

УЧЕБНИК
ДЛЯ ВУЗОВ

ПИТЕР®

Ю. Избачков В. Петров А. Васильев И. Телина



Информационные системы

3-е издание

Информационные системы
с использованием приложений Delphi

Методология проектирования
информационных систем

Описание стандарта SQL-92 ANSI

ДОПУЩЕНО
МИНИСТЕРСТВОМ ОБРАЗОВАНИЯ И НАУКИ РФ



Ю. Избачков В. Петров А. Васильев И. Телина

Информационные системы

3-е издание

Допущено Министерством образования и науки Российской Федерации
в качестве учебного пособия для студентов высших учебных заведений,
обучающихся по направлению подготовки дипломированных
специалистов «Информатика и вычислительная техника»



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

2011

ББК 32.973.233-018.2я7
УДК 004.6(075)
И74

Рецензенты:

Волков Ю. И., завкафедрой ЮНЕСКО Московского государственного института электронной техники, кандидат технических наук;
Чечурин С. Л., ректор Санкт-Петербургского института ЮНЕСКО по информационным технологиям в образовании при СПбГТУ.

Избачков Ю. С., Петров В. Н., Васильев А. А., Телина И. С.
И74 Информационные системы: Учебник для вузов. 3-е изд. — СПб.: Питер, 2011. — 544 с.: ил.

ISBN 978-5-49807-158-9

Учебник посвящен вопросам проектирования и разработки информационных систем. В нем рассматриваются современные методологии и технологии, применяемые при создании информационных систем, такие как RAD, CASE, COM, .NET, интернет-технологии, приводится подробное описание стандарта SQL-92 ANSI, излагаются теоретические сведения о реляционной модели данных, дается достаточно полное описание языков UML, SQL, HTML. В качестве инструментального средства разработки выбрана система объектно-ориентированного визуального программирования Delphi. В книге также рассмотрен ряд дополнительных вопросов: разработка справочной системы приложения, управление проектами приложений.

Допущено Министерством образования и науки Российской Федерации в качестве учебного пособия для студентов высших учебных заведений, обучающихся по направлению подготовки дипломированных специалистов «Информатика и вычислительная техника».

ББК 32.973.233-018.2я7
УДК 004.6(075)

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

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

ISBN 978-5-49807-158-9

© ООО Издательство «Питер», 2010

Краткое содержание

Введение	15
Часть I. Анализ и проектирование информационных систем	23
Глава 1. Информационные системы	24
Глава 2. Жизненный цикл информационных систем	41
Глава 3. Методология и технология разработки информационных систем	61
Глава 4. Реляционные базы данных	118
Глава 5. Управление реляционными базами данных	149
Глава 6. Проектирование структуры базы данных.	179
Часть II. Delphi — система быстрой разработки приложений	203
Глава 7. Delphi и объектно-ориентированное программирование	204
Глава 8. Средство быстрой разработки приложений Delphi	245
Глава 9. Компоненты для ввода и редактирования данных	273
Глава 10. Создание форм для ввода и редактирования данных	313
Часть III. Выборка данных	339
Глава 11. Выборка данных	340
Часть IV. Компоновка приложения и управление проектом	379
Глава 12. Система меню и панель инструментов приложения	380
Глава 13. Управление проектом и создание приложения	393
Глава 14. Коллективная разработка приложений	408
Глава 15. Справочная система приложения	431
Часть V. Программирование для Интернета	469
Глава 16. Особенности интернет-приложений	470
Глава 17. Разработка интернет-приложений	496
Алфавитный указатель.	522

Содержание

Введение	15
--------------------	----

Часть I. Анализ и проектирование информационных систем 23

Глава 1. Информационные системы 24

Основные понятия	24
----------------------------	----

Факторы, влияющие на развитие корпоративных информационных систем	24
---	----

Основные составляющие корпоративных информационных систем	26
---	----

Соотношение между составляющими информационной системы	26
--	----

Классификация информационных систем	28
---	----

Классификация по сфере применения	29
---	----

Классификация по способу организации	30
--	----

Области применения и примеры реализации информационных систем	35
---	----

Бухгалтерский учет	36
------------------------------	----

Управление финансовыми потоками	36
---	----

Управление складом, ассортиментом, закупками	36
--	----

Управление производственным процессом	36
---	----

Управление маркетингом	37
----------------------------------	----

Документооборот	37
---------------------------	----

Оперативное управление предприятием	37
---	----

Предоставление информации о фирме	37
---	----

Требования, предъявляемые к информационным системам	38
---	----

Гибкость	38
--------------------	----

Надежность	38
----------------------	----

Эффективность	39
-------------------------	----

Безопасность	40
------------------------	----

Глава 2. Жизненный цикл информационных систем 41

Общие сведения об управлении проектами	42
--	----

Понятие проекта	42
---------------------------	----

Классификация проектов	43
----------------------------------	----

Основные фазы проектирования информационной системы	44
---	----

Процессы, протекающие на протяжении жизненного цикла информационной системы	47
Основные процессы жизненного цикла	47
Вспомогательные процессы	49
Организационные процессы	49
Структура жизненного цикла информационной системы	50
Начальная стадия	50
Стадия уточнения	51
Стадия конструирования	51
Стадия перехода	51
Модели жизненного цикла информационной системы	51
Каскадная модель жизненного цикла информационной системы	52
Спиральная модель жизненного цикла	58

Глава 3. Методология и технология разработки информационных систем 61

Методология RAD — Rapid Application Development	63
Основные особенности методологии RAD	63
Объектно-ориентированный подход	64
Визуальное программирование	66
Событийное программирование	67
Фазы жизненного цикла в рамках методологии RAD	67
Фаза анализа и планирования требований	67
Фаза проектирования	68
Фаза построения	69
Фаза внедрения	70
Ограничения методологии RAD	70
Профили открытых информационных систем	71
Понятие профиля информационной системы	71
Принципы формирования профиля информационной системы	72
Структура профилей информационных систем	74
Стандарты и методики	78
Виды стандартов	79
Методика Oracle CDM	80
Международный стандарт ISO/IEC 12207: 1995-08-01	83
UML — универсальный язык моделирования	87

Глава 4. Реляционные базы данных 118

Базы данных: основные сведения	119
Основные функции СУБД	119
Эволюция систем управления базами данных	124
Реляционная модель данных	127
Базовые понятия реляционной модели данных	129
Связанные отношения	135
Основные свойства отношений	138

Реляционная система управления базами данных	139
Нормализация данных.	142
Цели нормализации	143
Нормальные формы	144
Глава 5. Управление реляционными базами данных	149
Краткая история языка SQL	149
Типы команд SQL	150
Типы данных SQL/92	151
Строковые типы.	151
Числовые типы	152
Типы для представления даты и времени	153
Управление объектами базы данных	154
Создание, модификация и удаление таблиц	154
Задание ограничений	156
Задание значений по умолчанию	163
Создание и удаление индексов	163
Работа с представлениями	165
Хранимые процедуры	168
Триггеры	170
Манипулирование данными	171
Добавление в таблицу новой информации	171
Изменение данных, хранящихся в таблице	173
Управление безопасностью базы данных	175
Привилегии пользователей.	175
Управление доступом к базе данных	176
Глава 6. Проектирование структуры базы данных	179
Концептуальное моделирование структуры данных.	179
Концептуальные модели данных.	180
Модель «сущность–связь»	181
Создание физической модели.	182
Общие сведения о CASE-средствах	183
Основные возможности CASE-средств.	183
Создание модели информационной системы	185
База данных Премьер	185
Создание новой модели базы данных в DBDesigner	188
Создание таблицы	189
Определение связей между таблицами	193
Документирование модели базы данных	197
Создание структуры базы данных.	198
Модификация структуры базы данных	200

Часть II. Delphi — система быстрой разработки приложений 203

Глава 7. Delphi и объектно-ориентированное программирование. 204

Основы языка Delphi	205
Структура программы в Delphi	205
Заголовок программы	206
Раздел объявления модулей	206
Раздел объявления меток	206
Раздел описания типов	206
Раздел переменных	207
Раздел констант.	207
Типы данных в Delphi	209
Простые типы	209
Структурные типы	213
Указательные типы	218
Вариантные типы	219
Операторы языка Delphi.	221
Оператор присваивания.	221
Оператор безусловного перехода.	221
Условный оператор.	222
Операторы цикла	223
Составной оператор	224
Процедуры и функции	225
Процедуры.	225
Функции	228
Модули Delphi.	229
Основы объектно-ориентированного программирования	230
Основные понятия и отличительные черты ООП	231
Основные концепции ООП	232
Поля, свойства и методы	235
Вложенные типы данных	241
Области видимости	242
Обработка исключительных ситуаций	243

Глава 8. Средство быстрой разработки приложений 245

Средства визуального программирования	245
Среда разработки Delphi	246
Главное окно Delphi IDE	248
Главное меню	249
Инспектор объектов	260
Редактор форм	261
Основные компоненты Delphi. Построение простых приложений.	262
Библиотека визуальных компонентов	262

Основные компоненты для построения простых приложений	263
Объединение элементов управления	271

Глава 9. Компоненты для ввода и редактирования

данных	273
Стандартные компоненты Delphi для ввода и редактирования данных	273
Многострочные текстовые поля	274
Списки	274
Комбинированные списки	275
Изображения	275
Стандартные окна диалога Delphi	276
Окна диалога для работы с файлами	276
Окно диалога для установки и настройки шрифтов	279
Окно диалога для выбора цвета	279
Окна диалога для работы с принтером	280
Работа с базами данных в Delphi	281
Доступ к данным с использованием dbGo	281
Доступ к данным с использованием dbExpress	291
Доступ к данным с использованием BDE	295
Работа с полями	302
Подключение базы данных к BDE	305
Компоненты Delphi для отображения и редактирования данных	306
Класс TDataSource	306
Модули данных	307
Класс TDBGrid	308
Компоненты для доступа к отдельным полям	309
Навигация по набору данных	311

Глава 10. Создание форм для ввода и редактирования

данных.	313
Формы в Delphi	313
Свойства класса TForm	314
Фреймы	318
Использование базовых классов для создания форм ввода	318
Размещение и удаление элементов управления	318
Выравнивание компонентов на форме	319
Изменение размеров и перемещение компонентов	320
Порядок обхода элементов	321
Настройка внешнего вида формы	322
Простые формы для ввода данных	322
Пример создания простой формы.	322
Табличные формы	327
Формы со вкладками.	333
Работа с многотабличными базами данных	336
Пример приложения со связанными таблицами	338

Часть III. Выборка данных 339**Глава 11. Выборка данных 340**

Использование SQL для выборки данных из таблицы 340

Компоненты Delphi, работающие с базами данных через

SQL-запросы 341

Компонент TADOQuery. 341

Пример использования компонентов доступа к данным через

SQL-запросы. 342

Язык запросов DQL 345

Простейшая форма оператора SELECT 346

Задание условий при выборке данных 347

Упорядочение данных 355

Использование вычисляемых полей 356

Псевдонимы полей 358

Функции агрегирования 359

Группировка данных 360

Выборка данных из нескольких таблиц 364

Подзапросы 368

Объединение запросов 369

Оператор UNION 370

Оператор UNION ALL 370

Упорядочение и группировка данных в составных запросах 371

Работа с представлениями данных 371

Создание представлений 372

Удаление представлений 373

Использование параметров в SQL-запросах 374

Часть IV. Компоновка приложения и управление проектом. 379**Глава 12. Система меню и панель инструментов приложения 380**

Планирование приложения. 380

Создание главного меню 381

Класс TMenuItem 383

Работа с редактором меню 384

Задание реакции на выбор команды меню 386

Создание контекстного меню 388

Панель инструментов 388

Класс TToolBar 388

Класс TToolButton. 389

Задание реакции на нажатие кнопки 391

Контейнеры для панелей инструментов 392

Глава 13. Управление проектом и создание приложения . 393

Структура проекта	393
Модуль формы проекта	394
Главный файл проекта	395
Файл описания формы проекта	395
Добавление к проекту форм и модулей	397
Класс TApplication	398
Управление формами проекта	399
Работа с группой проектов	401
Создание группы проектов	401
Управление группой проектов	402
Настройка параметров проекта	403
Вкладка Application	403
Вкладка Compiler	404
Вкладка Linker	405
Компиляция и запуск приложения	406
Команды компиляции проекта	406
Команды запуска приложения	407

Глава 14. Коллективная разработка приложений 408

Структура средств коллективного проектирования и решаемые ими задачи	408
Идентификация	409
Хранилище файлов и контроль за изменением файлов	409
Блокировки	410
Последовательность работы с PVCS	410
Программа TeamSource	411
Структура системы TeamSource	411
Идентификация проекта и его составляющих в TeamSource	411
Хранилище TeamSource	412
Работа с программой TeamSource	412
Первый запуск TeamSource	412
Настройка параметров программы TeamSource	413
Создание проекта	415
Настройка параметров проекта	418
Работа с проектом TeamSource	421
Использование закладок	429
Сборка проекта	430

Глава 15. Справочная система приложения 431

Основные компоненты справочной системы	431
Создание всплывающих подсказок	432
Создание строки состояния приложения	434
Создание файла справки в формате WinHelp 4	435

Основные элементы справочной системы WinHelp 4	435
Создание файла справки	438
Разработка текстов тем справочной системы.	438
Компиляция файла справки	444
Создание файла справки в формате HTML Help	452
Основные элементы HTML Help	452
Создание файла справки в формате HTML	453
Компиляция и тестирование файла справки	462
Использование справочной системы в приложениях	462
Подключение к приложению справочных файлов формата WinHelp	462
Использование в приложениях Delphi справочной системы HTML Help	465

Часть V. Программирование для Интернета 469

Глава 16. Особенности интернет-приложений 470

Основные сведения об Интернет	470
Протокол IP как основа Интернет	470
Многоуровневая сетевая модель	471
Уровень сетевого доступа	472
Межсетевой уровень.	472
Транспортный уровень	473
Уровень приложений.	473
Адресация в Интернет.	474
Доменная система имен	474
Порты и службы.	475
Унифицированный указатель ресурсов	475
Основы веб-программирования.	476
Основные понятия и термины	476
Веб-дизайн и веб-программирование	477
Протокол HTTP	478
Запрос клиента	479
Ответ сервера.	480
Язык HTML.	482
Структура HTML-документа.	483
Теги форматирования текста.	484
Гиперссылки.	487
Формы	488
Язык XML.	491
Типы веб-приложений.	492
CGI-сценарии	492
Расширения ISAPI	493
Доступ к базам данным с использованием Интернет	493

Глава 17. Разработка интернет-приложений	496
Разработка сценариев CGI	496
Запуск CGI-приложения	496
Простейшее CGI-приложение	497
Строка передаваемых параметров	499
Методы передачи и получения строки параметров	500
Использование специальных средств Delphi для разработки	
веб-приложений	505
Delphi Web Module	506
Компоненты для формирования ответа в формате HTML.	511
Компоненты для работы с базами данных	515
Алфавитный указатель	522

Введение

Программное обеспечение за полвека своего существования претерпело огромные изменения: от программ, способных выполнять только простейшие логические и арифметические операции, до сложных систем управления предприятиями. В развитии программного обеспечения всегда можно было выделить два основных направления:

- ❑ выполнение вычислений;
- ❑ накопление и обработка информации.

Хотя первоначально компьютеры предназначались главным образом для выполнения сложных математических расчетов (в первую очередь для расчетов, связанных с созданием ядерного оружия и ракетной техники), в настоящее время доминирующим является второе направление. Такое перераспределение основных функций, выполняемых вычислительной техникой, вполне понятно — гражданский бизнес гораздо более распространен, чем военные и научные вычисления, а снижение стоимости компьютеров сделало их доступными для совсем небольших предприятий и даже частных лиц.

Сегодня управление предприятием без компьютера просто невозможно. Компьютеры давно и прочно вошли в такие области управления, как бухгалтерский учет, управление складом, ассортиментом и закупками. Однако современный бизнес требует гораздо более широкого применения информационных технологий в управлении предприятием. Жизнеспособность и развитие информационных технологий объясняется тем, что современный бизнес крайне чувствителен к ошибкам в управлении. Интуиции, личного опыта руководителя и размеров капитала уже мало для того, чтобы быть первым. Для принятия любого грамотного управленческого решения в условиях неопределенности и риска необходимо постоянно держать под контролем различные аспекты финансово-хозяйственной деятельности, будь то: торговля, производство или предоставление каких-либо услуг. Поэтому современный подход к управлению предполагает вложение средств в информационные технологии. И чем крупнее предприятие, тем серьезнее должны быть подобные вложения. Они являются жизненной необходимостью — в жесткой конкурентной борьбе одержать победу сможет лишь тот, кто лучше оснащен и наиболее эффективно организован.

Информационные системы

Хотя информационные системы являются обычным программным продуктом, они имеют ряд существенных отличий от стандартных прикладных программ и систем.

В зависимости от предметной области информационные системы могут очень сильно различаться по своим функциям, архитектуре, реализации. Однако можно выделить ряд свойств, которые являются общими:

- ❑ информационные системы предназначены для сбора, хранения и обработки информации. Поэтому в основе любой из них лежит среда хранения и доступа к данным;
- ❑ информационные системы ориентируются на конечного пользователя, не обладающего высокой квалификацией в области применения вычислительной техники. Поэтому клиентские приложения информационной системы должны обладать простым, удобным, легко осваиваемым интерфейсом, который предоставляет конечному пользователю все необходимые для работы функции, но в то же время не дает ему возможность выполнять какие-либо лишние действия.

Таким образом, при разработке информационной системы приходится решать две основные задачи:

- ❑ задачу разработки базы данных, предназначенной для хранения информации;
- ❑ задачу разработки графического интерфейса пользователя клиентских приложений.

В данной книге рассматриваются оба аспекта разработки информационных систем, но большее внимание уделено второму.

База данных

Как уже отмечалось ранее, система управления базой данных (СУБД) является неотъемлемой частью любой информационной системы. Тип используемой СУБД обычно определяется масштабом такой системы — малые системы могут использовать локальные СУБД, в корпоративных же информационных системах потребуется мощная клиент-серверная СУБД, поддерживающая многопользовательскую работу.

В настоящее время наиболее широко распространены реляционные СУБД. Несмотря на очевидную привлекательность и растущую популярность объектно-ориентированных СУБД (ObjectStore, Objectivity, O2, Jasmin), пока все же преобладают реляционные базы данных, являющиеся хорошо отлаженными, развитыми, сопровождаемыми системами, поддерживающими стандарт SQL-92 (к таким системам относятся, например, Oracle, Informix, Sybase, DB2, MS SQL Server).

Традиционным методом организации информационных систем является двухзвенная архитектура клиент-сервер. В этом случае вся прикладная часть информационной системы размещается на рабочих станциях, а на стороне сервера осуществляется только доступ к базе данных. Чтобы разгрузить клиентскую рабочую станцию и уменьшить загрузку сети, применяются трехзвенные архитектуры клиент-сервер. В этой архитектуре кроме клиентской части системы и сервера базы данных вводится промежуточный сервер приложений. На

стороне клиента выполняются только интерфейсные действия, а вся логика обработки информации поддерживается на сервере приложений.

При разработке базы данных необходимо учитывать специфику той СУБД, для которой эта разработка проводится. Несмотря на существование стандартов ANSI SQL 92 и более поздних, практически все SQL-серверы используют свои реализации SQL, содержащие расширения стандарта. Тем не менее на начальном этапе, при разработке общей структуры базы данных (на уровне концептуальной модели), особенности используемой СУБД можно не учитывать.

CASE-средства

Первым шагом в проектировании информационной системы является получение формального описания предметной области, построение полных и непротиворечивых функциональных и информационных моделей информационной системы. Это логически сложная, трудоемкая и длительная по времени работа, требующая высокой квалификации участвующих в ней специалистов. Следует также учитывать, что в процессе создания и функционирования информационной системы потребности пользователей могут изменяться или уточняться, что еще более усложняет разработку и сопровождение таких систем. Модели информационных систем должны быть описаны средствами, понятными большинству участников проекта, как правило, с использованием универсального языка моделирования UML.

Указанные сложности способствовали появлению программно-технологических средств специального класса, так называемых CASE-средств, призванных повысить эффективность разработки программного обеспечения. Термин CASE (Computer Aided Software/System Engineering) используется в настоящее время в весьма широком смысле. Первоначальное значение термина CASE, ограниченное вопросами автоматизации разработки только лишь программного обеспечения, в настоящее время приобрело новый смысл, охватывающий процесс разработки сложных информационных систем в целом. В настоящее время под CASE-средствами понимаются программные средства, поддерживающие процессы создания и сопровождения информационных систем, включая анализ и формулировку требований, проектирование прикладного программного обеспечения и баз данных, генерацию кода, тестирование, документирование, обеспечение качества, конфигурационное управление и управление проектом, а также другие процессы.

Средства разработки

Еще один класс задач, решаемых при проектировании информационных систем, относится к созданию удобного и соответствующего целям информационной системы пользовательского интерфейса. Следует понимать, что задача эргономичности интерфейса не формализуется, но в то же время она является очень существенной. Пользователи часто судят о качестве системы в целом, исходя из качества ее интерфейса. Более того, от качества интерфейса зависит эффективность использования системы.

Разработка интерфейса всегда являлась трудоемкой задачей, отнимающей много времени у разработчиков. Однако в последние годы появились так называемые *средства визуальной разработки приложений*, в значительной мере упростившие задачу разработки графического интерфейса пользователя. Сейчас на рынке программных продуктов предлагается довольно много разнообразных средств визуальной разработки приложений, ориентированных на создание информационных систем. Все их можно условно разделить на два класса:

- специализированные средства — ориентированные исключительно на работу с вполне определенной СУБД и не предназначенные для разработки обычных приложений, не использующих базы данных. Примером средств такого рода может служить система Power Builder фирмы Sybase;
- универсальные средства, которые могут использоваться как для разработки информационных приложений, взаимодействующих с базами данных, так и для разработки любых других приложений, не использующих базы данных. Из таких средств наибольшей известностью пользуются системы Delphi фирмы Borland (CodeGear) и Visual Studio фирмы Microsoft.

Каждый из указанных классов имеет свои достоинства и недостатки, поэтому в общем случае трудно отдать предпочтение одному из них.

В предлагаемой книге в качестве средства разработки выбран продукт Borland Delphi, пользующийся большой популярностью в нашей стране. Delphi базируется на объектно-ориентированном языке Object Pascal, который наилучшим образом подходит для учебных целей вследствие своей строгости и простоты. Кроме того, в Object Pascal в полной мере реализованы все основные концепции объектно-ориентированного программирования.

ВНИМАНИЕ

В данном издании книги рассматривается бесплатная версия Turbo Delphi Explorer, которая совместима с Borland Developer Studio 2006, но имеет некоторые ограничения по функциональным возможностям.

Объектно-ориентированное программирование позволяет сделать любую систему более гибкой и динамичной, исключив необходимость в постоянном переписывании структуры базы данных и приложений.

Главное достоинство объектно-ориентированного проектирования заключается в возможности повторно использовать ранее написанный код. Кроме того, объектные системы несут в себе возможность модификации и развития. Применительно к базам данных это положение позволяет начать проектирование будущей системы, не имея исчерпывающего представления о предметной области. Поскольку получение детальной информации о предметной области — процесс весьма трудоемкий, то применение объектно-ориентированного подхода позволит сократить сроки и уменьшить стоимость разработки системы.

Для кого предназначена эта книга

Данная книга в первую очередь предназначена для начинающих программистов, не имеющих большого опыта разработки информационных систем. Ос-

новное внимание в книге уделяется вопросам разработки клиентской части информационных систем с использованием системы визуальной разработки приложений Borland Delphi. При этом обращается внимание на смещение акцентов в разработке информационных систем в сторону концептуального проектирования.

В книге содержится большое количество материала, посвященного вопросам разработки баз данных, в частности рассматриваются основные методологии проектирования информационных систем, приводится подробное описание стандарта ANSI SQL-92, излагаются теоретические сведения о реляционной модели данных. Таким образом, данную книгу можно рассматривать в качестве учебного пособия по информационным системам начального уровня.

Как составлена книга

Данная книга содержит семнадцать глав, которые сгруппированы в 5 частей.

Часть I. Анализ и проектирование информационных систем

В этой части книги (главы 1–6) излагаются базовые сведения об информационных системах предприятий и их проектировании. В первых трех главах приводится основная терминология и рассматриваются базовые понятия, знание которых необходимо для эффективного восприятия материала из последующих глав и других литературных источников. Далее рассматриваются вопросы проектирования и разработки одной из важнейших частей информационной системы — реляционной базы данных. В реляционных базах данных информация хранится в виде взаимосвязанных двумерных таблиц. Разработка структуры базы данных, обеспечивающей эффективный доступ к информации и ее обработку, в значительной степени определяет качество информационной системы в целом. Для упрощения процесса проектирования структуры базы данных и уменьшения времени разработки используются специальные программные средства проектирования, называемые CASE-средствами.

Каждая из представленных в этой части книги глав касается важных концептуальных понятий.

- ❑ **Глава 1, «Информационные системы».** В данной главе рассматриваются общие понятия и типы информационных систем, определяются их базовые свойства, а также формулируются задачи, решаемые при разработке таких систем, и проблемы, возникающие при их решении. Также обсуждаются наиболее типичные области применения информационных систем.
- ❑ **Глава 2, «Жизненный цикл информационных систем».** Здесь рассматриваются понятие жизненного цикла информационной системы и основные процессы, его сопровождающие. Также исследуются основные модели жизненного цикла информационных систем.
- ❑ **Глава 3, «Методология и технология разработки информационных систем».** В этой главе приводятся сведения о методологии быстрой разработки приложений — RAD (Rapid Application Development), рассматриваются фазы

жизненного цикла информационной системы в рамках методологии RAD. Приводятся сведения об основных международных и российских стандартах и методиках разработки информационных систем, в частности универсальном языке моделирования — стандарте описания информационных систем.

- **Глава 4, «Реляционные базы данных».** В этой главе приводятся основные сведения о реляционных базах данных. Рассматриваются важнейшие функции, выполняемые системами управления базами данных, дается краткая история их развития. Обсуждаются основы реляционной модели данных, нормальные формы данных и вопросы нормализации данных.
- **Глава 5, «Управление реляционными базами данных».** Здесь приводятся сведения о методах и средствах управления как информацией, хранящейся в базе данных, так и структурой самой базы данных. Рассматриваются средства языка управления базами данных SQL, предусмотренные стандартом ANSI 92.
- **Глава 6, «Проектирование структуры базы данных».** В данной главе рассматриваются понятия концептуальной и физической модели данных, а также средства анализа и проектирования баз данных (CASE-средства). Приводится пример разработки базы данных с использованием одного из наиболее популярных CASE-средств Power Designer.

Часть II. Delphi — система быстрой разработки приложений

Эта часть книги (главы 7–10) содержит базовые сведения об объектно-ориентированном и визуальном программировании — современном подходе к разработке приложений. Несмотря на то что основные концепции объектно-ориентированного программирования и первые объектно-ориентированные языки появились около 30 лет назад, сам подход оказался востребованным сравнительно недавно — в 90-х годах. Несколько позже появились средства визуальной разработки приложений, позволяющие быстро создавать графический интерфейс пользователя.

При разработке клиентских приложений информационных систем очень важным аспектом является создание удобного, интуитивно понятного интерфейса пользователя. А поскольку одной из основных функций клиентских приложений является ввод и редактирование данных, то следует обратить особое внимание на различные способы организации доступа к данным для их ввода и модификации. Особенностью здесь является тот факт, что редактируемые данные сохраняются в таблицах баз данных.

- **Глава 7, «Delphi и объектно-ориентированное программирование».** В этой главе излагаются основные концепции объектно-ориентированного программирования. Рассмотрение проводится на базе языка программирования Object Pascal, являющегося базовым языком системы визуальной разработки приложений Borland Delphi. В языке Object Pascal в полной мере реализованы все принципы объектно-ориентированного программирования. Строгость и ясность этого языка делают его идеальным для изучения концепций объ-

ектно-ориентированного программирования. В то же время этот язык обладает достаточной мощностью для разработки сложных приложений, в полной мере использующих все возможности операционной системы Windows.

- ❑ **Глава 8, «Средство быстрой разработки приложений Delphi».** Данная глава содержит начальные сведения о системе проектирования Delphi, а также подробное описание интегрированной среды системы визуальной разработки приложений Borland Delphi. Данный программный продукт пользуется заслуженной популярностью в России, сочетая в себе простоту и мощь.
- ❑ **Глава 9, «Компоненты для ввода и редактирования данных».** В этой главе рассматриваются компоненты для ввода и редактирования данных, входящие в стандартную библиотеку Borland Delphi.
- ❑ **Глава 10, «Создание форм для ввода и редактирования данных».** Данная глава является органическим продолжением предшествующей главы. Однако если в главе 9 рассматривались отдельные компоненты для ввода и редактирования данных, то здесь обсуждаются различные варианты их компоновки в формах, обеспечивающие наиболее эффективный и наглядный доступ к информации, хранящейся в базе данных.

Часть III. Выборка данных

Кроме редактирования данных, важной функцией клиентских приложений является также выполнение выборки хранящихся в базе данных по какому-либо критерию. Причем проблема не исчерпывается лишь выполнением выборки — полученные в результате выборки данные необходимо представить в удобном для пользователя виде. Рассмотрению этих задач — выборки данных и представления полученных результатов — и посвящена эта часть книги.

- ❑ **Глава 11, «Выборка данных».** В данной главе рассматриваются средства языка SQL, предназначенные для выполнения различного рода выборок данных из таблиц базы данных. Также здесь описываются компоненты библиотеки Borland Delphi, предназначенные для организации взаимодействия с базой данных с помощью операторов языка SQL.

Часть IV. Компоновка приложения и управление проектом

В предыдущих частях книги, посвященных созданию приложений, затрагивались лишь вопросы разработки отдельных фрагментов программ, выполняющих различные функции. В данной части книги рассматривается ряд вопросов, позволяющих придать приложению законченный вид. Кроме того, здесь обсуждаются вопросы организации коллективной разработки приложений, что может быть актуальным при выполнении сложных проектов.

- ❑ **Глава 12, «Система меню и панель инструментов приложения».** В этой главе рассматривается создание основных элементов интерфейса пользователя приложения — меню и панели инструментов.

- ❑ **Глава 13**, «Управление проектом и создание приложения». Здесь рассматривается структура проекта в Borland Delphi, основные свойства проекта, события компиляции и управления приложением.
- ❑ **Глава 14**, «Коллективная разработка приложений». Эта глава посвящена вопросам коллективной разработки приложений. Рассматриваются основные проблемы и принципы организации коллективной разработки приложений, а также средство контроля версий TeamSource, входящее в поставку Borland Delphi.
- ❑ **Глава 15**, «Справочная система приложения». В данной главе излагаются вопросы создания справочной системы приложения и организации ее взаимодействия с приложением, организации контекстно-зависимой справочной системы. Здесь вы познакомитесь с методами создания файлов справки как в формате WinHelp, так и в формате HTML Help.

Часть V. Программирование для Интернет

Глобальная сеть Интернет уже настолько прочно вошла в нашу жизнь, что публикация информации в WWW стала нормой, а не исключением. Поэтому организация взаимодействия информационной системы с веб-сервером является сейчас актуальной задачей.

- ❑ **Глава 16**, «Особенности интернет-приложений». В этой главе рассматриваются базовые технические особенности организации сети Интернет, а также основные понятия и термины веб-программирования. Излагаются основы протокола HTTP и языка разметки гипертекста HTML.
- ❑ **Глава 17**, «Разработка интернет-приложений». Здесь излагаются вопросы разработки веб-приложений в среде Borland Delphi. Особое внимание уделяется возможностям организации взаимодействия веб-сервера с системами управления базами данных.

От издательства

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

Мы будем рады узнать ваше мнение!

Файлы примеров вы можете найти на веб-сайте издательства на странице, посвященной этой книге.

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

Часть I

Анализ и проектирование информационных систем

Глава 1

Информационные системы

В данной главе рассматриваются общие понятия и типы информационных систем, определяют их базовые свойства, а также формулируются задачи, решаемые при разработке таких систем и возникающие при этом проблемы.

Основные понятия

Под информационной системой обычно понимается прикладная программная система, ориентированная на сбор, хранение, поиск и обработку текстовой и/или фактографической информации. Подавляющее большинство информационных систем работает в режиме диалога с пользователем.

В наиболее общем случае типовые программные компоненты, входящие в состав информационной системы, включают:

- ☐ диалоговый ввод-вывод;
- ☐ логику диалога;
- ☐ прикладную логику обработки данных;
- ☐ логику управления данными;
- ☐ операции манипулирования файлами и/или базами данных.

Корпоративной информационной системой (КИС) мы будем называть совокупность специализированного программного обеспечения и вычислительной аппаратной платформы, на которой установлено и настроено программное обеспечение.

Факторы, влияющие на развитие корпоративных информационных систем

В последнее время все больше руководителей начинают отчетливо осознавать важность построения на предприятии корпоративной информационной системы как необходимого инструментария для успешного управления бизнесом в современных условиях.

Можно выделить три наиболее важных фактора, существенно влияющих на развитие корпоративных информационных систем:

- ❑ развитие методик управления предприятием;
- ❑ развитие общих возможностей и производительности компьютерных систем;
- ❑ развитие подходов к технической и программной реализации элементов информационной системы.

Рассмотрим эти факторы более подробно.

Развитие методик управления предприятием

Теория управления предприятием представляет собой довольно обширный предмет для изучения и совершенствования. Это обусловлено широким спектром постоянных изменений ситуации на мировом рынке. Все время растущий уровень конкуренции вынуждает руководителей компаний искать новые методы сохранения своего присутствия на рынке и поддержания рентабельности своей деятельности. Такими методами могут быть диверсификация, децентрализация, управление качеством и многое другое. Современная информационная система должна отвечать всем нововведениям в теории и практике менеджмента. Несомненно, это самый главный фактор, так как построение продвинутой в техническом отношении системы, которая не отвечает требованиям по функциональности, не имеет смысла.

Развитие общих возможностей и производительности компьютерных систем

Прогресс в области наращивания мощности и производительности компьютерных систем, развитие сетевых технологий и систем передачи данных, широкие возможности интеграции компьютерной техники с самым разнообразным оборудованием позволяют постоянно наращивать производительность информационных систем и их функциональность.

Развитие подходов к технической и программной реализации элементов информационных систем

Параллельно с развитием аппаратной части информационных систем на протяжении последних лет происходит постоянный поиск новых, более удобных и универсальных методов их программно-технологической реализации. Можно выделить три наиболее существенных новшества, оказавших колоссальное влияние на развитие информационных систем в последние годы:

- ❑ *новый подход к программированию*: с начала 90-х годов объектно-ориентированное программирование фактически вытеснило модульное; до настоящего времени непрерывно совершенствуются методы построения объектных моделей. Благодаря внедрению объектно-ориентированных технологий программирования существенно сокращаются сроки разработки сложных информационных систем, упрощаются их поддержка и развитие;

- благодаря *развитию сетевых технологий* локальные информационные системы повсеместно вытесняются клиент-серверными и многоуровневыми реализациями;
- *развитие сети Интернет* открыло большие возможности работы с удаленными подразделениями, широкие перспективы электронной коммерции, обслуживания покупателей через Интернет и многое другое. Более того, определенные преимущества дает использование интернет-технологий в интрасетях предприятия (так называемые интранет-технологии).

ПРИМЕЧАНИЕ

Следует иметь в виду, что использование определенных технологий при построении информационных систем не является самоцелью разработчика. Выбор технологий должен производиться в зависимости от реальных потребностей.

Основные составляющие корпоративных информационных систем

В составе корпоративных информационных систем можно выделить две относительно независимые составляющие:

- *компьютерную инфраструктуру* организации, представляющую собой совокупность сетевой, телекоммуникационной, программной, информационной и организационной инфраструктур. Данная составляющая обычно называется *корпоративной сетью*;
- *взаимосвязанные функциональные подсистемы*, обеспечивающие решение задач организации и достижение ее целей.

Первая составляющая отражает системно-техническую, структурную сторону любой информационной системы. По сути, это основа для интеграции функциональных подсистем, полностью определяющая свойства информационной системы, характеризующие ее успешную эксплуатацию. Требования к компьютерной инфраструктуре едины и стандартизованы, а методы ее построения хорошо известны и многократно проверены на практике.

Вторая составляющая корпоративной информационной системы полностью относится к прикладной области и сильно зависит от специфики задач и целей предприятия. Данная составляющая полностью базируется на компьютерной инфраструктуре предприятия и определяет прикладную функциональность информационной системы. Требования к функциональным подсистемам сложны и зачастую противоречивы, так как выдвигаются специалистами из различных прикладных областей. Однако в конечном счете именно эта составляющая более важна для функционирования организации, так как для нее, собственно, и строится компьютерная инфраструктура.

Соотношение между составляющими информационной системы

Взаимосвязи между двумя указанными составляющими информационной системы достаточно сложны. С одной стороны, они в определенном смысле неза-

висимы. Например, организация сети и протоколы, используемые для обмена данными между компьютерами, абсолютно не зависят от того, какие методы и программы планируется использовать на предприятии для организации бухгалтерского учета.

