* хранящих эти классы.

На панели вкладки Diagram Editor имеется кнопка Add Unit Package (Добавить описание модуля).

Щелкнем на этой кнопке (пока ничего не произойдет, но курсор примет вид стрелочки с крестиком) и затем щелкнем мышкой на свободном месте окна редактора диаграмм или растянем в окне прямоугольник, определяющий размер и местоположение нового модуля на диаграмме. Б диалоговом окне Select Link будет предложено выбрать один из доступных модулей проекта. В нашем случае это будет единственный модуль MyLabelUnit.



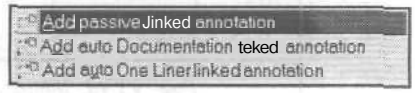
Выберем его и щелкнем на кнопке OK. В диалоговом окне Package Symbol (рис. 13.15) появятся текущие настройки для отображения данного модуля на диаграмме. Здесь надо установить флажок Show contained classes (Показывать содержащиеся в модуле классы) и щелкнуть на кнопке OK. Диаграмма примет примерно такой вид (рис. 13.16).

На ней наглядно видна как структура класса TMyLabel, так и местонахождение его реализации (модуль MyLabelUnit).

Отображение документации на диаграмме

Теперь перейдем к важному шагу включения созданных ранее примечаний и комментариев к проекту а текущую диаграмму. Для этого надо выбрать режим Add auto Documentation linked annotation (Включить режим автоматической связи с документацией). Для этого надо последовательно выполнить следующие действия.

- выбрать соответствующий пункт в раскрывающемся списке, связанном с кнопкой Add passive linked annotation;



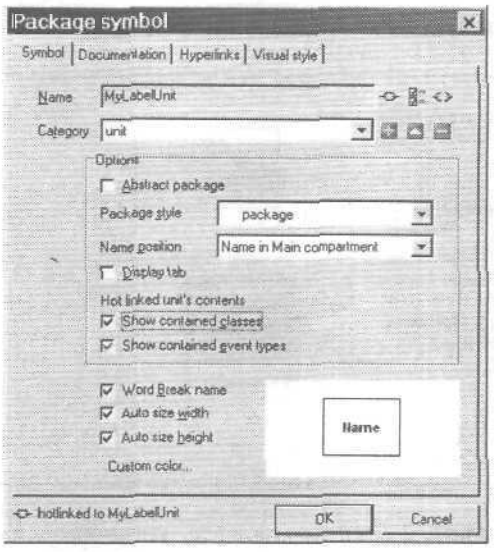


РИС. 13.15. *Настройка описания модуля*

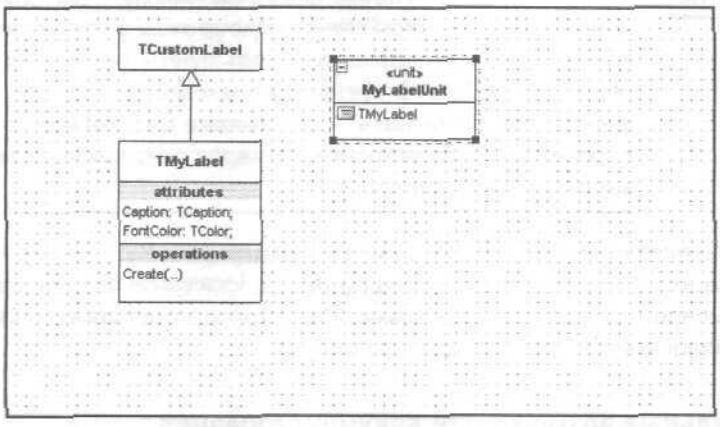


Рис. 13.16. *Добавление описания модуля на диаграмму*

- щелкнуть на прямоугольнике документируемого класса (TMyLabel);
- передвинуть курсор в сторону при нажатой левой кнопке мыши (при этом от объекта TMyLabel потянется пунктирная линия);
- щелкнуть кнопкой мыши в том месте, где мы хотим видеть блок комментария.

ЗАМЕЧАНИЕ В этом блоке выводится сопроводительный текст, введенный ранее в главном, многострочном окне редактора — не в однострочном поле One Liner!

В результате будет получена следующая диаграмма.

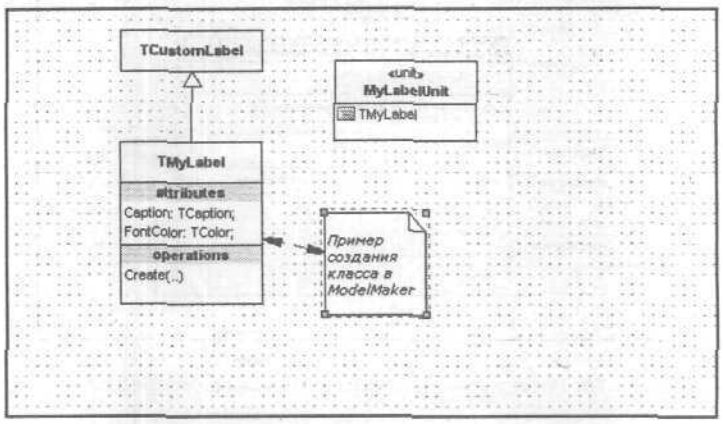


Рис. 13.17. *Добавление комментариев на диаграмму*

ЗАМЕЧАНИЕ Возможно, что текст примечания будет выведен нерусифицированным шрифтом. В током случае надо дважды щелкнуть на элементе примечания и на закладке Visual style [Стиль отображения) в разделе Fonts (Шрифты) выбрать подходящий шрифт в раскрывающемся списке Font name. А но закладке General можно настроить параметры автоподстройки размеров окна примечания и способ выравнивания и переноса слов.

Аналогичную операцию повторим с элементом диаграммы MyLabelUnit. В дальнейшем при каждом изменении текстов, связанных с классами, автоматически произойдет изменение текста на диаграмме. Редактор примечания можно быстро вызвать, выбрав примечание и нажав клавишу F2.

Дополнительные возможности документирования

В ходе рассматриваемого примера были затронуты основные приемы создания документации к проекту. Однако в ходе подготовки текстового сопровождения к различным элементам диаграммы переключаться между панелями редактора диаграмм Diagram Editor и подготовки документации Documentation не очень удобно. Система *ModelMart* позволяет открыть в редакторе диаграмм плавающее окно документирования. Оно вызывается командой View ➤ Floating Documentation (Просмотр > Плавающая документация) главного меню.

В нем автоматически отображается аннотация и многострочный текст текущего выбранного на диаграмме элемента. Эти сведения можно отредактировать и внести в диаграмму командой Save локального меню плавающего окна.

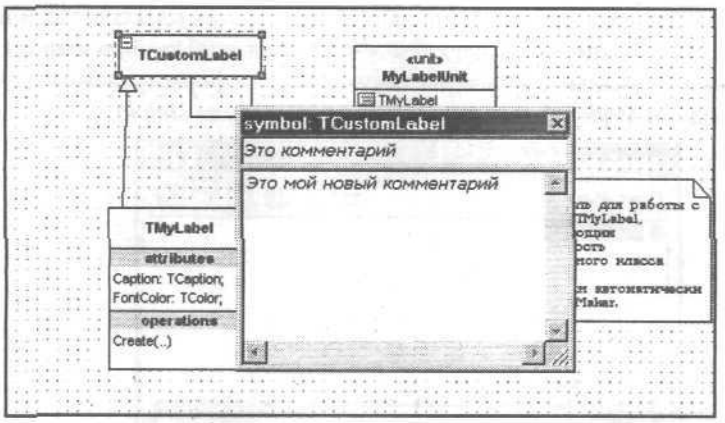


Рис. *13.18.* *Редактирование комментария на диаграмме*

Импорт диаграмм из существующих проектов

Возможности системы *ModelMart* по созданию диаграмм с нуля очень хороши, однако в большинстве случаев система используется в ситуациях, когда уже имеется достаточно большой объем наработок и желательно перенести эти наработки в *ModelMart* автоматически. Как правило, никаких проблем такая операция не вызывает.