С другой стороны, указанные составляющие в определенном смысле все же зависят друг от друга. Функциональные подсистемы в принципе не могут существовать без компьютерной инфраструктуры. В то же время компьютерная инфраструктура сама по себе достаточно ограничена, поскольку не обладает необходимой функциональностью. Невозможно эксплуатировать распределенную информационную систему при отсутствии сетевой инфраструктуры. С другой стороны, имея развитую инфраструктуру, можно предоставить сотрудникам организации ряд полезных общесистемных служб (например, электронную почту и доступ в Интернет), упрощающих работу и делающих ее более эффективной (в частности, за счет использования более развитых средств связи).

Таким образом, разработку информационной системы целесообразно начинать с построения компьютерной инфраструктуры (корпоративной сети) как наиболее важной составляющей, опирающейся на апробированные промышленные технологии и гарантированно реализуемой в разумные сроки в силу высокой степени определенности как в постановке задачи, так и в предлагаемых решениях.

ПРИМЕЧАНИЕ

Бессмысленно строить корпоративную сеть как некую самодостаточную систему, не принимая во внимание прикладную функциональность. Если в процессе создания системно-технической инфраструктуры не проводить анализ и автоматизацию управленческих задач, то средства, инвестированные в разработку корпоративной сети, не дадут впоследствии реальной отдачи.

Корпоративная сеть создается на многие годы вперед, капитальные затраты на ее разработку и внедрение настолько велики, что практически исключают возможность полной или частичной переделки существующей сети.

Функциональные подсистемы, в отличие от корпоративной сети, изменчивы по своей природе, так как в предметной области деятельности организации постоянно происходят более или менее существенные изменения. Функциональность информационных систем сильно зависит от организационно-управленческой структуры организации, распределения организационно-управленческих функций по подразделениям организации, принятых в организации финансовых технологий и схем, существующей практики документооборота и множества других факторов.

Разработку и внедрение функциональных подсистем можно выполнять постепенно. Например, сначала на наиболее важных и ответственных участках выполнять разработки, обеспечивающие прикладную функциональность системы (внедрять системы финансового учета, управления кадрами и т. п.), а затем распространять прикладные программные системы и на другие, первоначально менее значимые области управления предприятием.

Классификация информационных систем

Информационные системы классифицируются по разным признакам. Рассмотрим наиболее часто используемые способы классификации.

Классификация по масштабу

По масштабу информационные системы подразделяются на следующие группы (рис. 1.1):

- ☐ одиночные;
- ☐ групповые;
- ☐ корпоративные.

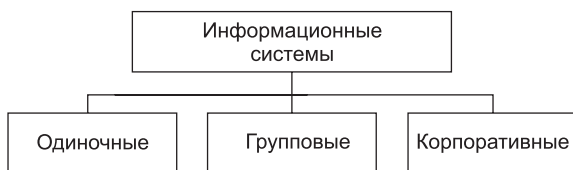


Рис. 1.1. Деление информационных систем по масштабу

Одиночные информационные системы

Одиночные информационные системы реализуются, как правило, на автономном персональном компьютере (сеть не используется). Такая система может содержать несколько простых приложений, связанных общим информационным фондом, и рассчитана на работу одного пользователя или группы пользователей, разделяющих по времени одно рабочее место. Подобные приложения создаются с помощью так называемых *настольных* или *локальных* систем управления базами данных (СУБД). Среди локальных СУБД наиболее известными являются Clarion, Clipper, FoxPro, Paradox, dBase и Microsoft Access.

Групповые информационные системы

Групповые информационные системы ориентированы на коллективное использование информации членами рабочей группы и чаще всего строятся на базе локальной вычислительной сети. При разработке таких приложений используются серверы баз данных (называемые также SQL-серверами) для рабочих групп. Существует довольно большое количество различных SQL-серверов, как коммерческих, так и свободно распространяемых. Среди них наиболее известны такие серверы баз данных, как Oracle, DB2, Microsoft SQL Server, InterBase, Sybase, Informix.

Корпоративные информационные системы

Корпоративные информационные системы являются развитием систем для рабочих групп, они ориентированы на крупные компании и могут поддерживать территориально разнесенные узлы или сети. В основном они имеют иерархическую структуру из нескольких уровней. Для таких систем характерна архитектура клиент-сервер со специализацией серверов или же многоуровневая архи-

тектура. При разработке таких систем могут использоваться те же серверы баз данных, что и при разработке групповых информационных систем. Однако в крупных информационных системах наибольшее распространение получили серверы Oracle, DB2 и Microsoft SQL Server.

Для групповых и корпоративных систем существенно повышаются требования к надежности функционирования и сохранности данных. Эти свойства обеспечиваются поддержкой целостности данных, ссылок и транзакций в серверах баз данных.

Классификация по сфере применения

По сфере применения информационные системы обычно подразделяются на четыре группы (рис. 1.2):

- ❑ системы обработки транзакций;
- ❑ системы принятия решений;
- ❑ информационно-справочные системы;
- ❑ офисные информационные системы.

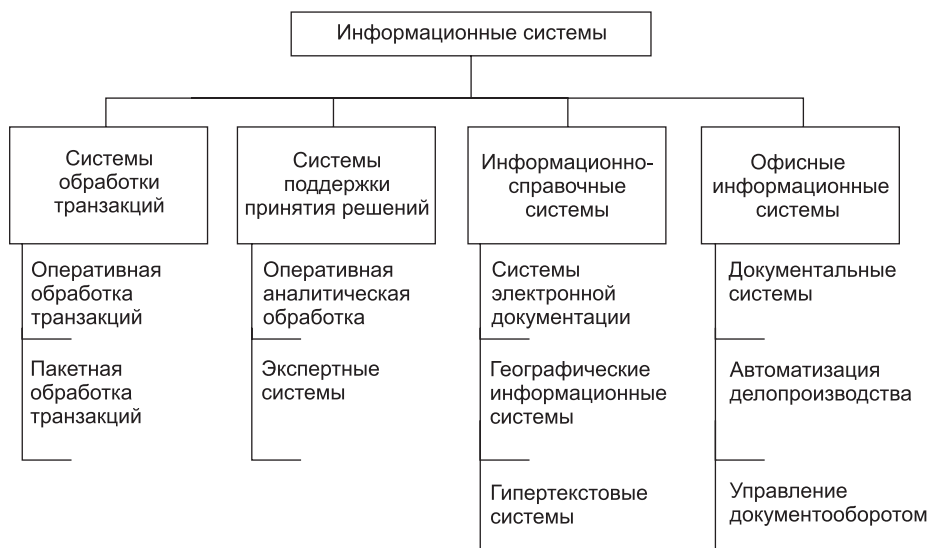


Рис. 1.2. Деление информационных систем по сфере применения

Системы обработки транзакций, в свою очередь, по оперативности обработки данных разделяются на пакетные информационные системы и оперативные информационные системы. В информационных системах организационного управления преобладает режим оперативной обработки транзакций — OLTP (OnLine Transaction Processing), для отражения актуального состояния предметной области в любой момент времени, а режим пакетной обработки используется достаточно редко. Для систем OLTP характерен регулярный (возможно

интенсивный) поток довольно простых транзакций, представляющих собой заказы, платежи, запросы и т. п. Важными требованиями для них являются:

- ❑ высокая производительность обработки транзакций;
- ❑ гарантированная доставка информации при удаленном доступе к БД по телекоммуникациям.

Системы поддержки принятия решений — DSS (Decision Support System) — представляют собой другой тип информационных систем, в которых с помощью довольно сложных запросов производится отбор и анализ данных в различных разрезах: временных, географических и по другим показателям.

Обширный класс *информационно-справочных систем* основан на гипертекстовых документах и мультимедиа. Наибольшее развитие такие информационные системы получили в сети Интернет.

Класс *офисных информационных систем* нацелен на перевод бумажных документов в электронный вид, автоматизацию делопроизводства и управление документооборотом.

ПРИМЕЧАНИЕ

Следует отметить, что приводимая классификация по сфере применения в достаточной степени условна. Крупные информационные системы очень часто обладают признаками всех перечисленных выше классов. Кроме того, корпоративные информационные системы масштаба предприятия обычно состоят из ряда подсистем, относящихся к различным сферам применения.

Классификация по способу организации

По способу организации групповые и корпоративные информационные системы подразделяются на следующие классы (рис. 1.3):

- ❑ системы на основе архитектуры файл-сервер;
- ❑ системы на основе архитектуры клиент-сервер;
- ❑ системы на основе многоуровневой архитектуры;
- ❑ системы на основе Интернет/интранет-технологий.

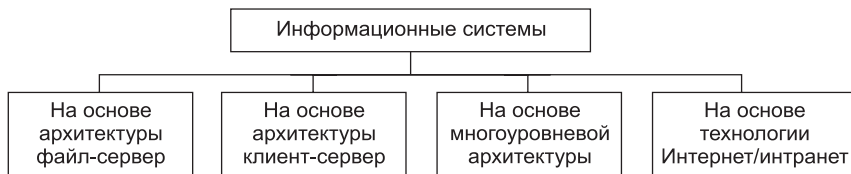


Рис. 1.3. Деление информационных систем по способу организации

В любой информационной системе можно выделить необходимые функциональные компоненты (табл. 1.1), которые помогают понять ограничения различных архитектур информационных систем. Рассмотрим более подробно особенности вариантов построения информационных приложений.

Таблица 1.1. Типовые функциональные компоненты информационной системы

Обозначение	Наименование	Характеристика
PS	Presentation Services (средства представления)	Обеспечиваются устройствами, принимающими ввод от пользователя и отображающими то, что сообщает ему компонент логики представления PL, с использованием соответствующей программной поддержки
PL	Presentation Logic (логика представления)	Управляет взаимодействием между пользователем и ЭВМ. Обработывает действия пользователя при выборе команды в меню, нажатии кнопки или выборе элемента из списка
BL	Business or Application Logic (прикладная логика)	Набор правил для принятия решений, вычислений и операций, которые должно выполнить приложение
DL	Data Logic (логика управления данными)	Операции с базой данных (SQL-операторы), которые нужно выполнить для реализации прикладной логики управления данными
DS	Data Services (операции с базой данных)	Действия СУБД, вызываемые для выполнения логики управления данными, такие как манипулирование данными, определения данных, фиксация или откат транзакций и т. п. СУБД обычно компилирует SQL-предложения
FS	File Services (файловые операции)	Дисковые операции чтения и записи данных для СУБД и других компонентов. Обычно являются функциями операционной системы (ОС)

Архитектура файл-сервер

Архитектура файл-сервер не имеет сетевого разделения компонентов диалога **PS** и **PL** и использует компьютер для функций отображения, что облегчает построение графического интерфейса. Файл-сервер только извлекает данные из файлов, так что дополнительные пользователи и приложения добавляют лишь незначительную нагрузку на центральный процессор. С добавлением каждого нового клиента растет общая вычислительная мощность сети.

Объектами разработки в файл-серверной системе являются компоненты приложения, определяющие логику диалога **PL**, а также логику обработки **BL** и управления данными **DL**. Разработанное приложение реализуется либо в виде законченного загрузочного модуля, либо в виде специального кода для интерпретации.

Однако такая архитектура имеет существенный недостаток: при выполнении некоторых запросов к базе данных клиенту могут передаваться большие объемы данных, загружающие сеть и приводящие к непредсказуемости времени реакции. Значительный сетевой трафик особенно сильно сказывается при организации удаленного доступа к базам данных на файл-сервере через низкоскоростные каналы связи. Одним из вариантов устранения данного недостатка является удаленное управление файл-серверным приложением в сети. При этом в локальной сети размещается сервер приложений, совмещенный с телекоммуникационным сервером (обычно называемым сервером доступа), в среде которого выполняются обычные файл-серверные приложения. Особенность состоит в том, что диалоговый ввод-вывод поступает от удаленных клиентов через

телекоммуникации. Приложения не должны быть слишком сложными, иначе велика вероятность перегрузки сервера, или же нужна очень мощная платформа для сервера приложений.

ПРИМЕЧАНИЕ

Одним из традиционных средств, на основе которых создаются файл-серверные системы, являются локальные СУБД. Однако такие системы, как правило, не отвечают требованиям обеспечения целостности данных (в частности, они не поддерживают транзакции). Поэтому при их использовании задача обеспечения целостности данных возлагается на программы клиентов, что приводит к усложнению клиентских приложений. Однако эти инструменты привлекают своей простотой, удобством использования и доступностью. Поэтому файл-серверные информационные системы до сих пор представляют интерес для малых рабочих групп и, более того, нередко используются в качестве информационных систем масштаба предприятия.

Архитектура клиент-сервер

Архитектура клиент-сервер предназначена для разрешения проблем файл-серверных приложений путем разделения компонентов приложения и размещения их там, где они будут функционировать наиболее эффективно. Особенностью архитектуры клиент-сервер является использование выделенных серверов баз данных, понимающих запросы на языке структурированных запросов SQL (Structured Query Language) и выполняющих поиск, сортировку и агрегирование информации.

Отличительная черта серверов БД — наличие справочника данных, в котором записана структура БД, ограничения целостности данных, форматы и даже серверные процедуры обработки данных по вызову или по событиям в программе. Объектами разработки в таких приложениях помимо диалога и логики обработки являются, прежде всего, реляционная модель данных и связанный с ней набор SQL-операторов для типовых запросов к базе данных.

Большинство конфигураций клиент-сервер использует двухуровневую модель, в которой клиент обращается к услугам сервера. Предполагается, что диалоговые компоненты **PS** и **PL** размещаются на клиенте, что позволяет обеспечить графический интерфейс. Компоненты управления данными **DS** и **FS** размещаются на сервере, а диалог (**PS, PL**), логика **BL** и **DL** — на клиенте. Двухуровневое определение архитектуры клиент-сервер использует именно этот вариант: приложение работает у клиента, СУБД — на сервере (рис. 1.4).

Поскольку эта схема предъявляет наименьшие требования к серверу, она обладает наилучшей масштабируемостью. Однако сложные приложения, вызывающие большое взаимодействие с БД, могут сильно загрузить как клиента, так и сеть. Результаты SQL-запроса должны вернуться клиенту для обработки, потому что там находится логика принятия решения. Такая схема приводит к дополнительному усложнению администрирования приложений, разбросанных по различным клиентским узлам.

Для сокращения нагрузки на сеть и упрощения администрирования приложений компонент **BL** можно разместить на сервере. При этом вся логика принятия решений оформляется в виде *хранимых процедур* и выполняется на сервере БД.

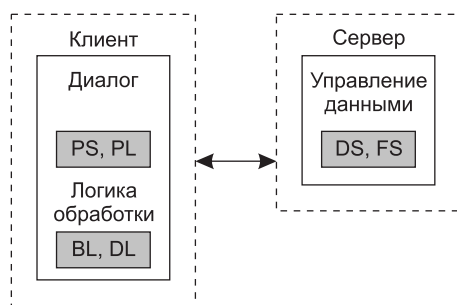


Рис. 1.4. Классический вариант клиент-серверной системы

Хранимая процедура — процедура с операторами SQL для доступа к БД, вызываемая по имени с передачей требуемых параметров и выполняемая на сервере БД. Хранимые процедуры могут компилироваться, что повышает скорость их выполнения и сокращает нагрузку на сервер.

Хранимые процедуры улучшают целостность приложений и БД, гарантируют актуальность коллективно используемых операций и вычислений. Улучшается сопровождение таких процедур, а также безопасность (нет прямого доступа к данным).

ПРИМЕЧАНИЕ

Следует помнить, что излишнее утяжеление хранимых процедур прикладной логикой может перегрузить сервер, что приведет к потере производительности. Эта проблема особенно актуальна при разработке крупных информационных систем, в которых к серверу может одновременно обращаться большое количество клиентов. Поэтому в большинстве случаев следует принимать компромиссные решения: часть логики приложения размещать на стороне сервера, часть — на стороне клиента. Такие клиент-серверные системы называются системами с разделенной логикой. Данная схема при удачном ее воплощении позволяет получить более сбалансированную загрузку клиентов и сервера, но при этом затрудняется сопровождение приложений.

Создание архитектуры клиент-сервер возможно и на основе многотерминальной системы. В этом случае в многозадачной среде сервера приложений выполняются программы пользователей, а клиентские узлы вырождены и представлены терминалами. Подобная схема информационной системы характерна для Unix.

В настоящее время архитектура клиент-сервер получила признание и широкое распространение как способ организации приложений для рабочих групп и информационных систем корпоративного уровня. Подобная организация работы повышает эффективность выполнения приложений за счет использования возможностей сервера БД, разгрузки сети и обеспечения контроля целостности данных.

Двухуровневые схемы архитектуры клиент-сервер могут привести к некоторым проблемам в сложных информационных приложениях с множеством

пользователей и запутанной логикой. Решением этих проблем может стать использование многоуровневой архитектуры.

Многоуровневая архитектура

Многоуровневая архитектура стала развитием архитектуры клиент-сервер и в своей классической форме состоит из трех уровней:

- нижний уровень представляет собой приложения клиентов, выделенные для выполнения функций и логики представлений **PS** и **PL** и имеющие программный интерфейс для вызова приложения на среднем уровне;
- средний уровень — это сервер приложений, на котором выполняется прикладная логика **BL** и с которого логика обработки данных **DL** вызывает операции с базой данных **DS**;
- на верхнем уровне находится удаленный специализированный сервер базы данных, выделенный для услуг обработки данных **DS** и файловых операций **FS** (без риска использования хранимых процедур).

Подобную концепцию обработки данных пропагандируют, в частности, фирмы Oracle, Sun, Borland и др.

Трехуровневая архитектура позволяет еще больше сбалансировать нагрузку на разные узлы и сеть, а также способствует специализации инструментов для разработки приложений и устраняет недостатки двухуровневой модели клиент-сервер.

Централизация логики приложения упрощает администрирование и сопровождение. Четко разделяются платформы и инструменты для реализации интерфейса и прикладной логики, что позволяет с наибольшей отдачей отдавать их специалистам узкого профиля. Наконец, изменения прикладной логики не затрагивают интерфейса, и наоборот. Но поскольку границы между компонентами **PL**, **BL** и **DL** размыты, прикладная логика может появиться на всех трех уровнях. Сервер приложений с помощью монитора транзакций обеспечивает интерфейс с клиентами и другими серверами, может управлять транзакциями и гарантировать целостность распределенной базы данных. Средства удаленного вызова процедур наиболее соответствуют идее распределенных вычислений: они обеспечивают из любого узла сети вызов прикладной процедуры, расположенной на другом узле, передачу параметров, удаленную обработку и возврат результатов.

С ростом сложности систем клиент-сервер необходимость трех уровней становится все более очевидной. Продукты для трехзвенной архитектуры, так называемые мониторы транзакций, являются относительно новыми. Эти инструменты в основном ориентированы на среду Unix, однако прикладные серверы можно строить на базе Microsoft Windows с использованием вызова удаленных процедур для организации связи клиентов с сервером приложений. На практике в локальной сети могут использоваться смешанные архитектуры (двухуровневые и трехуровневые) с одним и тем же сервером базы данных. С учетом глобальных связей архитектура может иметь больше трех звеньев. В настоящее время появились новые инструментальные средства для гибкой сегментации приложений клиент-сервер по различным узлам сети.

Таким образом, многоуровневая архитектура распределенных приложений позволяет повысить эффективность работы корпоративной информационной системы и оптимизировать распределение ее программно-аппаратных ресурсов. Но пока на российском рынке по-прежнему доминирует архитектура клиент-сервер.

Интернет/интранет-технологии

В развитии технологии Интернет/интранет основной акцент пока что делается на разработке инструментальных программных средств. В то же время наблюдается отсутствие развитых средств разработки приложений, работающих с базами данных. Компромиссным решением для создания удобных и простых в использовании и сопровождении информационных систем, эффективно работающих с базами данных, стало объединение Интернет/интранет-технологии с многоуровневой архитектурой. При этом структура информационного приложения приобретает следующий вид: браузер — сервер приложений — сервер баз данных — сервер динамических страниц — веб-сервер.

Благодаря интеграции Интернет/интранет-технологий и архитектуры клиент-сервер процесс внедрения и сопровождения корпоративной информационной системы существенно упрощается при сохранении достаточно высокой эффективности и простоты совместного использования информации.

Области применения и примеры реализации информационных систем

В последние несколько лет компьютер стал неотъемлемой частью управленческой системы предприятий. Современный подход к управлению предполагает вложение денег в информационные технологии. Причем чем крупнее предприятие, тем больше должны быть подобные вложения.

Благодаря стремительному развитию информационных технологий наблюдается расширение области их применения. Если раньше чуть ли не единственной областью, в которой применялись информационные системы, была автоматизация бухгалтерского учета, то сейчас наблюдается внедрение информационных технологий во множество других областей. Эффективное использование корпоративных информационных систем позволяет делать более точные прогнозы и избегать возможных ошибок в управлении.

Из любых данных и отчетов о работе предприятия можно извлечь массу полезных сведений. И информационные системы как раз и позволяют извлекать максимум пользы из всей имеющейся в компании информации.

Именно этим фактом и объясняются жизнеспособность и бурное развитие информационных технологий — современный бизнес крайне чувствителен к ошибкам в управлении, и для принятия грамотного управленческого решения в условиях неопределенности и риска необходимо постоянно держать под контролем различные аспекты финансово-хозяйственной деятельности предприятия (независимо от профиля его деятельности).

Поэтому можно вполне обоснованно утверждать, что в жесткой конкурентной борьбе большие шансы на победу имеет предприятие, использующее в управлении современные информационные технологии.

Рассмотрим наиболее важные задачи, решаемые с помощью специальных программных средств.

Бухгалтерский учет

Это классическая область применения информационных технологий и наиболее часто реализуемая на сегодняшний день задача. Такое положение вполне объяснимо. Во-первых, ошибка бухгалтера может стоить очень дорого, поэтому очевидна выгода использования возможностей автоматизации бухгалтерии. Во-вторых, задача бухгалтерского учета довольно легко формализуется, так что разработка систем автоматизации бухгалтерского учета не представляет технически сложной проблемы.

ПРИМЕЧАНИЕ

Тем не менее разработка систем автоматизации бухгалтерского учета является весьма трудоемкой. Это связано с тем, что к системам бухгалтерского учета предъявляются повышенные требования в отношении надежности и максимальной простоты и удобства в эксплуатации. Следует отметить также постоянные изменения в бухгалтерском и налоговом учете.

Управление финансовыми потоками

Внедрение информационных технологий в управление финансовыми потоками также обусловлено критичностью этой области управления предприятия к ошибкам. Неправильно построив систему расчетов с поставщиками и потребителями, можно спровоцировать кризис наличности даже при налаженной сети закупки, сбыта и хорошем маркетинге. И наоборот, точно просчитанные и жестко контролируемые условия финансовых расчетов могут существенно увеличить оборотные средства фирмы.

Управление складом, ассортиментом, закупками

Далее можно автоматизировать процесс анализа движения товара, тем самым отследив и зафиксировав те 20 % ассортимента, которые приносят 80 % прибыли. Это же позволит ответить на главный вопрос — как получать максимальную прибыль при постоянной нехватке средств?

«Заморозить» оборотные средства в чрезмерном складском запасе — самый простой способ сделать любое предприятие, производственное или торговое, потенциальным инвалидом. Можно проглядеть перспективный товар, вовремя не вложив в него деньги.

Управление производственным процессом

Управление производственным процессом представляет собой очень трудоемкую задачу. Основными механизмами здесь являются планирование и оптимальное управление производственным процессом.

Автоматизированное решение подобной задачи дает возможность грамотно планировать, учитывать затраты, проводить техническую подготовку производства, оперативно управлять процессом выпуска продукции в соответствии с производственной программой и технологией.

Очевидно, что чем крупнее производство, тем большее число бизнес-процессов участвует в создании прибыли, а значит, использование информационных систем жизненно необходимо.

Управление маркетингом

Управление маркетингом подразумевает сбор и анализ данных о фирмах-конкурентах, их продукции и ценовой политике, а также моделирование параметров внешнего окружения для определения оптимального уровня цен, прогнозирования прибыли и планирования рекламных кампаний. Решение большинства этих задач может быть формализовано и представлено в виде информационной системы, позволяющей существенно повысить эффективность управления маркетингом.

Документооборот

Документооборот является очень важным процессом деятельности любого предприятия. Хорошо отлаженная система учетного документооборота отражает реально происходящую на предприятии текущую производственную деятельность и дает управленцам возможность воздействовать на нее. Поэтому автоматизация документооборота позволяет повысить эффективность управления.

Оперативное управление предприятием

Информационная система, решающая задачи оперативного управления предприятием, строится на основе базы данных, в которой фиксируется вся возможная информация о предприятии. Такая информационная система является инструментом для управления бизнесом и обычно называется корпоративной информационной системой.

Информационная система оперативного управления включает в себя массу программных решений автоматизации бизнес-процессов, имеющих место на конкретном предприятии.

Предоставление информации о фирме

Активное развитие сети Интернет привело к необходимости создания корпоративных серверов для предоставления различного рода информации о предприятии. Практически каждое уважающее себя предприятие сейчас имеет свой веб-сервер. Веб-сервер предприятия решает ряд задач, из которых можно выделить две основные:

- ☐ создание имиджа предприятия;
- ☐ максимальная разгрузка справочной службы компании путем предоставления потенциальным и уже существующим абонентам возможности получения необходимой информации о фирме, предлагаемых товарах, услугах и ценах.

Кроме того, использование веб-технологий открывает широкие перспективы для электронной коммерции и обслуживания покупателей через Интернет.

Требования, предъявляемые к информационным системам

Информационная система должна соответствовать требованиям гибкости, надежности, эффективности и безопасности.

Гибкость

Гибкость, способность к адаптации и дальнейшему развитию подразумевает возможность приспособления информационной системы к новым условиям, новым потребностям предприятия. Выполнение этих условий возможно, если на этапе разработки информационной системы использовались общепринятые средства и методы документирования, так что по прошествии определенного времени будет возможно разобраться в структуре системы и внести в нее соответствующие изменения, даже если все разработчики или их часть по каким-либо причинам не смогут продолжить работу.

ПРИМЕЧАНИЕ

Следует иметь в виду, что психологически легче разобраться в собственных разработках, пусть даже созданных давно, чем разбираться в чужих решениях, не всегда на первый взгляд логичных. Поэтому рекомендуется фазу сопровождения системы доверить лицам, которые ее проектировали.

Любая информационная система рано или поздно морально устареет и станет вопрос о ее модернизации или полной замене. Разработчики информационных систем, как правило, не являются специалистами в прикладной области, для которой разрабатывается система. Участие в модернизации или создании новой системы той же группы проектировщиков существенно сократит сроки модернизации.

Вместе с тем возникает риск применения устаревших решений при модернизации системы. Рекомендация в таком случае одна — внимательнее относиться к поиску разработчиков информационных систем.

Надежность

Надежность информационной системы подразумевает ее функционирование без искажения находящейся в ней информации, потери данных по «техническим причинам». Требование надежности обеспечивается созданием резервных копий хранимой информации, выполнением операций журнализации, поддержанием качества каналов связи и физических носителей информации, использованием современных программных и аппаратных средств. Сюда же следует отнести защиту от случайных потерь информации в силу недостаточной квалификации персонала.

Эффективность

Система будет эффективной, если с учетом выделенных ей ресурсов она позволяет решать возложенные на нее задачи в минимальные сроки.

В любом случае оценка эффективности будет производиться заказчиком исходя из вложенных в разработку средств и соответствия представленной информационной системы его ожиданиям.

Негативной оценки эффективности информационной системы со стороны заказчика можно избежать, если представители заказчика будут привлекаться к процессу проектирования системы на всех его стадиях. Такой подход позволяет многим конечным пользователям уже на этапе проектирования адаптироваться к изменениям условий работы, которые иначе были бы приняты враждебно.

Активное сотрудничество с заказчиком с ранних этапов проектирования позволяет уточнить его потребности. Часто встречается ситуация, что заказчик чего-то хочет, но сам не знает чего именно. Чем раньше будут учтены дополнения заказчика, тем дешевле и в более короткие сроки система будет завершена.

Кроме того, заказчик, не являясь специалистом в области разработки информационных систем, может не знать о новых информационных технологиях. Контакты с заказчиком во время разработки информационной системы могут подтолкнуть его к модернизации своих аппаратных средств, применению новых методов ведения бизнеса, что отвечает как потребностям заказчика, так и проектировщика. Первый получает увеличение эффективности своего предприятия. Второй — расширение возможностей, применяемых при проектировании информационной системы.

Эффективность системы обеспечивается оптимизацией данных и методов их обработки, применением оригинальных разработок, идей, методов проектирования (в частности, спиральной модели проектирования информационной системы, о которой речь пойдет в следующих главах).

Не следует забывать о том, что работать с системой придется обычным людям, являющимся специалистами в своей предметной области, но зачастую обладающим весьма средними навыками в работе с компьютерами. Интерфейс информационных систем должен быть им интуитивно понятен. В свою очередь разработчик-программист должен понимать характер выполняемых конечным пользователем операций. Рекомендациями в этом случае могут служить повышение эффективности управления разработкой информационных систем, улучшение информированности разработчиков о предметной области.

СОВЕТ

Желательно предоставление им возможности самим попробовать себя в роли конечных пользователей до начала эксплуатации информационной системы. Встречались случаи, когда такой подход приводил к отказу от использования на рабочем месте оператора манипулятора «мышь», что, в свою очередь, давало увеличение производительности оператора в несколько раз.

Безопасность

Под безопасностью прежде всего подразумевается свойство системы, в силу которого посторонние лица не имеют доступа к информационным ресурсам организации, кроме тех, которые для них предназначены. Защита информации от постороннего доступа обеспечивается управлением доступом к ресурсам системы, использованием современных программных средств защиты информации. В крупных организациях целесообразно создавать подразделения, основным направлением деятельности которых было бы обеспечение информационной безопасности, в менее крупных организациях назначать сотрудника, ответственного за данный участок работы.

Система, не отвечающая требованиям безопасности, несет в себе риск причинения ущерба интересам заказчика, прежде всего имущественным.

ПРИМЕЧАНИЕ

В связи с этим следует отметить, что согласно действующему в России законодательству ответственность за вред, причиненный ненадлежащим качеством работ или услуг, несет исполнитель, то есть в нашем случае разработчик информационной системы. Поэтому обеспечение безопасности информационной системы заказчика в худшем случае обернется для исполнителя судебным преследованием, в лучшем — потерей клиента и утрате деловой репутации.

Кроме злого умысла при обеспечении безопасности информационных систем приходится сталкиваться еще с несколькими факторами. В частности, современные информационные системы являются достаточно сложными программными продуктами. При их проектировании с высокой вероятностью возможны ошибки, вызванные большим объемом программного кода, несовершенством компиляторов, несовершенством человеческого фактора, несовместимостью с используемыми программами сторонних разработчиков в случае их модификации и т. п. Поэтому за фазой разработки информационной системы неизбежно следует фаза ее сопровождения в процессе эксплуатации, в которой происходит выявление скрытых ошибок и их исправление.

ПРИМЕЧАНИЕ

При проектировании одной информационной системы курс доллара США в одной из процедур обозначен константой. На момент ввода в эксплуатацию информационной системы курс доллара был стабилен. Ошибка была выявлена только через некоторое время в период роста курса.

Требование безопасности обеспечивается использованием новейших средств разработки информационных систем, методов защиты информации, применением современных аппаратных средств, паролирования и журнализации, постоянного мониторинга состояния безопасности операционных систем и средств их защиты.

И наконец, самый важный фактор, влияющий на процесс разработки, — знания и опыт коллектива разработчиков информационных систем.

Глава 2

Жизненный цикл информационных систем

Разработка корпоративной информационной системы, как правило, выполняется для вполне определенного предприятия. Особенности предметной деятельности предприятия, безусловно, будут оказывать влияние на структуру информационной системы. Но в то же время структуры разных предприятий в целом похожи между собой. Каждая организация, независимо от рода ее деятельности, состоит из ряда подразделений, непосредственно осуществляющих тот или иной вид деятельности компании. И эта ситуация справедлива практически для всех организаций, каким бы видом деятельности они ни занимались.

Любую организацию можно рассматривать как совокупность взаимодействующих элементов (подразделений), каждый из которых может иметь свою, достаточно сложную, структуру. Взаимосвязи между подразделениями тоже достаточно сложны. В общем случае можно выделить три вида связей между подразделениями предприятия:

- ❑ *функциональные связи* — каждое подразделение выполняет определенные виды работ в рамках единого бизнес-процесса;
- ❑ *информационные связи* — подразделения обмениваются информацией (документами, факсами, письменными и устными распоряжениями и т. п.);
- ❑ *внешние связи* — некоторые подразделения взаимодействуют с внешними системами, причем их взаимодействие также может быть как информационным, так и функциональным.

Общность структуры разных предприятий позволяет сформулировать некоторые единые принципы построения корпоративных информационных систем.

В общем случае процесс разработки информационной системы может быть рассмотрен с двух точек зрения:

- ❑ по содержанию действий разработчиков (групп разработчиков). В данном случае рассматривается статический аспект процесса разработки, описываемый в терминах основных потоков работ: исполнители, действия, последовательность действий и т. п.;

- по времени или по стадиям жизненного цикла разрабатываемой системы. В данном случае рассматривается динамическая организация процесса разработки, описываемая в терминах циклов, стадий, итераций и этапов.

Общие сведения об управлении проектами

Информационная система предприятия разрабатывается как некоторый *проект*. Многие особенности управления проектами и фазы их разработки (фазы жизненного цикла) являются общими, не зависящими не только от предметной области, но и от характера проекта (не важно, инженерный это проект или экономический). Поэтому имеет смысл вначале рассмотреть ряд общих вопросов управления проектами.

Понятие проекта

Проект — это ограниченное по времени целенаправленное изменение отдельной системы с изначально четко определенными целями, достижение которых определяет завершение проекта, а также с установленными требованиями к срокам, результатам, риску, рамкам расходования средств и ресурсов и к организационной структуре.

ПРИМЕЧАНИЕ

Обычно для сложного понятия (каким, в частности, является понятие проекта) трудно дать однозначную формулировку, которая полностью охватывает все признаки вводимого понятия. Поэтому приведенное определение не претендует на единственность и полноту.

Можно выделить следующие основные отличительные признаки проекта как объекта управления:

- изменчивость — целенаправленный перевод системы из существующего в некоторое желаемое состояние, описываемое в терминах целей проекта;
- ограниченность конечной цели;
- ограниченность продолжительности;
- ограниченность бюджета;
- ограниченность требуемых ресурсов;
- новизна для предприятия, для которого реализуется проект;
- комплексность — наличие большого числа факторов, прямо или косвенно влияющих на прогресс и результаты проекта;
- правовое и организационное обеспечение — создание специфической организационной структуры на время реализации проекта.

Рассматривая планирование проектов и управление ими, необходимо четко осознать, что речь идет об управлении неким динамическим объектом. Поэтому система управления проектом должна быть достаточно гибкой, чтобы допускать возможность модификации без глобальных изменений в рабочей программе.

В системном плане проект может быть представлен «черным ящиком», входом которого являются технические требования и условия финансирования, а итогом работы — достижение требуемого результата (рис. 2.1). Выполнение работ обеспечивается наличием необходимых ресурсов:

- ☐ материалов;
- ☐ оборудования;
- ☐ человеческих ресурсов.

Эффективность работ достигается за счет управления процессом реализации проекта, которое обеспечивает распределение ресурсов, координацию выполняемой последовательности работ и компенсацию внутренних и внешних возмущающих воздействий.

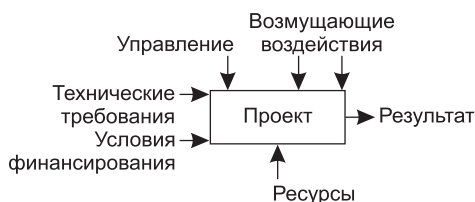


Рис. 2.1. Представление проекта в виде «черного ящика»

С точки зрения теории систем управления проект как объект управления должен быть наблюдаемым и управляемым, то есть выделяются некоторые характеристики, по которым можно постоянно контролировать ход выполнения проекта (свойство *наблюдаемости*). Кроме того, необходимы механизмы своевременного воздействия на ход реализации проекта (свойство *управляемости*).

Свойство управляемости особенно актуально в условиях неопределенности и изменчивости предметной области, которые нередко сопутствуют проектам по разработке информационных систем (более подробно проблемы получения полного формального описания предметной области будут обсуждаться в конце данной главы). Для обоснования целесообразности и осуществимости проекта, анализа хода его реализации, а также для заключительной оценки степени достижения поставленных целей проекта и сравнения фактических результатов с запланированными существует ряд характеристик проекта. К важнейшим из них относятся технико-экономические показатели:

- ☐ объем работ;
- ☐ сроки выполнения;
- ☐ себестоимость;
- ☐ экономическая эффективность, обеспечиваемая реализацией проекта;
- ☐ социальная и общественная значимость проекта.

Классификация проектов

Проекты могут сильно отличаться по сфере приложения, составу, предметной области, масштабам, длительности, составу участников, степени сложности,

значимости результатов и т. п. Проекты могут быть классифицированы по разным различным признакам. Отметим основные из них.

Класс проекта определяется по составу и структуре проекта. Обычно различают:

- ☐ монопроект (отдельный проект, который может быть любого типа, вида и масштаба);
- ☐ мультипроект (комплексный проект, состоящий из ряда монопроектов и требующий применения многопроектного управления).