ЗАМЕЧАНИЕ Процесс создания модели на основе исходных текстов часто называют обратным реинжинирингом. Под прямым реинжинирингом подразумевается соответственно генерация исходных текстов из модели.

Сохраним текущий проект и создадим новый командой главного меню File ➤ New From Default. Теперь попробуем импортировать в него только что созданный нами класс TMylabel. Для этого щелкнем на кнопке Import source file in new model (Импорт исходного файла в новую модель) на главной панели быстрых кнопок.

Выберем каталог, где хранится модуль MyLabelUnit, и укажем этот файл. *ModelMart* автоматически загрузит его в текущий проект. Щелкнем на кнопке Add Diagram (Добавить диаграмму) редактора диаграмм и дадим имя новой диаграмме (в качестве типа укажем Class Diagram). Затем щелкнем на кнопке OK и перетащим класс TMyLabel из Просмотрщика классов (с вкладки Classes) в окно редактора, как мы это уже делали раньше, получив в итоге новую диаграмму. Видимость элементов класса можно настроить, дважды щелкнув мышью в диаграмме на названии класса.

Вы можете поэкспериментировать с любыми *Delphi*-файлами, ранее созданными вами или другими разработчиками (например, из каталога примеров *Delphi),* и полюбоваться красивым и удобным представлением структуры описанных в них классов.

Импортировать таким образом можно как одни описания классов, так и их реализацию. Б диалоговом окне, открывающемся после нажатия кнопки импорта, можно настраивать самые разные характеристики процесса формирования диаграмм и построения модели *ModelMart* из исходного кода модуля *Delphi*.

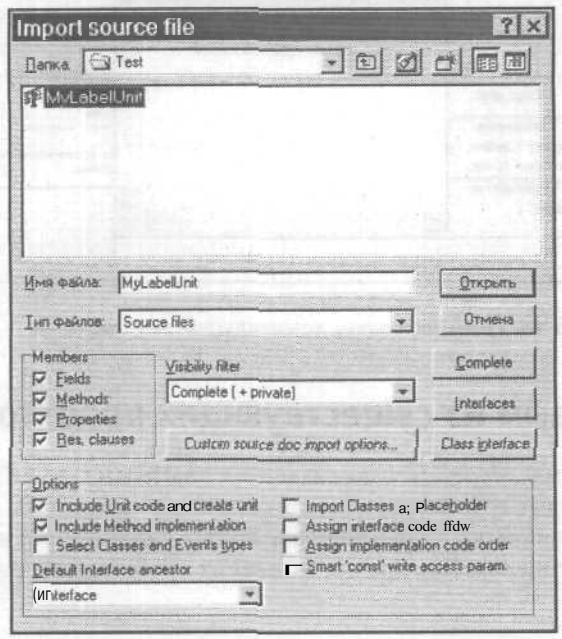


Рис. 13.19. *Импорт исходного файла*

В разделе Members с помощью флажков задаются те элементы класса (поля, свойства, методы), которые будут переданы в новый класс. В большинстве случаев требуются все элементы, но в ситуациях, когда на основе существующего класса надо создать новый класс, например абстрактный или схожий с существующим, удобнее отказаться от детального описания и переносить в новый класс только методы или *public*-свойства. Фильтр видимости Visibility filter дает возможность задать уровень видимости отбираемых элементов класса: Complete (Все), Inheriting client (Все, кроме private-элементов) или Using client (Только public и published).

Детальные настройки процесса импорта описываются в разделе Options.

- Include Unit code and create unit — флажок устанавливается, если на основании импортируемых сведений надо создать новый модуль *Delphi* и внести в него существующую реализацию модуля.
- Include Method implementation — флажок устанавливается, если надо импортировать вместе со свойствами также и реализацию методов. Если данный флажок снят, то импортируемые методы будут пустыми.
- Select Classes and Events types — если флажок установлен, разработчику будет предложено дополнительное диалоговое окно, в котором можно вручную указать классы и события, подлежащие импорту.

- Import Classes as Placeholder — флажок устанавливается, когда импортируемые классы должны быть абстрактными. При попадании в модель *ModelMart* они помечаются как замороженные. Данная возможность фактически используется для импорта интерфейсов классов.
- Assign Interface / Assign Implementation Code Order — когда флажок установлен, *ModelMart* не будет изменять после импорта порядок следования элементов класса в интерфейсе и реализации класса на стандартный (например, в алфавитном порядке). Такой стандартный порядок задастся в настройках проекта на вкладке Code generation диалогового окна, открываемого командой Options ➤ Project options.
- Smart «const» Write Access Parameter Import — флажок устанавливается, если надо добавлять характеристику const (константа) параметрам методов изменения значений свойств. Данный флажок рекомендуется устанавливать только при необходимости.
- Default Interface Ancestor - флажок определяет, будет ли использоваться по умолчанию родительский интерфейс IInterface или IUnknown.

Вот пример импортированного модуля MyLabelUnit со снятым флажком Include Method implementation:

```
unit MyLabelUnit;
interface
uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls;
type
  TMyLabel = class (TCustomLabel)
  private
    FFontColorr TColor;
    FOnFontChange: TNotifyEvent;
    function GetFontColor: TColor;
    procedure SetFontColor(AValue: TColor) ;
  protected
    procedure Paint; override;
  public
    constructor Create(aOwner: TComponent); override;
  published
    property Caption;
    property Fontcolor: TColor read GetFontColor write
      SetFontColor;
```

продолжение ⇨

```
    property OnFontChange: TNotifyEvent read FOnFontChange
     write FOnFontChange;
  end;

procedure Register;
implementation
procedure Register;
begin
  RegisterComponents("MODELMARTTest", [TMyLabel]);
end;

{
********** TMyLabel **********
}
constructor TMyLabel.Create(aOwner: TComponent);
begin
end;

function TMyLabel.GetFontColor: TColor;
begin
end;

procedure TMyLabel.Paint;
begin
end;

procedure TMyLabel.SetFontColor(AValue: TColor);
begin
end;
end.
```

Этот модуль будет успешно компилироваться, однако все методы в нем пустые.

Проблемы импорта исходных текстов

Импорт кода всегда оказывается успешным, если он осуществляется на основе ранее сгенерированного системой *ModelMart* кода *Delphi*, который вручную либо не модифицировался, либо модифицировался аккуратно (не удалялись комментарии

*ModelMart*и сохранялись все сгенерированные *ModelMart*элементы оформления). Вот основные правила, которыми руководствуется синтаксический анализатор *ModelMart*при построении модели *ModelMart* на базе текста *Delphi*.