Тип проекта определяется по основным сферам деятельности, в которых осуществляется проект. Можно выделить пять основных типов проекта:

- ☐ технический;
- ☐ организационный;
- ☐ экономический;
- ☐ социальный;
- ☐ смешанный.

ПРИМЕЧАНИЕ

Разработка информационных систем относится, скорее всего, к техническим проектам, которые имеют следующие особенности:

- главная цель проекта четко определена, но отдельные цели должны уточняться по мере достижения частных результатов;
- срок завершения и продолжительность проекта определены заранее, желательно их точное соблюдение, однако они также могут корректироваться в зависимости от полученных промежуточных результатов и общего прогресса проекта.

Масштаб проекта определяется по размерам бюджета и количеству участников:

- ☐ мелкие проекты;
- ☐ малые проекты;
- ☐ средние проекты;
- ☐ крупные проекты.

Можно также рассматривать масштабы проектов в более конкретной форме — отраслевые, корпоративные, ведомственные, проекты одного предприятия.

Основные фазы проектирования информационной системы

Каждый проект, независимо от сложности и объема работ, необходимых для его выполнения, проходит в своем развитии определенные состояния: от состояния, когда «проекта еще нет», до состояния, когда «проекта уже нет». Совокупность ступеней развития от возникновения идеи до полного завершения проекта принято разделять на *фазы (стадии, этапы)*.

В определении количества фаз и их содержания имеются некоторые различия, поскольку эти характеристики во многом зависят от условий осуществле-

ния конкретного проекта и опыта основных участников. Тем не менее логика и основное содержание процесса разработки информационной системы почти во всех случаях являются общими.

Можно выделить следующие фазы развития информационной системы:

- ☐ формирование концепции;
- ☐ разработка технического задания;
- ☐ проектирование;
- ☐ изготовление;
- ☐ ввод системы в эксплуатацию.

Рассмотрим каждую из них более подробно.

ПРИМЕЧАНИЕ

Вторую и частично третью фазы принято называть *фазами системного проектирования*, а последние две (иногда сюда включают и фазу проектирования) — *фазами реализации*.

Концептуальная фаза

Главным содержанием работ на этой фазе является определение целей проекта, разработка его концепции, включающая:

- ☐ формирование идеи, постановку целей;
- ☐ формирование ключевой команды проекта;
- ☐ изучение мотивации и требований заказчика и других участников;
- ☐ сбор исходных данных и анализ существующего состояния;
- ☐ определение основных требований и ограничений, необходимых материальных, финансовых и трудовых ресурсов;
- ☐ сравнительную оценку альтернатив;
- ☐ представление предложений, их экспертизу и утверждение.

Разработка технического предложения

Главным содержанием этой фазы является разработка технического предложения и переговоры с заказчиком о заключении контракта. Общее содержание работ этой фазы:

- ☐ разработка основного содержания проекта, базовой структуры проекта;
- ☐ разработка и утверждение технического задания;
- ☐ планирование, декомпозиция базовой структурной модели проекта;
- ☐ составление сметы и бюджета проекта, определение потребности в ресурсах;
- ☐ разработка календарных планов и укрупненных графиков работ;
- ☐ подписание контракта с заказчиком;
- ☐ ввод в действие средств коммуникации участников проекта и контроля за ходом работ.

Проектирование

На этой фазе определяются подсистемы, их взаимосвязи, выбираются наиболее эффективные способы выполнения проекта и использования ресурсов. Характерные работы этой фазы:

- ☐ выполнение базовых проектных работ;
- ☐ разработка частных технических заданий;
- ☐ выполнение концептуального проектирования;
- ☐ составление технических спецификаций и инструкций;
- ☐ представление проектной разработки, экспертиза и утверждение.

Разработка

На этой фазе производится координация и оперативный контроль работ по проекту, осуществляется изготовление подсистем, их объединение и тестирование. Основное содержание:

- ☐ выполнение работ по разработке программного обеспечения;
- ☐ выполнение подготовки к внедрению системы;
- ☐ контроль и регулирование основных показателей проекта.

Ввод системы в эксплуатацию

На этой фазе проводятся испытания, опытная эксплуатация системы в реальных условиях, ведутся переговоры о результатах выполнения проекта и о возможных новых контрактах. Основные виды работ:

- ☐ комплексные испытания;
- ☐ подготовка кадров для эксплуатации создаваемой системы;
- ☐ подготовка рабочей документации, сдача системы заказчику и ввод ее в эксплуатацию;
- ☐ сопровождение, поддержка, сервисное обслуживание;
- ☐ оценка результатов проекта и подготовка итоговых документов;
- ☐ разрешение конфликтных ситуаций и закрытие работ по проекту;
- ☐ накопление опытных данных для последующих проектов, анализ опыта, состояния, определение направлений развития.

ПРИМЕЧАНИЕ

Начальные фазы проекта имеют решающее влияние на достигаемый результат, так как на них принимаются основные решения, определяющие качество информационной системы. При этом обычно 30 % вклада в конечный результат проекта вносят фазы концепции и предложения, 20 % — фаза проектирования, 20 % — фаза изготовления, 30 % — фаза сдачи объекта и завершения проекта.

Кроме того, на обнаружение ошибок, допущенных на стадии системного проектирования, расходуется примерно в два раза больше времени, чем на последующих фазах, а их исправление обходится в пять раз дороже. Поэтому на

начальных стадиях проекта разработку следует выполнять особенно тщательно. Наиболее часто на начальных фазах допускаются следующие ошибки:

- ☐ ошибки в определении интересов заказчика;
- ☐ концентрация на маловажных, сторонних интересах;
- ☐ неправильная интерпретация исходной постановки задачи;
- ☐ неправильное или недостаточное понимание деталей;
- ☐ неполнота функциональных спецификаций (системных требований);
- ☐ ошибки в определении требуемых ресурсов и сроков;
- ☐ редкая проверка на согласованность этапов и отсутствие контроля со стороны заказчика (нет привлечения заказчика).

Процессы, протекающие на протяжении жизненного цикла информационной системы

Понятие жизненного цикла является одним из базовых понятий методологии проектирования информационных систем. Жизненный цикл информационной системы представляет собой непрерывный процесс, начинающийся с момента принятия решения о ее создании и заканчивается в момент полного изъятия ее из эксплуатации.

Существует международный стандарт, регламентирующий жизненный цикл информационных систем, — ISO/IEC 12207.

ПРИМЕЧАНИЕ

ISO — International Organization of Standardization (Международная организация по стандартизации). IEC — International Electrotechnical Commission (Международная комиссия по электротехнике).

Стандарт ISO/IEC 12207 определяет структуру жизненного цикла, содержащую процессы, действия и задачи, которые должны быть выполнены во время создания информационной системы. Согласно данному стандарту структура жизненного цикла основывается на трех группах процессов:

- ☐ основные процессы жизненного цикла (приобретение, поставка, разработка, эксплуатация, сопровождение);
- ☐ вспомогательные процессы, обеспечивающие выполнение основных процессов (документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, оценка, аудит, разрешение проблем);
- ☐ организационные процессы (управление проектами, создание инфраструктуры проекта, определение, оценка и улучшение самого жизненного цикла, обучение).

Рассмотрим каждую из указанных групп более подробно.

Основные процессы жизненного цикла

Среди основных процессов жизненного цикла наибольшую важность имеют три: разработка, эксплуатация и сопровождение. Каждый процесс характеризуется

определенными задачами и методами их решения, исходными данными, полученными на предыдущем этапе, и результатами.

Разработка

Разработка информационной системы включает в себя все работы по созданию информационного программного обеспечения и его компонентов в соответствии с заданными требованиями. Разработка информационного программного обеспечения также включает:

- ☐ оформление проектной и эксплуатационной документации;
- ☐ подготовку материалов, необходимых для проведения тестирования разработанных программных продуктов;
- ☐ разработку материалов, необходимых для организации обучения персонала.

Разработка является одним из важнейших процессов жизненного цикла информационной системы и, как правило, включает в себя стратегическое планирование, анализ, проектирование и реализацию (программирование).

Эксплуатация

Эксплуатационные работы можно подразделить на подготовительные и основные. К подготовительным относятся:

- ☐ конфигурирование базы данных и рабочих мест пользователей;
- ☐ обеспечение пользователей эксплуатационной документацией;
- ☐ обучение персонала.

Основные эксплуатационные работы включают:

- ☐ непосредственно эксплуатацию;
- ☐ локализацию проблем и устранение причин их возникновения;
- ☐ модификацию программного обеспечения;
- ☐ подготовку предложений по совершенствованию системы;
- ☐ развитие и модернизацию системы.

Сопровождение

Службы технической поддержки играют весьма заметную роль в жизни любой корпоративной информационной системы. Наличие квалифицированного технического обслуживания на этапе эксплуатации информационной системы является необходимым условием для решения поставленных перед ней задач, причем ошибки обслуживающего персонала могут приводить к явным или скрытым финансовым потерям, сопоставимым со стоимостью самой этой системы.

Основными предварительными действиями при подготовке к организации технического обслуживания информационной системы являются следующие:

- ☐ выделение наиболее ответственных узлов системы и определение для них критичности простоя. Это позволит выделить наиболее критичные состав-

ляющие информационной системы и оптимизировать распределение ресурсов для технического обслуживания;

- ❑ определение задач технического обслуживания и их разделение на внутренние (решаемые силами обслуживающего подразделения) и внешние (решаемые специализированными сервисными организациями). Таким образом производится четкое определение круга исполняемых функций и разделение ответственности;
- ❑ проведение анализа имеющихся внутренних и внешних ресурсов, необходимых для организации технического обслуживания в рамках описанных задач и разделения компетенции. Основные критерии для анализа: наличие гарантии на оборудование, состояние ремонтного фонда, квалификация персонала;
- ❑ подготовка плана организации технического обслуживания, в котором необходимо определить этапы исполняемых действий, сроки их исполнения, затраты на этапах, ответственность исполнителей.

Обеспечение качественного технического обслуживания информационной системы требует привлечения специалистов высокой квалификации, которые в состоянии решать не только каждодневные задачи администрирования, но и быстро восстанавливать работоспособность системы при сбоях.

Вспомогательные процессы

Среди вспомогательных процессов одно из главных мест занимает управление конфигурацией. Это один из вспомогательных процессов, поддерживающих основные процессы жизненного цикла информационной системы, прежде всего процессы разработки и сопровождения. При разработке проектов сложных информационных систем, состоящих из многих компонентов, каждый из которых может разрабатываться независимо и, следовательно, иметь несколько вариантов реализации и/или несколько версий одной реализации, возникает проблема учета их связей и функций, создания единой структуры и обеспечения развития всей системы. Управление конфигурацией позволяет организовывать, систематически учитывать и контролировать внесение изменений в различные компоненты информационной системы на всех стадиях ее жизненного цикла.

Организационные процессы

Управление проектом связано с вопросами планирования и организации работ, создания коллективов разработчиков и контроля за сроками и качеством выполняемых работ. Техническое и организационное обеспечение проекта включает:

- ❑ выбор методов и инструментальных средств для реализации проекта;
- ❑ определение методов описания промежуточных состояний разработки;
- ❑ разработку методов и средств испытаний созданного программного обеспечения;
- ❑ обучение персонала.

Обеспечение качества проекта связано с проблемами верификации, проверки и тестирования компонентов информационной системы.

Верификация — это процесс определения соответствия текущего состояния разработки, достигнутого на данном этапе, требованиям этого этапа.

Проверка — это процесс определения соответствия параметров разработки исходным требованиям. Проверка отчасти совпадает с тестированием, которое проводится для определения различий между действительными и ожидавшимися результатами и оценки соответствия характеристик информационной системы ее первоначальным требованиям.

Структура жизненного цикла информационной системы

Полный жизненный цикл информационной системы включает в себя, как правило, стратегическое планирование, анализ, проектирование, реализацию, внедрение и эксплуатацию. В общем случае жизненный цикл можно, в свою очередь, разбить на ряд стадий. В принципе это деление на стадии достаточно произвольно. Мы рассмотрим один из вариантов такого деления, предлагаемый корпорацией Rational Software. Это одна из ведущих фирм на рынке программного обеспечения средств разработки информационных систем (среди которых большой популярностью заслуженно пользуется универсальное CASE-средство Rational Rose). Согласно методологии, предлагаемой Rational Software, жизненный цикл информационной системы подразделяется на четыре стадии:

- ☐ начало;
- ☐ уточнение;
- ☐ конструирование;
- ☐ передача в эксплуатацию.

Границы каждой стадии определены некоторыми моментами времени, в которые необходимо принимать определенные критические решения и в которые, следовательно, должны быть достигнуты определенные ключевые цели.

Начальная стадия

На начальной стадии устанавливается область применения системы и определяются граничные условия. Для этого необходимо идентифицировать все внешние объекты, с которыми должна взаимодействовать разрабатываемая система, и определить характер этого взаимодействия на высоком уровне. На начальной стадии идентифицируются все функциональные возможности системы и производится описание наиболее существенных из них.

Деловое применение включает:

- ☐ критерии успеха разработки;
- ☐ оценку риска;
- ☐ оценку ресурсов, необходимых для выполнения разработки;
- ☐ календарный план с указанием сроков завершения основных этапов.

Стадия уточнения

На этой стадии проводится анализ прикладной области, разрабатывается архитектурная основа информационной системы.

При принятии любых решений, касающихся архитектуры системы, необходимо принимать во внимание всю разрабатываемую систему в целом. Это означает, что необходимо описать большинство функциональных возможностей системы и учесть взаимосвязи между отдельными ее составляющими.

В конце стадии уточнения проводится анализ архитектурных решений и способов устранения главных элементов риска, содержащихся в проекте.

Стадия конструирования

На стадии конструирования разрабатывается законченное изделие, готовое к передаче пользователю.

По окончании этой стадии определяется работоспособность разработанного программного обеспечения.

Стадия перехода

На стадии перехода производится передача разработанного программного обеспечения пользователям. При эксплуатации разработанной системы в реальных условиях часто возникают различного рода проблемы, которые требуют дополнительных работ по внесению корректив в разработанный продукт. Это, как правило, связано с обнаружением ошибок и недоработок.

В конце стадии перехода необходимо определить, достигнуты цели разработки или нет.

Модели жизненного цикла информационной системы

Моделью жизненного цикла информационной системы будем называть некоторую структуру, определяющую последовательность осуществления процессов, действий и задач, выполняемых на протяжении жизненного цикла информационной системы, а также взаимосвязи между этими процессами, действиями и задачами.

В стандарте ISO/IEC 12207 не конкретизируются в деталях методы реализации и выполнения действий и задач, входящих в процессы жизненного цикла информационной системы, а лишь описываются структуры этих процессов. Это вполне понятно, так как регламенты стандарта являются общими для любых моделей жизненного цикла, методологий и технологий разработки. Модель же жизненного цикла зависит от специфики информационной системы и условий, в которых она создается и функционирует. Поэтому не имеет смысла предлагать какие-либо конкретные модели жизненного цикла и методы разработки информационных систем для общего случая, без привязки к определенной предметной области.

К настоящему времени наибольшее распространение получили следующие две основные модели жизненного цикла:

- ☐ каскадная модель, иногда также называемая моделью «водопад» (waterfall);
- ☐ спиральная модель.

Каскадная модель жизненного цикла информационной системы

Каскадная модель демонстрирует классический подход к разработке различных систем в любых прикладных областях. Для разработки информационных систем данная модель широко использовалась в 1970-х и первой половине 1980-х годов. Каскадные методы проектирования хорошо описаны в зарубежной и отечественной литературе разных направлений: методических монографиях, стандартах, учебниках. Организация работ по каскадной схеме официально рекомендовалась и широко применялась в различных отраслях. Таким образом, наличие не только теоретических оснований, но и промышленных методик и стандартов, а также использование этих методов в течение десятилетий позволяет называть каскадные методы классическими.

Каскадная модель предусматривает последовательную организацию работ. При этом основной особенностью является разбиение всей разработки на этапы, причем переход с одного этапа на следующий происходит только после того, как будут полностью завершены все работы на предыдущем этапе. Каждый этап завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков.

Основные этапы разработки по каскадной модели

За десятилетия существования модели «водопад» разбиение работ на стадии и названия этих стадий менялись. Кроме того, наиболее разумные методики и стандарты избегали жесткого и однозначного приписывания определенных работ к конкретным этапам. Тем не менее все же можно выделить ряд устойчивых этапов разработки, практически не зависящих от предметной области (рис. 2.2.):

- ☐ анализ требований заказчика;
- ☐ проектирование;
- ☐ разработка;
- ☐ тестирование и опытная эксплуатация;
- ☐ сдача готового продукта.

На первом этапе проводится исследование проблемы, которая должна быть решена, четко формулируются все требования заказчика. Результатом, получаемым на данном этапе, является техническое задание (задание на разработку), согласованное со всеми заинтересованными сторонами.

На втором этапе разрабатываются проектные решения, удовлетворяющие всем требованиям, сформулированным в техническом задании. Результатом данного этапа является комплект проектной документации, содержащей все необходимые данные для реализации проекта.

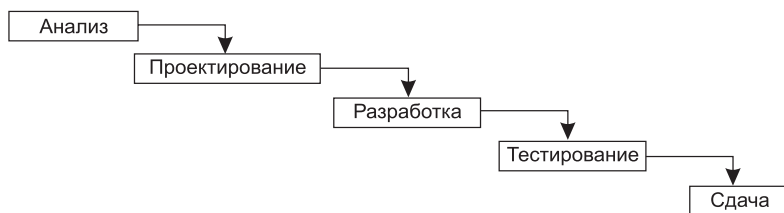


Рис. 2.2. Каскадная модель разработки

Третий этап — реализация проекта. Здесь осуществляется разработка программного обеспечения (кодирование) в соответствии с проектными решениями, полученными на предыдущем этапе. Методы, используемые для реализации, не имеют принципиального значения. Результатом выполнения данного этапа является готовый программный продукт.

На четвертом этапе проводится проверка полученного программного обеспечения на предмет соответствия требованиям, заявленным в техническом задании. Опытная эксплуатация позволяет выявить различного рода скрытые недостатки, проявляющиеся в реальных условиях работы информационной системы.

Последний этап — сдача готового проекта. Главная задача этого этапа — убедить заказчика, что все его требования реализованы в полной мере.

Этапы работ в рамках каскадной модели часто также называют частями «проектного цикла» системы. Такое название возникло потому, что этапы состоят из многих итерационных процедур уточнения требований к системе и вариантов проектных решений. Жизненный цикл самой системы существенно сложнее и больше. Он может включать в себя произвольное число циклов уточнения, изменения и дополнения уже принятых и реализованных проектных решений. В этих циклах происходит развитие информационной системы и модернизация отдельных ее компонентов.

Основные достоинства каскадной модели

Каскадная модель имеет ряд положительных сторон, благодаря которым она хорошо зарекомендовала себя при выполнении различного рода инженерных разработок и получила широкое распространение. Рассмотрим основные достоинства модели «водопад»:

- ❑ на каждом этапе формируется законченный набор проектной документации, отвечающий критериям полноты и согласованности. На заключительных этапах также разрабатывается пользовательская документация, охватывающая все предусмотренные стандартами виды обеспечения информационной системы: организационное, методическое, информационное, программное, аппаратное;
- ❑ выполняемые в логичной последовательности этапы работ позволяют планировать сроки завершения и соответствующие затраты.

Каскадная модель изначально разрабатывалась для решения различного рода инженерных задач и не потеряла своего значения для прикладной области

до настоящего времени. Кроме того, каскадный подход хорошо зарекомендовал себя и при построении определенных информационных систем. Имеются в виду системы, для которых в самом начале разработки можно достаточно точно и полно сформулировать все требования, с тем чтобы предоставить разработчикам свободу выбора реализации, наилучшей с технической точки зрения. К таким информационным системам, в частности, относятся сложные расчетные системы, системы реального времени.

Тем не менее, несмотря на все свои достоинства, каскадная модель имеет ряд недостатков, ограничивающих ее применение при разработке информационных систем. Причем эти недостатки делают ее либо полностью неприменимой, либо приводят к увеличению сроков разработки и стоимости проекта. В настоящее время многие неудачи программных проектов объясняются именно применением последовательного процесса разработки.

Недостатки каскадной модели

Перечень недостатков каскадной модели при ее использовании для разработки информационных систем достаточно обширен. Вначале просто перечислим их, а затем рассмотрим основные из них более подробно:

- ☐ существенная задержка получения результатов;
- ☐ ошибки и недоработки на любом из этапов выясняются, как правило, на последующих этапах работ, что приводит к необходимости возврата на предыдущие стадии;
- ☐ сложность распараллеливания работ по проекту;
- ☐ чрезмерная информационная перенасыщенность каждого из этапов;
- ☐ сложность управления проектом;
- ☐ высокий уровень риска и ненадежность инвестиций.

Задержка получения результатов обычно считается главным недостатком каскадной схемы. Данный недостаток проявляется в основном в том, что вследствие последовательного подхода к разработке согласование результатов с заинтересованными сторонами производится только после завершения очередного этапа работ. Поэтому может оказаться, что разрабатываемая информационная система не соответствует требованиям пользователей. Причем такие несоответствия могут возникать на любом этапе разработки — искажения могут преднамеренно вноситься и проектировщиками-аналитиками, и программистами, так как они не обязательно хорошо разбираются в тех предметных областях, для которых производится разработка информационной системы.

Кроме того, используемые при разработке информационной системы модели автоматизируемого объекта, отвечающие критериям внутренней согласованности и полноты, могут в силу различных причин устареть за время разработки (например, из-за внесения изменений в законодательство, колебания курса валют и т. п.). Это относится и к функциональной модели, и к информационной модели, и к проектам интерфейса пользователя, и к пользовательской документации.

Возврат на более ранние стадии. Данный недостаток каскадной модели в общем-то является одним из проявлений предыдущего. Поэтапная и последовательная работа над проектом может быть следствием того, что ошибки, допущенные на более ранних этапах, как правило, обнаруживаются только на последующих стадиях работы над проектом. Поэтому, после того как ошибки проявятся, проект возвращается на предыдущий этап, перерабатывается и снова передается на последующую стадию. Это может служить причиной срыва графика работ и усложнения взаимоотношений между группами разработчиков, выполняющих отдельные ее этапы.

Самым же неприятным является то, что недоработки предыдущего уровня могут обнаруживаться не сразу на последующем уровне, а позднее (например, на стадии опытной эксплуатации могут проявиться ошибки в описании предметной области). Это означает, что часть проекта должна быть возвращена на начальный уровень работы. Вообще работа может быть возвращена с любого этапа на любой предыдущий этап, поэтому в реальном случае каскадная схема разработки имеет вид, приведенный на рис. 2.3.

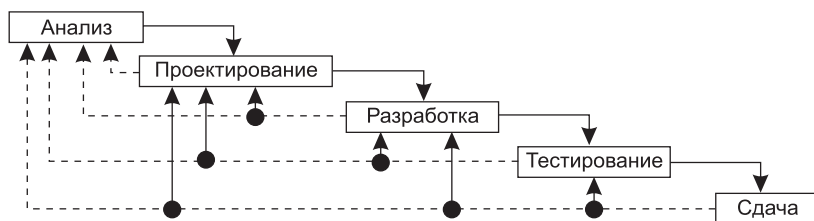


Рис. 2.3. Реальный процесс разработки по каскадной схеме

Одной из причин данной ситуации является то, что в качестве экспертов, участвующих в описании предметной области, часто выступают будущие пользователи системы, которые нередко не могут четко сформулировать то, что они хотели бы получить. Кроме того, заказчики и исполнители часто неправильно понимают друг друга вследствие того, что исполнители обычно не являются специалистами в предметной области решаемой задачи, а заказчики далеки от программирования.

Сложность параллельного ведения работ. Отмеченные выше проблемы возникают вследствие того, что работа над проектом строится в виде цепочки последовательных шагов. Причем даже в том случае, когда разработку некоторых частей проекта (подсистем) можно вести параллельно, при использовании каскадной схемы распараллеливание работ весьма затруднительно. Сложности параллельного ведения работ связаны с необходимостью постоянного согласования различных частей проекта. Чем сильнее взаимозависимость отдельных частей проекта, тем чаще и тщательнее должна выполняться синхронизация, тем сильнее зависимы друг от друга группы разработчиков. Поэтому преимущества параллельного ведения работ просто теряются.

Отсутствие параллелизма негативно сказывается и на организации работы всего коллектива разработчиков. Работа одних групп сдерживается другими. Пока производится анализ предметной области, проектировщики, разработчики

и те, кто занимается тестированием и администрированием, почти не имеют работы. Кроме того, при последовательной разработке крайне сложно внести изменения в проект после завершения этапа и передаче проекта на следующую стадию. Так, например, если после передачи проекта на следующий этап группа разработчиков нашла более эффективное решение, оно не может быть использовано. Это связано с тем, что более раннее решение уже, возможно, реализовано и связано с другими частями проекта. Поэтому исключается (или, по крайней мере, существенно затрудняется) доработка проекта после его передачи на следующий этап.

Информационная перенасыщенность. Проблема информационной перенасыщенности возникает вследствие сильной зависимости между различными группами разработчиков. Данная проблема заключается в том, что при внесении изменений в одну из частей проекта необходимо оповещать всех разработчиков, которые использовали или могли использовать эту часть в своей работе. Когда система состоит из большого количества взаимосвязанных подсистем, то синхронизация внутренней документации становится важной самостоятельной задачей.

Причем синхронизация документации на каждую часть системы — это не более чем процесс оповещения групп разработчиков. Самим же разработчикам необходимо ознакомиться с изменениями и оценить, не сказались ли эти изменения на уже полученных результатах. Все это может потребовать проведения повторного тестирования и даже внесения изменений в уже готовые части проекта. Причем эти изменения, в свою очередь, должны быть отражены во внутренней документации и быть разосланы другим группам разработчиков. Как следствие, объем документации по мере разработки проекта растет очень быстро, так что требуется все больше времени для составления документации и ознакомления с ней.

Следует также отметить, что, кроме изучения нового материала, не отпадает и необходимость в изучении старой информации. Это связано с тем, что вполне вероятна ситуация, когда в процессе выполнения разработки изменяется состав группы разработчиков (этот процесс носит название *ротации кадров*). Новым разработчикам необходима информация о том, что было сделано до них. Причем чем сложнее проект, тем больше времени требуется, чтобы ввести нового разработчика в курс дела.

Сложность управления проектом при использовании каскадной схемы в основном обусловлена строгой последовательностью стадий разработки и наличием сложных взаимосвязей между различными частями проекта.

Последовательность разработки проекта приводит к тому, что одни группы разработчиков должны ожидать результатов работы других команд. Поэтому требуется административное вмешательство для того, чтобы согласовать сроки работы и состав передаваемой документации.

В случае же обнаружения ошибок в выполненной работе необходим возврат к предыдущим этапам выполнения проекта. Это приводит к дополнительным сложностям в управлении проектом. Разработчики, допустившие просчет или ошибку, вынуждены прервать текущую работу (над новым проектом) и заняться исправлением ошибок. Следствием этого обычно является срыв сроков выполнения как исправляемого, так и нового проектов. Требовать же от команды

разработчиков ожидания окончания следующей стадии разработки нерационально, так как это приводит к существенным потерям рабочего времени.

Упростить взаимодействие между группами разработчиков и уменьшить информационную перенасыщенность документации можно, уменьшая количество связей между отдельными частями проекта. Однако это обычно весьма непросто. Далеко не каждую информационную систему можно разделить на несколько слабо связанных подсистем.

Высокий уровень риска. Чем сложнее проект, тем больше продолжительность каждого из этапов разработки и тем сложнее взаимосвязи между отдельными частями проекта, количество которых также увеличивается. Причем результаты разработки можно реально увидеть и оценить лишь на этапе тестирования, то есть после завершения анализа, проектирования и разработки — этапов, выполнение которых требует значительного времени и средств. Как уже было отмечено выше, запоздалая оценка создает значительные проблемы при выявлении ошибок анализа и проектирования — требуется возврат проекта на предыдущие стадии и повторение процесса разработки. Однако возврат на предыдущие стадии может быть связан не только с ошибками, но и с изменениями, произошедшими за время выполнения разработки в предметной области или в требованиях заказчика. Причем возврат проекта вследствие этих причин на доработку не гарантирует, что предметная область снова не изменится к тому моменту, когда будет готова следующая версия проекта. Фактически это означает, что существует вероятность того, что процесс разработки «заикнется» и никогда не дойдет до сдачи в эксплуатацию. Расходы на проект будут постоянно расти, а сроки сдачи готового продукта — постоянно откладываться.

ПРИМЕЧАНИЕ

Кроме рассмотренных выше существует еще один серьезный недостаток, присущий каскадной модели разработки, на который также следует обратить внимание. Этот недостаток связан с конфликтом (не всегда явным) между разработчиками, участвующими в выполнении проекта. Этот конфликт обусловлен тем, что возврат части проекта на предыдущую стадию обычно сопровождается поиском причин и виновных. А так как однозначно персонифицировать ответственного за ошибки далеко не всегда возможно, то попытки поиска виноватых могут сильно усложнить отношения в коллективе. Как следствие, в рабочей группе часто ценится не тот руководитель, который имеет высокую квалификацию и большой опыт, а тот, кто умеет «отстоять» своих подчиненных, обеспечить им более удобные условия работы и т. п. В результате появляется опасность снижения и квалификации, и творческого потенциала всей команды. Соответственно, техническое руководство проектом начинает в большей степени подменяться организационным руководством, все более детальной проработкой должностных инструкций и все более формальным исполнением этих инструкций. Тот, кто не умеет организовать работу, обречен бороться за дисциплину. И здесь возникает проблема несовместимости дисциплины и творчества. Чем строже дисциплина, тем менее творческой становится атмосфера в коллективе. И такое положение вещей может привести к тому, что наиболее одаренные кадры со временем покинут коллектив.

Поэтому можно утверждать, что сложные проекты, разрабатываемые по каскадной схеме, имеют повышенный уровень риска. Этот вывод подтверждается

практикой: по сведениям консалтинговой компании The Standish Group, в США более 31 % проектов корпоративных информационных систем (IT-проектов) заканчивается неудачей; почти 53 % IT-проектов завершается с перерасходом бюджета (в среднем на 189 %, то есть почти в два раза); и только 16,2 % проектов укладывается и в срок и в бюджет.

Спиральная модель жизненного цикла

Спиральная модель в отличие от каскадной предполагает итерационный процесс разработки информационной системы. При этом возрастает значение начальных этапов жизненного цикла, таких как анализ и проектирование. На этих этапах проверяется и обосновывается реализуемость технических решений путем создания прототипов.

Итерации

Каждая итерация представляет собой законченный цикл разработки, приводящий к выпуску внутренней или внешней версии изделия (или подмножества конечного продукта), которое совершенствуется от итерации к итерации, чтобы стать законченной системой (рис. 2.4).

Таким образом, каждый виток спирали соответствует созданию фрагмента или версии программного изделия, на нем уточняются цели и характеристики проекта, определяется его качество, планируются работы следующего витка спирали. На каждой итерации углубляются и последовательно конкретизируются детали проекта, в результате чего выбирается обоснованный вариант, который доводится до окончательной реализации.

Использование спиральной модели позволяет осуществлять переход на следующий этап выполнения проекта, не дожидаясь полного завершения работы на текущем — недоделанную работу можно будет выполнить на следующей итерации. Главная задача каждой итерации — как можно быстрее создать работоспособный продукт, который можно показать пользователям системы. Таким образом существенно упрощается процесс внесения уточнений и дополнений в проект.

Преимущества спиральной модели

Спиральный подход к разработке программного обеспечения позволяет преодолеть большинство недостатков каскадной модели и, кроме того, обеспечивает ряд дополнительных возможностей, делая процесс разработки более гибким.

Рассмотрим преимущества итерационного подхода более подробно:

- ❑ итерационная разработка существенно упрощает внесение изменений в проект при изменении требований заказчика;
- ❑ при использовании спиральной модели отдельные элементы информационной системы интегрируются в единое целое постепенно. При итерационном подходе интеграция производится фактически непрерывно. Поскольку интеграция начинается с меньшего количества элементов, то возникает гораздо меньше проблем при ее проведении (по некоторым оценкам, при использовании каскадной модели разработка интеграция занимает до 40 % всех затрат в конце проекта);

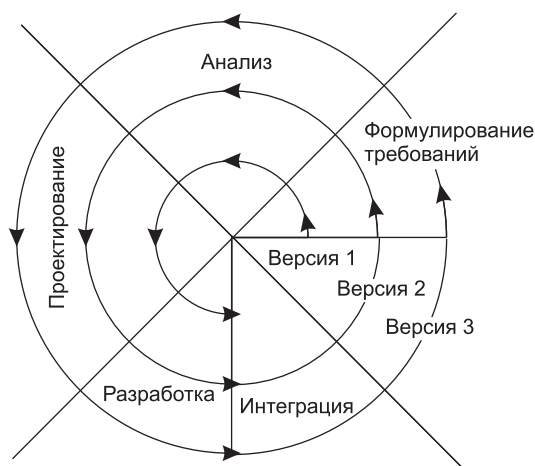


Рис. 2.4. Спиральная модель жизненного цикла информационной системы

- ❑ уменьшение уровня рисков. Данное преимущество является следствием предыдущего, так как риски обнаруживаются именно во время интеграции. Поэтому уровень рисков максимален в начале разработки проекта. По мере продвижения разработки ожидаемый риск уменьшается. Данное утверждение справедливо при любой модели разработки, однако при использовании спиральной модели уменьшение уровня рисков происходит с наибольшей скоростью. Это связано с тем, что при итерационном подходе интеграция выполняется уже на первой итерации, и при этом выявляются многие аспекты проекта, такие как пригодность используемых инструментальных средств и программного обеспечения, квалификация разработчиков и т. п. На рис. 2.5 приведены графики зависимости уровня рисков от времени разработки при использовании каскадного и итерационного подходов;
- ❑ итерационная разработка обеспечивает большую гибкость в управлении проектом, давая возможность внесения тактических изменений в разрабатываемое изделие. Например, можно сократить сроки разработки за счет уменьшения функциональности системы или использовать в качестве составных частей системы продукцию сторонних фирм вместо собственных разработок. Это может быть актуальным в условиях конкурентной борьбы, когда необходимо противостоять продвижению изделия, предлагаемого конкурентами;
- ❑ итерационный подход упрощает повторное использование компонентов (позволяет использовать компонентный подход к программированию — более подробно об этом мы будем говорить в следующей главе). Это обусловлено тем, что гораздо проще выявить (идентифицировать) общие части проекта, когда они уже частично разработаны, чем пытаться выделить их в самом начале проекта. Анализ проекта после проведения нескольких начальных итераций позволяет выявить общие, многократно используемые компоненты, которые на последующих итерациях будут совершенствоваться;

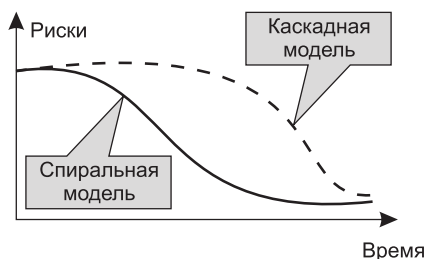


Рис. 2.5. Зависимость рисков от времени разработки

- спиральная модель позволяет получить более надежную и устойчивую систему. Это связано с тем, что по мере развития системы ошибки и слабые места обнаруживаются и исправляются на каждой итерации. Одновременно могут корректироваться критические параметры эффективности, что при использовании каскадной модели выполняется только перед внедрением системы;
- итерационный подход позволяет совершенствовать процесс разработки — анализ, проводимый в конце каждой итерации, позволяет проводить оценку того, что должно быть изменено в организации разработки, и улучшить ее на следующей итерации.

Проблемы, возникающие при использовании спиральной модели

Основная проблема спирального цикла — определение момента перехода на следующий этап. Для ее решения необходимо ввести временные ограничения на каждый из этапов жизненного цикла. Иначе процесс разработки может превратиться в бесконечное совершенствование уже сделанного. При итерационном подходе полезно следовать принципу «лучшее — враг хорошего». Поэтому завершение итерации должно производиться строго в соответствии с планом, даже если не вся запланированная работа закончена.

Планирование работ обычно проводится на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.

Глава 3

Методология и технология разработки информационных систем

Методология создания информационных систем заключается в организации процесса их построения и обеспечении управления этим процессом для того, чтобы гарантировать выполнение требований как к самой системе, так и к характеристикам процесса разработки.

Основными задачами, решение которых должна обеспечивать методология создания корпоративных информационных систем (с помощью соответствующего набора инструментальных средств), являются следующие:

- ❑ обеспечение создания информационных систем, отвечающих целям и задачам предприятия и соответствующих предъявляемым к ним требованиям по автоматизации деловых процессов;
- ❑ гарантия создания системы с заданными параметрами в течение заданного времени в рамках оговоренного заранее бюджета;
- ❑ простота сопровождения, модификации и расширения системы с целью обеспечения ее соответствия изменяющимся условиям работы предприятия;
- ❑ обеспечение создания корпоративных информационных систем, отвечающих требованиям открытости, переносимости и масштабируемости;
- ❑ возможность использования в создаваемой системе разработанных ранее и применяемых на предприятии средств информационных технологий (программного обеспечения, баз данных, средств вычислительной техники, телекоммуникаций).

Методологии, технологии и инструментальные средства проектирования (CASE-средства) составляют основу проекта любой информационной системы. Методология реализуется через конкретные технологии и поддерживающие их стандарты, методики и инструментальные средства, которые обеспечивают выполнение процессов жизненного цикла информационных систем.

Основное содержание технологии проектирования составляют технологические инструкции, состоящие из описания последовательности технологических операций, условий, в зависимости от которых выполняется та или иная операция, и описаний самих операций.

Технология проектирования может быть представлена как совокупность трех составляющих:

- ☐ заданной последовательности выполнения технологических операций проектирования;
- ☐ критериев и правил, используемых для оценки результатов выполнения технологических операций;
- ☐ графических и текстовых средств (нотаций), используемых для описания проектируемой системы.

Каждая технологическая операция должна обеспечиваться следующими материальными и информационными ресурсами:

- ☐ данными, полученными от предыдущей операции (или исходными данными), представленными в стандартном виде;
- ☐ методическими материалами, инструкциями, нормативами и стандартами;
- ☐ программными и техническими средствами;
- ☐ исполнителями.

Результаты выполнения операции должны представляться в некотором стандартном виде, обеспечивающем их адекватное восприятие при выполнении следующей технологической операции (на которой они будут использоваться в качестве исходных данных).

Можно сформулировать следующий ряд общих требований, которым должна удовлетворять технология проектирования, разработки и сопровождения информационных систем:

- ☐ поддерживать полный жизненный цикл информационной системы;
- ☐ обеспечивать гарантированное достижение целей разработки системы с заданным качеством и в установленное время;
- ☐ обеспечивать возможность разделения крупных проектов на ряд подсистем — декомпозицию проекта на составные части, разрабатываемые группами исполнителей ограниченной численности, с последующей интеграцией составных частей;

ПРИМЕЧАНИЕ

Декомпозиция проекта позволяет повысить эффективность работ. Подсистемы, на которые разбивается проект, должны быть слабо связаны по данным и функциям. Каждая подсистема разрабатывается отдельной группой разработчиков. При этом необходимо обеспечить координацию работ и исключить дублирование результатов, получаемых каждой проектной группой.

- ☐ технология должна обеспечивать возможность ведения работ по проектированию отдельных подсистем небольшими группами (3–7 человек). Это обу-

словлено принципами управляемости коллектива и повышения производительности за счет минимизации числа внешних связей;

- ❑ обеспечивать минимальное время получения работоспособной системы;

ПРИМЕЧАНИЕ

Здесь имеется в виду не реализация информационной системы в целом, а разработка ее отдельных подсистем. Как правило, даже при наличии полностью завершенного проекта внедрение разработанной системы проводится последовательно, по отдельным подсистемам. Реализация же всей системы в сжатые сроки может потребовать привлечения большого числа разработчиков, при этом эффект может оказаться ниже, чем при реализации отдельных подсистем в более короткие сроки меньшим числом разработчиков.

- ❑ предусматривать возможность управления конфигурацией проекта, ведения версий проекта и его составляющих, возможность автоматического выпуска проектной документации и синхронизацию ее версий с версиями проекта;
- ❑ обеспечивать независимость выполняемых проектных решений от средств реализации системы — системы управления базами данных, операционной системы, языка и системы программирования.

Методология RAD — Rapid Application Development

На начальном этапе существования компьютерных информационных систем их разработка велась на традиционных языках программирования. Однако по мере возрастания сложности разрабатываемых систем и увеличения запросов пользователей (чему в значительной степени способствовал прогресс в области вычислительной техники, а также появление удобного графического интерфейса пользователя в системном программном обеспечении) потребовались новые средства, обеспечивающие значительное сокращение сроков разработки. Это послужило предпосылкой к созданию целого направления в области программного обеспечения — инструментальных средств для быстрой разработки приложений. Развитие этого направления привело к появлению на рынке программного обеспечения средств автоматизации практически всех этапов жизненного цикла информационных систем.

Основные особенности методологии RAD

Методология разработки информационных систем, основанная на использовании средств быстрой разработки приложений, получила в последнее время широкое распространение и приобрела название *методологии быстрой разработки приложений* — RAD (Rapid Application Development). Данная методология охватывает все этапы жизненного цикла современных информационных систем.

RAD — это комплекс специальных инструментальных средств быстрой разработки прикладных информационных систем, позволяющих оперировать с определенным набором графических объектов, функционально отображающих отдельные информационные компоненты приложений.

Под методологией быстрой разработки приложений обычно понимается процесс разработки информационных систем, основанный на трех основных элементах:

- ❑ небольшой команде программистов (обычно от 2 до 10 человек);
- ❑ тщательно проработанном производственном графике работ, рассчитанном на сравнительно короткий срок разработки (от 2 до 6 мес.);
- ❑ итерационной модели разработки, основанной на тесном взаимодействии с заказчиком — по мере выполнения проекта разработчики уточняют и реализуют в продукте требования, выдвигаемые заказчиком.

При использовании методологии RAD большое значение имеют опыт и профессионализм разработчиков. Группа разработчиков должна состоять из профессионалов, имеющих опыт в анализе, проектировании, программировании и тестировании программного обеспечения.

Основные принципы методологии RAD можно свести к следующему:

- ❑ используется итерационная (спиральная) модель разработки;
- ❑ полное завершение работ на каждом из этапов жизненного цикла не обязательно;
- ❑ в процессе разработки информационной системы необходимо тесное взаимодействие с заказчиком и будущими пользователями;
- ❑ необходимо применение CASE-средств и средств быстрой разработки приложений;
- ❑ необходимо применение средств управления конфигурацией, облегчающих внесение изменений в проект и сопровождение готовой системы;
- ❑ необходимо использование прототипов, позволяющее полнее выяснить и реализовать потребности конечного пользователя;
- ❑ тестирование и развитие проекта осуществляются одновременно с разработкой;
- ❑ разработка ведется немногочисленной и хорошо управляемой командой профессионалов;
- ❑ необходимо грамотное руководство разработкой системы, четкое планирование и контроль выполнения работ.

Объектно-ориентированный подход

Средства RAD дали возможность реализовывать совершенно иную по сравнению с традиционной технологией создания приложений: информационные объекты формируются как некие действующие модели (прототипы), чье функционирование согласовывается с пользователем, а затем разработчик может перейти непосредственно к формированию законченных приложений, не теряя из виду общей картины проектируемой системы.

Возможность использования подобного подхода в значительной степени является результатом применения принципов объектно-ориентированного проектирования. Применение объектно-ориентированных методов позволяет преодолеть одну из главных трудностей, возникающих при разработке сложных систем — ко-

лоссальный разрыв между реальным миром (предметной областью описываемой проблемы) и имитирующей средой.

Использование объектно-ориентированных методов позволяет создать описание (модель) предметной области в виде совокупности объектов — сущностей, объединяющих данные и методы обработки этих данных (процедуры). Каждый объект обладает своим собственным поведением и моделирует некоторый объект реального мира. С этой точки зрения объект является вполне осязаемой вещью, которая демонстрирует определенное поведение.

В объектном подходе акцент переносится на конкретные характеристики физической или абстрактной системы, являющейся предметом программного моделирования. Объекты обладают целостностью, которая не может быть нарушена. Таким образом, свойства, характеризующие объект и его поведение, остаются неизменными. Объект может только менять состояние, управляться или становиться в определенное отношение к другим объектам.

Широкое распространение объектно-ориентированное программирование получило с появлением визуальных средств проектирования, которые обеспечивают слияние (инкапсуляцию) данных с процедурами, описывающими поведение реальных объектов, в объекты программ, которые могут быть отображены определенным образом в графической пользовательской среде. Это позволило приступить к созданию программных систем, максимально похожих на реальные, и добиваться наивысшего уровня абстракции. В свою очередь, объектно-ориентированное программирование позволяет создавать более надежные коды, так как у объектов программ существует точно определенный и жестко контролируемый интерфейс.

При разработке приложений с помощью инструментов RAD используется множество готовых объектов, сохраняемых в общедоступном хранилище. Однако обеспечивается и возможность разработки новых объектов. При этом новые объекты могут разрабатываться как на основе существующих, так и «с нуля».

Инструментальные средства RAD обладают удобным графическим интерфейсом пользователя и позволяют на основе стандартных объектов формировать простые приложения без написания кода программы. Это является большим преимуществом RAD, так как в значительной степени сокращает рутинную работу по разработке интерфейсов пользователя (при использовании обычных средств разработка интерфейсов представляет собой достаточно трудоемкую задачу, отнимающую много времени). Высокая скорость разработки интерфейсной части приложений позволяет быстро создавать прототипы и упрощает взаимодействие с конечными пользователями.

Таким образом, инструменты RAD позволяют разработчикам сконцентрировать усилия на сущности реальных деловых процессов предприятия, для которого создается информационная система. В итоге это приводит к повышению качества разрабатываемой системы.

ПРИМЕЧАНИЕ

В данном разделе мы лишь поверхностно рассмотрели особенности и преимущества объектно-ориентированных методов проектирования. Более подробно этот вопрос будет обсуждаться далее.

Визуальное программирование

Применение принципов объектно-ориентированного программирования позволило создать принципиально новые средства проектирования приложений, называемые средствами *визуального программирования*. Визуальные инструменты RAD позволяют создавать сложные графические интерфейсы пользователя вообще без написания кода программы. При этом разработчик может на любом этапе наблюдать то, что закладывается в основу принимаемых решений.

Визуальные средства разработки оперируют в первую очередь со стандартными интерфейсными объектами — окнами, списками, текстами, которые легко можно связать с данными из базы данных и отобразить на экране монитора. Другая группа объектов представляет собой стандартные элементы управления — кнопки, переключатели, флажки, меню и т. п., с помощью которых осуществляется управление отображаемыми данными. Все эти объекты могут быть стандартным образом описаны средствами языка, а сами описания сохранены для дальнейшего повторного использования.

В настоящее время существует довольно много различных визуальных средств разработки приложений. Но все они могут быть разделены на две группы — универсальные и специализированные.

Среди универсальных систем визуального программирования сейчас наиболее распространены такие как Borland Delphi и Visual Basic. Универсальными мы их называем потому, что они не ориентированы на разработку только приложений баз данных — с их помощью могут быть разработаны приложения почти любого типа, в том числе и информационные приложения. Причем программы, разрабатываемые с помощью универсальных систем, могут взаимодействовать практически с любыми системами управления базами данных. Это обеспечивается как использованием драйверов ODBC или OLE DB, так и применением специализированных средств (компонентов).

Специализированные средства разработки ориентированы только на создание приложений баз данных. Причем, как правило, они привязаны к вполне определенным системам управления базами данных. В качестве примера таких систем можно привести Power Builder фирмы Sybase (естественно, предназначенный для работы с СУБД Sybase Anywhere Server) и Visual FoxPro фирмы Microsoft.

Поскольку задачи создания прототипов и разработки пользовательского интерфейса, по существу, слились, программист получил непрерывную обратную связь с конечными пользователями, которые могут не только наблюдать за созданием приложения, но и активно участвовать в нем, корректировать результаты и свои требования. Это также способствует сокращению сроков разработки и является важным психологическим аспектом, который привлекает к RAD все большее число пользователей.

Визуальные инструменты RAD позволяют максимально сблизить этапы создания информационных систем: анализ исходных условий, проектирование системы, разработка прототипов и окончательное формирование приложений становятся сходными, так как на каждом этапе разработчики оперируют визуальными объектами.

Событийное программирование

Логика приложения, построенного с помощью RAD, является событийно-ориентированной. Это означает следующее: каждый объект, входящий в состав приложения, может генерировать события и реагировать на события, генерируемые другими объектами. Примерами событий могут быть: открытие и закрытие окон, нажатие кнопки, нажатие клавиши клавиатуры, движение мыши, изменение данных в базе данных и т. п.

Разработчик реализует логику приложения путем определения обработчика каждого события — процедуры, выполняемой объектом при наступлении соответствующего события. Например, обработчик события «нажатие кнопки» может открыть диалоговое окно. Таким образом, управление объектами осуществляется с помощью событий.

Обработчики событий, связанных с управлением базой данных (DELETE, INSERT, UPDATE), могут реализовываться в виде триггеров на клиентском или серверном узле. Такие обработчики позволяют обеспечить ссылочную целостность базы данных при операциях удаления, вставки и обновления, а также автоматическую генерацию первичных ключей.

Фазы жизненного цикла в рамках методологии RAD

При использовании методологии быстрой разработки приложений жизненный цикл информационной системы состоит из четырех фаз:

- ☐ фаза анализа и планирования требований;
- ☐ фаза проектирования;
- ☐ фаза построения;
- ☐ фаза внедрения.

Рассмотрим каждую из них более подробно.

Фаза анализа и планирования требований

На данной фазе выполняются следующие работы:

- ☐ определяются функции, которые должна выполнять разрабатываемая информационная система;
- ☐ определяются наиболее приоритетные функции, требующие разработки в первую очередь;
- ☐ проводится описание информационных потребностей;

ПРИМЕЧАНИЕ

Определение указанных выше требований выполняется совместно будущими пользователями системы и разработчиками.

- ☐ ограничивается масштаб проекта;
- ☐ определяются временные рамки для каждой из последующих фаз;

- в заключение определяется сама возможность реализации данного проекта в установленных рамках финансирования, на имеющихся аппаратных и программных средствах.

Если реализация проекта принципиально возможна, то результатом фазы анализа и планирования требований будет список функций разрабатываемой информационной системы с указанием их приоритетов и предварительные функциональные и информационные модели системы.

Фаза проектирования

На фазе проектирования необходимым инструментом являются CASE-средства, используемые для быстрого получения работающих прототипов приложений.

ПРИМЕЧАНИЕ

Термин CASE (Computer Aided Software/System Engineering) используется в настоящее время в весьма широком смысле. Первоначальное значение термина CASE ограничивалось лишь вопросами автоматизации разработки программного обеспечения. Однако в дальнейшем значение этого термина расширилось и приобрело новый смысл, охватывающий процесс разработки сложных информационных систем в целом. Теперь под термином «CASE-средства» понимаются программные средства, поддерживающие процессы создания и сопровождения информационных систем, включая анализ и формулировку требований, проектирование прикладного программного обеспечения и баз данных, генерацию кода, тестирование, документирование, обеспечение качества, конфигурационное управление и управление проектом, а также другие процессы. Подробному рассмотрению CASE-технологий в данной книге посвящена глава 6 «Проектирование структуры базы данных».

Прототипы, созданные с помощью CASE-средств, анализируются пользователями, которые уточняют и дополняют те требования к системе, которые не были выявлены на предыдущей фазе. Таким образом, на данной фазе также необходимо участие будущих пользователей в техническом проектировании системы.

Далее на этой фазе проводится анализ и при необходимости корректировка функциональной модели системы. Детально рассматривается каждый процесс системы. При необходимости для каждого элементарного процесса создается частичный прототип: экран, диалог или отчет (это позволяет устранить неясности или неоднозначности). Затем определяются требования разграничения доступа к данным.

ПРИМЕЧАНИЕ

Для построения всех моделей и прототипов должны быть использованы именно те CASE-средства, которые будут затем применяться при построении системы. Данное требование связано с тем, что при передаче информации о проекте с этапа на этап может произойти фактически неконтролируемое искажение данных. Применение единой среды хранения информации о проекте позволяет избежать этой опасности.

После детального рассмотрения процессов определяется количество функциональных элементов разрабатываемой системы. Это позволяет разделить информационную систему на ряд подсистем, каждая из которых реализуется одной командой разработчиков за приемлемое для RAD-проектов время (порядка полутора месяцев). С использованием CASE-средств проект распределяется между различными командами — делится функциональная модель.

На этой же фазе происходит определение набора необходимой документации.

Результатами данной фазы являются:

- ☐ общая информационная модель системы;
- ☐ функциональные модели системы в целом и подсистем, реализуемых отдельными командами разработчиков;
- ☐ точно определенные с помощью CASE-средства интерфейсы между автономно разрабатываемыми подсистемами;
- ☐ построенные прототипы экранов, диалогов и отчетов.

ПРИМЕЧАНИЕ

Одной из особенностей применения методологии RAD на данной фазе является то, что каждый созданный прототип развивается в часть будущей системы. Таким образом, на следующую фазу передается более полная и полезная информация. (При традиционном подходе использовались средства прототипирования, не предназначенные для построения реальных приложений, поэтому разработанные прототипы не могли быть использованы на последующих фазах и просто «выбрасывались» после того, как выполняли задачу устранения неясностей в проекте.)

Фаза построения

На фазе построения выполняется собственно быстрая разработка приложения. На данной фазе разработчики производят итеративное построение реальной системы на основе полученных ранее моделей, а также требований нефункционального характера. Разработка приложения ведется с использованием визуальных средств программирования. Формирование программного кода частично выполняется с помощью автоматических генераторов кода, входящих в состав CASE-средств. Код генерируется на основе разработанных моделей.

На фазе построения также требуется участие пользователей системы, которые оценивают получаемые результаты и вносят коррективы, если в процессе разработки система перестает удовлетворять определенным ранее требованиям. Тестирование системы осуществляется непосредственно в процессе разработки.

После окончания работ каждой отдельной командой разработчиков производится постепенная интеграция данной части системы с остальными, формируется полный программный код, выполняется тестирование совместной работы данной части приложения с остальными, а затем тестирование системы в целом.

Завершается физическое проектирование системы, а именно:

- ☐ определяется необходимость распределения данных;
- ☐ производится анализ использования данных;
- ☐ производится физическое проектирование базы данных;

- ❑ определяются требования к аппаратным ресурсам;
- ❑ определяются способы увеличения производительности;
- ❑ завершается разработка документации проекта.

Результатом данной фазы является готовая информационная система, удовлетворяющая всем требованиям пользователей.

Фаза внедрения

Фаза внедрения в основном сводится к обучению пользователей разработанной информационной системы.

Так как фаза построения достаточно непродолжительна, планирование и подготовка к внедрению должны начинаться заранее, еще на этапе проектирования системы.

ПРИМЕЧАНИЕ

Приведенная схема разработки информационной системы не является универсальной. Вполне возможны различные отклонения от нее. Это связано с зависимостью схемы выполнения проекта от тех условий, при которых начинается разработка (например, создается совершенно новая система или на предприятии уже существует некоторая информационная система). Во втором случае существующая система может либо использоваться в качестве прототипа новой, либо интегрироваться в новую разработку в качестве одной из подсистем.

Ограничения методологии RAD

Несмотря на все свои достоинства, методология RAD тем не менее (как, впрочем, и любая другая методология) не может претендовать на универсальность. Ее применение наиболее эффективно при выполнении сравнительно небольших систем, разрабатываемых для вполне определенного предприятия.

При разработке же типовых систем, не являющихся законченным продуктом, а представляющих собой совокупность типовых элементов информационной системы, большое значение имеют такие показатели проекта, как управляемость и качество, которые могут войти в противоречие с простотой и скоростью разработки. Это связано с тем, что типовые системы обычно централизованно сопровождаются и могут быть адаптированы к различным программно-аппаратным платформам, системам управления базами данных, коммуникационным средствам, а также интегрироваться с существующими разработками. Поэтому для такого рода проектов необходим высокий уровень планирования и жесткая дисциплина проектирования, строгое следование заранее разработанным протоколам и интерфейсам, что снижает скорость разработки.

Методология RAD неприменима не только для создания типовых информационных систем, но и для построения сложных расчетных программ, операционных систем или программ управления сложными инженерно-техническими объектами — программ, требующих написания большого объема уникального кода.

Методология RAD не может быть использована для разработки приложений, в которых интерфейс пользователя является вторичным, то есть отсутст-

ует наглядное определение логики работы системы. Примерами таких приложений могут служить приложения реального времени, драйверы или службы.

Совершенно неприемлема методология RAD для разработки систем, от которых зависит безопасность людей, например систем управления транспортом или атомных электростанций. Это обусловлено тем, что итеративный подход, являющийся одной из основ RAD, предполагает, что первые версии системы не будут полностью работоспособны, что в данном случае может привести к серьезнейшим катастрофам.

Профили открытых информационных систем

Создание, сопровождение и развитие современных сложных информационных систем базируется на методологии построения таких систем как *открытых*. Такие системы создаются в процессе информатизации всех основных сфер современного общества: органов государственного управления, финансово-кредитной сферы, информационного обслуживания предпринимательской деятельности, производственной сферы, науки, образования. Развитие и использование открытых информационных систем неразрывно связано с применением определенных норм на основе методологии функциональной стандартизации информационных технологий.

Понятие профиля информационной системы

При создании и развитии сложных, распределенных, тиражируемых информационных систем требуется гибкое формирование и применение гармонизированных совокупностей базовых стандартов и нормативных документов разного уровня, выделение в них требований и рекомендаций, необходимых для реализации заданных функций системы. Для унификации и регламентирования такие совокупности базовых стандартов должны адаптироваться и конкретизироваться применительно к определенным классам проектов, функций, процессов и компонентов системы. В связи с этим выделилось и сформировалось понятие *профиля* информационной системы как основного инструмента функциональной стандартизации.

Профиль — это совокупность нескольких (или подмножество одного) базовых стандартов с четко определенными и гармонизированными подмножествами обязательных и факультативных возможностей, предназначенная для реализации заданной функции или группы функций.

Профиль формируется исходя из функциональных характеристик объекта стандартизации. В нем выделяются и устанавливаются допустимые возможности и значения параметров каждого базового стандарта и/или нормативного документа, входящего в профиль.

Профиль не должен противоречить использованным в нем базовым стандартам и нормативным документам. Он должен применять выбранные из альтернативных вариантов необязательные возможности и значения параметров в пределах допустимых.

На базе одной совокупности базовых стандартов могут формироваться и утверждаться различные профили для разных проектов информационных систем.

Ограничения базовых документов профиля и их согласованность, проведенная разработчиками профиля, должны обеспечивать качество, совместимость и корректное взаимодействие отдельных компонентов системы, соответствующих профилю, в заданной области его применения.

Базовые стандарты и профили в зависимости от проблемно-ориентированной области применения информационных систем могут использоваться как непосредственные директивные, руководящие или рекомендательные документы, а также как нормативная база, необходимая при выборе или разработке средств автоматизации технологических этапов или процессов создания, сопровождения и развития информационных систем.

Обычно рассматривают две группы профилей:

- регламентирующие архитектуру и структуру информационной системы;
- регламентирующие процессы проектирования, разработки, применения, сопровождения и развития системы.

В зависимости от области применения профили могут иметь разные категории и соответственно разные статусы утверждения:

- профили конкретной информационной системы, определяющие стандартизованные проектные решения в пределах данного проекта;
- профили информационной системы, предназначенные для решения некоторого класса прикладных задач.

Профили информационных систем унифицируют и регламентируют только часть требований, характеристик, показателей качества объектов и процессов, выделенных и формализованных на базе стандартов и нормативных документов. Другая часть функциональных и технических характеристик системы определяется заказчиками и разработчиками творчески, без учета положений нормативных документов.

Принципы формирования профиля информационной системы

Использование профилей информационных систем призвано решить следующие задачи:

- снижение трудоемкости проектов;
- повышение качества компонентов информационной системы;
- обеспечение расширяемости и масштабируемости разрабатываемых систем;
- обеспечение возможности функциональной интеграции в информационную систему задач, которые раньше решались отдельно;
- обеспечение переносимости прикладного программного обеспечения.

В зависимости от того, какие из указанных задач являются наиболее приоритетными, производится выбор стандартов и документов для формирования профиля.

Актуальность использования профилей информационных систем обусловлена современным состоянием стандартизации информационных технологий, которое характеризуется следующими особенностями:

- ❑ существует множество международных и национальных стандартов, которые не полностью и неравномерно удовлетворяют потребности в стандартизации объектов и процессов создания и применения сложных информационных систем;
- ❑ длительные сроки разработки, согласования и утверждения международных и национальных стандартов приводят к их консерватизму и хроническому отставанию от современных информационных технологий;
- ❑ функциональными стандартами поддерживаются и регламентированы только самые простые объекты и рутинные, массовые процессы: телекоммуникации, программирование, документирование программ и данных. Наиболее сложные и творческие процессы создания и развития крупных распределенных информационных систем — системный анализ и проектирование, интеграция компонентов и систем, испытания и сертификация — почти не поддерживаются требованиями и рекомендациями стандартов из-за трудности их формализации и унификации;
- ❑ совершенствование и согласование нормативных и методических документов в ряде случаев позволяет создать на их основе национальные и международные стандарты.

Подходы к формированию профилей информационных систем могут быть различными. В международной функциональной стандартизации информационных технологий принято довольно жесткое понятие профиля. Считается, что его основой могут быть только международные и национальные, утвержденные стандарты. Использование стандартов де-факто и нормативных документов фирм не допускается. При таком подходе затруднены унификация, регламентирование и параметризация множества конкретных функций и характеристик сложных объектов архитектуры и структуры современных информационных систем.

Другой подход к разработке и применению профилей информационных систем состоит в использовании совокупности адаптированных и параметризованных базовых международных и национальных стандартов и открытых спецификаций, отвечающих стандартам де-факто и рекомендациям международных консорциумов.

Эталонная модель среды открытых систем (OSE/RM) определяет разделение любой информационной системы на две составляющие: *приложения* (прикладные программы и программные комплексы) и *среду*, в которой эти приложения функционируют.

Между приложениями и средой определяются стандартизованные интерфейсы — Application Program Interface (API), которые являются необходимой частью профилей любой открытой системы. Кроме того, в профилях могут быть определены унифицированные интерфейсы взаимодействия функциональных частей друг с другом и интерфейсы взаимодействия между компонентами среды системы. Спецификации выполняемых функций и интерфейсов взаимодействия могут быть оформлены в виде профилей компонентов системы. Таким образом, профили информационной системы с иерархической структурой могут включать в себя:

- ❑ стандартизованные описания функций, выполняемых данной системой;
- ❑ функции взаимодействия системы с внешней для нее средой;
- ❑ стандартизованные интерфейсы между приложениями и средой информационной системы;
- ❑ профили отдельных функциональных компонентов, входящих в систему.

Для эффективного использования конкретного профиля необходимо:

- ❑ выделить объединенные логической связью проблемно-ориентированные области функционирования, где могут применяться стандарты, общие для одной организации или группы организаций;
- ❑ идентифицировать стандарты и нормативные документы, варианты их использования и параметры, которые необходимо включить в профиль;
- ❑ документально зафиксировать участки конкретного профиля, где требуется создание новых стандартов или нормативных документов, и идентифицировать характеристики, которые могут оказаться важными для разработки недостающих стандартов и нормативных документов этого профиля;
- ❑ формализовать профиль в соответствии с его категорией, включая стандарты, различные варианты нормативных документов и дополнительные параметры, которые непосредственно связаны с профилем;
- ❑ опубликовать профиль и/или продвигать его по формальным инстанциям для дальнейшего распространения.

При использовании профилей важное значение имеет обеспечение проверки корректности их применения путем тестирования, испытаний и сертификации. Для этого требуется создание технологии контроля и тестирования в процессе применения профиля. Данная технология должна поддерживаться совокупностью методик, инструментальных средств, составом и содержанием оформляемых документов на каждом этапе выполнения проекта.

Использование профилей способствует унификации при разработке тестов, проверяющих качество и взаимодействие компонентов проектируемой информационной системы. Профили должны определяться таким образом, чтобы тестирование их реализации можно было проводить по возможности наиболее полно по стандартизованной методике. При этом возможно применение ранее разработанных методик, так как международные стандарты и профили являются основой для создания общепризнанных аттестационных тестов.

Структура профилей информационных систем

Разработка и применение профилей являются органической частью процессов проектирования, разработки и сопровождения информационных систем. Профили характеризуют каждую конкретную информационную систему на всех стадиях ее жизненного цикла, задавая согласованный набор базовых стандартов, которым должна соответствовать система и ее компоненты.

Стандарты, важные с точки зрения заказчика, должны задаваться в ТЗ на проектирование системы и составлять ее первичный профиль. То, что не задано в ТЗ, первоначально остается на усмотрение разработчика системы, который,

руководствуясь требованиями ТЗ, может дополнять и развивать профили системы и впоследствии согласовывать их с заказчиком. Таким образом, профиль конкретной системы не является статичным, он развивается и конкретизируется в процессе проектирования информационной системы и оформляется в составе документации проекта системы.

В профиль конкретной системы включаются спецификации компонентов, разработанных в составе данного проекта, и спецификации использованных готовых программных и аппаратных средств, если эти средства не специфицированы соответствующими стандартами. После завершения проектирования и испытаний системы, в ходе которых проверяется ее соответствие профилю, профиль применяется как основной инструмент сопровождения системы при эксплуатации, модернизации и развитии.

Общая структура профиля информационной системы

Формирование и применение профилей конкретных информационных систем выполняется на основе использования международных и национальных стандартов, ведомственных нормативных документов, а также стандартов де-факто при условии доступности соответствующих им спецификаций. Для обеспечения корректного применения профилей их описания должны содержать:

- ☐ определение целей использования данного профиля;
- ☐ точное перечисление функций объекта или процесса стандартизации, определяемого данным профилем;
- ☐ формализованные сценарии применения базовых стандартов и спецификаций, включенных в данный профиль;
- ☐ сводку требований к информационной системе или к ее компонентам, определяющих их соответствие профилю, и требований к методам тестирования соответствия;
- ☐ нормативные ссылки на конкретный набор стандартов и других нормативных документов, составляющих профиль, с точным указанием применяемых редакций и ограничений, способных повлиять на достижение корректного взаимодействия объектов стандартизации при использовании данного профиля;
- ☐ информационные ссылки на все исходные документы.

На стадиях жизненного цикла информационной системы выбираются и затем применяются основные функциональные профили:

- ☐ профиль прикладного программного обеспечения;
- ☐ профиль среды информационной системы;
- ☐ профиль защиты информации в информационной системе;
- ☐ профиль инструментальных средств, встроенных в информационную систему.

Профиль прикладного программного обеспечения

Прикладное программное обеспечение всегда является проблемно-ориентированным и определяет основные функции информационной системы. Функциональные профили системы должны включать в себя согласованные базовые

стандарты. При использовании функциональных профилей информационных систем следует еще иметь в виду согласование этих профилей между собой. Необходимость такого согласования возникает, в частности, при использовании стандартизованных API, в том числе интерфейсов приложений со средой их функционирования и со средствами защиты информации. При согласовании функциональных профилей возможны также уточнения профиля среды системы и профиля встраиваемых инструментальных средств создания, сопровождения и развития прикладного программного обеспечения.

Профиль среды информационной системы

Профиль среды информационной системы должен определять ее архитектуру в соответствии с выбранной моделью обработки данных.

Стандарты интерфейсов приложений со средой (API) должны быть определены по функциональным областям профилей информационной системы. Декомпозиция структуры среды функционирования системы на составные части, выполняемая на стадии эскизного проектирования, позволяет детализировать профиль среды информационной системы по функциональным областям эталонной модели OSE/RM:

- ❑ область графического пользовательского интерфейса;
- ❑ область реляционных или объектно-ориентированных СУБД (например, стандарт языка SQL-92 и спецификации доступа к разным базам данных);
- ❑ область операционных систем с учетом сетевых функций, выполняемых на уровне операционной системы;
- ❑ область телекоммуникационной среды в части услуг и служб прикладного уровня: электронной почты, доступа к удаленным базам данных, передачи файлов, доступа к файлам и управления файлами.

Профиль среды распределенной системы должен включать стандарты протоколов транспортного уровня, стандарты локальных сетей (например, стандарт Ethernet IEEE 802.3 или стандарт Fast Ethernet IEEE 802.3u), а также стандарты средств сопряжения проектируемой информационной системы с сетями передачи данных общего назначения.

Выбор аппаратных платформ информационной системы связан с определением их параметров: вычислительной мощности серверов и рабочих станций в соответствии с проектными решениями по разделению функций между клиентами и серверами; степени масштабируемости аппаратных платформ; надежности. Профиль среды должен содержать стандарты, определяющие параметры технических средств и способы их измерения (например, стандартные тесты измерения производительности).

Профиль защиты информации

Профиль защиты информации должен обеспечивать реализацию политики информационной безопасности, разрабатываемой в соответствии с требуемой категорией безопасности и критериями безопасности, заданными в ТЗ на систему. Построение профиля защиты информации в распределенных системах клиент-сервер методически связано с точным определением компонентов системы,

ответственных за те или иные функции, службы и услуги, и средств защиты информации, встроенных в эти компоненты. Функциональная область защиты информации включает в себя следующие функции защиты, реализуемые разными компонентами системы:

- ☐ функции, реализуемые операционной системой;
- ☐ функции защиты от несанкционированного доступа, реализуемые на уровне программного обеспечения промежуточного слоя;
- ☐ функции управления данными, реализуемые СУБД;
- ☐ функции защиты программных средств, включая средства защиты от вирусов;
- ☐ функции защиты информации при обмене данными в распределенных системах, включая криптографические функции;
- ☐ функции администрирования средств безопасности.

Профиль защиты информации должен включать указания на методы и средства обнаружения в применяемых аппаратных и программных средствах недекларированных возможностей. Профиль должен также включать указания на методы и средства резервного копирования и восстановления информации при отказах и сбоях аппаратуры системы.

Профиль инструментальных средств

Профиль инструментальных средств, встроенных в информационную систему, должен отражать решения по выбору методологии и технологии создания, сопровождения и развития информационной системы. В этом профиле должны содержаться ссылки на описание выбранной методологии и технологии, выполненное на стадии эскизного проектирования системы.

Состав инструментальных средств определяется на основании решений и нормативных документов об организации сопровождения и развития информационной системы. При этом должны быть учтены правила и порядок, регламентирующие внесение изменений в действующие системы. Функциональная область профиля инструментальных средств, встроенных в систему, охватывает функции централизованного управления и администрирования:

- ☐ контроль производительности и корректности функционирования системы в целом;
- ☐ управление конфигурацией прикладного программного обеспечения, тиражированием версий;
- ☐ управление доступом пользователей к ресурсам системы и конфигурацией ресурсов;
- ☐ перенастройка приложений в связи с изменениями прикладных функций информационной системы;
- ☐ настройка пользовательских интерфейсов (экранных форм и отчетов);
- ☐ ведение баз данных системы;
- ☐ восстановление работоспособности системы после сбоев и аварий.

Дополнительные ресурсы, необходимые для функционирования встроенных инструментальных средств, такие как минимальный и рекомендуемый объем оперативной памяти, размеры требуемого дискового пространства и т. п., должны быть учтены в разделе проекта, относящемся к среде информационной системы.

Выбор инструментальных средств, встроенных в систему, должен производиться в соответствии с требованиями профиля среды. Ссылки на соответствующие стандарты, входящие в профиль среды, должны содержаться и в профиле инструментальных средств.

В этом профиле должны также содержаться ссылки на требования к средствам тестирования, которые необходимы для процессов сопровождения и развития системы и должны быть в нее встроены. В число встроенных в информационную систему средств тестирования должны входить средства функционального тестирования приложений, тестирования интерфейсов, системного тестирования и тестирования серверов/клиентов при максимальной нагрузке.

Стандарты и методики

Одним из важных условий эффективного использования информационных технологий является внедрение корпоративных стандартов. Они представляют собой соглашение о единых правилах организации технологии или управления. При этом за основу корпоративных могут приниматься отраслевые, национальные и даже международные стандарты.

Однако высокая динамика развития информационных технологий приводит к быстрому устареванию существующих стандартов и методик разработки информационных систем. Так, например, в связи со значительным прогрессом в области программного обеспечения и средств вычислительной техники наблюдается рост размеров и сложности информационных систем. При этом существенно меняются требования как к основным функциям и сервисным возможностям систем, так и к динамике изменения этих функций. В этих условиях применение классических способов разработки и обеспечения качества информационных систем становится малоэффективным и не приводит к уровню качества, адекватному реальным требованиям.

Для устранения вышеназванных недостатков в настоящее время разработаны стандарты открытых систем (в первую очередь стандарты на интерфейсы различных видов, включая лингвистические, и на протоколы взаимодействия). Однако разработка систем в новых условиях требует также новых методов проектирования и новой организации проектных работ. Проектирование и методическая поддержка организации разработки информационных систем (включая программное обеспечение (ПО) и базы данных (БД)) традиционно поддерживаются многими стандартами и фирменными методиками. Вместе с тем известно, что требуется адаптивное планирование разработки, в том числе в динамике процесса ее выполнения. Одним из способов адаптивного проектирования является разработка и применение профилей жизненного цикла информационных систем и программного обеспечения. Корпоративные стандарты образуют целостную систему, которая включает три их вида:

- стандарты на продукты и услуги;
- стандарты на процессы и технологии;
- стандарты на формы коллективной деятельности, или управленческие стандарты.

Виды стандартов

Существующие на сегодняшний день стандарты можно, отчасти условно, разделить на несколько групп по следующим признакам:

- *по предмету стандартизации.* К этой группе можно отнести функциональные стандарты (на языки программирования, интерфейсы, протоколы) и стандарты на организацию жизненного цикла создания и использования информационных систем и программного обеспечения;
- *по утверждающей организации.* Здесь можно выделить официальные международные, официальные национальные или национальные ведомственные стандарты (например, ГОСТы, ANSI, IDEF0/1), стандарты международных консорциумов и комитетов по стандартизации (например, консорциума OMG), стандарты «де-факто» — официально никем не утвержденные, но фактически действующие (например, стандартом «де-факто» долгое время были: язык взаимодействия с реляционными базами данных SQL и язык программирования C), фирменные стандарты (например, Microsoft ODBC);
- *по методическому источнику.* К этой группе относятся различного рода методические материалы ведущих фирм-разработчиков программного обеспечения, фирм-консультантов, научных центров, консорциумов по стандартизации.