1. Удаляются все добавленные программистом комментарии, директивы компилятора и пустые строки в интерфейсном описании класса. Например, давайте внесем следующие изменения в класс TMyLabel:

```
type
  TMyLabel = class (TCustomLabel)
  private
    FFontColor: TColor; // цвет метки

{$IFNDEF DEMO_LABEL}
    FOnFontChange: TNotifyEvent;
{$ENDIF}

    function GetFontColor: TColor;
    procedure SetFontColor(AValue: TColor);
  protected
    procedure Paint; override;
  public
    constructor Create(aOwner: TComponent); override;
  published
    property Caption;
    property FontColor: TColor read GetFontColor write
     SetFontColor;
    property OnFontChange: TNotifyEvent read FOnFontChange
     write FOnFontChange;
  end;
```

(добавлен комментарий «цвет метки» и директива условной компиляции, выделенная пустыми строками). Тогда после команды *Delphi* ModelMart > Convert to Model (Преобразование в модель ModelMart), перехода в *ModelMart* и выполнения генерации кода нз основе сформированной модели описание этого класса автоматически обновится и примет такой стандартный вид:

```
type
  TMyLabel = class (TCustomLabel)
  private
    FFontColor: TColor;
    FOnFontChange: TNotifyEvent;
```

продолжение ☞

```
    function GetFontColor: TColor;
    procedure SetFontColor(AValue: TColor);
  protected
    procedure Paint; override;
  public
    constructor Create(aOwner: TComponent); override;
  published
    property Caption;
    property Fontcolor: TColor read GetFontColor write
     SetFontColor;
    property OnFontChange: TNotifyEvent read FOnFontChange
     write FOnFontChange;
  end;
```

При этом, очевидно, будет нарушена логика создаваемого приложения, так как наличие события FOnFontChange в классе теперь не зависит от настроек препроцессора Delphi. Если программист подразумевал, что при определенной препроцессорной переменной DEMO_LABEL компилятор как минимум выдаст ошибку (такой прием сам по себе не совсем корректен и приведен только для наглядности), то в измененном системой *ModelMart* тексте данная особенность проекта учтена не будет. Кроме того, перегруженный в *Delphi c* диска после перегенерации файл с исходным текстом модуля нельзя будет в редакторе «откатить назад» командой отмены редактирования Undo, даже если включены настройки редактора Delphi Undo after save (Поддерживается откат после сохранения на диск). Ведь в данном случае происходит не исправление текста в редакторе, а загрузка **из** внешней по отношению к *Delphi* среды.

2. Каждая группа переменных одного типа в *var*-разделе переменных метода должна находится на отдельной строке, обладать явно указанным типом и быть при этом выровнена по левому краю. Например, если переписать реализацию метода SetFontColor так (происходят выделения трех составляющих цвета-параметра в три переменные r, g и b):

```
procedure TMyLabel.SetFontColor(AValue: TColor);
var r,g: Integer;
      b: Integer;
begin
  r := GetRValue(ColorToRGB(AValue));
  g := GetGValue(ColorToRGB(AValue));
  b := GetBValue(ColorToRGB(AValue));
  FFontColor := AValue;
  Repaint;
```

```
  if Assigned(FOnFontChange) then OnFontChange(Self);
end;
```

затем передать модуль в *ModelMart*и выполнить генерацию, то *var*-секция метода будет выровнена:

```
var
  r, g: Integer;
  b: Integer;
```

3. Если в качестве типа одного из элементов класса указана процедура, то такое описание не будет корректно импортировано в *ModelMart.* Это достаточно естественное, хотя и жесткое, требование **языка** Паскаль. Так. два абсолютно одинаково описанных массива:

```
var
  ModelsA: array[1..10] of TModel;
  ModelsB: array[1..10] of TModel;
```

в Паскале считаются данными разных типов, хотя и совпадающих по своей структуре. В таких случаях полагается описывать такую структуру как тип данных:

```
type TModelArray = array[1..10] of TModel;
var
  ModelsA, ModelsB: TModelArray;
```

Аналогично обстоит дело и с типами процедур. Объявление в классе элемента FMyProc:

```
type
  TMyLabel = class (TCustomLabel)
...
public
FMyProc: procedure of object;
end;
```

вызовет при попытке импорта сообщение *ModelMart*об ошибке. В этом случае надо просто сделать описание procedure of object новым **типом:**

```
type
  TMyProc = procedure of object;
  TMyLabel = class (TCustomLabel)
...
public
FMyProc: TMyProc;
end;
```

4. Требования к интерфейсу класса (отсутствие комментариев, директив компилятора и пустых строк) в полной мере относятся и к описанию методов в разделе реализации модуля.
5. *ModelMart* не допускает описания локальных переменных в начале и середине тела модуля, если они не являются элементами класса. Так, если ввести локальную переменную x:

```
var x: integer;
procedure TMyLabel.SetFontColor(AValue: TColor);
var
  r, g: Integer;
  b: Integer;
begin
  r := GetRValue(ColorToRGB(AValue));
  g := GetGValue(ColorToRGB(AValue));
  b := GetBValue(ColorToRGB(AValue));
  x := b;
  FFontColor := AValue;
  Repaint;
  if Assigned(FOnFontChange) then OnFontChange(Self);
end;
```

передать данный код в *ModelMart* и выполнить генерацию, то *ModelMart* переместит описание переменной x в самый конец модуля:

```
procedure TMyLabel.SetFontColor(AValue: TColor);
Var
  r, g: Integer;
  b: Integer;
begin
  r := GetRValue(ColorToRGB(AValue));
  g := GetGValue(ColorToRGB(AValue));
  b := GetBValue(ColorToRGB(AValue));
  X := b;
  FFontColor := AValue;
  Repaint;
  if Assigned(FOnFontChange) then OnFontChange(Self);
end;

...
```

```
var x: integer;
end.
```

Соответственно, если в классе TMyLabel нет поля x, то компилятор выдаст ошибку.

6. Не допускается использовать препроцессорные директивы включения файлов $INCLUDE.

7. Фактически все препроцессорные директивы игнорируются *ModelMart,* однако есть исключения. Директивы условной компиляции могут охватывать только участки реализации методов и не должны затрагивать части, связанные с заголовками. Так, описание

```
{$IFDEF DEMO_LABEL}
procedure TMyLabel.SetFontColor(AValue: TColor);
begin
// ничего не делать
end;
{$ELSE)
procedure TMyLabel. SetFontColor(AValue: TColor);
begin
FFontColor := AValue;
end;
{$ENDIF}
```

недопустимо при работе с *ModelMart.* Его необходимо заменить на

```
procedure TMyLabel.SetFontColor(AValue: TColor);
begin
{$IFDEF DEMO_LABEL}
// ничего не делать
{$ELSE)
FFontColor 1= AValue;
{$ENDIF}
end;
```

Необходимо, чтобы заголовок мегода не попал в условную директиву. Как уже говорилось, *ModelMart* не интерпретирует директивы препроцессора. В процессе импорта он просто проверяет парность логических связок begin ... end, case ... end, try end и тому подобных, невзирая на наличие таких директив.

8. Не обрабатываются написанные на ассемблере методы (*ModelMart* не содержит анализатора ассемблерного кода). Следующая конструкция считается при работе с *ModelMart* недопустимой:

```
procedure TMyLabel.NewFontColor; assembler;
```

```
asm
...
end;
```

9. Не разрешены выражения, следующие после спецификатора индекса в описании свойств:

   Выражение `COLOR_BASE + 5` в описании свойства

   ```
   property MyGreenColor: TColor index MY_COLOR_BASE + 5 read
    GetMyColorIndex;
   ```

   придется заменить на константу:

   ```
   property MyGreenColor: TColor index MY_GREEN_COLOR read
    GetMyCoOlorIndex;
   ```

Работа с диаграммами

Модель *ModelMart* может содержать любое число диаграмм любых доступных видов, как того требует *UML*. При этом диаграммы не обязательно связывать друг с другом в рамках данной модели. Каждой диаграмме дается название, при этом допускается совпадение названий.

Список всех диаграмм модели можно получить с помощью команды View ➤ Diagrams (Просмотр ➤ Диаграммы) главного меню или нажатием клавиши F5.

Диаграммы хранятся в модели *ModelMart* в виде дерева, хотя просматривать их можно по разному. В Просмотрщике объектов на закладке Diagrams можно просмотреть текущее дерево диаграмм и изменить метод просмотра, указав подходящий вариант в локальном меню командами Order By ➤ Hierarchy (Упорядочить ➤ По иерархии), Order By > Name (Упорядочить ➤ По имени) или Order By ➤ Name, Type (Упорядочить > По имени и типу). Допускается произвольная структура дерева. Для этого в момент создания новой диаграммы в поле Parent диалогового окна добавления можно выбрать любую доступную диаграмму проекта.

Каждая диаграмма отличается способом визуализации на экране и способом представления содержащихся в ней символов.

Ассоциации

Элементы модели связываются между собой ассоциациями *(association)* или отношениями. Эти ассоциации определяют некоторую смысловую связь между разными понятиями модели (как минимум между двумя — если один из связанных элементов удаляется, то исчезают и ассоциации с ним). Ассоциации в *ModelMart* имеют направленность (от начального элемента к целевому), хотя в некоторых типах ассоциаций оно может явно не показываться.

Допускается задание ассоциаций не только между сущностями (например, классами) но и между другими ассоциациями. Так, документирующая связь *(documentation link)* предназначена для описания различных элементов модели *ModelMart* и позволяет создавать такое комментирующее описание, в частности, для ассоциаций.

Создается ассоциация так. Выбирается подходящий тип связи. Курсор устанавливается на начальном элементе, нажимается левая кнопка мыши и курсор перемещается к конечному элементу, затем кнопка мыши отпускается. В окне редактора диаграмм возникает новая связь.

Ассоциации можно дать имя. Оно отображается в середине ассоциации в пунктирном прямоугольнике. Имя указывается либо в поле Discriminator на вкладке Association редактора свойств ассоциации, который вызывается двойным щелчком на линии связи, либо нажатием клавиши F2 и редактированием названия прямо Б окне Diagrams.

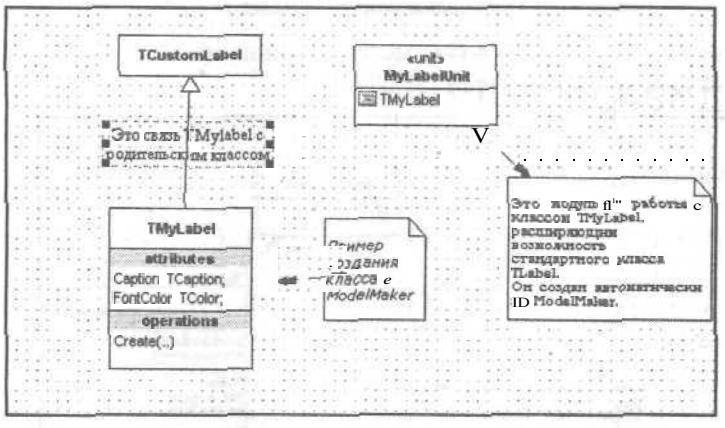


Рис. 13.20. *Ассоциация г. названием*

Представление ассоциаций на экране

Ассоциации крепятся к каждому из элементов диаграммы с помощью якорей *(anchor)* — черных прямоугольников, располагаемых, как правило, в центре каждой из связываемых сторон (рис. 13.21). Мышкой можно перетаскивать якоря, а восстанавливаются они в исходное центрированное состояние командой Association ➤ Reset Anchors (Ассоциация ➤ Сброс якорей) локального меню ассоциации.

Не всегда бывает удобно использовать для связи прямые линии. Диаграмма может быть большой и сложной, и ее отдельные элементы начнут пересекаться видимыми связями. Иногда требуется состыковать несколько ассоциаций в одном месте, что также подразумевает более гибкое представление ассоциации на экране.

ModelMart допускает добавление в существующие связи между элементами диаграммы узлов образа *(shape node)*. Выделим любую ассоциацию на экране, подведем к нему курсор, нажмем клавишу CTRL и, не отпуская ее, потянем курсор в сторону.

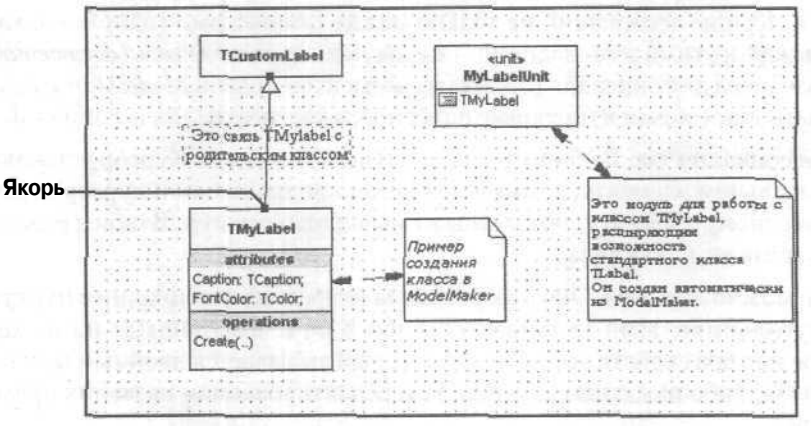


Рис. *13-21.* *Установка якорей на диаграмму*

Когда клавиша будет отпущена, в новом месте возникнет новый узел образа, а ассоциация изменит свою структуру (рис. 13.22).

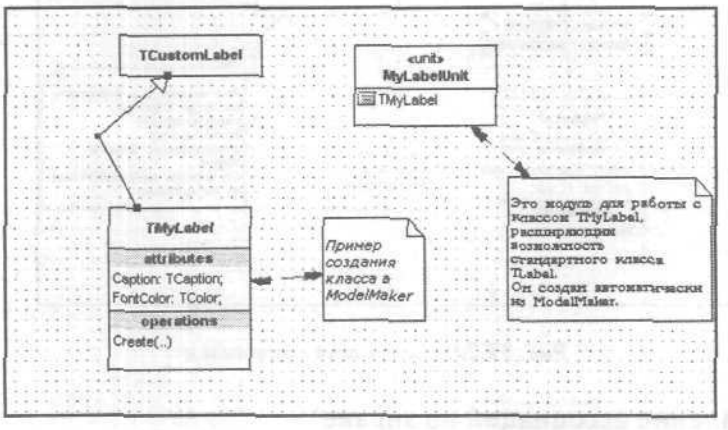


Рис. 13.22. *Изменение формы связи*

С помощью подобных узлов ассоциации можно придавать произвольную форму (рис. 13.23).

Если ассоциацию выровнять по прямой линии, то промежуточные узлы, которые станут не нужными (окажутся на прямой), исчезнут. Узлы можно удалять и по одному командой Assotiation > Remove shape node (Ассоциация ➤ Удалить узел образа) либо Assotiation > Remove all shape nodes (Ассоциация ➤ Удалить все узлы образа) локального меню.