ПРИМЕЧАНИЕ

Необходимо иметь в виду, что, хотя это и не очевидно, в каждую из указанных выше групп и подгрупп входят стандарты, существенно различающиеся по степени обязательности для различных организаций; конкретности и детализации содержащихся требований; открытости и гибкости, а также возможностям адаптации к конкретным условиям.

Ниже мы рассмотрим методику Oracle CDM (Custom Development Method) по разработке прикладных информационных систем под заказ, Международный стандарт ISO/IEC 12207:1995-08-01 01 на организацию жизненного цикла продуктов программного обеспечения и язык UML (Unified Modeling Language — универсальный язык моделирования), принятый консорциумом OMG (Object Management Group) как стандарт описания программного обеспечения.

Поскольку рассматриваемые стандарты представляют собой весьма объемные документы, изложенные на десятках и даже сотнях страниц, мы рассмотрим их лишь на уровне общей структуры и основных особенностей.

Методика Oracle CDM

Одним из уже сложившихся направлений деятельности фирмы ORACLE стала разработка методологических основ и производство инструментальных средств для автоматизации процессов разработки сложных прикладных систем, ориентированных на интенсивное использование баз данных. Методика Oracle CDM является развитием давно разработанной версии Oracle CASE-Method, применяемой в CASE-средстве Oracle CASE (в новых версиях — Designer/2000).

Основу CASE-технологии и инструментальной среды фирмы ORACLE составляют:

- ❑ методология структурного нисходящего проектирования, при которой разработка прикладной системы представляется в виде последовательности четко определенных этапов;
- ❑ поддержка всех этапов жизненного цикла прикладной системы, начиная с самых общих описаний предметной области до получения и сопровождения готового программного продукта;
- ❑ ориентация на реализацию приложений в архитектуре клиент-сервер с использованием всех особенностей современных серверов баз данных, включая декларативные ограничения целостности, хранимые процедуры, триггеры баз данных, и с поддержкой в клиентской части всех современных стандартов и требований к графическому интерфейсу конечного пользователя;
- ❑ наличие централизованной базы данных, *репозитария*, для хранения спецификаций проекта прикладной системы на всех этапах ее разработки. Такой репозиторий представляет собой базу данных специальной структуры, работающую под управлением СУБД ORACLE;
- ❑ возможность одновременной работы с репозитарием многих пользователей. Такой многопользовательский режим почти автоматически обеспечивается стандартными средствами СУБД ORACLE. Централизованное хранение проекта системы и управление одновременным доступом к нему всех участников разработки поддерживают согласованность действий разработчиков и не допускают ситуацию, когда каждый проектировщик или программист работает со своей версией проекта и модифицирует ее независимо от других;
- ❑ автоматизация последовательного перехода от одного этапа разработки к следующему. Для этого предусмотрены специальные утилиты, с помощью которых можно по спецификациям концептуального уровня (модели предметной области) автоматически получать первоначальный вариант спецификации уровня проектирования (описание структуры базы данных и состава программных модулей), чтобы на его основе после всех необходимых уточнений и дополнений автоматически генерировать готовые к выполнению программы;
- ❑ автоматизация различных стандартных действий по проектированию и реализации приложения: предусматривается генерация многочисленных отчетов по содержимому репозитария, обеспечивающих полное документирование текущей версии системы на всех этапах ее разработки; с помощью

специальных процедур предоставляется возможность проверки спецификаций на полноту и непротиворечивость.

Общая структура

Жизненный цикл формируется из определенных этапов (фаз) проекта и процессов, каждый из которых выполняется в течение нескольких этапов.

Методика Oracle CDM определяет следующие фазы жизненного цикла информационной системы:

- ☐ стратегия;
- ☐ анализ (формулирование детальных требований к прикладной системе);
- ☐ проектирование (преобразование требований в детальные спецификации системы);
- ☐ реализация (написание и тестирование приложений);
- ☐ внедрение (установка новой прикладной системы, подготовка к началу эксплуатации);
- ☐ эксплуатация (поддержка приложения и слежение за ним, планирование будущих функциональных расширений).

Первый этап связан с моделированием и анализом процессов, описывающих деятельность организации, технологические особенности работы. Целью является построение моделей существующих процессов, выявление их недостатков и возможных источников усовершенствования. Этот этап не является обязательным в случае, когда существующая технология и организационные структуры четко определены, хорошо понятны и не требуют дополнительного изучения и реорганизации.

ПРИМЕЧАНИЕ

Более точным названием первого этапа, вероятно, было бы «определение требований».

На втором этапе разрабатываются детальные концептуальные модели предметной области, описывающие информационные потребности организации, особенности функционирования и т. п. Результатом являются модели двух типов:

- ☐ информационные, отражающие структуру и общие закономерности предметной области;
- ☐ функциональные, описывающие особенности решаемых задач.

На третьей стадии (этапе проектирования) на основании концептуальных моделей вырабатываются технические спецификации будущей прикладной системы — определяются структура и состав базы данных, специфицируется набор программных модулей. Первоначальный вариант проектных спецификаций может быть получен автоматически с помощью специальных утилит на основании данных концептуальных моделей.

На этапе реализации создаются программы, отвечающие всем требованиям проектных спецификаций.

ПРИМЕЧАНИЕ

Использование генераторов приложений, входящих в состав DESIGNER/2000, позволяет полностью автоматизировать этот этап, существенно сократить сроки разработки системы и повысить ее качество и надежность.

Методика Oracle CDM выделяет следующие процессы, протекающие на протяжении жизненного цикла информационной системы:

- ☐ определение производственных требований;
- ☐ исследование существующих систем;
- ☐ определение технической архитектуры;
- ☐ проектирование и построение базы данных;
- ☐ проектирование и реализация модулей;
- ☐ конвертирование данных;
- ☐ документирование;
- ☐ тестирование;
- ☐ обучение;
- ☐ переход к новой системе;
- ☐ поддержка и сопровождение.

Процессы состоят из последовательностей задач, которые для разных процессов связаны с помощью явно обозначенных ссылок.

Особенности методики Oracle CDM

Отметим основные особенности методики Oracle CDM, определяющие область ее применения и присущие ей ограничения.

1. Степень адаптивности CDM ограничивается тремя моделями жизненного цикла:
 - *классическая* — предусматривает все этапы;
 - *быстрая разработка* — ориентированна на использование инструментов моделирования и программирования Oracle;
 - *облегченный подход* — рекомендуется в случае малых проектов и возможности быстро прототипировать приложения.
2. Методика не предусматривает включение дополнительных задач, которые не оговорены в CDM, и их привязку к остальным. Также исключено удаление задачи (и порождаемых ею документов), не предусмотренное ни одной из трех моделей жизненного цикла, и изменение последовательности выполнения задач по сравнению с предложенной.
3. Все модели жизненного цикла являются по сути каскадными. Даже «облегченный подход», несмотря на итерационность выполнения действий по прототипированию, сохраняет общий последовательный и детерминированный порядок выполнения задач.

4. Методика не является обязательной, но может считаться фирменным стандартом. При формальном применении степень обязательности полностью соответствует ограничениям возможностей адаптации.
5. Прикладная система рассматривается в основном как программно-техническая система — например, возможность выполнения организационно-структурных преобразований, практически всегда происходящих при переходе к новой информационной системе, в этой методике отсутствует.
6. CDM теснейшим образом опирается на использование инструментария Oracle, несмотря на утверждения о простом приспособлении CDM к проектам, в которых используется другой комплект инструментальных средств.
7. Методика Oracle CDM представляет собой вполне конкретный материал, детализированный до уровня заготовок проектных документов, рассчитанных на прямое использование в проектах информационных систем с опорой на инструментальные средства и СУБД фирмы Oracle.

Международный стандарт ISO/IEC 12207: 1995-08-01

Первая редакция ISO 12207 была подготовлена в 1995 году объединенным техническим комитетом ISO/IEC JTC1 «Информационные технологии, подкомитет SC7, проектирование программного обеспечения».

По определению, ISO 12207 — базовый стандарт процессов жизненного цикла ПО, ориентированный на различные виды ПО и типы проектов автоматизированных систем, в которых ПО является одной из составных частей. Стандарт определяет стратегию и общий порядок в создании и эксплуатации ПО, он охватывает жизненный цикл от концептуализации идей до завершения проекта.

Целесообразность совместного использования стандартов на информационные системы и на ПО обуславливается одним из положений ISO 12207, согласно которому процессы, используемые во время жизненного цикла ПО, должны быть совместимы с процессами, используемыми во время жизненного цикла автоматизированной системы.

Согласно ISO 12207, система — это объединение одного или нескольких процессов, аппаратных средств, программного обеспечения, оборудования и людей для обеспечения возможности удовлетворения определенных потребностей или целей.

ПРИМЕЧАНИЕ

В отличие от Oracle CDM, стандарт ISO 12207 в равной степени ориентирован на организацию действий каждой из двух сторон: поставщика (разработчика) и покупателя (пользователя); он может быть применен и в том случае, когда обе стороны — из одной организации.

Общая структура

В стандарте ISO 12207 не предусмотрено каких-либо этапов (фаз или стадий) жизненного цикла информационной системы. Данный стандарт определяет лишь ряд процессов, причем по сравнению с Oracle CDM стандарт ISO 12207 состоит из гораздо более крупных обобщенных процессов: приобретение,

поставка, разработка и т. п. Несколько утрируя, можно сказать, что один процесс ISO 12207 сопоставим со всеми процессами Oracle CDM вместе взятыми.

Согласно ISO 12207, каждый процесс подразделяется на ряд действий, а каждое действие — на ряд задач.

Очень важной особенностью ISO 12207 по сравнению с CDM является то, что каждый процесс, действие или задача инициируется и выполняется другим процессом по мере необходимости, причем нет заранее определенных последовательностей (естественно, при сохранении логики связей по исходным сведениям задач и т. п.).

Основные и вспомогательные процессы жизненного цикла

В стандарте ISO 12207 описаны пять основных процессов жизненного цикла программного обеспечения:

- ❑ *процесс приобретения* определяет действия предприятия-покупателя, которое приобретает информационную систему, программный продукт или службу программного обеспечения;
- ❑ *процесс поставки* определяет действия предприятия-поставщика, которое снабжает покупателя системой, программным продуктом или службой программного обеспечения;
- ❑ *процесс разработки* определяет действия предприятия-разработчика, которое разрабатывает принцип построения программного изделия и программный продукт;
- ❑ *процесс функционирования* определяет действия предприятия-оператора, которое обеспечивает обслуживание системы в целом (а не только программного обеспечения) в процессе ее функционирования в интересах пользователей. В дополнении к действиям, которые определяются разработчиком в инструкциях по эксплуатации (эта деятельность разработчика предусмотрена во всех трех рассматриваемых стандартах), определяются действия оператора по консультированию пользователей, получению обратной связи и др., которые он планирует сам и берет на себя соответствующие обязанности;
- ❑ *процесс сопровождения* определяет действия персонала, обеспечивающего сопровождение программного продукта, то есть управление модификациями программного продукта, поддержку его актуального состояния и функциональной пригодности; сюда же относятся установка программного изделия на вычислительной системе и его удаление.

Кроме основных, стандарт ISO 12207 оговаривает 8 вспомогательных процессов, которые являются неотъемлемой частью всего жизненного цикла программного изделия и обеспечивают должное качество проекта программного обеспечения. К вспомогательным процессам относятся:

- ❑ процесс решения проблем;
- ❑ процесс документирования;
- ❑ процесс управления конфигурацией;
- ❑ процесс обеспечения качества;

- ☐ процесс верификации;
- ☐ процесс аттестации;
- ☐ процесс совместной оценки;
- ☐ процесс аудита.

В стандарте ISO 12207 также определяются четыре организационных процесса:

- ☐ процесс управления;
- ☐ процесс создания инфраструктуры;
- ☐ процесс совершенствования;
- ☐ процесс обучения.

ПРИМЕЧАНИЕ

Под процессом совершенствования в стандарте ISO 12207 понимается не совершенствование информационной системы или программного обеспечения, а улучшение самих процессов приобретения, разработки, обеспечения качества и т. д., реально осуществляемых в организации.

И наконец, в стандарте ISO 12207 определен один особый процесс, называемый процессом адаптации, который определяет основные действия, необходимые для адаптации этого стандарта к условиям конкретного проекта.

Особенности стандарта ISO 12207

Все сказанное выше позволяет сформулировать следующие особенности стандарта ISO 12207.

1. Стандарт ISO 12207 имеет динамический характер, обусловленный способом определения последовательности выполнения процессов и задач, при котором один процесс при необходимости вызывает другой или его часть. Такой характер позволяет реализовать любую модель жизненного цикла.

ПРИМЕЧАНИЕ

Согласно стандарту ISO 12207, модель жизненного цикла — это структура, содержащая процессы, действия и задачи, которые осуществляются в ходе разработки, функционирования и сопровождения программного продукта в течение всей жизни системы, от определения требований до завершения ее использования.

2. Стандарт ISO 12207 обеспечивает максимальную степень адаптивности. Множество процессов и задач сконструировано так, что возможна их адаптация в соответствии с конкретными проектами информационных систем. Адаптация сводится к исключению процессов, видов деятельности и задач, неприемлемых в конкретном проекте.

ПРИМЕЧАНИЕ

Согласно ISO 12207, добавление уникальных или специфических процессов, действий и задач должно быть оговорено в контракте между сторонами. Причем «контракт» понимается в самом широком смысле — от юридически оформленного документа до неформального соглашения. Это соглашение может быть определено даже единственной стороной — как задача, поставленная самому себе.

3. Стандарт принципиально не содержит описания конкретных методов действий, а тем более заготовок решений или документации. Он лишь описывает архитектуру процессов жизненного цикла программного обеспечения, но не конкретизирует в деталях, как реализовывать или выполнять услуги и задачи, включенные в процессы. Данный стандарт не предписывает имена, форматы или точное содержание получаемой документации. Решения такого типа принимаются сторонами, использующими стандарт.
4. Обеспечение качества разных процессов выполняется с разной предусмотренной степенью организационной независимости контролирующей деятельности вплоть до обязательных требований к полной независимости проверяющего персонала от какой-либо прямой ответственности за проверяемые объекты. В отличие от CDM, контроль этого вида предусмотрен на самых ранних шагах разработки, начиная с анализа системных требований посредством их проверок на соответствие потребностям приобретения.
5. Степень обязательности рассматриваемого стандарта следующая: после решения организации о применении ISO 12207 организация берет на себя ответственность за указание минимального набора требуемых процессов и задач, которые обеспечивают согласованность с этим стандартом.
6. Стандарт содержит предельно мало описаний, направленных на проектирование базы данных. Это можно считать оправданным, так как разные системы и разные прикладные комплексы программного обеспечения могут не только использовать весьма специфические типы баз данных, но и вообще не использовать базу данных.

Ценность стандарта ISO 12207 в том, что он содержит наборы задач, характеристик качества, критериев оценки и т. п., дающие всесторонний охват проектных ситуаций. Например, при выполнении анализа требований к системе предусматривается, что:

- ☐ рассматривается область применения системы для определения требований, предъявляемых к ней;
- ☐ спецификация требований системы должна описывать: функции и возможности системы, области ее применения, организационные требования и требования пользователя, безопасность, защищенность, человеческие факторы, эргономику, связи, операции и требования сопровождения; проектные ограничения и квалификационные требования.

Далее, при выполнении анализа требований к программному обеспечению предусмотрено 11 классов характеристик качества, которые используются позже при обеспечении качества.

При этом разработчик должен установить и документировать в виде требований к программному обеспечению следующие спецификации и характеристики:

- ☐ функциональные и возможные спецификации, включая исполнение, физические характеристики и условия среды эксплуатации, при которых единица программного обеспечения должна быть выполнена;
- ☐ внешние связи (интерфейсы) с единицей программного обеспечения;

- ❑ квалификационные требования;
- ❑ спецификации надежности, включая спецификации, связанные с методами функционирования и сопровождения, воздействия окружающей среды и вероятностью травмы персонала;
- ❑ спецификации защищенности, включая спецификации, связанные с компрометацией точности информации;
- ❑ человеческие факторы спецификаций по инженерной психологии (эргономике), включая связанные с ручным управлением, взаимодействием человека и оборудования, ограничениями на персонал и областями, нуждающимися в концентрированном человеческом внимании, которые являются чувствительными к ошибкам человека и обучению;
- ❑ определение данных и требований к базе данных;
- ❑ установочные и приемочные требования поставляемого программного продукта в местах функционирования и сопровождения (эксплуатации);
- ❑ документацию пользователя;
- ❑ требования к интерфейсу пользователя.

ПРИМЕЧАНИЕ

Согласно стандарту ISO 12207, квалификационные требования — это набор критериев или условий, которые должны быть удовлетворены для того, чтобы квалифицировать программный продукт как подчиняющийся (удовлетворяющий условиям) его спецификациям и готовый для использования в целевой окружающей среде.

Хотя стандарт не предписывает конкретной модели жизненного цикла или метода разработки, он определяет, что стороны-участники при использовании стандарта ответственны за:

- ❑ выбор модели жизненного цикла для разрабатываемого проекта;
- ❑ адаптацию процессов и задач стандарта к этой модели;
- ❑ выбор и применение методов разработки программного обеспечения;
- ❑ выполнение действий и задач, подходящих для проекта программного обеспечения.

UML — универсальный язык моделирования

Универсальный язык моделирования, разработка которого началась с середины 90-х годов прошлого века с применением нескольких методов и нотаций описания информационных систем на базе объектно-ориентированной технологии, в настоящее время является общепринятым стандартом при документировании процесса разработки информационных систем и программного обеспечения. Как стандарт UML принят консорциумом OMG, в который входят все ведущие производители программного обеспечения.

Наиболее весомый вклад в разработку языка внесли известные специалисты Грэди Буч (Grady Booch), Джим Румбах (Jim Rumbaugh) и Ивар Якобсон (Ivar

Jacobson), на базе объединения методик каждого из них собственно и возник UML.

Язык UML постоянно совершенствуется. В настоящее время текущей спецификацией языка является версия 2.0. Кроме этого, сам язык предоставляет пользователю возможности расширения ядра языка для нужд конкретного производителя, хотя это и не рекомендуется делать по причине достаточности возможностей языка.

Первоначально UML создавался для упрощения описания информационных систем на базе применения объектно-ориентированной технологии. Сейчас средствами языка можно описывать и программное обеспечение, оперирующее в основном потоками данных и методами их обработки (процедурное, алгоритмическое проектирование).

Предшественники UML

Причиной, побудившей к созданию универсального языка описания программного обеспечения, явилась постоянно возрастающая сложность проектируемых информационных систем, которая, в свою очередь, диктуется усовершенствованием, увеличением сложности решаемых задач.

Когда количество объектов информационной системы не превышает 6–8 (то есть психологического предела, при котором человек еще способен оперировать информацией без записи и дополнительной тренировки), сложности при проектировании системы преодолимы без использования специальных средств. Такую информационную систему (для одного рабочего места, для небольшой компании) способен создать один человек. Когда же число объектов достигает тысяч и десятков тысяч, а число состояний и переходов между ними достигает миллионов, ни один специалист, каким бы образованным и опытным он ни был, не способен охватить всю систему целиком. К примеру, система навигации будет размещаться на тысячах автомобилей, морских и речных судах, железнодорожных составах, десятках искусственных спутников Земли и будет использовать тысячи компьютеров, а также всевозможные сети связи.

Такие информационные системы под силу создать только группе разработчиков, как правило, с разделением обязанностей на аналитиков, проектировщиков, программистов, тестеров и, конечно, менеджеров-руководителей проекта. А там, где трудится коллектив, просто необходим понятный всем инструмент общения. Для инженеров-механиков таким инструментом, понятным как его коллегам, так и простым рабочим, являются машиностроительные чертежи, для строителей — всевозможные эскизы, планы и т. д., для инженеров-электронщиков — электрические схемы, топологические чертежи. Для программистов инструментом такого общения долгое время являлись алгоритмы и естественный человеческий язык. Каждый, кто хоть немного разбирается в программировании, понимает, что за исключением очень простых примеров, без пояснений разобраться в коде программ, написанных не им самим, а часто и им самим, но давно, практически невозможно.

Диаграммы на языке UML можно назвать «иллюстрациями к программному коду».

Создание диаграмм аналогично созданию проекта в строительстве — можно обойтись и без него, например при строительстве сарая на дачном участке, однако чем больше здание, тем труднее это делать и тем неопределеннее конечный результат. Информационная система, описанная диаграммами UML, для разработчика показывает результат, который необходимо достичь в процессе проектирования.

Проблема коммуникации внутри коллектива разработчиков и необходимость документирования результатов проектирования с целью возможности в дальнейшем внести в систему усовершенствования является не единственной побудительной причиной создания UML.

Абсолютное большинство информационных систем, используемых в наши дни, в той или иной степени использует объектно-ориентированную технологию. Ключевым моментом в эффективности функционирования таких информационных систем является удачно построенная иерархия объектов, позволяющая использовать преимущества объектно-ориентированной технологии — инкапсуляцию, наследование и полиморфизм (подробнее об объектно-ориентированной разработке информационных систем далее в этой книге).

Выявление объектов-сущностей, их свойств и построение их иерархии является в большинстве случаев нетривиальной задачей, решение которой происходит с помощью методов объектно-ориентированного анализа.

Иерархия объектов должна отражать закономерности функционирования предметной области — то есть области деятельности человека, для которой создается информационная система. Прямолинейное проектирование, как правило, не является оптимальным. Из практики известно, что самыми удачными решениями являются «элегантные» системы, авторам которых удалось найти «изящную» комбинацию, необычный, неординарный ход, «изюминку» в построении иерархии объектов информационной системы.

Построение иерархии объектов требует опыта и усердной работы аналитиков и системных проектировщиков. Для облегчения их интеллектуального труда весьма кстати пришлось возможности нотации (*lam. notatio* — обозначение, система записи), с помощью которых можно легко и понятно фиксировать возникшие в сознании идеи. Этой цели служат всевозможные кружочки, квадратiki, стрелки и линии, с помощью которых человек пытается зафиксировать свои мысли в объектной форме — на листе бумаги или в электронном документе. С появлением соответствующих методик, а впоследствии и UML такая запись становится стандартизированной и понятной другим людям.

ПРИМЕЧАНИЕ

Проблемой, ставящей под угрозу осуществление крупного проекта, является уход из команды разработчиков одного или нескольких ведущих специалистов. Использование стандарта документирования процесса разработки минимизирует риски срыва работы над проектом в таких случаях.

Универсальный язык моделирования возник не на пустом месте. С середины 1980-х годов ведутся разработки способов, позволяющих автоматизировать процесс создания иерархии объектов. Некоторые из них, например CRC-карточки, не потеряли своей актуальности.

Словарь предметной области

Метод заключается в построении словаря терминов, используемого специалистами, для которых создается система, с целью наиболее полного понимания поставленных задач проектировщиками, которые, как правило, специалистами в этой области не являются.

Словарь содержит наименование термина и его краткую расшифровку. По мере работы над системой словарь предметной области может дополняться. Составлением словаря занимаются те из проектировщиков и аналитиков, кто имеет непосредственный контакт с конечными пользователями.

Использование в работе такого словаря весьма полезно и эффективно при решении практически любых задач.

Диаграммы сущность—связь

Метод построения диаграмм сущность—связь основан на выявление форм взаимосвязи и взаимодействия сущностей. Подробнее метод описан в главе 6 настоящей книги, где его использование показано на примере проектирования структуры базы данных.

Метод Аббота

Метод заключается в описании задачи на простом человеческом языке и анализе полученного текста. Существительные в этом случае принимаются как вероятные кандидаты на роль объектов-сущностей, а глаголы — как методы этих сущностей.

Этот метод поддается автоматизации, в частности такие системы были построены в Токийском технологическом институте и фирмой Fujitsu.

CRC-карточки

Аббревиатура CRC означает Class-Responsibilities-Collaborators (Класс—Ответственность—Участники). Метод впервые предложили Бек и Каннингхэм для обучения объектно-ориентированному программированию.

CRC-карточки представляют собой обычные картонные карточки 10 на 15 сантиметров, на которых карандашом сверху пишется название класса, слева — за что отвечает этот объект, справа — с какими классами он взаимодействует (сотрудничает, обменивается сообщениями).

В ходе анализа появляются новые карточки, в старые вносятся изменения. Может возникнуть ситуация, когда один и тот же класс (объект) будет слишком большим, что на стадии реализации системы приведет к постоянному его использованию, громоздкости его кода; в этом случае целесообразно разбить его на несколько классов или передать часть функций другому классу.

ПРИМЕЧАНИЕ

Возможна ситуация, что аналогичные сообщения будут передаваться между различными классами. Возможно выделение таких сообщений в отдельный класс для удобства отладки и внесения изменений. Так поступают, например, при обращениях к базе данных с помощью SQL-запросов, выделяя такие транзакции в отдельный класс.

Карточки раскладываются в разном порядке, что помогает определить возможные варианты наследования свойств и методов, движения потоков данных.

Метод Буча

Метод Буча явился основой создания UML. Предложенная им графическая нотация достаточно распространена и наряду с UML используется в системах автоматизации процесса разработки программного обеспечения, в частности в системе Rational Rose.

Применяемые в методике Буча обозначения несколько отличаются от обозначений, принятых в UML. Так, класс в нотации UML представляет собой прямоугольник, в методике Буча — облако, каким его рисуют дети, что по замыслу автора символизирует абстрактность этого понятия. Кроме того, язык UML более формализован за счет наличия метамодели языка.

ПРИМЕЧАНИЕ

Мы привели далеко не полный перечень методов, использовавшихся для описания и моделирования информационных систем. Именно их большое количество послужило побудительной причиной создания унифицированного метода.

Структура UML

В структуре универсального языка моделирования выделяют две основные составляющие:

- метамодель;
- правила построения диаграмм.

Метамодель представляет собой описание общей структуры языка, основных понятий объектно-ориентированного проектирования: класс, объект, событие, ассоциация, автомат, наследование и прочих, а также методов расширения ядра UML. Описания используемых терминов в общем совпадают с определениями, приводимыми в этой книге. Мы не будем подробно останавливаться на них.

ПРИМЕЧАНИЕ

Описание метамодели приводится на естественном человеческом языке с применением конструкций самого языка UML, что, однако, не создает трудностей при ее изучении.

Наличие метамодели придает языку UML строгость и выгодно отличает его от других методик.

Основной интерес для проектировщика представляют правила построения диаграмм UML, основными разновидностями которых являются:

- диаграммы прецедентов (use case diagram, диаграммы вариантов использования);
- диаграммы классов (class diagram);
- диаграммы состояний (statechart diagram);
- диаграммы активности (activity diagram, диаграммы деятельности);

- ❑ диаграммы взаимодействия (interaction diagrams):
 - диаграммы последовательности (sequence diagram);
 - диаграммы кооперации (collaboration diagram, диаграммы сотрудничества);
- ❑ диаграммы компонентов (component diagram);
- ❑ диаграммы развертывания (deployment diagram).

Именно диаграммы в нотации UML служат удобным средством передачи информации об особенностях построения информационной системы между участниками проекта. Часто задание на проектирование рядовому программисту представляется именно в виде набора диаграмм UML.

ПРИМЕЧАНИЕ

Юридически к категории программы для ЭВМ относится не только код на каком-либо языке программирования и исполняемый машинный код, но и все материалы, полученные в ходе процесса разработки программы, в первую очередь это касается документированных решений, применяемых при проектировании системы, как на естественном человеческом языке, так и с помощью UML. Это еще один аргумент к активному использованию UML при разработках информационных систем, так как в случае возникновения спора UML-диаграммы будут удобными доказательствами необходимого по закону признака оригинальности программы для ЭВМ.

При проектировании информационной системы, как правило, составляется множество диаграмм одного и того же вида: множество диаграмм прецедентов, несколько диаграмм классов, множество диаграмм активности.

Необязательно всегда составлять все диаграммы. UML создан для облегчения процесса разработки, а не для утомительного документирования всех шагов разработки. Некоторые из диаграмм могут отсутствовать.

Последовательность построения диаграмм также свободна.

Среди всех диаграмм следует выделить диаграмму классов, так как исходя из нее возможна автоматическая генерация части программного кода будущей информационной системы, содержащей описания классов и объектов (предварительную декларацию, предварительное объявление в разделе типов и переменных), а также заголовки методов объектов, реализацию которых необходимо писать вручную. А также следует выделить диаграммы последовательности и диаграммы кооперации, которые являются взаимно обратимыми, то есть могут быть преобразованы друг в друга. Такие возможности предоставляют системы автоматизированного проектирования, наиболее удачной и известной из которых является система Rational Rose фирмы Rational Software.

ПРИМЕЧАНИЕ

При построении диаграмм UML общепринято использование языка объектных ограничений (Object Constraint Language, OCL), разработанного фирмой IBM. Язык OCL похож на язык SQL, но создан специально для навигации и получения данных из объектов. Ввиду ограниченности объема книги мы его рассматривать не будем.

Диаграмма прецедентов

Диаграмма прецедентов служит для выявления и формального представления требований заказчика к проектируемой системе, то есть какие возможности система будет представлять конечному пользователю, какая информация необходима для обработки запроса пользователя. При этом механизм функционирования системы от пользователя скрыт и на диаграмме прецедентов не показывается.

Конечный пользователь (actor, актер), в роли которого может выступать человек (например, покупатель или оператор), техническое устройство (например, мобильный телефон), изображается в виде стилизованной фигурки человека (рис. 3.1).

Если же в качестве пользователя выступает сама система, что возможно, например, при предоставлении каких-либо функций определенному классу, для изображения актера в этом случае рекомендуется использовать соответствующее обозначение класса.

Пользователь использует систему определенным образом. Такое использование — прецедент (вариант использования) — обозначается на диаграмме овалом, внутри которого пишется наименование варианта использования (рис. 3.2).



Рис. 3.1. Графическое изображение пользователя



Рис. 3.2. Графическое изображение прецедента

Для пояснения содержания диаграмм используют примечания (notes), обозначаемые в виде листа бумаги с загнутым углом (рис. 3.3).

Текст примечания записывается внутри этого листа. Примечание соединяется пунктирной линией с тем элементом диаграммы, к которому оно относится.

ПРИМЕЧАНИЕ

Используемые на диаграммах обозначения являются общими для всех видов диаграмм. Так, обозначение примечания или прецедента на всех диаграммах будет одинаковым.

Еще одним наиболее часто используемым на диаграммах прецедентов элементом является интерфейс.

Интерфейс — это совокупность операций, предоставляемых классом или компонентом. Интерфейс описывает поведение класса или компонента, видимое извне. Он определяет только описание (спецификации) операций класса или компонента, но никогда не определяет физические реализации операций.

Интерфейс представляет собой сущность, которая предоставляет пользователю возможность совершить определенное действие, получить информацию. Пользователю интерфейс может быть доступен в качестве датчика, обращения

к базе данных, кнопки, бланка заявления — то есть устройства или определенной операции. Графически интерфейс обозначается небольшим кружком, рядом с которым указывается его наименование (рис. 3.4).

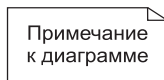


Рис. 3.3. Графическое изображение примечания

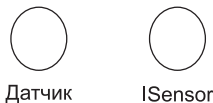


Рис. 3.4. Графическое изображение интерфейса

В нотации UML английские имена интерфейсов принято начинать с буквы I.

ПРИМЕЧАНИЕ

Относительно имен компонентов диаграмм разработчиками программного обеспечения выработана рекомендация: изначально, при построении основ системы использовать имена компонентов на русском языке, что делает разработку более понятной. В дальнейшем постепенно, а к завершению разработки полностью заменить названия на английские, которые могут быть восприняты компиляторами.

Между компонентами диаграммы прецедентов могут существовать различные отношения. Отношения могут быть между пользователями и прецедентами, между несколькими пользователями. Пользователь может взаимодействовать с несколькими вариантами использования.

В нотации UML определены следующие виды отношений между компонентами на диаграммах прецедентов:

- ❑ *отношение ассоциации* (association relationship) — устанавливает роль пользователя по отношению к системе. обозначается сплошной линией между пользователем и прецедентом (рис. 3.5, а);
- ❑ *отношение расширения* (extend relationship) — определяет взаимосвязь прецедента с прецедентом, возможности которого он может использовать. Графически обозначается пунктирной стрелкой с пометкой «extend» от дополняющего прецедента к расширяемому. Случай, изображенный на рис. 3.5, б может означать, что при определенных условиях прецедент В может быть дополнен прецедентом А. На практике это может означать, например, дополнительные, помимо обычных, меры к идентификации личности человека;
- ❑ *отношение обобщения* (generalization relationship) — показывает, что компонент (пользователь или прецедент) является частным случаем другого компонента. Графически обозначается непрерывной стрелкой от общего к частному (рис. 3.5, в);
- ❑ *отношение включения* (include relationship) — указывает на включение прецедента в качестве составной части другого прецедента. Один и тот же прецедент может быть включен в несколько более крупных прецедентов. Графически данное отношение обозначается пунктирной линией со стрелкой, направленной от базового прецедента к включаемому с пометкой «include» (рис. 3.5, г).

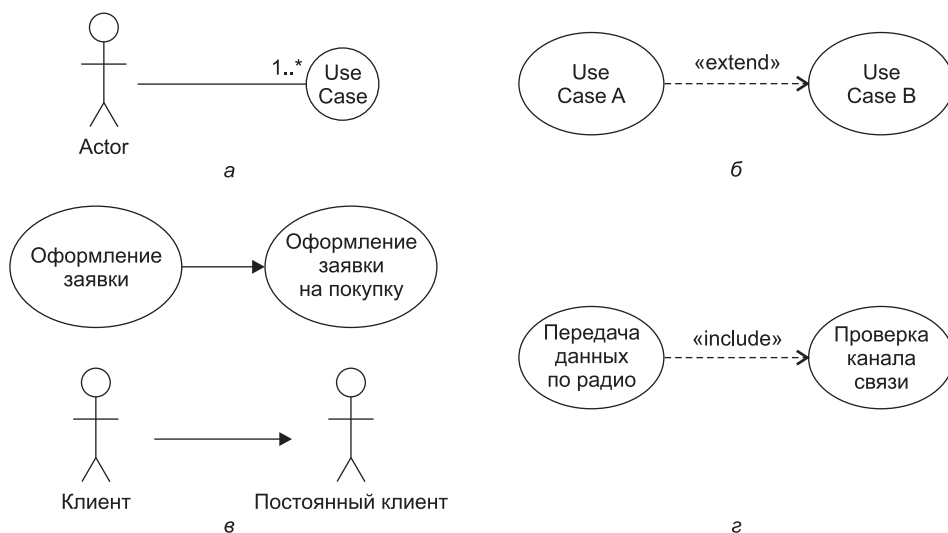


Рис. 3.5. Графическое изображение отношений на диаграммах прецедентов

Цифры над стрелкой (см. рис. 3.5, *а*) обозначают кратность ассоциации (multiplicity) и показывают количество возможных компонентов данной ассоциации. Случай на рисунке означает, что один и тот же пользователь может использовать систему данным образом любое количество (обозначается «звездочкой») раз.

Проиллюстрируем изложенное выше на примере действий дежурного врача при поступлении пациента в больницу через приемный покой. Дежурный врач организует прием пациента, который включает оформление истории болезни, проведение анализов, первичный осмотр, оповещает родственников пострадавшего. В случае тяжелого состояния пациента он направляется в реанимацию. Если состояние пациента безнадежно, от родственников испрашивается согласие на трансплантацию органов. Разрабатываемая информационная система должна автоматизировать выдачу направлений на анализы, выдавать пакет документов для оформления согласия родственников. Истории болезни в организации ведутся в бумажной форме (результаты анализов в историю болезни вклеиваются). На рис. 3.6 представлен возможный вариант диаграммы прецедентов для данного случая.

Диаграмма прецедентов в таком виде наглядно представляет первичные требования заказчика — в данном случае медицинского учреждения. Совокупность таких диаграмм в идеале должна служить полным описанием требований к функциональности системы. На практике требования могут изменяться. Графическое представление требований к системе значительно уменьшает сроки их согласования между заказчиком и разработчиками.

ПРИМЕЧАНИЕ

На диаграммах прецедентов не указывается, в какой последовательности выполняются операции. Данная информация может содержаться на диаграммах активности, взаимодействия, состояний.

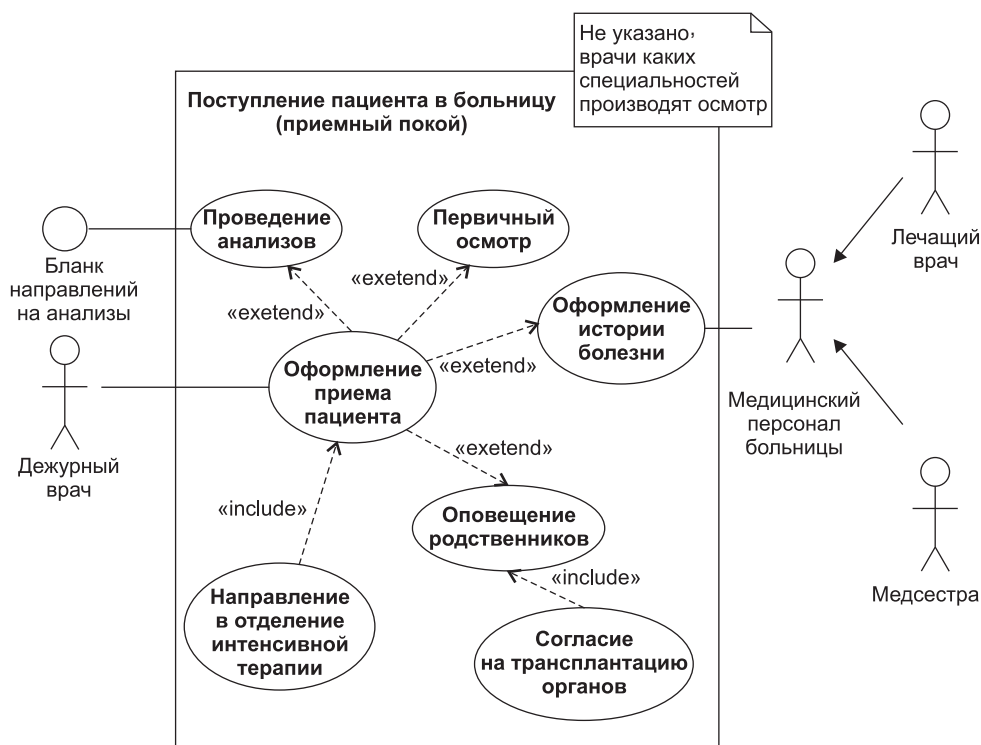


Рис. 3.6. Прием пациента в больницу

Пакеты

Для сложных систем возникает необходимость разложить их на несколько составляющих, причем таким образом, чтобы это разбиение отражалось в обозначении компонентов. Для этих целей в языке UML служат пакеты (рис. 3.7).

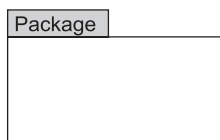


Рис. 3.7. Графическое изображение пакета

Компоненты, относящиеся к определенному пакету, могут быть доступны вне пакета, если указать имя компонента с указанием на его принадлежность определенному пакету через двойное двоеточие. Например, `Имя_Пакета::Имя_Компонента`. То есть пакет представляет собой самостоятельное пространство имен.

Как правило, элементы модели, входящие в пакет, логически связаны между собой.

Если количество классов в системе достаточно велико, иногда прибегают к построению диаграмм пакетов, разбивая систему на части на более высоком уровне, чем диаграммы классов.

Диаграмма классов

Под классом в языке UML понимается множество объектов, обладающих одинаковой структурой, свойствами, отношениями с другими объектами.

Графически в самом общем виде класс изображается прямоугольником с четырьмя секциями (рис. 3.8). Любая из секций, кроме имени класса, может отсутствовать. В этом случае она не изображается либо оставляется пустой. Как правило, на начальном этапе проектирования разработчик располагает только общими представлениями о будущей структуре системы. В дальнейшем, по мере разработки, уточняются роли, свойства каждого класса, что находит отражение на соответствующих диаграммах классов.

Абстрактным классом называется класс, который не имеет экземпляров. Их весьма удобно использовать при конструировании иерархии классов в качестве промежуточных элементов. Все, что касается абстрактных классов, в языке UML выделяется курсивом (в нотации Буча — абстрактный класс помечается буквой А в треугольнике, обращенном вершиной вниз).

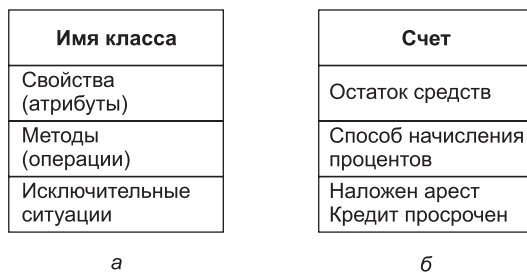


Рис. 3.8. Графическое изображение класса

Имя класса в языке UML принято писать полужирным шрифтом в самой верхней секции графического изображения класса (абстрактного класса — полужирным шрифтом и курсивом). Здесь же указывается информация об отношении этого класса к абстрактным классам, от которых он образован, а также служебная информация о процессе проектирования класса (лицо, ответственное за проектирование класса, язык реализации, номер версии и т. д.).

Свойства класса — это параметры, значения которых определяют состояние экземпляров класса (объектов).

В общем виде формат записи свойства можно представить в виде строки:

видимость имя_свойства [кратность] : тип_свойства = значение_по_умолчанию

где:

□ видимость — видимость свойства для других классов, принимает следующие значения:

○ public (или значок +) — свойство класса доступно любому другому классу;

- `protected` (или значок #) — свойство класса доступно только экземплярам этого класса и потомкам данного класса;
- `private` (или значок -) — свойство класса доступно только экземплярам этого класса.

Отсутствие указания на категорию доступа означает, что видимость не указывается.

ПРИМЕЧАНИЕ

Как уже отмечалось, диаграмма классов может использоваться для автоматической генерации программного кода на выбранном языке программирования. Следует иметь в виду, что в разных языках значения видимости свойства (атрибута), задаваемые по умолчанию, являются разными.

- `тип` — определяет тип свойства. Тип свойства выбирается исходя из языка реализации проекта (C++, Delphi, Java и др.).
- `кратность` — показывает диапазон принимаемых свойством значений с учетом типа (так, если в качестве кратности свойства `имя_клиента` указан диапазон `1..5` и тип `String`, то это свойство может принимать в том числе следующие значения: Иван Петров, Уильям Джефферсон Клинтон, Мехти Асадулла оглы Мамедов, то есть содержащие не более 5 слов; если же в качестве кратности свойства `доход` указан диапазон `0..*` и тип `Currency`, то значение свойства `доход` может принимать любое положительное значение в денежном формате).
- `значение_по_умолчанию` — определяет начальное значение свойства, которое в последующем можно изменять программно. Если же в строке определения свойства вместо знака `=` указать `==`, то значение по умолчанию программно изменить будет нельзя.

Запись свойства класса Менеджер

Заработная плата [`0..*`] : `Currency` = 700 \$

означает, что по умолчанию всем сотрудникам, занимающим эту должность, первоначально устанавливается заработная плата 700 долларов.

ПРИМЕЧАНИЕ

Язык UML кроме проектирования информационных систем нашел применение в сфере аналитики бизнеса. Отображенная на диаграммах структура фирмы и ее деятельность позволяют выявить, например, перегруженность определенных отделов, участки, тормозящие скорость принятия управленческих решений. На основе полученных данных вырабатываются рекомендации по совершенствованию деятельности организаций. Диаграммы UML в этом случае служат инструментом анализа и иллюстраций выводов.

Методы класса — это способы обработки свойств класса. Они характеризуют поведение экземпляров класса.

Формат записи обозначения метода можно представить в виде:
видимость имя_метода (список параметров) : тип значения {строка-свойство},
где:

- видимость — определяется аналогично видимости свойств;
- имя_метода — является его идентификатором. Наличие круглых скобок обязательно;
- список параметров — перечень разделенных запятой формальных параметров, каждый из которых может быть представлен в следующем виде:
вид_параметра имя_параметра : тип_параметра = значение по умолчанию
 - вид_параметра: входной (передаваемый методу), выходной (возвращаемый методом), универсальный (значение параметра может изменяться методом). Обозначается соответственно одним из трех слов: in, out или inout;
 - тип_параметра — определяется языком реализации класса;
- тип_значения — определяется языком реализации класса;
- строка-свойство — указывает значение свойства, которое может относиться к данному параметру. Метод, не изменяющий состояние объекта, обозначается строкой-свойством {query}. Строка-свойство {concurrency = sequential} указывает на необходимость обращения к методу во время вызова другого метода, {concurrency = concurrent} — возможность параллельного вызова методов. Строка-свойство {concurrency = guarded} обращает внимание на необходимость принятия дополнительных мер по контролю исключительных ситуаций.

Имя и тип метода, с областью действия на весь класс подчеркивается. Например, изменение фона одного окна программы автоматически изменяет цвет фона всех окон системы, как это имеет место при щелчке правой кнопкой мыши на свободном участке рабочего стола и изменении вида окон Windows на вкладке Оформление окна Свойства:Экран. По умолчанию областью действия метода определяется экземпляр класса (объект). В приведенном примере изменение цвета фона окна (формы) затронет только это окно.

Абстрактные методы — то есть методы, не задействованные в данном классе, выделяются курсивом или помечаются строкой-свойством {abstract}.

Примером обозначения метода «открыть» класса File служить запись:
+ open ()

этот метод не возвращает никакого результата своего выполнения.

Запись:

издать приказ (сотрудник) : приказ по форме 1-А "отпуск"}

может означать, что результатом вызова этого метода является автоматическая генерация приказа по организации, в котором сотруднику предоставляется ежегодный оплачиваемый отпуск.

Между компонентами диаграммы классов могут существовать различные отношения.

В нотации UML определены следующие виды отношений между компонентами на диаграммах классов.

- *Отношение зависимости* (dependency relationship) — указывает на общую взаимосвязь компонентов диаграммы. Графически эта связь изображается пунктирной стрелкой от зависимого класса к независимому (рис. 3.9).

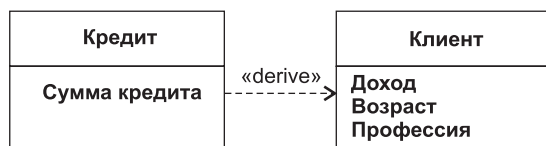


Рис. 3.9. Графическое изображение отношения зависимости

Ключевое слово (стереотип) над стрелкой означает, что свойства класса кредит, в частности сумма кредита, выдаваемая банком заемщику, может быть определена исходя из характеристики заемщика (свойств класса Клиент): ежемесячного дохода, возраста и др.

Ключевое слово является необязательным элементом. Кроме значения «derive», может принимать значения:

- access — указывает на доступность свойств и методов независимого класса для зависимого;
 - import — открытые свойства и методы независимого класса (источника) становятся частью зависимого класса, как если бы они были объявлены непосредственно в нем;
 - bind — класс может использовать другой класс в качестве шаблона;
 - refine — зависимый класс уточняет класс-источник в силу исторических причин.
- *Отношение ассоциации* (association relationship) — устанавливает некоторую связь между классами системы. Графически это отношение обозначается сплошной линией между классами (рис. 3.10).

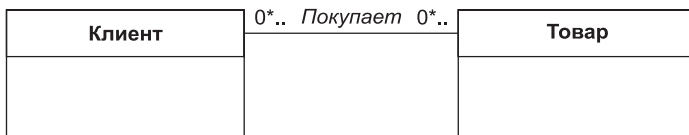


Рис. 3.10. Графическое изображение отношения ассоциации

Пример на рисунке означает, что клиенты делают покупки. При этом любой покупатель может купить любой товар или не купить никакой, и любой товар может быть продан любому покупателю или не быть купленным никем.

Отношение ассоциации может связывать большее количество классов (N-арная ассоциация). На диаграмме классов такая ассоциация изображается ромбом (рис. 3.11).

Приведенная ассоциация указывает, что покупатель может приобрести товар у любого из десяти поставщиков, входящих в данную систему продаж.

Частными случаями отношений ассоциации являются: исключаяющая ассоциация, отношение агрегации и отношения композиции:

- *исключаяющая ассоциация* (xor-association) — указывает на возможность связи определенного класса с только одним из нескольких классов (рис. 3.12);

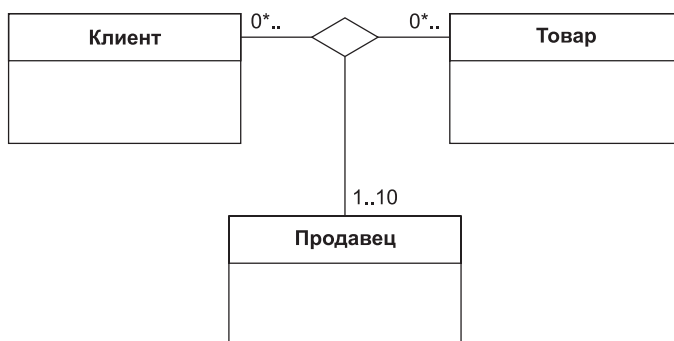


Рис. 3.11. N-арная ассоциация

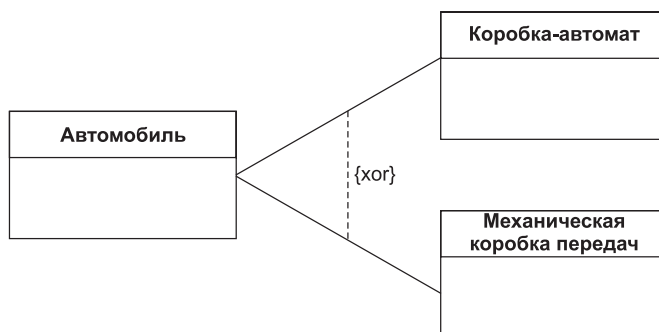


Рис. 3.12. Графическое изображение исключаящей ассоциации

- *отношение агрегации* означает включение нескольких классов в другой класс. Графическое изображение отношения агрегации показано на рис. 3.13 и означает, что в состав класса Сервиз в качестве самостоятельных единиц могут входить тарелки и другие столовые приборы;



Рис. 3.13. Графическое изображение отношения агрегации

ПРИМЕЧАНИЕ

Не все языки программирования поддерживают такие конструкции.

- *отношение композиции* показывает, что компонент состоит из нескольких частей, которые, в отличие от отношения агрегации, не могут использоваться отдельно. Графическое изображение отношения композиции похоже на изображение отношения агрегации (рис. 3.14), что отражает их сходство по сути. Так, составными частями класса Газета будут редакционные статьи, фотоиллюстрации, реклама, которые не могут быть использованы отдельно от самой газеты, если под газетой понимать лист бумаги. Отношение композиции может иметь кратность.

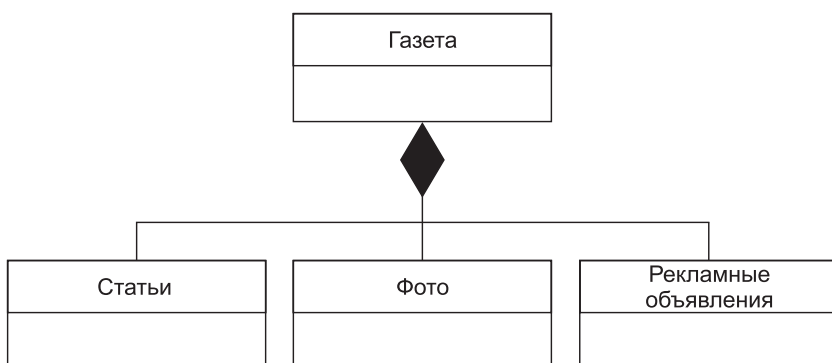


Рис. 3.14. Графическое изображение отношения композиции

- *Отношение обобщения* (generalization relationship) показывает, что компонент (пользователь или прецедент) является частным случаем другого компонента. Графически обозначается непрерывной стрелкой от частного к общему (рис. 3.15). Отношениями обобщения показывается наследование классов.

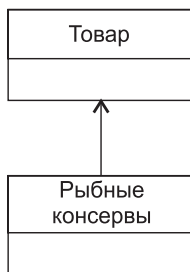


Рис. 3.15. Графическое изображение отношения обобщения

В приведенном примере класс Рыбные консервы унаследует свойства и методы более общего класса Товар.

ПРИМЕЧАНИЕ

Язык UML является средством документирования и иллюстрации удачных идей и решений в области проектирования информационных систем. Так, например, диаграмма, представленная на рис. 3.16, выразительно показывает идею множественного наследования, реализованную в некоторых языках программирования.

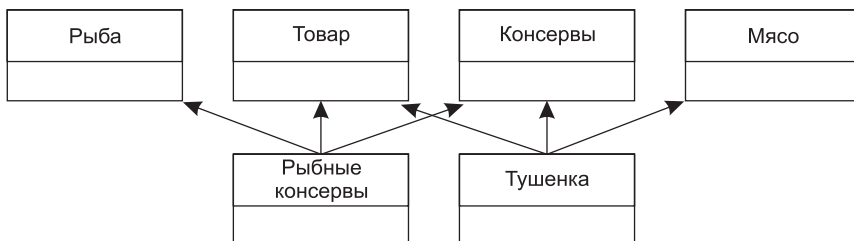


Рис. 3.16. Множественное наследование

Классы Рыбные консервы и Тушенка унаследуют свойства и методы более общих классов Товар и Консервы, в то же время они унаследуют соответственно свойства и методы классов Рыба и Мясо (последние принято называть «примесными классами»). Такое решение может значительно упростить процесс разработки информационной системы, сделать ее более устойчивой к вносимым изменениям за счет сокращения избыточности и локализации общих структур.

ПРИМЕЧАНИЕ

Множественное наследование классов реализовано далеко не во всех языках программирования. В языке C++ это имеет место, в Object Pascal — нет. Это обусловлено возможностью возникновения ситуации, когда два или несколько классов, имея единого предка, по-разному реализуют одноименный метод, и при множественном наследовании возникнет проблема выбора той или иной реализации метода (рис. 3.17), которую придется разрешать вручную (как это делается в C++).

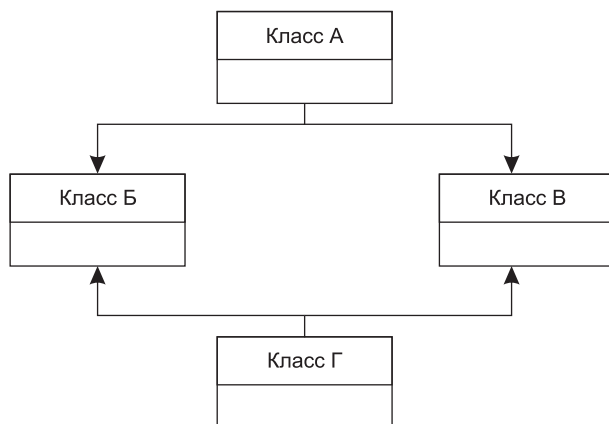


Рис. 3.17. Проблема множественного наследования классов

Отношение обобщения может содержать идентификатор, поясняющий его:

- {complete} — на диаграмме показаны все классы-потомки;
- {incomplete} — на диаграмме указаны не все классы-потомки;
- {disjoint} — не допускает множественного наследования;
- {overlapping} — допускает множественное наследование.

Кроме классов на диаграммах данного вида могут присутствовать интерфейсы (рис. 3.18), объекты (экземпляры классов) (рис. 3.19), параметризованные классы (метаклассы, шаблоны) (рис. 3.20).



Рис. 3.18. Графическое изображение интерфейса

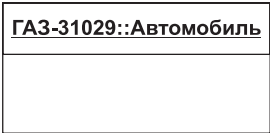


Рис. 3.19. Графическое изображение объекта на диаграмме классов

Интерфейс на диаграмме классов показывается прямоугольником с двумя секциями. В первой записывается имя интерфейса, ключевое слово «interface» и служебная информация. Вторая секция предназначена для записи методов интерфейса.

Под объектом в языке UML понимается отдельный экземпляр или пример класса, структура и поведение которого полностью определяются порождающим этот объект классом. Объекты показываются прямоугольниками, как и классы, при этом имя объекта подчеркивается и содержит указание на класс объекта.

Параметризованный класс (метакласс, шаблон) представляет собой семейство классов, каждый член которого может отличаться от других каким-либо параметром, указываемым в пунктирном прямоугольнике в правом верхнем углу значка класса.

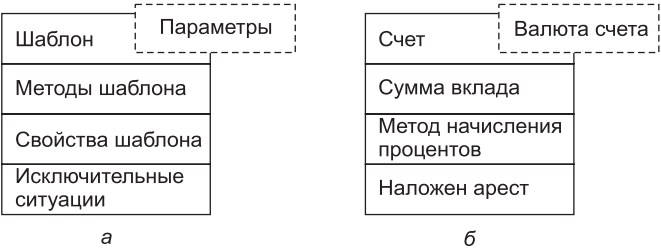


Рис. 3.20. Графическое изображение метакласса

В примере на рис. 3.20, б метакласс может служить основой для создания классов Счет в рублях, Счет в долларах США, Счет в евро, Мультивалютный счет.

Диаграмма состояний

В процессе функционирования информационной системы объекты системы, свойства объектов системы будут принимать различные значения (объекты будут принимать различные состояния). Состояния объектов, переходы объектов из одного состояния в другое, сообщения, которыми обмениваются объекты, составляют динамическую составляющую информационной системы.

Для моделирования динамики информационной системы служат диаграммы состояний, деятельности, последовательности и кооперации, каждая из которых показывает будущую систему в своем ракурсе.

Диаграмма состояний описывает процесс изменения одного класса, а точнее — одного экземпляра класса. На диаграмме показываются состояния и переходы между состояниями объекта под воздействием внешних факторов.

Состояния (state) на диаграмме состояний показываются прямоугольниками с закругленными вершинами, в котором может быть одна (обязательная) секция, где записывается имя состояния (рекомендуется использовать для обозначения глаголы и глагольные формы: причастия, деепричастия), а также несколько других секций, в которых указываются действия объекта в этом состоянии.

Начальное состояние показывается черным кружком, конечное — черным кружком в белом кольце (рис. 3.21). На диаграмме объект из начального состояния переходит в некоторое состояние, в котором выполняется «действие 1», а в последующем переходит в конечное состояние.



Рис. 3.21. Простейшая диаграмма состояний

Начальное и конечное состояния могут отсутствовать. Примером отсутствия конечного состояния может служить, когда система запускается один раз, а дальше будет функционировать в непрерывном режиме. Примером отсутствия начального состояния может служить описание уже функционирующей системы, о которой неизвестно, когда она возникла. Примером отсутствия как конечного состояния, так и начального могут служить циклические изменения какого-либо объекта.

Внутренние действия состояния описываются в следующем формате:

метка / действие

Метка представляет собой одно из следующих ключевых слов:

- ☐ entry — действие выполняется в момент перехода объекта в данное состояние;
- ☐ exit — действие выполняется в момент выхода объекта из данного состояния;

- ❑ **do** — действие выполняется во время нахождения объекта в данном состоянии;
- ❑ **include** — означает обращение к подсостоянию (substate) данного состояния объекта.

Составное состояние (composite state), или суперсостояние, — такое состояние объекта, которое включает в себя несколько подсостояний. Использование подсостояний удобно, например, когда на диаграмме требуется показать состояния объекта в зависимости от одного из его свойств. При этом другие свойства объекта также могут изменяться, и если такие состояния и переходы между ними показывать в качестве самостоятельных состояний, то диаграмма будет перегружена обозначениями, затрудняющими восприятие смысла диаграммы — демонстрации переходов между состояниями в зависимости от конкретного свойства. Графическое изображение составного состояния показано на рис. 3.22, *а* и *б*.



Рис. 3.22. Графическое изображение составного состояния

Частными случаями составного состояния являются последовательные состояния (состояния А, Б, В на рис. 3.23) и параллельные состояния (состояние Г по отношению к последовательности состояний А, Б, В рис. 3.23).

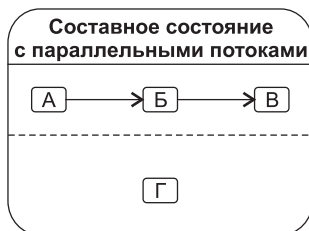


Рис. 3.23. Графическое изображение составного состояния

Историческое состояние (history state) — это составное состояние, последующие переходы в которое означают переход к подсостоянию, в котором объект находился в последний раз. Например, система ведет журнал сбоев (рис. 3.24). При втором и последующем обращении к этому состоянию нет необходимости еще раз создавать журнал сбоев, переход приведет сразу к подсостоянию Запись в журнал. Обозначается символом **H** в кружке.



Рис. 3.24. Графическое изображение исторического состояния

Переходы между состояниями в языке UML полагаются мгновенными, то есть на переход из одного состояния объекта в другое время не затрачивается.

Переходы между состояниями показываются на диаграммах состояний стрелками, над которыми могут указываться событие (event), вызвавшее этот переход, условие допустимости этого перехода (сторожевое условие), действия, сопровождающие этот переход, в формате:

событие(список параметров события) [сторожевое условие] выполняемые действия

События на диаграммах состояний выполняют роль триггеров — переход не произойдет, пока не будет выполнено соответствующее событие. Если рядом со стрелкой, обозначающей переход, не указано событие, его специфицирующее, то такой переход называется «нетриггерным», то есть переход будет выполняться сразу после выполнения действий в этом состоянии.

Сторожевое условие (guard condition) — некоторое логическое выражение, являющееся дополнительным условием срабатывания перехода. Переход не произойдет, если сторожевое условие принимает значение false, даже если соответствующее событие наступило. Сторожевые условия удобно использовать, когда из одного состояния при наступлении одного и того же события возможен переход в несколько других состояний в зависимости от какого-либо условия. Например, клиент выбрал в интернет-магазине товары (состояние — выбор) и по окончании послал команду оплатить покупку (событие). Товары будут доставлены клиенту, если на его счете есть достаточная сумма (сработает переход в состояние, в котором будет выполняться действие доставки), если же необходимая сумма отсутствует, возможны переходы в другие состояния.

Выполняемые действия — это действия, сопровождающие переход.

Переходы могут быть:

- простые:
 - между различными состояниями объекта;
 - в себя (в то же состояние) — на диаграммах показывается стрелкой, начинающейся и заканчивающейся у состояния (рис. 3.25). При таком переходе каждый раз будут выполняться соответствующие входные и выходные действия;
- сложные:
 - параллельные переходы — переходы с несколькими исходными или конечными состояниями объекта. Обозначаются жирной вертикальной или горизонтальной чертой (линиями синхронизации) (рис. 3.26);



Рис. 3.25. Графическое изображение перехода в себя

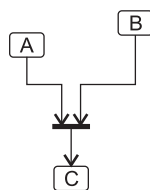


Рис. 3.26. Графическое изображение параллельного перехода

- переходы между подсостояниями разных состояний. Пример такого перехода показан на рис. 3.27, переход из A2 в B1.

В общем случае переходы на диаграммах состояний принимаются асинхронными, то есть не зависящими от других переходов. Однако при проектировании может возникнуть необходимость показать синхронность переходов между состояниями объекта. Это достигается введением синхронизирующих состояний, обозначаемых символом * в окружности (рис. 3.28). Объект не перейдет в подсостояние B, пока не произойдет переход из C в D, который «синхронизирует» переход A — B.

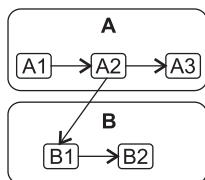


Рис. 3.27. Пример перехода между подсостояниями разных состояний

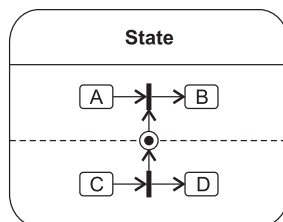


Рис. 3.28. Пример использования синхронизирующего состояния

ПРИМЕЧАНИЕ

Нотация UML представляет широкие возможности для иллюстрации процесса разработки информационных систем, что видно на примере диаграмм состояний: существует возможность показать составные состояния, сложные переходы и т. д. В то же время чрезмерное использование сложных конструкций в конечном итоге приводит только к усложнению понимания диаграмм — цели, прямо противоположной обозначенной нами. Такой же эффект дает перегрузка диаграмм объектами и связями между ними. Отсюда следует общая рекомендация к построению диаграмм — без необходимости не усложнять диаграммы, не пытаться на одной диаграмме показать все аспекты функционирования системы сразу.

Диаграмма активности

Диаграммы активности позволяют показать движения потоков данных в проектируемой системе.

Диаграммы активности напоминают хорошо знакомые многим алгоритмы, только несколько модифицированные. Пример диаграммы активности показан

на рис. 3.29. Диаграмма демонстрирует процесс рассмотрения заявки на получение кредита в некоторой организации.

Поясним применяемые обозначения. Начальное и конечное состояния изображаются аналогично обозначениям на диаграмме состояний. Прямоугольником с округлыми боковыми сторонами изображается действие над данными (состояние действия). Внутри такого прямоугольника указывается производимое действие, это может быть текст, математическое выражение и т. д.

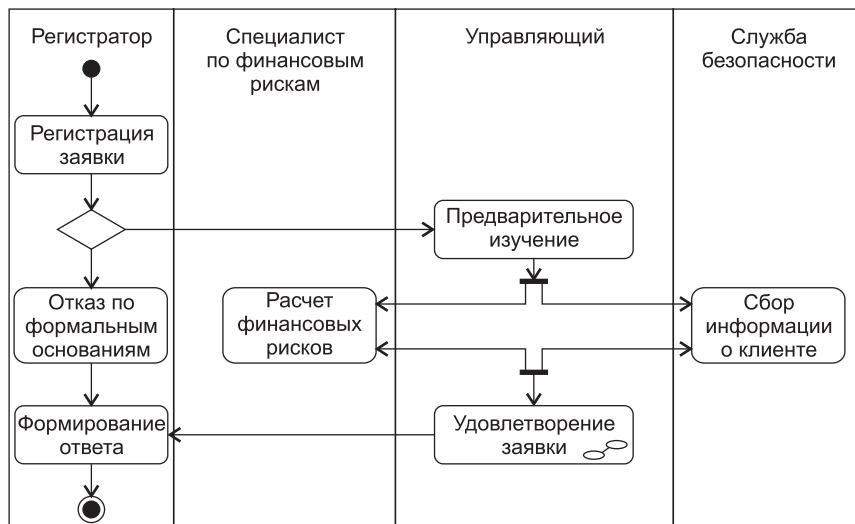


Рис. 3.29. Пример диаграммы активности

Действие может содержать несколько поддействий, показывать которые на данной определенной диаграмме нецелесообразно. В этом случае используют обозначение вложенности действий. Так, в приведенном примере действие Удовлетворение заявки может включать в себя несколько поддействий: оценка достоверности данных о клиенте, правильности расчетов, непосредственно принятие решения.

Движение данных (переходы) показывается сплошными стрелками. Возле стрелок может быть указано сторожевое условие. Если движение данных предусматривает ветвление, то указание условия обязательно (в примере на рис. 3.29 — не показано). Символ ветвления графически изображается ромбом, из которого выходят две или более стрелки. Разделение потока данных и слияние параллельных потоков данных показывается сплошной жирной чертой, к которой подходят стрелки движения потоков данных.

Диаграммы активности позволяют показать разделение ответственности различных субъектов за выполнение операций путем введения *дорожек* (swimlanes). В приведенном примере таких дорожек четыре: Регистратор, Специалист по финансовым рискам, Управляющий, Служба безопасности.

Передаваемые данные могут быть указаны на диаграммах активности в явном виде. Например, передача запроса между отделами организации показана

на рис. 3.30 в квадратных скобках внутри обозначения объекта Заказ. В качестве передаваемых данных может быть указан пользователь. Например, при построении карты веб-сайта.

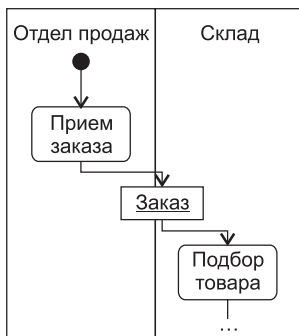


Рис. 3.30. Движение заказа между отделами

Стрела времени на диаграммах активности в явном виде не показывается.

Диаграмма последовательности

Идеология объектно-ориентированного программирования заключается в описании поставленной задачи образами некоторых самостоятельных сущностей (объектов), которые в процессе функционирования системы обмениваются сообщениями. Диаграммы последовательности служат инструментом отображения такого обмена.

Основными компонентами диаграмм активности являются: пользователь, объект (графически изображаемый как и на других диаграммах UML — прямоугольником с подчеркнутым именем), линия жизни объекта (object lifeline) — вертикальная пунктирная линия, сообщение (message) (изображается горизонтальной стрелкой), фокус управления (focus of control) — прямоугольник на линии жизни объекта.



Рис. 3.31. Примеры диаграмм последовательности

Примеры диаграмм последовательности приведены на рис. 3.31. В первом случае пользователь создает объект, который через некоторое время уничтожается (символ уничтожения объекта — жирный крест). Второй пример наглядно демонстрирует идею и параметры замера температурно-влажностного режима в помещении хранилища музея.

Фокус управления показывает, какой именно элемент на определенный момент находится в активном состоянии (действует). Фокус управления может быть как у одного, так и одновременно у нескольких элементов системы. Он может передаваться от одного объекта другому. На рис. 3.32 приведен пример передачи фокуса управления от объекта А объектам Б или В в зависимости от условия $x = 0$ или $x > 0$, записываемого возле символа ветвления.

Сообщения на диаграммах последовательности могут быть помечены идентификаторами, поясняющими их смысловую нагрузку (стереотипы):

- call — объект вызывает другой объект;
- return — возврат значения вызвавшему объекту;
- create — создание объекта (пример на рис. 3.31 а);
- destroy — объект посылает сообщение, результатом которого будет уничтожение другого объекта;
- send — посылка асинхронного сигнала.

Рекурсия (самовывоз) объекта на диаграммах последовательности может быть показана как сообщение вызова объекта, обращенное самому себе или как специальный символ на фокусе управления (рис. 3.33).

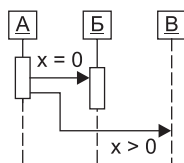


Рис. 3.32. Передача фокуса управления

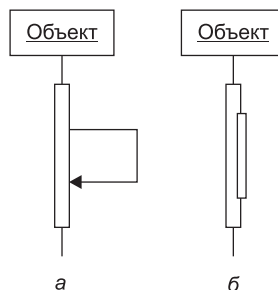


Рис. 3.33. Варианты изображения рекурсии

Диаграмма сотрудничества

Диаграмма сотрудничества, как и диаграмма последовательности, является разновидностью диаграмм взаимодействия. Современные программные средства, используемые для построения диаграмм, содержат средства автоматического преобразования диаграмм данных видов друг в друга.

Если диаграмма последовательности ориентирована на отображение временных аспектов взаимодействия, то диаграмма сотрудничества (диаграмма кооперации) показывает структурные особенности взаимодействия между объектами и является развитием идеи построения диаграмм сущность—связь.

Диаграммы сотрудничества бывают двух видов.

- ❑ *Диаграммы сотрудничества уровня спецификаций* оперируют классами, пользователями, кооперациями и ролями, которые играют пользователи и классы. Пример диаграммы сотрудничества уровня спецификаций приведен на рис. 3.34. Окружность — кооперация, пунктирная линия — роль пользователя в кооперации, стрелка — отношение обобщения, общее для всех диаграмм языка UML.

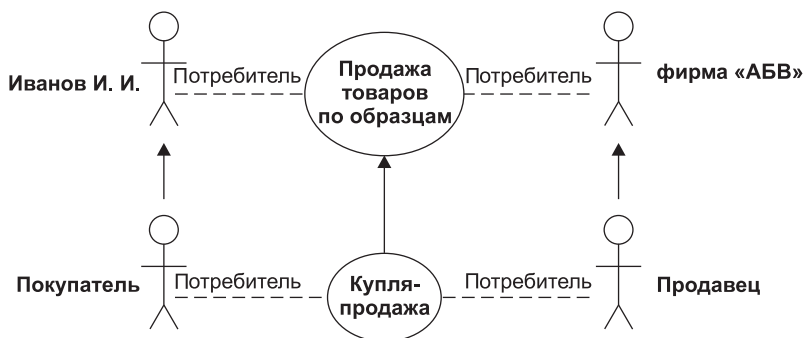


Рис. 3.34. Пример диаграммы сотрудничества уровня спецификаций

Кооперация определяет взаимодействие классов. Участвуя в кооперации, классы совместно производят некоторый кооперативный результат.

- ❑ *Диаграммы сотрудничества уровня примеров* — оперируют экземплярами классов (объектами) связями между ними и сообщениями, которыми обмениваются объекты.

Объекты на диаграммах сотрудничества обозначаются так же, как и на других диаграммах UML — прямоугольниками. Однако на диаграммах данного вида имя объекта может дополняться его ролью в сотрудничестве (рис. 3.35), любая из трех частей имени объекта может отсутствовать.



Рис. 3.35. Графическое изображение объектов на диаграммах сотрудничества

Объекты, которые могут управлять другими объектами, называются активными (active object) и помечаются словом {active}.

Для обозначения группы объектов, которым адресован один и тот же сигнал, вводится понятие мультиобъект, изображаемый графически двумя прямоугольниками, налагаемыми друг на друга (рис. 3.36).

В отличие от мультиобъекта, составной объект представляет собой объект, составными частями которого являются другие объекты (рис. 3.37).

Между объектами диаграммы сотрудничества существуют связи (link), по которым объекты посылают друг другу сообщения (message). Связи не имеют

названий (в терминах UML — «анонимные»), но могут быть специфицированы ключевыми словами (стереотипами):

- `association` — связь представляет собой некоторую зависимость объектов;
- `parameter` — объект полагается параметром метода;
- `local` — локальная переменная метода. Область видимости ограничена соседним объектом;
- `global` — глобальная переменная. Область видимости ограничена диаграммой сотрудничества;
- `self` — связь объекта с самим собой.