Рекуррентные (круговые) ассоциации

Ассоциация может быть рекуррентной — начинающейся и кончающейся на одном и том же элементе модели. Создается такая ассоциация как и обычная связь, только

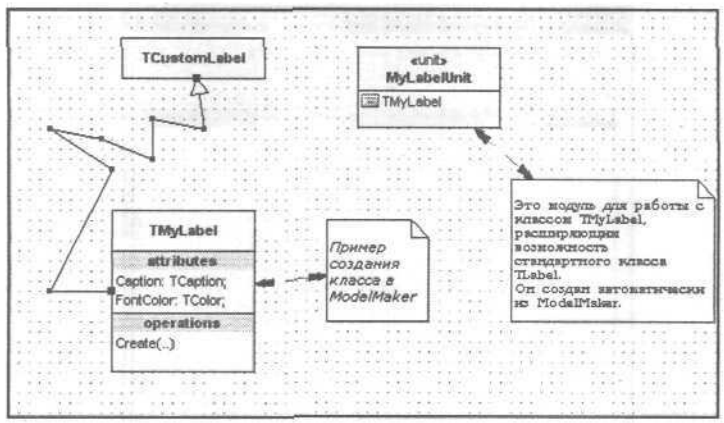


Рис. 13.23. Выбор произвольного вида связи

вместо конечного элемента снова выбирается начальный. При этом возникшая ассоциация имеет круговую структуру, а к ассоциации будут автоматически добавлены два узла образа, чтобы обеспечить замкнутость графической линии.

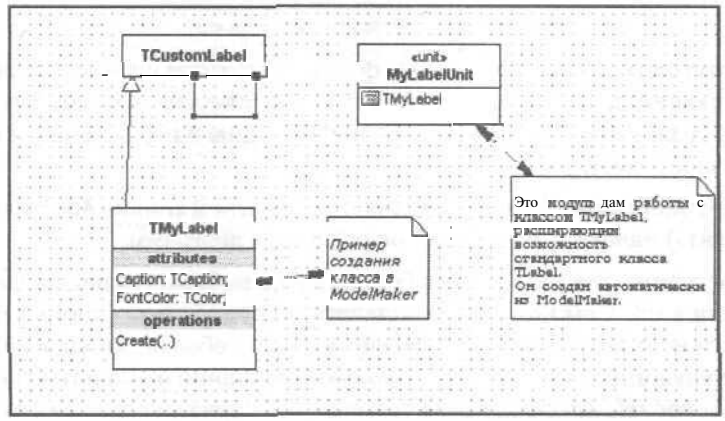


Рис. 13.24. Выбор произвольного вида связи

Гиперсвязи

Гиперсвязи *(hyperlinks)* — одна из важнейших особенностей всех систем моделирования. В принципе все элементы диаграммы имеют гиперсвязи, объединяющие классы, их элементы, модули и т. д. Гиперсвязи элементов диаграммы можно посмотреть на закладке Hyperlinks диалогового окна настройки свойств.

Первая строка в списке считается гиперсвязью по умолчанию. Она выделяется шрифтом с подчеркиванием. Эта гиперсвязь обычно отображается в виде значка в прямоугольнике класса после названия элемента. Ее наличие определяется в стиле представления (см. ниже).

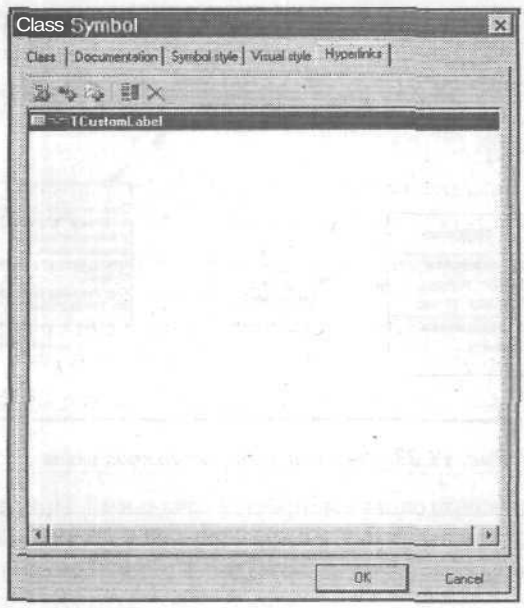


Рис. *13.25. Просмотр гиперсвязей*

Если щелкнуть мышью на этом значке, фокус диаграммы переместится на объект, соответствующий данной гиперсвязи. Так, если гиперсвязь указывает на другую диаграмму, редактор сохранит текущую информацию и переключится на соответствующую диаграмму.

Добавление гиперсвязи объекту выполняется нажатием кнопки Add Hyperlink (Добавить Гиперсвязь) панели горячих кнопок редактора диаграмм.

В дереве доступных для ссылки сущностей модели выбирается подходящая и нажимается кнопка OK. Ссылки можно задавать на диаграммы (кнопка Add diagram reference), элемент кода (класс, интерфейс, модуль, событие; кнопка Add code model reference) и внешний документ (произвольный внешний файл или адрес в Интернете; кнопка Add reference to external document). В случае, когда у объекта имеется несколько гиперсвязей, можно выбрать подходящую в разделе Navigation (Навигация) локального меню.

Стили содержимого и представления диаграмм на экране

Каждая диаграмма, а также все входящие в нее элементы, в том числе и ассоциации, обладает собственными стилями визуального представления. Они определяют, как соответствующие им элементы отображаются на экране. Если стиль элемента явно не задан, он берется из стиля родительской диаграммы. Если явно не задан стиль диаграммы, он берется из стиля представления проекта *ModelMart*. Соответственно, если вносятся изменения в стиль проекта, они вызывают изменения во

всех входящих в него диаграммах. А изменения в стилях диаграмм приводят к изменению стилей входящих в них элементов. Стили всех уровней можно именовать и затем повторно использовать.

Основные свойства стиля

Одни свойства стиля подходят ко всем элементам и диаграммам, другие — только к конкретным типам. К общедоступным свойствам относятся:

- размер и название основного шрифта для названий элементов и ассоциаций (такие характеристики шрифта, как наклон или толщина, менять не допускается, потому что они используются в *UML* с определенными смысловыми целями);
- размер и название шрифта, которым выполняется вся остальная текстовая информация;
- цветовая палитра различных деталей оформления;
- способы представления различных элементов диаграмм.

Редактирование стиля представления

Практически все диалоговые окна редактирования элементов диаграмм имеют вкладку Visual style. На ней выполняется редактирование стиля представления конкретного элемента проекта (рис. 13.26).

В раскрывающемся списке Parent style указывается, какой стиль будет использоваться для настроек представления по умолчанию. Для элементов диаграммы это будет, соответственно, либо <diagram style> (стиль диаграммы), либо <project style> (стиль проекта). Если требуется установить все настройки родительского стиля, достаточно нажать кнопку Revert, расположенную справа от списка.

Настройки шрифта выбираются в разделе Fonts, цветовые настройки — в разделе Color Palette. Способ отображения деталей оформления (таких, как показ стрелок на диаграммах) указывается в разделе Options.

Некоторые параметры элементов диаграмм можно задавать непосредственно с панели быстрых кнопок редактора диаграмм. Раскрывающийся список Parent style позволяет выбрать подходящий родительский стиль для текущего выбранного элемента на диаграмме.

Ближайшая к нему кнопка справа (Revert to parent style) сбрасывает собственные настройки этого элемента, делая их равными стилю выбранного родителя.

Следующая кнопка Assign custom color (Установить пользовательский цвет) дает возможность выбрать определенный нестандартный цвет текущего элемента. Такая возможность полезна для выделения более важных или особенных элементов диаграммы.

Список доступных стилей представления можно редактировать. Для этого предназначен Менеджер стилей (Style Manager). Он вызывается командой Visual style ➤ Style manager (Стиль представления ➤ Менеджер стилей) локального меню диаграммы.

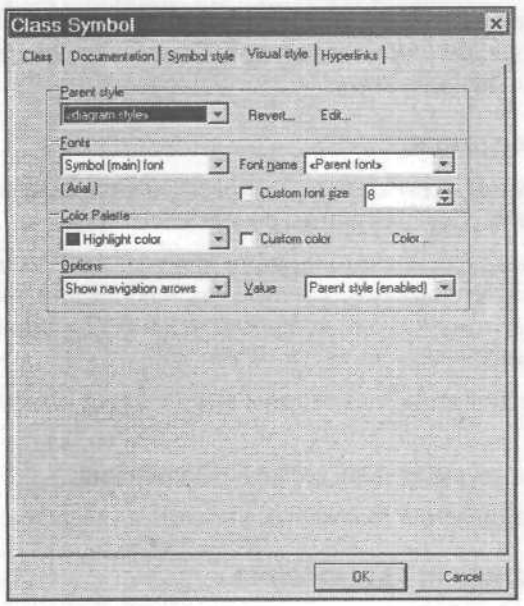


Рис. 13.26. Настройка стиля представления

Кнопка Add new style позволяет сформировать новый стиль на основе одного из существующих и разместить его в произвольном месте дерева стилей. Это дерево стилей в Менеджере стилей можно импортировать в *XML-файл* или экспортировать из *XML-файла*. Для этой цели используют кнопки Import styles и Export styles.

Печать диаграмм

Почти всегда диаграммы проекта выводятся на печать и добавляются к проектной документации. Они упрощают процесс общения с заказчиком, позволяя вести с ним диалог в рамках единой системы понятий. Поэтому все *CASE*-системы снабжены удобными и гибкими средствами настройки печати диаграмм. В *ModelMart*, в частности, имеется возможность создания стилей печати по аналогии со стилями представления.

Надо учитывать, что при печати немало нужных деталей оформления может быть потеряно. Так, на черно-белом принтере будет недоступна возможность выделения сущностей модели разными цветами. Кроме того, не печатаются разные мелкие значки (символы раскрытия вложенных определений, якоря и т. д.). В любой момент можно посмотреть, как будет выглядеть диаграмма на листе бумаги, нажав кнопку Use Printing Style (Использовать стиль представления, принятый для печати). Окно редактора диаграмм сразу же примет новый вид (рис. 13.27).

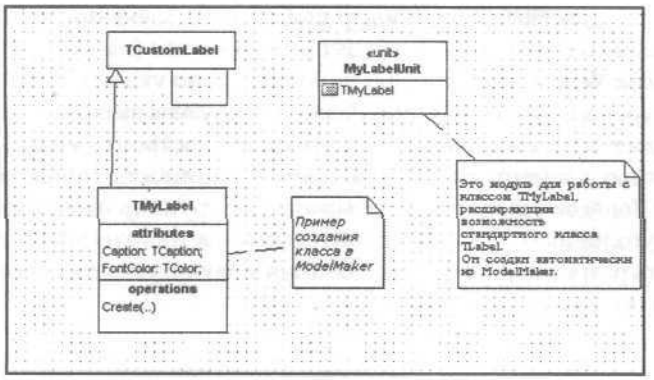


Рис. 13.27. *Вид окна редактора диаграмм, подготовленного для печати*

Стили содержимого

Стили содержимого по аналогии со стилями представления позволяют определять, как будут отображаться различные элементы диаграммы на экране. Однако они не задают способ описания внешнего вида, а определяют, какие составляющие конкретной сущности будут отображаться. Эти стили также можно объединять в иерархию и наследовать. Однако есть и отличие: стилям содержимого нельзя давать имена.

Настройки конкретного стиля выполняются в диалоговом окне свойств некоторого элемента диаграммы на закладке Symbol style (рис. 13.28).

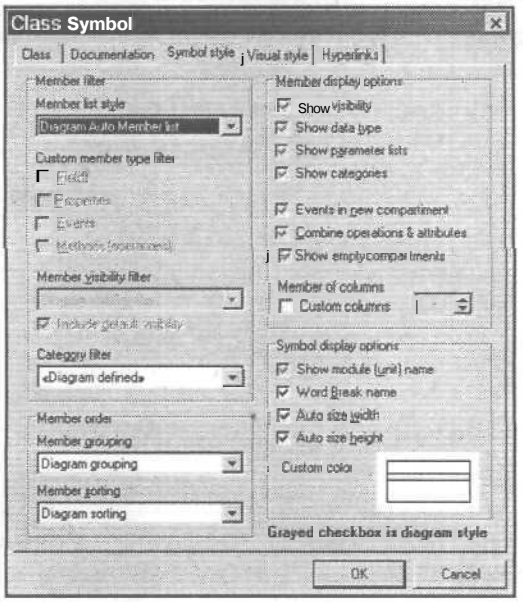


Рис. ***13.28.*** *Редактирование стиля содержимого*

Раздел Class symbol member filter (Фильтр отображения элементов класса) позволяет определить, какие элементы класса будут отображаться (поля, свойства, события, методы). Список Member visibility filter дополнительно указывает допустимый для отображения уровень инкапсуляции. Раздел Class symbol member display options (Настройки отображения элементов класса) позволяет настроить видимые составные части конкретных элементов класса, например видимость типов данных. Раздел Member order (Порядок элементов) указывает, как группировать элементы класса внутри прямоугольника класса. Дополнительные настройки (Symbol display options) позволяют дать разрешение на показ названия модуля, указать способ выравнивания размера и пользовательский цвет.

Что нового мы узнали?

В этом уроке мы научились

- использовать *ModelMart* для проектирования классов *Delphi*;
- применять средство автоматической генерации исходных текстов;
- организовать документирование проекта;
- работать с *UML*-диаграммами;
- импортировать проекты *Delphi в ModelMart;*
- настраивать стили представления и содержимого диаграмм.

Словарь сокращений

ActiveX

Технологический стандарт компании *Microsoft,* описывающий структуру платформонезависимых программных объектов. Первоначально разработан для использования в Интернете и ориентировался на взаимодействие объектов с приложениями *Microsoft.* Является расширением стандарта *COM,* В настоящее время поддерживается многими разработчиками программного обеспечения.

ADO

ActiveX Data Object — технология досгупа к данным, основанная на технологии *ActiveX.* Используется при создании одно- или двухуровневых приложений.

ANSI

American National Standard Institute — Национальный институт стандартизации США. Один из известных международных разработчиков систем кодирования данных. В частности, ввел в действие систему кодирования *ASCII.*

API

Application Program Interface — стандартизированный программный интерфейс, с помощью которого можно обращаться к внутренним функциям программы из других приложений.

ASCII

American Standard Code for InformationInterchange — американский стандарт кодирования данных для информационного обмена. Система кодирования, закрепляющая первые 128 стандартных двоичных кодов (от 0 до 127) за символами английского алфавита, цифрами, знаками препинания, знаками арифметических действий и некоторыми стандартными числовыми командами управления электронными устройствами. Введен в действие институтом *ANSI.*

ASP

Active Server Pages — одна из технологий создания динамически обновляемых *Web*-страниц. Обновление происходит за счет включения в *Web*-страницу сценариев на языках типа *Java,* выполняющихся на стороне сервера.

AVI

Audio Video Interleaved — один из стандартных форматов представления мультимедийных данных (звук и видео). Поддерживается большинством мультимедийных приложений.

BDE

Borland Database Engine — «фирменный» механизм связи между программным приложением и источником данных (базой данных), разработанный компанией *Borland.* Механизм реализован в виде набора библиотек *BDE x.x.* Для программ, написанных на языке *Delphi (Object Pascal),* он обеспечивает простой и удобный доступ к различным базам данных, независимо от их архитектуры.