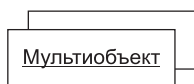


Рис. 3.36. Графическое изображение мультиобъекта

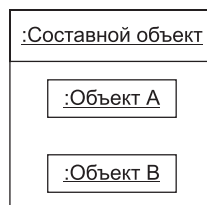


Рис. 3.37. Графическое изображение составного объекта

Приведенные стереотипы требуют пояснения. Связь между объектами в информационной системе на уровне программирования на определенном языке осуществляется посредством передачи параметров (переменных) от одного объекта другому. Например, объект *Отдел продаж* передает объекту *Склад* некоторый принятый в организации документ (переменная), в котором сообщает о необходимости выделения той или иной номенклатуры продукции. Значение передаваемых параметров является содержанием передаваемого посредством связи сообщения.

Сообщения на диаграммах сотрудничества изображаются стрелками вдоль связей. Порядок передачи сообщений может быть определен явным указанием номера сообщения возле стрелки. Вид сообщения несет дополнительную нагрузку в виде определения ролей взаимодействующих объектов. В зависимости от этого сообщения графически изображаются:

- сплошной линией с треугольной стрелкой (рис. 3.38, *а*) — сообщение означает вызов процедуры (метода объекта) или вызов другого потока управления;
- сплошной линией с обычной стрелкой (рис. 3.38, *б*) — простой поток управления (простая передача данных);
- сплошной линией с полустрелкой (рис. 3.38, *в*) — такие сообщения не имеют заранее обусловленного времени передачи, как правило асинхронны;
- пунктирной линией с обычной стрелкой (рис. 3.38 *г*) — возврат значения из процедуры.

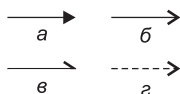


Рис. 3.38. Графическое изображение сообщений на диаграммах сотрудничества

Сообщение записывается в определенном формате. Например, запись 1. 2 / [пароль истинный] 3.2 Форма_1А := найти_сведения (Фамилия, Имя, Отчество) означает, что данное сообщение будет передано только после сообщений с номерами 1 и 2 (предшествующие сообщения), при условии истинности введенного пароля (сторожевое условие), в потоке последовательных сообщений будет занимать место между сообщениями 3.1 и 3.3 между данными объектами, при этом возможна параллельная передача сообщения с другими сообщениями, имеющими номер 3.2 (номер сообщения), сообщение вызывает метод нахождения сведений о человеке (имя сообщения) по фамилии, имени и отчеству (список аргументов) и предполагает предоставление карточки по форме 1А на запрашиваемое лицо (возвращаемое значение).

В заключение приведем пример диаграммы сотрудничества — диаграмму, иллюстрирующую отношения по расчетам чеками (§ 5, гл. 46 Гражданского кодекса Российской Федерации) (рис. 3.39).

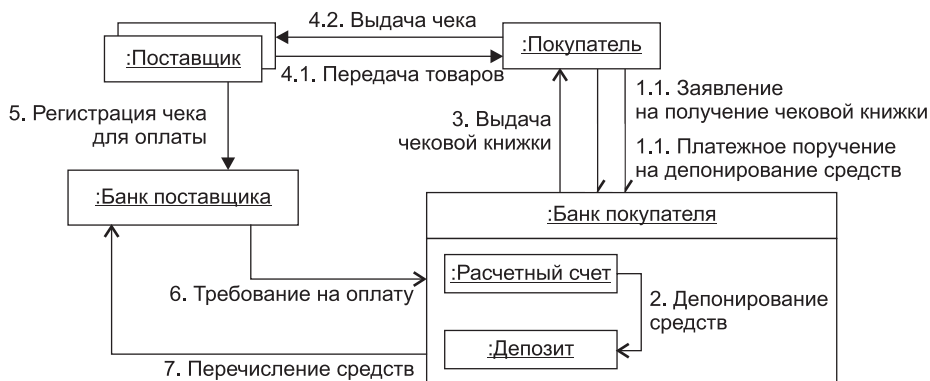


Рис. 3.39. Расчеты чеками

Приведенный пример существенно упрощен. В частности, не показаны действия в случае неоплаты чека, опущены параметры сообщений.

Диаграмма компонентов

Информационные системы на уровне программного кода могут состоять из множества приложений, файлов справок, исходных текстов, веб-документов, динамических библиотек. Как именно будут распределены классы, их экземпляры по файлам, каковы взаимосвязи между файлами, позволяют отобразить диаграммы компонентов.

Основным обозначением, используемым на диаграммах компонентов, является компонент (component). Графически изображается прямоугольником со встроенными слева прямоугольными секциями. Внутри прямоугольника указывается имя компонента, в качестве которого принято использовать имена исполняемых файлов, баз данных и т. д., а также служебная информация: версия, язык реализации, разработчик (рис. 3.40).

ПРИМЕЧАНИЕ

Диаграммы компонентов играют существенную роль при оптимизации быстродействия системы — выявляются наиболее часто используемые компоненты, выбирается их оптимальное распределение по модулям. На завершающих стадиях разработки информационной системы может возникнуть ситуация, когда незначительные изменения реализации одного из компонентов потребуют перекомпиляции всех компонентов, созданных на его основе. В этом случае пренебрежительное отношение к диаграммам данного вида может существенно сказаться на сроках сдачи проекта. Особенно это касается крупных приложений с числом модулей от тысячи и выше.

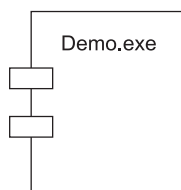


Рис. 3.40. Графическое изображение компонента

Иногда перед именем компонента указывается его спецификация:

- ☐ library — библиотека;
- ☐ table — база данных, отдельная таблица;
- ☐ file — исходный текст программ;
- ☐ document — документ;
- ☐ executable — исполняемый файл.

Для ряда компонентов приняты специальные обозначения: динамических библиотек (рис. 3.41, а), файлов справки (рис. 3.41, б), исходных текстов программ (рис. 3.41, в), веб-документов (рис. 3.41, г). Эти обозначения, а также названия исполняемых модулей иногда называют *артефактами*.

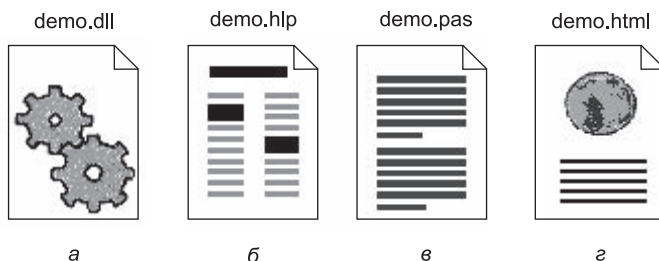


Рис. 3.41. Графическое изображение артефактов

Между компонентами и другими обозначениями диаграммы компонентов существуют отношения зависимости (рис. 3.42), показывающие, что один компонент зависит от другого и при его изменении тоже должен измениться, а также отношения реализации, изображаемые сплошной линией (рис. 3.43, а) либо

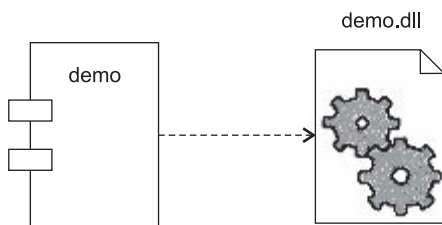


Рис. 3.42. Графическое изображение отношения зависимости

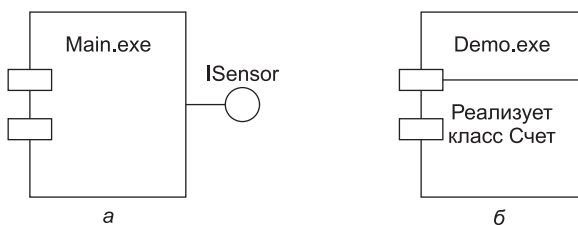


Рис. 3.43. Графическое изображение отношений реализации

указанием на соответствующие элементы внутри обозначения компонента (рис. 3.43, б).

Диаграмма развертывания

Диаграммы развертывания служат для иллюстрации физического размещения информационной системы на серверах, компьютерах пользователей, искусственных спутниках Земли, поездах, морских судах, внутри ЭВМ, отображения физических каналов передачи информации между компонентами информационной системы.

Основным обозначением, используемым на диаграммах данного вида, является узел (node). Графически изображается аксонометрической проекцией куба (рис. 3.44). Внутри обозначения узла указывается его имя (например, Сервер) и развертываемые на нем компоненты, обозначения которых аналогичны обозначениям диаграммы компонентов.

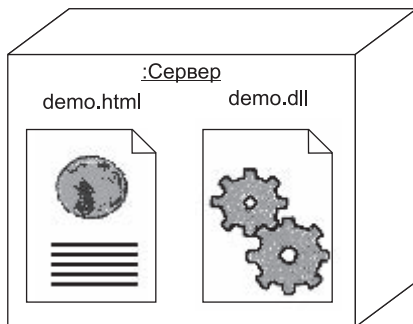


Рис. 3.44. Графическое изображение узла

Между узлами сплошными линиями показывают соединения, которые могут содержать пояснения относительно характера реализации связи между компонентами (рис. 3.45)

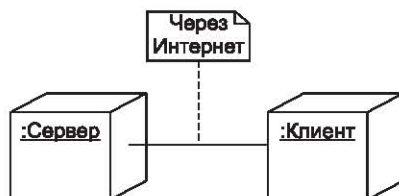


Рис. 3.45. Графическое изображение соединения

Глава 4

Реляционные базы данных

По мере развития вычислительной техники изменялись и основные направления ее использования. Первоначально средства вычислительной техники подразумевалось использовать для выполнения различного рода математических вычислений, которые невозможно провести «вручную» за разумное время. Развитие этого направления привело к развитию разделов математики, связанных с численными методами вычислений, и к появлению алгоритмических языков, удобных для реализации алгоритмов численных методов и ориентированных на выполнение математических расчетов (одним из наиболее популярных языков программирования такого типа является Fortran, до сих пор широко применяющийся для научных расчетов).

Затем, по мере увеличения возможностей и уменьшения стоимости вычислительных средств, получило развитие второе направление, связанное с использованием компьютеров в автоматизированных информационных системах. Здесь вычислительные возможности компьютеров отходят на второй план — основные функции вычислительных средств в информационных системах состоят в поддержке надежного хранения информации, выполнении специфических для данного приложения преобразований информации и/или вычислений, предоставлении пользователям удобного и легко осваиваемого интерфейса.

ПРИМЕЧАНИЕ

Несмотря на то что сложность вычислений, выполняемых в информационных системах, несоизмеримо ниже, чем при проведении научных расчетов, требования к вычислительной мощности компьютеров в таких системах увеличиваются. Это связано с тем, что объемы обрабатываемой информации, как правило, достаточно велики, а сама информация имеет сложную структуру. Этим же объясняется существенное увеличение требований к объему как оперативной памяти, так и устройств постоянного хранения информации.

Со временем именно второе направление, связанное с хранением и обработкой данных, стало доминирующим, особенно после появления персональных компьютеров. Использование персональных компьютеров для выполнения сложных научных расчетов сейчас является скорее исключением. Интересно также

отметить, что современные персональные компьютеры, оборудованные процессорами с громадными тактовыми частотами (на сегодняшний день рядовой дешевый процессор работает на частоте около 2 ГГц), при решении сложных научных задач могут даже уступать по вычислительным возможностям «большим» компьютерам 15–20-летней давности.

Базы данных: основные сведения

Развитие компьютерных технологий, связанных с хранением и обработкой данных, привело к появлению в конце 1960-х — начале 1970-х годов специализированного программного обеспечения, получившего название *систем управления базами данных* (СУБД) (*DataBase Management Systems — DBMS*). СУБД позволяют структурировать, систематизировать и организовывать данные для их компьютерного хранения и обработки. Именно системы управления базами данных являются основой практически любой информационной системы.

СУБД можно определить как некую систему управления данными, обладающую следующими свойствами:

- ☐ поддержание логически согласованного набора файлов;
- ☐ обеспечение языка манипулирования данными;
- ☐ восстановление информации после разного рода сбоев;
- ☐ обеспечение параллельной работы нескольких пользователей.

Основные функции СУБД

К основным функциям, выполняемым системами управления базами данных, обычно относят следующие:

- ☐ непосредственное управление данными во внешней памяти;
- ☐ управление буферами оперативной памяти;
- ☐ управление транзакциями;
- ☐ журнализацию (протоколирование);
- ☐ поддержку языков баз данных.

Рассмотрим каждую из указанных функций более подробно.

Непосредственное управление данными во внешней памяти

Функция непосредственного управления данными во внешней памяти включает обеспечение необходимых структур внешней памяти (как правило магнитных дисков), как для хранения данных, непосредственно входящих в базу данных, так и для служебных целей, например для ускорения доступа к данным в некоторых случаях (обычно для этого используются *индексы*). Причем пользователи базы данных в общем случае не нужно знать, использует ли СУБД файловую систему и, если использует, как организованы файлы. Обычно СУБД поддерживает собственную систему именования объектов базы данных. В зависимости от способа реализации СУБД может либо использовать

возможности существующих файловых систем, либо работать с устройствами внешней памяти на низком уровне.

Управление буферами оперативной памяти

Объем информации, хранящейся в базе данных, с которой работает СУБД, обычно достаточно велик и практически всегда превышает доступный объем оперативной памяти. При этом время доступа к данным, хранящимся в оперативной памяти, существенно меньше, чем к данным, хранящимся на устройствах внешней памяти. Очевидно, что если при обращении к любому элементу данных будет производиться обмен с внешней памятью, то вся система будет работать со скоростью устройства внешней памяти.

Увеличения скорости обмена данными можно достичь, используя буферизацию данных в оперативной памяти. При этом, даже если операционная система производит общесистемную буферизацию (как в случае ОС Unix), этого недостаточно для целей СУБД, которая располагает гораздо большей информацией о полезности буферизации той или иной части базы данных. Поэтому в СУБД обычно поддерживается собственный набор буферов оперативной памяти с собственным механизмом замены буферов.

ПРИМЕЧАНИЕ

Следует отметить, что существует направление развития СУБД, ориентированное на постоянное присутствие в оперативной памяти всей информации из базы данных. Это направление основывается на предположении, что в будущем объем оперативной памяти компьютеров будет настолько велик, что буферизация станет не нужна.

Управление транзакциями

Транзакцией называется последовательность операций над базой данных, рассматриваемых СУБД как единое целое. Если все операции успешно выполнены, то транзакция также считается успешно выполненной и СУБД *фиксирует* (COMMIT) все изменения данных, произведенные этой транзакцией (то есть заносит изменения во внешнюю память). Если же хотя бы одна операция транзакции заканчивается неудачей, то транзакция считается невыполненной и производится *откат* (ROLLBACK) — отмена всех изменений данных, произведенных в ходе выполнения транзакции, и возврат базы данных к состоянию до начала выполнения транзакции.

Управление транзакциями необходимо для поддержания логической целостности базы данных.

Поддержка механизма транзакций является обязательным условием для однопользовательских, а тем более для многопользовательских СУБД. То свойство, что каждая транзакция начинается при целостном состоянии базы данных и оставляет это состояние целостным после своего завершения, делает очень удобным использование понятия транзакции как единицы активности пользователя по отношению к базе данных. При соответствующем управлении параллельно выполняющимися транзакциями со стороны СУБД каждый из пользователей может в принципе ощущать себя единственным пользователем СУБД.

С управлением транзакциями в многопользовательской СУБД связаны важные понятия *сериализации транзакций* и *серияльного плана выполнения смеси транзакций*. Под сериализацией параллельно выполняющихся транзакций понимается такое планирование их работы, при котором суммарный результат смеси транзакций эквивалентен результату их некоторого последовательного выполнения. Серияльный план выполнения смеси транзакций — это такой план, который приводит к сериализации транзакций. Понятно, что если удастся добиться действительно серияльного выполнения смеси транзакций, то для каждого пользователя, по инициативе которого образована транзакция, присутствие других транзакций будет незаметно (если не считать некоторого замедления работы по сравнению с однопользовательским режимом).

Существует несколько базовых алгоритмов сериализации транзакций. В централизованных СУБД наиболее распространены алгоритмы, основанные на синхронизационных захватах объектов базы данных. При использовании любого алгоритма сериализации возможны конфликты между несколькими транзакциями по доступу к объектам базы данных. В этом случае для поддержания сериализации необходимо выполнить откат одной или нескольких транзакций. Это один из случаев, когда пользователь многопользовательской СУБД может реально (и достаточно неприятно) ощутить присутствие в системе транзакций других пользователей.

Журнализация

Одним из основных требований к СУБД является надежность хранения данных во внешней памяти. Под надежностью хранения понимается то, что СУБД должна быть в состоянии восстановить последнее согласованное состояние БД после любого аппаратного или программного сбоя.

Аппаратные сбои обычно подразделяются на два вида:

- ❑ *мягкие сбои* связаны с внезапной остановкой работы компьютера. Обычно являются следствием внезапного выключения питания или «зависания» операционной системы;
- ❑ *жесткие сбои* характеризуются потерей информации на носителях внешней памяти.

Программные сбои обычно возникают вследствие ошибок в программах. Причем эти ошибки могут быть как в самой СУБД, что может привести к аварийному завершению ее работы, так и в пользовательской программе. Первый случай можно рассматривать как разновидность мягкого аппаратного сбоя. Во втором случае незавершенной остается только одна транзакция.

В любом случае для восстановления информации в базе данных необходимо иметь некоторую дополнительную информацию. Таким образом, для поддержания надежности хранения данных требуется избыточность данных. Причем та часть информации, которая используется для восстановления, должна храниться особо надежно. Наиболее распространенным методом поддержания такой избыточной информации является ведение журнала изменений базы данных.

Журнал представляет собой особую часть базы данных, недоступную пользователям СУБД и поддерживаемую с особой тщательностью (иногда используются две копии журнала, располагаемые на разных физических дисках), в которую поступают записи обо всех изменениях основной части базы данных.

В разных СУБД изменения базы данных журналируются на разных уровнях: иногда запись в журнале соответствует некоторой логической операции изменения базы данных, иногда — минимальной внутренней операции модификации страницы внешней памяти. Могут также использоваться одновременно оба подхода.

Во всех случаях придерживаются стратегии «упреждающей» записи в журнал (так называемого протокола Write Ahead Log — WAL). Несколько утрированно можно сказать, что эта стратегия заключается в том, что запись об изменении любого объекта базы данных должна быть занесена в журнал до того, как будет выполнено и зафиксировано изменение этого объекта. Если в СУБД корректно соблюдается протокол WAL, то с помощью журнала можно решить все проблемы восстановления базы данных после любого сбоя.

Самая простая ситуация восстановления — индивидуальный откат транзакции. Строго говоря, для этого не требуется общесистемный журнал изменений базы данных. Достаточно для каждой транзакции поддерживать локальный журнал операций модификации базы данных, выполненных в этой транзакции, и производить откат транзакции путем выполнения обратных операций, следуя от конца локального журнала. Некоторые СУБД так и делают, но большинство локальные журналы не поддерживают, а индивидуальный откат транзакции выполняют по общесистемному журналу, для чего все записи, относящиеся к одной транзакции, связывают обратным списком (от конца к началу).

При мягком сбое во внешней памяти основной части базы данных могут находиться объекты, модифицированные транзакциями, не закончившимися к моменту сбоя, и могут отсутствовать объекты, модифицированные транзакциями, которые к моменту сбоя успешно завершились (по причине использования буферов оперативной памяти, содержимое которых при мягком сбое пропадает). При соблюдении протокола WAL во внешней памяти журнала должны гарантированно находиться записи, относящиеся к операциям модификации обоих видов объектов. Целью процесса восстановления после мягкого сбоя является приведение внешней памяти основной части базы данных в такое состояние, которое возникло бы при фиксации в ней изменений всех завершившихся транзакций и которое не содержало бы никаких следов незаконченных транзакций. Для того чтобы этого добиться, сначала производят откат незавершенных транзакций, а потом повторно воспроизводят те операции завершенных транзакций, результаты которых не отображены во внешней памяти.

Для восстановления базы данных после жесткого сбоя используют журнал и архивную копию базы данных. Архивная копия — это полная копия базы данных к моменту начала заполнения журнала (хотя имеется много вариантов трактовки смысла архивной копии). Для нормального восстановления базы данных после жесткого сбоя, естественно, необходимо, чтобы журнал не пропал. Тогда восстановление базы данных состоит в том, что, исходя из архивной копии, по журналу воспроизводится работа всех транзакций, которые закончи-

лись к моменту сбоя. В принципе, можно даже воспроизвести работу незавершенных транзакций и продолжить их работу после завершения восстановления. Однако в реальных системах это обычно не делается, поскольку процесс восстановления после жесткого сбоя является достаточно длительным.

Поддержка языков баз данных

Для работы с информацией, хранящейся в базе данных, используются специальные языки, носящее общее название *языков баз данных*. Чаще всего выделяется два языка:

- *язык определения схем данных* (Schema Definition Language, SDL) — служит главным образом для определения логической структуры базы данных;
- *язык манипулирования данными* (Data Manipulation Language, DML) — содержит набор операторов манипулирования данными, то есть операторов, позволяющих заносить данные в базу, а также удалять, модифицировать или выбирать существующие данные.

Несколько разных специализированных языков баз данных поддерживалось лишь в ранних СУБД. В современных СУБД обычно поддерживается единый интегрированный язык, содержащий все необходимые средства для работы с базой данных, начиная от ее создания, и обеспечивающий базовый пользовательский интерфейс с базами данных. Стандартным языком наиболее распространенных в настоящее время реляционных СУБД является язык SQL (Structured Query Language). Таким образом, указанные выше языки баз данных на сегодняшний день фактически являются подмножествами единого стандартного языка SQL.

Язык SQL позволяет определять схему реляционной базы данных и манипулировать данными. При этом именование объектов базы данных (для реляционной базы данных — именование таблиц и их полей) поддерживается на языковом уровне в том смысле, что компилятор языка SQL производит преобразование имен объектов в их внутренние идентификаторы на основании специально поддерживаемых служебных таблиц-каталогов.

Язык SQL содержит специальные средства определения ограничений целостности базы данных. Опять же, ограничения целостности хранятся в специальных таблицах-каталогах, и обеспечение контроля целостности базы данных производится на языковом уровне — при компиляции операторов модификации базы данных компилятор SQL на основании имеющихся в базе данных ограничений целостности генерирует соответствующий программный код.

Специальные операторы языка SQL позволяют определять так называемые *представления* базы данных, фактически являющиеся хранимыми в базе данных запросами (результатом любого запроса к реляционной базе данных является таблица) с именованными столбцами, называемыми *полями*. Для пользователя представление является такой же таблицей, как любая базовая таблица, хранимая в базе данных, но с помощью представлений можно ограничить или, наоборот, расширить видимость данных для конкретного пользователя. Поддержка представлений производится также на языковом уровне.

Наконец, авторизация доступа к объектам базы данных производится также на основе специального набора операторов SQL. Идея состоит в том, что для выполнения операторов SQL разного вида пользователь должен обладать различными полномочиями. Пользователь, создавший таблицу базы данных, обладает полным набором полномочий для работы с данной таблицей. В число этих полномочий входит полномочие на передачу всех или части полномочий другим пользователям, включая полномочие на передачу полномочий. Полномочия пользователей описываются в специальных таблицах-каталогах, контроль полномочий поддерживается на языковом уровне.

ПРИМЕЧАНИЕ

Здесь дается лишь общее представление о языке SQL. Более подробно данный язык и его функции будут рассматриваться ниже.

Эволюция систем управления базами данных

На эволюцию СУБД существенное влияние оказывает бурное развитие микроэлектронных технологий и связанное с этим развитие персональных компьютеров. Темпы развития персональных компьютеров за последние 15–20 лет существенно превышают темпы развития «больших» ЭВМ. Область применения персональных компьютеров за последние несколько лет существенно расширилась. Можно выделить следующие основные причины этой тенденции:

- ❑ цена персональных компьютеров значительно ниже, чем больших ЭВМ;
- ❑ по функциональным возможностям персональные компьютеры превосходят большие ЭВМ;
- ❑ существенно уменьшился разрыв между производительностью персональных компьютеров и больших ЭВМ. Кроме того, для многих задач работы с данными производительность компьютера не является решающим фактором;
- ❑ архитектура систем на основе персональных компьютеров обладает большей гибкостью и мобильностью, а сфера их использования значительно шире области применения больших ЭВМ.

Общая тенденция движения от отдельных mainframe-систем к открытым распределенным системам оказала огромное влияние на развитие архитектур СУБД и поставила перед их разработчиками ряд сложных проблем. Главная проблема состояла в технологической сложности перехода от централизованного управления данными на одном компьютере и СУБД, использовавшей собственные модели, форматы представления данных и языки доступа к данным, к распределенной обработке данных в неоднородной вычислительной среде, состоящей из соединенных в сеть компьютеров различных моделей и производителей.

Постепенный переход от вычислительных систем на основе больших ЭВМ и централизованного управления данными к распределенным системам на основе персональных компьютеров, а также внедрение персональных компьютеров практически во все сферы деятельности привели и к изменению подходов к организации систем управления базами данных. В истории развития и совер-

шенствования систем управления базами данных можно условно выделить три основных этапа. Кратко рассмотрим каждый из них.

СУБД первого поколения

Первый этап был связан с созданием первого поколения СУБД, опиравшихся на иерархическую и сетевую модели данных (на основе спецификаций CODASYL). В этот период времени на рынке вычислительной техники доминировали большие вычислительные машины (*mainframe*), такие как система IBM 360/370, которые в совокупности с СУБД первого поколения составили аппаратно-программную платформу больших информационных систем. СУБД первого поколения были в подавляющем большинстве закрытыми системами: отсутствовал стандарт внешних интерфейсов и не обеспечивалась переносимость прикладных программ.

Ранние СУБД, с сегодняшней точки зрения, имели массу недостатков, из которых наиболее существенными были следующие:

- ☐ сложность использования;
- ☐ необходимость знать физическую организацию базы данных;
- ☐ сильная зависимость прикладных систем от физической организации базы данных;
- ☐ перегрузка логики прикладных систем деталями организации доступа к базе данных;
- ☐ отсутствие средств автоматизации проектирования баз данных;
- ☐ очень высокая стоимость.

Среди достоинств СУБД первого поколения можно отметить:

- ☐ наличие развитых средств управления данными во внешней памяти на низком уровне;
- ☐ возможность построения эффективных прикладных систем вручную;
- ☐ возможность экономии памяти за счет совместного использования объектов (в сетевых системах).

Несмотря на все свои недостатки, СУБД первого поколения оказались весьма долговечными: разработанное на их основе программное обеспечение используется по сей день, и большие ЭВМ по-прежнему хранят огромные массивы актуальной информации. Главной причиной этого является, вероятно, экономический фактор — в свое время в аппаратное и программное обеспечение больших ЭВМ были вложены огромные средства: в результате многие продолжают их использовать, несмотря на морально устаревшую архитектуру. В то же время перенос данных и программ с больших ЭВМ на компьютеры нового поколения сам по себе представляет сложную техническую проблему и требует значительных затрат.

Реляционные СУБД

Началом второго этапа в эволюции СУБД можно считать публикации в начале 1970-х годов ряда статей Э. Кодда, в которых выдвигались, по сути, революционные идеи, существенно изменившие устоявшиеся представления о базах данных.

Будучи математиком по образованию, Кодд предложил использовать для обработки данных аппарат теории множеств (объединение, пересечение, разность, декартово произведение). Он показал, что любое представление данных сводится к совокупности двумерных таблиц особого вида, известного в математике как *отношение* (по-английски — *relation*, отсюда и название — *реляционные* базы данных).

Одна из главных идей Кодда заключалась в том, что связь между данными должна устанавливаться в соответствии с их внутренними логическими взаимоотношениями.

ПРИМЕЧАНИЕ

В СУБД первого поколения для связи записей из разных файлов использовались физические указатели или адреса на диске. Это означало, что в том случае, когда в разных файлах хранится логически связанная информация, а физическая связь между этими файлами отсутствует, то для получения выборки (извлечения информации) из такой базы данных необходимо использовать низкоуровневые средства работы с файлами. В случае же реляционной базы данных сама СУБД поддерживает извлечение информации из базы данных на основе логических связей, и при работе с базой данных нет необходимости напрямую программировать работу с файлами. Естественно, это существенно упрощает работу с базами данных.

Второй важный принцип, предложенный Коддом, заключается в том, что в реляционных системах одной командой могут обрабатываться целые файлы данных, в то время как в ранних СУБД одной командой обрабатывалась только одна запись. Реализация этого принципа существенно повысила эффективность программирования баз данных.

Реализация реляционных принципов в СУБД сделала возможным разработку простых языков запросов, доступных для изучения пользователями, не являющимися специалистами в области программирования. Таким образом, благодаря снижению требований к квалификации существенно расширился круг пользователей баз данных.

ПРИМЕЧАНИЕ

На начальном этапе развития реляционных баз данных было разработано несколько языков запросов, среди которых наиболее известны такие как QBE — Query by Example (запрос по образцу), Quel — Query Language (язык запросов) и SQL — Structured Query Language (структурированный язык запросов). Среди этих языков на сегодняшний день наибольшее распространение имеет SQL, который в 1986 году был принят в качестве стандарта ANSI языков реляционных баз данных. Последнее обновление этого стандарта было принято в 1992 году, и язык запросов, соответствующий этому стандарту, обычно обозначается как SQL-92.

Сейчас реляционные базы данных получили очень широкое распространение, и фактически их можно рассматривать как стандарт СУБД для современных информационных систем.

Объектно-ориентированные СУБД

Несмотря на большую популярность реляционных СУБД, развитие технологии управления данными на них не остановилось. Развитие реляционных баз дан-

ных и обеспечение возможностей решения более сложных задач привели к появлению объектно-ориентированных баз данных, для которых характерно использование идей объектно-ориентированного подхода, управления распределенными базами данных, активного сервера базы данных, языков программирования четвертого поколения, фрагментации и параллельной обработки запросов, технологии тиражирования данных, многопоточной архитектуры и других революционных достижений в области обработки данных.

Объектно-ориентированный подход имеет ряд преимуществ для разработчика, из которых можно отметить следующие:

- ❑ возможность разбить систему на совокупность независимых сущностей (объектов) и провести их строгую независимую спецификацию;
- ❑ простоту эволюции системы за счет использования таких элементов объектного подхода как наследование и полиморфизм;
- ❑ возможность объектного моделирования системы, позволяющую проследить поведение реальных сущностей предметной области уже на ранних стадиях разработки.

Объектная модель данных более близка сущностям реального мира. Объекты можно сохранить и использовать непосредственно, не раскладывая их по таблицам. Типы данных определяются разработчиком и не ограничены набором predetermined типов.

При занесении сложного объекта в реляционную базу обязательна процедура декомпозиции его данных для того, чтобы разместить их в таблицах. При чтении объекта из реляционной базы он собирается из отдельных элементов и только затем пригоден для использования. В объектных же СУБД данные объекта, а также методы изменения этих данных помещаются в хранилище как единое целое.

Использование объектной модели представления данных (и соответственно объектно-ориентированной СУБД) наиболее привлекательно для информационных систем корпоративного уровня, разработка которых ведется методами объектного проектирования.

ПРИМЕЧАНИЕ

Несмотря на все достоинства объектно-ориентированных СУБД, их использование далеко не всегда оправдано. Нередко декомпозиция данных объекта не вызывает никаких проблем и вполне логична. В этом случае использование реляционной модели может быть более эффективно. Кроме того, ведущие производители реляционных СУБД IBM и Oracle доработали свои продукты (DB2 и Oracle соответственно), добавив объектную надстройку над реляционным ядром системы. Таким образом, работая с этими СУБД, можно использовать ту или иную модель данных в зависимости от конкретной ситуации. Вероятно, что в обозримом будущем рынок корпоративных систем пока останется за гибридными объектно-реляционными СУБД.

Реляционная модель данных

Реляционная модель данных была предложена Э. Коддом, известным американским специалистом в области баз данных. Основные концепции этой модели

были впервые опубликованы в 1970 году в статье «A Relational Model of Data for Large Shared Data Banks» (CACM, 1970, Vol. 13, № 6). Реляционная модель позволила решить одну из важнейших задач в управлении базами данных — обеспечить независимость представления и описания данных от прикладных программ, следствием чего было бы существенное упрощение проектирования и программирования баз данных. Поэтому после опубликования работ Кодда начались активные исследования по созданию реляционной системы управления базами данных. В результате этих исследований во второй половине 1970-х годов был создан ряд коммерческих и некоммерческих реляционных СУБД.

К основным достоинствам реляционного подхода к управлению базой данных следует отнести:

- ❑ наличие небольшого набора абстракций, которые позволяют сравнительно просто моделировать большую часть распространенных предметных областей и допускают точные формальные определения, оставаясь интуитивно понятными;
- ❑ наличие простого и в то же время мощного математического аппарата, опирающегося главным образом на теорию множеств и математическую логику и обеспечивающего теоретический базис реляционного подхода к организации баз данных;
- ❑ возможность манипулирования данными без необходимости знания конкретной физической организации баз данных во внешней памяти.

Несмотря на все свои достоинства, реляционные системы далеко не сразу получили широкое признание. Хотя уже во второй половине 1970-х годов появились первые прототипы реляционных СУБД, долгое время считалось невозможным добиться эффективной реализации таких систем. Однако постепенное накопление методов и алгоритмов организации реляционных баз данных и управления ими привели к тому, что уже в середине 1980-х годов реляционные системы практически вытеснили с мирового рынка ранние СУБД.

В настоящее время реляционные СУБД остаются одними из наиболее распространенных, несмотря на некоторые присущие им недостатки. Сейчас основным предметом критики реляционных СУБД является не их недостаточная эффективность, а некоторая ограниченность таких систем при использовании в так называемых нетрадиционных областях (наиболее распространенными примерами являются системы автоматизации проектирования), в которых требуются предельно сложные структуры данных. Причем эта ограниченность реляционных СУБД является прямым следствием их простоты и проявляется лишь в отдельных предметных областях. Вторым часто отмечаемым недостатком реляционных баз данных является невозможность адекватного отражения семантики предметной области — возможности представления знаний о семантической специфике предметной области в реляционных системах очень ограничены.

На устранение именно этих недостатков в основном и направлены исследования по созданию объектно-ориентированных баз данных.

Базовые понятия реляционной модели данных

Термин «реляционный» (от англ. *relation* — отношение) указывает прежде всего на то, что такая модель хранения данных построена на взаимоотношении составляющих ее частей, которые удобно представлять в виде двумерной таблицы. Кодд показал, что набор отношений (таблиц) может быть использован для хранения данных об объектах реального мира и моделирования связей между ними. Таким образом, реляционная модель данных представляет информацию в виде совокупности взаимосвязанных таблиц, которые принято называть *отношениями*, или *реляциями*.

Основными понятиями реляционной модели данных являются:

- ☐ тип данных;
- ☐ домен;
- ☐ атрибут;
- ☐ кортеж;
- ☐ ключ.

Рассмотрим смысл этих понятий на примере отношения (таблицы) СТУДЕНТЫ, содержащего информацию о студентах некоторого вуза (табл. 4.1).

Таблица 4.1. Пример отношения СТУДЕНТЫ реляционной базы данных

№_студенческого_билета	Имя	Дата_рождения	Курс	Специальность
23980282	Алексеев Д. А.	12.03.1982	2	Биология
22991380	Яковлев Н. В.	25.12.1979	4	Физика
22657879	Михайлов В. В.	29.02.1979	5	Математика
24356783	Афанасьев А. В.	19.08.1983	1	Иностранный язык
24350283	Кузнецов В. И.	03.10.1982	1	Физика
23125681	Смирнов А. Д.	26.03.1981	3	История

Тип данных

Понятие *тип данных* в реляционной модели данных полностью эквивалентно соответствующему понятию в алгоритмических языках. Набор поддерживаемых типов данных определяется СУБД и может сильно различаться в разных системах. Однако практически все СУБД поддерживают следующие типы данных:

- ☐ целочисленные;
- ☐ вещественные;
- ☐ строковые;
- ☐ специализированные типы данных для денежных величин;
- ☐ специальные типы данных для временных величин (дата и/или время);
- ☐ типы двоичных объектов (данный тип не имеет аналога в языках программирования; обычно для его обозначения используется аббревиатура BLOB — Binary Large Object).

ПРИМЕЧАНИЕ

Достаточно активно развивается подход к расширению возможностей реляционных систем абстрактными типами данных (соответствующими возможностями обладают, например, системы семейства Ingres/Postgres).

В рассматриваемом примере используются три типа данных — строковый (столбцы Имя и Специальность), временной тип (столбец Дата_рождения) и целочисленный тип (Курс и №_студенческого_билета).

Домен

Наименьшая единица данных реляционной модели — это отдельное *атомарное* (неразложимое) для данной модели значение данных. *Доменом* называется множество атомарных значений одного и того же типа. Иными словами, домен представляет собой допустимое потенциальное множество значений данного типа.