CASE

Computer Aided System/Software Engineering — концепция применения специализированного инструментального программного обеспечения для повышения эффективности работы программистов. *CASE*-системы не только предоставляют средства для автоматизации проектирования и разработки ПО, но и средства для эффективного управления деятельностью участников проекта.

CGI

Common Gateway Interface — стандартный метод взаимодействия *Web*-модулей с *Web*-сервером. Предназначен для расширения функциональности Web-серверов.

COM

Component Object Model — технологический стандарт, описывающий структуру объекта и способы взаимодействия объектов друг с другом и с приложениями в среде *Windows. COM*-объект напоминает компонент системы программирования *Delphi* и представляет собой законченный объект со своими свойствами и методами, который может встраиваться в приложения или распространяться как отдельный программный продукт.

COM+

Расширение технологии *COM,* позволяющее взаимодействовать объектам, расположенным на разных компьютерах.

CORBA

Common Object Request Broker Architecture — открытая технология поддержки взаимодействия распределенных приложений, работающих на разных (в отличие от технологии *DCOM)* компьютерных платформах.

DCOM

Distributed Component Object Model — расширение технологического стандарта *COM,* предназначенное для создания распределенных приложений, работающих на платформе *Windows,*

DDE

Dynamic Data Exchange — устаревшая технология динамического обмена данными между приложениями, реализованная в 16-разрядных версиях *Windows.* Технология основана на обмене информационными сообщениями (символьными строками) между приложениями-серверами и приложениями-клиентами. Продолжает поддерживаться в современных системах программирования.

DICT

Dictionary Server Protocol — протокол словарного поиска, поддерживает запросы и ответы к словарным базам данных на естественном языке.

DLL

Dynamic Linking Lybrary — динамически подключаемые библиотеки. Технология совместного использования различными приложениями общих ресурсов (данных, программного кода) путем динамического подключения библиотек, в которых они хранятся.

DNS

Domain Name Service — служба доменных имен. Служба Интернета, выполняющая преобразование символьной формы записи адресов Интернета в числовую форму (в форму *IP*-адреса).

DOM

Document Object Model — объектная модель документа. Определяет методы редактирования документа, представленного в формате *XML*.

FTP

File Transfer Protocol — протокол передачи файлов. Основа одноименной службы Интернета.

HTML

HyperText Markup Language — язык разметки гипертекста. Позволяет функционально помечать элементы документов специальными тегами, определяющими способ форматирования документа для конкретного устройства вывода. Используется в качестве базового средства простейшего оформления документов, представляемых в службе *WWW* Интернета (*Web*-страниц).

HTTP

HyperText TransferProtocol — протокол передачи гипертекста. Основополагающий протокол взаимодействия клиентских и серверных приложений в Интернет-службе *World Wide Web*. Согласно модели взаимодействия открытых систем *OSI* относится к прикладному уровню.

ICMP

Internet Control Message Protocol — протокол управляющих сообщений. Служебный протокол Интернета, используемый для диагностики качества работы удаленных соединений.

IDL

Interface Definition Language — платформонезависимый язык описания интерфейсов, предназначенный для обеспечения взаимодействия приложений, в том числе и в распределенной вычислительной среде.

IIS

Internet Information Server — серверное приложение для Интернета от компании *Microsoft*. *Web*-сервер, работающий на платформе *Windows*.

IMAP4

Internet Messages Access Protocol — протокол доступа к сообщениям электронной почты в Интернете. Используется для получения клиентом сообщений электронной почты с почтового сервера. Во многом аналогичен протоколу *POP3,* но, в отличие от него, допускает расширенные возможности для клиента по управлению почтовой базой сервера. В частности, позволяет анализировать сообщения электронной почты, находящиеся на стороне сервера, и выполнять с ними определенные действия (фильтровать, удалять и т. п.) без загрузки на клиентский компьютер.

IP

Internet Protocol — протокол межсетевого взаимодействия. Один из двух основных протоколов Интернета *(см. также TCP).* Согласно модели взаимодействия открытых систем *(см. OSI),* рекомендованной Международным институтом стандартизации (си. *ISO),* относится к сетевому уровню взаимодействия. Определяет единый принцип адресации в межсетевом пространстве.

IRC

Internet Relay Chat — один из прикладных протоколов Интернета и название одноименной службы. Предназначен для двустороннего обмена краткими сообщениями между пользователями в режиме реального времени.

ISAPI

Internet Server Application Program Interface —способ организации взаимодействия между Wei-сервером и его модулями, альтернативный *CGI.* Характерное отличие заключается в том, что если несколько клиентов одновременно обращаются к одному и тому же приложению *CGI,* на стороне сервера запускаются несколько параллельных процессов (каждый в своем пространстве адресов), что ведет к общему снижению производительности. Приложения *ISAPI* имеют характер динамических библиотек *(см. DLL),* которые передаются клиенту и работают на его оборудовании, в адресном пространстве его браузера.

ISO

International Standard Organisation — Международный институт стандартизации. Стандарты, утвержденные этой организацией, имеют характер рекомендаций для разработчиков программных и технических систем.

ITE

Integrated Translation Environment — интегрированная оболочка перевода. Служебное программное средство, встроенное в систему *Delphi,* предназначенное для обслуживания работ по переводу информационных ресурсов разрабатываемых приложений на другие национальные языки.

MCI

Media Control Interface — платформонезависимая спецификация требований к мультимедийному программному обеспечению, опубликованная компанией *Microsoft* в 1990 г. Обеспечивает единые требования к функционированию программ, управляющих работой мультимедийного оборудования или воспроизводящих мультимедийные данные.

MIDAS

Miltitier Distributed Application Service Suite — набор служб поддержки распределенных многоуровневых приложений. Технология компании *Borland,* позволяющая, в частности, создавать клиентские приложения, предназначенные для работы с удаленными серверами баз данных через посредство промежуточного сервера приложений. В данном случае клиентское приложение может не иметь инструментальных средств для работы с базами данных — их предоставит удаленный сервер приложений.

MIME

Multipurpose Internet Mail Extensions — многоцелевой стандарт расширений почты Интернета. Стандарт выполняет следующие функции: связывает тип данных, хранящихся в файле, с расширением имени файла; предоставляет единый универсальный метод кодирования произвольных данных, вложенных в сообщения электронной почты; определяет способ представления специальных символов, которые могут встречаться в сообщениях электронной почты. Поддерживается практически всеми современными коммуникационными приложениями.

MMX

MultiMedia extensions — расширение системного набора команд для процессоров *Pentium,* введенное компанией *Intel* ориентированное на более эффективную обработку данных, характерных для мультимедийных источников (графика, видео, звук).

MTS

Microsoft Transaction Server — сервер транзакций. Программное приложение серверного типа, разработанное компанией *Microsoft* и предназначенное для обслуживания и синхронизации приложений, совместно выполняющих распределенные вычисления по технологии *DCOM*.

NDR

Network Data Representation — сетевое представление данных, используемое в ряде расширенных протоколов передачи данных между объектами.

NNTP

NetworkNews TransferProtocol — базовый протокол прикладного уровня для функционирования службы телеконференций (*Usenet*) в Интернете.

NSAPI

Netscape Server Application Program Interface —программный интерфейс, используемый при создании коммуникационных приложений, работающих совместно с серверами *Netscape Web Servers*.

ODBC

Open Database Connectivity Interface — стандартный протокол (интерфейс), определяющий принципы взаимодействия приложений с базами данных.

OLAP

On-Line Analytical Processing — механизм взаимодействия клиентского и серверного программного обеспечения с целью проведения многомерного анализа данных (и принятия решений), поступающих из распределенных источников.

OLE

Object Linking and Embedding — «фирменная» технология обмена данными между приложениями *(технология внедрения и связывания объектов),* реализованная компанией *Microsoft в* своих операционных системах и приложениях. Широко поддержана другими разрабочиками программного обеспечения.

OSI

Open System Interconnections — модель взаимодействия открытых систем. Модель анализа архитектуры компьютерных сетей, предусматривающая раздельное рассмотрение в них семи уровней от физического (низший уровень, связанный с физической линией связи) до прикладного (высший уровень, связанный с конкретным коммуникационным приложением). Модель рекомендована Международным институтом стандартизации /50.

OTS

Object Transaction Service — служба транзакций объектов. Программный комплекс, предназначенный для обслуживания и синхронизации приложений, совместно выполняющих распределенные вычисления по технологии *CORBA*.

PDF

Portable Document Format — компактный формат представления документов. Разработан компанией *Adobe Systems* для компактного хранения и эффективной транспортировки документов, имеющих полиграфическое качество. Характерная особенность формата — независимость от текущей конфигурации шрифтовых наборов на компьютере потребителя, поскольку все необходимые для воспроизведения на экране или принтере сведения об использованных шрифтах содержатся в самом документе.

POP3

Post Office Protocol — протокол почтового отделения, версия 3. Используется клиентом для получения сообщений электронной почты с почтового сервера. Наряду с протоколом *SMTP* является одним из основных протоколов службы электронной почты в Интернете. От протокола *SMTP* отличается тем, что проверяет права клиентского программного обеспечения на получение сообщений, хранящихся в почтовой базе сервера (*см. также IMAP4*).

RDS

Recordset Data Space — описание структуры набора данных, передаваемого между приложениями.

RIO

Remote Invokable Object — удаленно вызваемый объект. Динамически готовит функциональные возможности в зависимости от предоставленного интерфейса взаимодействия.

RSA

Один из алгоритмов асимметричной криптографии. В частности, может использоваться при формировании электронной цифровой подписи. Назван по первым буквам фамилий разработчиков: *Ron Rivest, Adi Shamir, Leonard Adleman.*

RFC

Request for Comments — категория формальных документов, в которых описываются технические спецификации протоколов, концепций, принципов организации служб Интернета. Имеют характер технических предложений. Публикуются в Интернете заинтересованными сторонами. Некоторые *RFC* играют чисто информационную роль, в то время как отдельные *RFC* по результатам общественного обсуждения и после внесения изменений и дополнений фактически приобретают черты промышленных стандартов.

RTF

Rich TextFormat — формат «обогащенного» текста. Общепринятый формат представления текста, сохраняющий простейшие элементы форматирования, параметры печатной страницы и параметры взаимодействия текста со встроенными иллюстрациями.

SDK

Software Development Kit — программный пакет, предназначенный для деятельности программистов, разрабатывающих приложения для заданной платформы. Обычно включает в себя библиотеки кода, инструментальные программные средства и документацию.

SNTP

Simple Network Time Protocol — упрощеная версия протокола *NTP*, предназначенного для точной (миллисекунды) синхронизации системного времени на компьютерах, входящих в сеть. Физически на клиентских компьютерах в фоновом режиме работает клиентская программа, периодически опрашивающая серверы времени и подстраивающая показания системных часов.

SMTP

Simple Mail Transfer Protocol —упрощенный протокол передачи сообщений электронной почты. Используется для передачи сообщений от клиента к серверу, а так-

же между серверами. Наряду с протоколом *POP3* является одним из основных протоколов службы электронной почты в Интернете. От протокола *POP3* отличается тем, что проверка прав клиентского программного обеспечения при подключении к серверу не обязательна.

SOAP

Simple ObjectAccess Protocol — высокоуровневый протокол, определяющий механизм взаимодействия приложений, в том числе и действующих на разных платформах, через Интернет, на базе *XML*.

SQL

Structured Query Language — высокоуровневый язык построения запросов *к* базам данных. Разработан компанией *IBM* и предназначен для создания запросов при работе с базами данных разных производителей на большинстве компьютерных платформ. Обеспечивает единый стандартный синтаксис формирования запросов.

TCP

Transfer Control Protocol — протокол управления передачей сообщений. Один из двух основных протоколов Интернета *(см. также IP).* Согласно модели взаимодействия открытых систем *OSI,* рекомендованной Международным институтом стандартизации *ISO,* относится к транспортному уровню взаимодействия. Является протоколом пакетной связи. Определяет требования к формированию транспортируемых пакетов и порядок восстановления сообщений из них.

UDP

User Datagram Protocol — в Интернете стандартный протокол пакетной передачи данных транспортного уровня (см. *OSI*). Альтернатива протоколу *TCP,* отличающаяся тем, что сервер не запрашивает у клиента квитанции, подтверждающей успешное получение пакетов, что дает возможность ускорить передачу данных с соответствующим снижением надежности передачи. Применяется для передачи данных потокового типа — звука, музыки, видео.

UML

Unified Modelling Language — универсальный язык моделирования, предназначенный для визуального построения графических моделей программных систем с целью их анализа и дальнейшего автоматического перевода в программный код конкретной среды разработки.

URL

Uniform Resource Locator — *унифицированный указатель ресурса*. Способ символьной адресации к информационным объектам, принятый в службах Интернета.

UUE

Unix to Unix Encoding — система кодирования, разработанная для обмена данными между приложениями операционной системы *Unix*. Система закрепляет один из возможных принципов представления произвольных октетов данных в семиразрядной форме. Используется, в частности, для пересылки файлов произвольного формата в качестве вложений в сообщения электронной почты. На платформе *IBM PC* используется в устаревших приложениях, не поддерживающих более совершенный стандарт *MIME*.

VCL

Visual Component Library — библиотека визуальных компонентов *Delphi*.

WSDL

Web Service Description Language — язык описания услуги *WebServices* в формате *XML*.

WWW

World Wide Web — наиболее развитая и известная служба Интернета, основанная на протоколе передачи гипертекста *HTTP*.

WYSIWYG

What You See Is What You Get — принцип полного соответствия графического образа документа на экране его печатной копии, выполненной на принтере. В дословном переводе: «Что видишь, то и получишь».

XML

Extendable Markup Language — расширяемый язык разметки документов. Отличается от языка *HTML* тем, что не имеет фиксированного набора тегов разметки. Создатель документа может создавать и описывать собственные теги для конкретных областей применения. Позволяет описать форматирование произвольного документа независимо от конкретного приложения (и его операционной системы), в котором он создан или воспроизводится.

Указатель компонентов

Бобровский Сергеи Игоревич

Delphi 7. Учебный курс

Главный редактор	*Е. Строганова*
Заведующий редакцией	*И. Корнеев*
Руководитель проекта	*С Симонович*
Научный редактор	*Г. Евсеев*
Литературный редактор	*В. Мураховский*
Художник	*Н. Биржаков*
Верстка	*А. Алексеев*

Лицензия ИД №05784 от 07.09.01.

Подписано к печати 25.09.03. Формат 70x100/16. Усл. п. л. 59,34.

Доп. тираж 5000. Заказ 348

ООО «Питер Принт», 196105, Санкт-Петербург, ул. Благодатная, д. 67в.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том *1;* 95 3005 — литература учебная.

Отпечатано с готовых диапозитивов в ФГУП ордена Трудового Красного Знамен н «Техническая книга» Министерства РФ по делам печати, телерадиовещания н средств массовых коммуникаций.

198005. Санкт-Петербург, Измайловский пр., 29.