В нашем примере можно для каждого столбца таблицы определить домен:

- ❑ домены Имена и Специальности для столбцов Имя и Специальность соответственно будут базироваться на строковом типе данных — в число их значений могут входить только те строки, которые могут изображать имя и название специальности (в частности, такие строки не должны начинаться с мягкого знака);
- ❑ домен Даты_рождения для столбца Дата_рождения определяется на базовом временном типе данных — данный домен содержит только допустимый диапазон дат рождения студентов;
- ❑ домены Номера_курсов и Номера_студенческих_билетов базируются на целочисленном типе — в число его значений могут входить только те целые числа, которые могут обозначать номер курса университета (обычно от 1 до 6) и номер студенческого билета (обязательно положительное число).

ПРИМЕЧАНИЕ

Понятие домена более специфично для баз данных, хотя и имеет некоторые аналогии с диапазонными типами и множествами, имеющимися в ряде языков программирования. В самом общем виде домен определяется заданием некоторого базового типа данных, к которому относятся элементы домена, и произвольного логического выражения, применяемого к элементу типа данных. Если вычисление этого логического выражения дает результат «истина», то элемент данных является элементом домена.

Следует отметить также семантическую нагрузку понятия домена: данные считаются сравнимыми только в том случае, когда они относятся к одному домену. Если же значения двух атрибутов берутся из различных доменов, то их сравнение, вероятно, лишено смысла. В нашем примере значения доменов Номера_курсов и Номера_студенческих_билетов основаны на одном типе данных — целочисленном, но не являются сравнимыми.

ПРИМЕЧАНИЕ

Понятие домена используется далеко не во всех СУБД. В качестве примера реляционных баз данных, использующих домены, можно привести Oracle и InterBase.

Атрибуты, схема отношения, схема базы данных

Столбцы отношения называют *атрибутами*, им присваиваются имена, по которым к ним затем производится обращение.

Список имен атрибутов отношения с указанием имен доменов (или типов, если домены не поддерживаются) называется *схемой отношения*.

Схема нашего отношения СТУДЕНТ запишется так:

```
СТУДЕНТ {№_студенческого_билета Номера_студенческих_билетов  
Имя Имена,  
Дата_рождения Даты_рождения,  
Курс Номера_курсов,  
Специальность Специальности}
```

Степень отношения — это число его атрибутов. Отношение степени один называют унарным, степени два — бинарным, степени три — тернарным, ..., а степени n — n -арным.

Степень отношения СТУДЕНТЫ равна пяти, то есть оно является 5-арным.

Схемой базы данных называется множество именованных схем отношений.

Кортеж

Кортеж, соответствующий данной схеме отношения, представляет собой множество пар {имя атрибута, значение}, которое содержит одно вхождение каждого имени атрибута, принадлежащего схеме отношения. Аргумент Значение является допустимым значением домена данного атрибута (или типа данных, если понятие домена не поддерживается). Тем самым степень кортежа, то есть число элементов в нем, совпадает со степенью соответствующей схемы отношения. Иными словами, кортеж — это набор именованных значений заданного типа.

Таким образом, отношение по сути является множеством кортежей, соответствующим одной схеме отношения.

ПРИМЕЧАНИЕ

Схему отношения иногда называют также *заголовком* отношения, а отношение как набор кортежей — *телом* отношения.

ПРИМЕЧАНИЕ

Понятие схемы отношения напоминает понятие структурного типа данных в языках программирования (структура в C/C++, запись в Pascal). Однако в реляционных базах данных имя схемы отношения всегда совпадает с именем соответствующего отношения-экземпляра. В классических реляционных базах данных после определения схемы базы данных изменяются только отношения-экземпляры. В них могут появляться новые и удаляться или модифицироваться существующие кортежи. Однако во многих реализациях допускается и изменение схемы базы данных: определение новых и изменение существующих схем отношения. Это принято называть *эволюцией схемы базы данных*.

Кардинальным числом, или *мощностью отношения* называется число его кортежей. Мощность отношения СТУДЕНТЫ равна 6. В отличие от степени отношения, кардинальное число отношения изменяется во времени.

Пустые значения

В некоторых случаях какой-либо атрибут отношения может быть неприменим. Например, в рассматриваемом в качестве примера отношении **СТУДЕНТЫ** может также храниться информация о потенциальных абитуриентах, посещающих подготовительные курсы вуза. В этом случае неприменимыми оказываются атрибуты №_студенческого_билета и Специальность. Абитуриенты еще не поступили в вуз и, следовательно, не имеют студенческого билета. Кроме того, иногда при вводе информации в строку реляционной таблицы некоторые данные могут быть неизвестны и выясняться позже. Для нашего примера — это данные атрибута Специальность, так как при поступлении на подготовительные курсы абитуриент еще не определился окончательно, на какую специальность он будет поступать.

В обоих указанных случаях в поля, соответствующие неприменимым или неизвестным атрибутам, ничего не заносится, и строка записывается в базу данных с пустыми значениями этих атрибутов.

Следует понимать, что пустое значение — это не ноль и не пустая строка, а *неизвестное* значение атрибута, которое *не определено* в данный момент времени и, в принципе, может быть определено позднее.

ПРИМЕЧАНИЕ

Для обозначения пустых значений полей используется слово NULL.

Ключи отношения

Поскольку отношение с математической точки зрения является множеством, а множества по определению не содержат совпадающих элементов, то никакие два кортежа отношения не могут быть дубликатами друг друга в любой произвольно заданный момент времени. Таким образом, в отношении всегда должен присутствовать некоторый атрибут (или набор атрибутов), однозначно определяющий каждый кортеж отношения и обеспечивающий уникальность строк таблицы. Такой атрибут (или набор атрибутов) называется *первичным ключом* отношения.

Более строго определить понятие первичного ключа можно следующим образом:

если R — отношение с атрибутами A_1, A_2, \dots, A_n , то множество атрибутов $K = (A_i, A_j, \dots, A_k)$ отношения R является первичным ключом этого отношения тогда и только тогда, когда удовлетворяются два независимых от времени условия:

- *уникальность*: в произвольный момент времени никакие два различных кортежа отношения R не имеют одного и того же значения для A_i, A_j, \dots, A_k ;
- *минимальность*: ни один из атрибутов A_i, A_j, \dots, A_k не может быть исключен из K без нарушения уникальности.

Для каждого отношения свойством уникальности обладает по крайней мере полный набор его атрибутов. Однако требуется обеспечить и условие минимальности. Поэтому, как правило, в отношении всегда имеется один атрибут, обладающий свойством уникальности и являющийся первичным ключом.

В зависимости от количества атрибутов, входящих в ключ, различают простые и сложные (или составные) ключи.

Простой ключ — ключ, содержащий только один атрибут. В общем случае операции объединения выполняются быстрее в том случае, когда в качестве ключа используется самый короткий и самый простой из возможных типов данных. С этой точки зрения наилучшим образом подходит целочисленный тип, который имеет аппаратную поддержку для выполнения над ним логических операций.

Сложный, или составной, ключ — ключ, состоящий из нескольких атрибутов.

ПРИМЕЧАНИЕ

Набор атрибутов, обладающий свойством уникальности, но не обладающий минимальностью, называется *суперключом*. Суперключ — сложный (составной) ключ с большим числом столбцов, чем необходимо для того, чтобы быть уникальным идентификатором. Такие ключи нередко используются на практике, так как избыточность может оказаться полезной пользователю.

В зависимости от того, содержит ли атрибут, являющийся первичным ключом, какую-либо информацию, различают искусственные и естественные ключи.

Искусственный, или суррогатный, ключ — ключ, созданный самой СУБД или пользователем с помощью некоторой процедуры, который сам по себе не содержит информации. Искусственный ключ используется для создания уникальных идентификаторов строк, когда сущность должна быть описана полностью, чтобы однозначно идентифицировать конкретный элемент. Искусственный ключ часто используют вместо значимого сложного ключа, который является слишком громоздким, чтобы использоваться в реальной базе данных.

Естественный ключ — ключ, в который включены значимые атрибуты и который, таким образом, содержит информацию.

ПРИМЕЧАНИЕ

В рассматриваемом нами примере в качестве первичного ключа отношения СТУДЕНТЫ можно рассматривать атрибут №_студенческого_билета. Причем данный ключ будет естественным, так как он несет вполне определенную информацию.

Каждый из типов первичных ключей имеет свои преимущества и недостатки; их обсуждению посвящено большое количество публикаций. Мы не будем проводить подробное их сравнение, а отметим лишь основные плюсы и минусы каждого из видов ключей.

Основными достоинствами естественных ключей является то, что они несут вполне определенную информацию и их использование не приводит к необходимости добавлять в таблицы атрибуты, значения которых не имеют никакого смысла и используются лишь для связи между отношениями. Иными словами, использование естественных ключей позволяет получить более компактную форму таблиц (в которых не будет избыточных, неинформативных данных) и более естественные связи между ними.

Основным же недостатком естественных ключей является то, что их использование весьма затруднительно в случае изменчивости предметной области. Следует понимать, что значения атрибутов первичного ключа не должны изменяться. То есть однажды заданное значение первичного ключа для кортежа не может быть позже изменено. Такое требование ставится в основном для поддержания целостности базы данных. Связь между отношениями обычно устанавливается именно по первичному ключу, и его изменение приведет к нарушению этих связей или к необходимости изменения записей в нескольких таблицах. Даже в сравнительно простых базах данных это может вызвать ряд трудноразрешимых проблем.

ПРИМЕЧАНИЕ

В некоторых реляционных СУБД допускается изменение первичного ключа. Иногда это бывает действительно полезно. Однако прибегать к этому следует лишь в случае крайней необходимости.

Типичным примером изменчивой предметной области, где для сущности невозможно определить естественный ключ, который был бы неизменен, является любая область, когда в качестве сущности выступает человек.

Второй, довольно существенный недостаток естественных ключей состоит в том, что, как правило, уникальные естественные ключи являются составными и содержат строковые атрибуты. Как уже отмечалось выше, максимальная скорость выполнения операций над данными обеспечивается при использовании простых целочисленных ключей. Таким образом, с точки зрения быстродействия системы естественные ключи часто оказываются неоптимальными.

Оба недостатка естественных ключей можно преодолеть, определив в отношениях суррогатные ключи, представляющие собой некоторый универсальный атрибут, как правило целочисленного типа, который не зависит ни от предметной области, ни, тем более, от структуры отношения, которое он идентифицирует. Таким образом можно обеспечить уникальность и неизменность ключа (раз он никаким образом не зависит от предметной области, то никогда не возникнет необходимость изменять его). Однако за это приходится платить избыточностью данных в таблицах.

ПРИМЕЧАНИЕ

Следует заметить, что во многих практических реализациях реляционных СУБД допускается нарушение свойства уникальности кортежей для промежуточных отношений, порождаемых неявно при выполнении запросов. Такие отношения являются не множествами, а мультимножествами, что в ряде случаев позволяет добиться определенных преимуществ, но иногда приводит к серьезным проблемам.

В любой из таблиц может оказаться несколько наборов атрибутов, которые можно выбрать в качестве ключа. Такие наборы называются *потенциальными*, или *альтернативными*, ключами.

Нередко в отношениях определяются так называемые *вторичные ключи*. Вторичный ключ представляет собой комбинацию атрибутов, отличную от комбинации, составляющей первичный ключ. Причем вторичные ключи не обяза-

тельно обладают свойством уникальности. При их определении могут задаваться следующие ограничения;

- UNIQUE — ограничение уникальности, значения вторичных ключей при данном ограничении не могут дублироваться;
- NOT NULL — при данном ограничении ни один из атрибутов, входящих в состав вторичного ключа, не может принимать значение NULL.

Перекрывающиеся ключи — сложные ключи, которые имеют один или несколько общих столбцов.

Связанные отношения

В реляционной модели данные представляются в виде совокупности *взаимосвязанных* таблиц. Подобное взаимоотношение между таблицами называется *связью* (*relationship*). Таким образом, еще одним важным понятием реляционной модели является связь между отношениями.

Для рассмотрения связанных отношений воспользуемся рассмотренным ранее примером — отношением СТУДЕНТЫ. Данное отношение может быть связано с отношением УСПЕВАЕМОСТЬ, в котором содержатся сведения об успеваемости студентов по разным предметам. Фрагмент такого отношения может иметь вид, приведенный в табл. 4.2.

Таблица 4.2. Фрагмент отношения УСПЕВАЕМОСТЬ, связанного с отношением СТУДЕНТЫ

№_студенческого_билета	Предмет	Оценка
...
23980282	Высшая математика	4
23980282	Философия	5
22991380	Высшая математика	3
22991380	Философия	NULL
22657879	Общая физика	5
24356783	Общая физика	NULL
...

Атрибут №_студенческого_билета таблицы УСПЕВАЕМОСТЬ содержит идентификатор студента (в данном примере в качестве такого идентификатора используется номер студенческого билета). Если нужно узнать имя студента, соответствующее строкам в таблице УСПЕВАЕМОСТЬ, то следует поискать это же значение идентификатора студента в поле №_студенческого_билета таблицы СТУДЕНТЫ и в найденной строке прочесть значение поля Имя. Таким образом, связь между таблицами СТУДЕНТЫ и УСПЕВАЕМОСТЬ устанавливается по атрибуту №_студенческого_билета.

При рассмотрении связанных таблиц важное значение имеет понятие *внешнего ключа*. Рассмотрим его более подробно.

Внешние ключи отношения

В базах данных одни и те же имена атрибутов часто используются в разных отношениях. В рассматриваемом примере атрибут №_студенческого_билета присутствует

как в отношении СТУДЕНТЫ, так и в отношении УСПЕВАЕМОСТЬ. В этом примере атрибут №_студенческого_билета иллюстрирует понятие *внешнего ключа* (*foreign key*).

Внешний ключ — это атрибут (или множество атрибутов) одного отношения, являющийся ключом другого (или того же самого) отношения.

Внешние ключи используются для установления логических связей между отношениями. Связь между двумя таблицами устанавливается путем присваивания значений внешнего ключа одной таблицы значениям ключа другой.

Так же как и любые другие ключи, внешние ключи могут быть простыми либо составными.

Часто связь между отношениями устанавливается по первичному ключу, то есть значениям внешнего ключа одного отношения присваиваются значения первичного ключа другого отношения. Однако это не является обязательным — в общем случае связь может устанавливаться также и с помощью вторичных ключей. Кроме того, при установлении связей между таблицами необязательно требование уникальности ключа, по которому устанавливается связь.

ПРИМЕЧАНИЕ

Атрибуты внешнего ключа не обязательно должны иметь те же имена, что атрибуты ключа, которым они соответствуют. Например, в нашем примере можно было дать атрибуту №_студенческого_билета таблицы УСПЕВАЕМОСТЬ другое имя, например Студенческий_билет.

Внешний ключ может ссылаться и на ту же таблицу, к которой он принадлежит. В этом случае внешний ключ называется *рекурсивным*.

Условия целостности данных

Чтобы информация, хранящаяся в базе данных, была однозначной и непротиворечивой, в реляционной модели устанавливаются некоторые *ограничительные условия*. Ограничительные условия — это правила, определяющие возможные значения данных. Они обеспечивают логическую основу для поддержания корректных значений данных в базе. Ограничения целостности позволяют свести к минимуму ошибки, возникающие при обновлении и обработке данных.

Важнейшими ограничениями целостности данных являются:

- ☐ категорийная целостность;
- ☐ ссылочная целостность.

Ограничение категорийной целостности заключается в следующем. Кортежи отношения представляют в базе данных элементы определенных объектов реального мира, или, в соответствии с терминологией реляционных СУБД, *категорий*. Например, строка таблицы СТУДЕНТЫ представляет конкретного студента. Первичный ключ таблицы однозначно определяет каждый кортеж и, следовательно, каждый элемент категории. Таким образом, для извлечения данных, содержащихся в строке таблицы, или для манипулирования этими данными необходимо знать значение ключа для этой строки. Поэтому строка не может быть занесена в базу данных до тех пор, пока не будут определены все атрибуты ее первичного ключа. Это правило называется правилом категорийной целост-

ности и кратко формулируется следующим образом: никакой атрибут первичного ключа строки не может быть пустым.

Второе условие накладывает на внешние ключи ограничения для обеспечения целостности данных, называемые *ссылочной целостностью*.

Если две таблицы связаны между собой, то внешний ключ таблицы должен содержать только те значения, которые уже имеются среди значений ключа, по которому осуществляется связь. Если корректность значений внешних ключей не контролируется СУБД, то может нарушиться ссылочная целостность данных. Это можно пояснить на рассматриваемом примере следующим образом. Если удалить из таблицы *СТУДЕНТЫ* строку (например, при отчислении студента), имеющую хотя бы одну связанную с ней строку в таблице *УСПЕВАЕМОСТЬ*, то это приведет к тому, что в таблице *УСПЕВАЕМОСТЬ* останутся записи об успеваемости студента, который уже отчислен. Такая же ситуация будет наблюдаться и в том случае, если внешнему ключу таблицы *УСПЕВАЕМОСТЬ* ошибочно будет присвоено значение, отсутствующее в значениях ключа связанной таблицы.

Ограничения категорической и ссылочной целостности должны поддерживаться СУБД. Для соблюдения целостности сущности достаточно гарантировать отсутствие в любом отношении кортежей с одним и тем же значением первичного ключа. Что же касается ссылочной целостности, то ее обеспечение выглядит несколько сложнее. При обновлении ссылающегося отношения (при вставке новых кортежей или модификации значения внешнего ключа в существующих кортежах) достаточно следить за тем, чтобы не появлялись некорректные значения внешнего ключа. А вот при удалении кортежа из отношения, на которое ведет ссылка, возможно использование одного из трех подходов, каждый из которых поддерживает целостность по ссылкам:

- ❑ первый подход заключается в том, что запрещается производить удаление кортежа, на который существуют ссылки (то есть сначала нужно либо удалить ссылающиеся кортежи, либо соответствующим образом изменить значения их внешнего ключа);
- ❑ при втором подходе при удалении кортежа, на который имеются ссылки, во всех ссылающихся кортежах значение внешнего ключа автоматически становится неопределенным;
- ❑ третий подход (называемый также *каскадным удалением*) состоит в том, что при удалении кортежа из отношения, на которое ведет ссылка, из ссылающегося отношения автоматически удаляются все ссылающиеся кортежи.

В развитых реляционных СУБД обычно можно выбрать способ поддержания ссылочной целостности для каждой отдельной ситуации определения внешнего ключа. Конечно, для принятия такого решения необходимо анализировать требования конкретной прикладной области.

ПРИМЕЧАНИЕ

Хотя большинство современных СУБД обеспечивает ссылочную целостность данных, все же следует помнить, что существуют реляционные СУБД, в которых не выполняются ограничения ссылочной целостности. Это, как правило, ранние разработки локальных реляционных СУБД — FoxPro версии 2.6 и ниже, версии dBase для DOS.

Типы связей между таблицами

При установлении связи между двумя таблицами одна из них будет являться *главной* (*master*), а вторая — *подчиненной* (*detail*). Различие между ними несколько упрощенно можно пояснить следующим образом. В главной таблице всегда доступны все содержащиеся в ней записи. В подчиненной же таблице доступны только те записи, у которых значение атрибутов внешнего ключа совпадает со значением соответствующих атрибутов *текущей записи главной таблицы*. Причем изменение текущей записи главной таблицы приведет к изменению множества доступных записей подчиненной таблицы, а изменение текущей записи в подчиненной таблице не вызовет никаких изменений ни в одной из таблиц.

ПРИМЕЧАНИЕ

На практике часто связывают более двух таблиц. Одна и та же таблица может быть главной по отношению к одной таблице и подчиненной по отношению к другой. Или у одной главной таблицы может находиться в подчинении не одна, а несколько таблиц. Однако подчиненная таблица не может управляться двумя таблицами. Таким образом, у главной таблицы может быть несколько подчиненных, но у подчиненной таблицы может быть только одна главная.

Различают четыре типа связей между таблицами реляционной базы данных:

- ❑ *один к одному* — каждой записи одной таблицы соответствует только одна запись другой таблицы;
- ❑ *один ко многим* — одной записи главной таблицы могут соответствовать несколько записей подчиненной таблицы;
- ❑ *многие к одному* — нескольким записям главной таблицы может соответствовать одна и та же запись подчиненной таблицы;
- ❑ *многие ко многим* — одна запись главной таблицы связана с несколькими записями подчиненной таблицы, а одна запись подчиненной таблицы связана с несколькими записями главной таблицы.

ПРИМЕЧАНИЕ

Различие между типами связей «один ко многим» и «многие к одному» зависит от того, какая из таблиц выбирается в качестве главной, а какая — в качестве подчиненной. Например, если из связанных таблиц СТУДЕНТЫ и УСПЕВАЕМОСТЬ в качестве главной выбрать таблицу СТУДЕНТЫ, то получим тип связи «один ко многим». Если же выбрать в качестве главной таблицу УСПЕВАЕМОСТЬ, получится тип связи «многие к одному».

Основные свойства отношений

Рассмотрим теперь некоторые важнейшие свойства отношений реляционной модели данных.

Отсутствие упорядоченности кортежей

В таблицах реляционной базы данных информация хранится в неупорядоченном виде. Упорядочивание в принципе не поддерживается СУБД, и такое поня-

тие как порядковый номер кортежа не имеет никакого смысла. Свойство отсутствия упорядоченности кортежей отношения также является следствием определения отношения как множества кортежей. Отсутствие требования к поддержанию порядка на множестве кортежей отношения дает СУБД дополнительную гибкость при хранении баз данных во внешней памяти и при выполнении запросов к базе данных.

ПРИМЕЧАНИЕ

При проведении выборки данных из базы (с использованием, например, языка SQL) и отображении результатов этой выборки можно потребовать сортировки результирующей таблицы в соответствии со значениями некоторых атрибутов. Однако это не противоречит принципу отсутствия упорядоченности, так как результат выборки не является отношением, а представляет собой некоторый упорядоченный список кортежей.

Отсутствие упорядоченности атрибутов

Атрибуты отношений также не упорядочены, поскольку по определению схема отношения есть множество пар {имя атрибута, имя домена}. Для ссылки на значение атрибута в кортеже отношения всегда используется имя атрибута. Это свойство теоретически позволяет, например, модифицировать схемы существующих отношений не только путем добавления новых атрибутов, но и путем удаления существующих атрибутов. Однако в большинстве существующих систем такая возможность не допускается, и хотя упорядоченность набора атрибутов отношения явно не требуется, часто в качестве их неявного порядка используется последовательность в линейной форме определения схемы отношения.

Атомарность значений атрибутов

Значения всех атрибутов являются атомарными. Это следует из определения домена как потенциального множества значений простого типа данных, то есть среди значений домена не могут содержаться множества значений (отношения).

Реляционная система управления базами данных

Реляционная база данных — это совокупность отношений, содержащих всю информацию, которая должна храниться в базе данных. Однако пользователи могут воспринимать такую базу данных как совокупность таблиц. Таким образом, реляционную базу данных можно рассматривать как хранилище данных, содержащее набор двумерных связанных таблиц. Набор средств для управления подобным хранилищем называется *реляционной системой управления базами данных*. Реляционная СУБД может содержать утилиты, приложения, службы, библиотеки, средства создания приложений и другие компоненты.

Еще раз подчеркнем, что в реляционной базе данных таблицы связаны между собой; это позволяет с помощью единственного запроса найти все необходимые данные (которые могут находиться в нескольких таблицах). Будучи связанной посредством общих ключевых полей, информация в реляционной базе данных может объединяться из множества таблиц в единый результирующий набор.

Свойства таблиц реляционной базы данных

Так как таблицы в реляционной СУБД являются отношениями реляционной модели данных, то и свойства этих таблиц являются свойствами отношений, которые мы уже рассмотрели выше. Кратко сформулируем эти свойства еще раз:

- ❑ каждая таблица состоит из однотипных строк и имеет уникальное имя;
- ❑ строки имеют фиксированное число полей (столбцов) и значений (множественные поля и повторяющиеся группы недопустимы). Иначе говоря, в каждой позиции таблицы на пересечении строки и столбца всегда имеется в точности одно значение или NULL;
- ❑ строки таблицы обязательно отличаются друг от друга хотя бы единственным значением, что позволяет однозначно идентифицировать любую строку;
- ❑ столбцам таблицы присваиваются уникальные имена и в каждом из них размещаются однородные значения данных (даты, фамилии, целые числа или денежные суммы);
- ❑ полное информационное содержание базы данных представляется в виде явных значений данных и такой метод представления является единственным. В частности, не существует каких-либо специальных «связей» или указателей, соединяющих одну таблицу с другой;
- ❑ при выполнении операций с таблицей ее строки и столбцы можно обрабатывать в любом порядке безотносительно к их информационному содержанию. Этому способствует наличие имен таблиц и их столбцов, а также возможность выделения любой строки или любого набора строк с указанными признаками.

Индексы

Выше мы рассмотрели понятие ключей таблиц базы данных. В большинстве реляционных СУБД ключи реализуются с помощью объектов, называемых *индексами*.

Индекс представляет собой указатель на данные, размещенные в реляционной таблице. Можно провести аналогию индекса таблицы базы данных с указателем, обычно помещаемым в конце книги. Чтобы найти в книге страницы, относящиеся к некоторой теме, проще всего обратиться к указателю, в котором устанавливается соответствие между перечисленными в алфавитном порядке темами и номерами страниц, и сразу определить страницы, которые следует просмотреть. Чтобы без указателя найти все страницы, относящиеся к нужной теме, пришлось бы просматривать всю книгу. Индекс базы данных предназначен для аналогичных целей — чтобы ускорить поиск информации в таблице базы данных. Индекс предоставляет информацию о точном физическом расположении данных в таблице.

ПРИМЕЧАНИЕ

Мы отмечали, что записи в реляционных таблицах не упорядочены. Тем не менее любая запись в конкретный момент времени имеет вполне определенное физическое местоположение в файле базы данных, хотя оно и может изменяться при изменении информации, хранящейся в базе данных.

При создании индекса в нем сохраняется информация о местонахождении записей, относящихся к индексируемому столбцу таблицы. При добавлении в таблицу новых записей или удалении существующих индекс также модифицируется.

При выполнении запроса к базе данных, в условие поиска которого входит индексированный столбец, поиск значений производится в первую очередь в индексе. Если этот поиск оказывается успешным, то в индексе устанавливается точное местоположение искомых данных в таблице базы данных.

Рассмотрим пример индекса. На рис. 4.1 показан фрагмент таблицы **СТУДЕНТЫ** и индекса, построенного по полю **Имя** данной таблицы. При выполнении поиска по имени студента, просматривая индекс, можно сразу определить порядковый номер записи, содержащей необходимую информацию, и затем быстро найти в таблице сами данные. Если бы у таблицы отсутствовал индекс по полю **Имя**, то выполнение поиска по имени студента потребовало бы просмотра всей таблицы. Таким образом, использование индексов снижает время выборки данных.

Имя	Расположение
Алексеев Д. А.	1
Афанасьев А. В.	4
Кузнецов В. И.	5
Михайлов А. И.	1000
Михайлов В. В.	3
Смирнов А. Д.	6
...	...
Яковлев Н. В.	2

Расположение	...	Расположение	...	Курс	...
1		Алексеев Д. А.		2	
2		Яковлев Н. В.		4	
3		Михайлов В. В.		5	
4		Афанасьев А. В.		1	
5		Кузнецов В. И.		1	
6		Смирнов А. Д.		3	
...		
1000		Михайлов А. И.		3	

Рис. 4.1. Поиск информации в таблице с помощью индекса

ПРИМЕЧАНИЕ

Ускорение поиска информации при использовании индекса может показаться неочевидным — ведь количество записей в индексе совпадает с количеством записей в таблице. Однако следует учитывать два обстоятельства:

- обращение к индексу выполняется быстрее, чем к таблице;
- в индексе записи хранятся в упорядоченном виде (в рассматриваемом примере — в алфавитном порядке) и поэтому при поиске информации в индексе нет необходимости просматривать все данные до конца индекса.

Различают несколько типов индексов. Наиболее часто выделяют три типа:

- ☐ простые;
- ☐ составные;
- ☐ уникальные.

Простые индексы представляют собой простейший и вместе с тем наиболее распространенный тип индекса. Простой индекс строится на основе только одного столбца реляционной таблицы (индекс, приведенный на рис. 4.1, является простым).

Составные индексы строятся по двум и более столбцам реляционной таблицы. При создании составного индекса необходимо принимать во внимание, что последовательность столбцов, по которым создается индекс, влияет на скорость поиска данных.

ПРИМЕЧАНИЕ

Последовательность столбцов в составном индексе указывается при его создании и никаким образом не связана с последовательностью столбцов в таблице.

Можно назвать два условия оптимальности следования столбцов в составном индексе:

- ❑ первым следует помещать столбец, содержащий наиболее ограничивающее значение (то есть содержащий меньшее количество повторов);
- ❑ первым следует помещать столбец, содержащий данные, которые наиболее часто задаются в условиях поиска.

Сформулированные условия оптимальности часто являются противоречивыми, так что между ними следует находить разумный компромисс.

Уникальные индексы не допускают введения в таблицу дублирующих значений. Уникальные индексы используются не только с целью повышения скорости поиска, но и для поддержания целостности данных. Уникальный индекс может быть как простым, так и составным.

ПРИМЕЧАНИЕ

Следует серьезно относиться к планированию индексов. Неправильное применение индексов может привести к снижению производительности системы. Мы уже говорили о том, что физическое местоположение записей может изменяться в процессе редактирования данных пользователями, а также в результате манипуляций с файлами базы данных, проводимых самой СУБД (таких как сжатие данных, сборка «мусора» и др.). Обычно при этом происходят соответствующие изменения и в индексе, а это увеличивает время, требующееся СУБД для проведения таких операций. Поэтому обычно не следует индексировать:

- столбцы, данные в которых подвержены частому изменению;
 - столбцы, содержащие большое количество пустых значений;
 - столбцы, содержащие небольшое количество уникальных значений;
 - небольшие таблицы;
 - поля большого размера.
-

Нормализация данных

Нормализация представляет собой процесс реорганизации данных путем ликвидации повторяющихся групп и иных противоречий с целью приведения таблиц к виду, позволяющему осуществлять непротиворечивое и корректное редактирование данных.

Окончательная цель нормализации сводится к получению такого проекта базы данных, в котором *каждый факт появляется лишь в одном месте*, то есть исключена избыточность информации. Таким образом, нормализацию можно

также определить как процесс, направленный на уменьшение избыточности информации в реляционной базе данных.

Цели нормализации

Избыточность информации устраняется не столько с целью экономии памяти, сколько для исключения возможной противоречивости хранимых данных и упрощения управления ими.

Использование ненормализованных таблиц может привести к нарушению целостности данных (противоречивости информации) в базе данных. Обычно различают следующие проблемы, возникающие при использовании ненормализованных таблиц:

- ☐ избыточность данных;
- ☐ аномалии обновления;
- ☐ аномалии удаления;
- ☐ аномалии ввода.

Чтобы проиллюстрировать проблемы, возникающие при работе с ненормализованными базами данных, рассмотрим в качестве примера таблицу СОТРУДНИКИ, содержащую информацию о сотрудниках некоей организации. Структура этой таблицы приведена на рис. 4.2.

Код сотрудника
Имя
Фамилия
Отчество
Дата рождения
Адрес
Телефон
Должность
Разряд
Зарплата
Рейтинг
Дата приема
Дата увольнения

Рис. 4.2. Структура ненормализованной таблицы СОТРУДНИКИ

Избыточность данных

Избыточность данных проявляется в том, что в нескольких записях таблицы базы данных повторяется одна и та же информация. Например, один человек может работать на двух (или даже более) должностях. Но в таблице, приведенной на рис. 4.2, каждой должности соответствует запись, и в этой записи содержится информация о личных данных сотрудника, эту должность занимающего. Таким образом, если сотрудник работает на нескольких должностях, то его личные

данные будут дублироваться несколько раз, что приведет к неоправданному увеличению занимаемого объема внешней памяти.

Аномалии обновления

Аномалии обновления тесно связаны с избыточностью данных. Предположим, что у сотрудника, работающего на нескольких должностях, изменился адрес. Чтобы информация, содержащаяся в таблице, была корректной, необходимо будет внести изменения в несколько записей. Если же исправление будет внесено не во все записи, то возникнет несоответствие информации, которое и называется аномалией обновления.

Аномалии удаления

Аномалии удаления возникают при удалении записей из ненормализованной таблицы. Пусть, например, в организации проводится сокращение штатов и некоторые должности аннулируются. При этом следует удалить соответствующие записи в рассматриваемой таблице. Однако удаление приведет к потере информации о сотруднике, занимавшем эту должность. Такая потеря информации и называется аномалией удаления. (Для нашего случая можно привести и другой пример — удаление записи при увольнении сотрудника приведет к потере информации о должности, которую он занимал.)

Аномалии ввода

Аномалии ввода возникают при добавлении в таблицу новых записей и обычно возникают, когда для некоторых полей таблицы заданы ограничения NOT NULL. В таблице, рассматриваемой в качестве примера, имеется поле Рейтинг, в котором содержится информация об уровне квалификации сотрудника, устанавливаемом по результатам его работы. При приеме на работу нового сотрудника установить уровень его квалификации невозможно, так как он еще не выполнял никаких работ в организации. Если для этого поля задать ограничение NOT NULL, то в таблицу нельзя будет ввести информацию о новом сотруднике. Это и называется аномалией ввода.

Выводы

Очевидно, что аномалии обновления, удаления и ввода крайне нежелательны. Чтобы свести к минимуму возможность появления такого рода аномалий, и используется нормализация.

Нормальные формы

Теория нормализации основана на концепции *нормальных форм*. Каждой нормальной форме соответствует некоторый определенный набор ограничений, и отношение находится в некоторой нормальной форме, если оно удовлетворяет свойственному данной форме набору ограничений.

В теории реляционных баз данных обычно выделяется следующая последовательность нормальных форм:

- первая нормальная форма (1NF);
- вторая нормальная форма (2NF);

- ❑ третья нормальная форма (3NF);
- ❑ нормальная форма Бойса—Кодда (BCNF);
- ❑ четвертая нормальная форма (4NF);
- ❑ пятая нормальная форма, или нормальная форма проекции-соединения (5NF или PJ/NF).

Основные свойства нормальных форм:

- ❑ каждая следующая нормальная форма в некотором смысле лучше предыдущей;
- ❑ при переходе к следующей нормальной форме свойства предыдущих нормальных форм сохраняются.

В основе процесса проектирования лежит метод нормализации — декомпозиция отношения, находящегося в предыдущей нормальной форме, в два или более отношения, удовлетворяющих требованиям следующей нормальной формы.

Наиболее важные на практике нормальные формы отношений основываются на фундаментальном в теории реляционных баз данных понятии *функциональной зависимости*. Функционально зависимым считается такой атрибут, значение которого однозначно определяется значением другого атрибута. Функционально зависимые атрибуты обозначаются следующим образом: $X \rightarrow Y$. Эта запись означает, что если два кортежа в таблице имеют одно и то же значение атрибута X , то они имеют одно и то же значение атрибута Y . Атрибут, указываемый в левой части, называется *детерминантом*.

ПРИМЕЧАНИЕ

Первичный ключ таблицы является детерминантом, так как его значение однозначно определяет значение любого атрибута таблицы.

Первая нормальная форма

Ограничение первой нормальной формы — значения всех атрибутов отношения должны быть атомарными. Данное требование является базовым требованием классической реляционной модели данных, поэтому любая реляционная таблица (в том числе и таблица, структура которой изображена на рис. 4.2) по определению уже находится в первой нормальной форме.

Вторая нормальная форма

Отношение находится во второй нормальной форме в том и только в том случае, когда это отношение находится в первой нормальной форме и каждый неключевой атрибут полностью зависит от первичного ключа.

ПРИМЕЧАНИЕ

Неключевым называется любой атрибут отношения, не входящий в состав первичного ключа.

Чтобы перейти от первой нормальной формы ко второй, нужно выполнить следующие шаги.

- 1. Определить, на какие части можно разбить первичный ключ, так чтобы некоторые из неключевых полей зависели от одной из этих частей (причем эти части могут содержать несколько атрибутов).
- 2. Создать новую таблицу для каждой такой части ключа и группы зависящих от нее полей и переместить их в эту таблицу. Часть бывшего первичного ключа станет при этом первичным ключом новой таблицы.
- 3. Удалить из исходной таблицы поля, перемещенные в другие таблицы, кроме тех из них, которые станут внешними ключами.

В нашем примере для приведения таблицы СОТРУДНИКИ ко второй нормальной форме ее следует разделить на две таблицы. Первичный ключ исходной таблицы состоит из двух атрибутов — Код сотрудника и Должность. Все же личные данные о сотрудниках зависят только от атрибута Код сотрудника. Атрибуты, соответствующие этим данным, мы и выделим в качестве одной из таблиц, которую назовем ФИЗИЧЕСКИЕ ЛИЦА. Информацию же о должностях и их оплате вынесем в другую таблицу, которой присвоим имя СОТРУДНИКИ. Схема приведения таблицы ко второй нормальной форме приведена на рис. 4.3.



Рис. 4.3. Приведение таблицы ко второй нормальной форме

Полученные две таблицы связаны между собой по полю Код физического лица, которое является первичным ключом для таблицы ФИЗИЧЕСКИЕ ЛИЦА и внешним ключом для таблицы СОТРУДНИКИ. Данное поле отсутствовало в исходной таблице и было добавлено при проведении нормализации.

Третья нормальная форма

Рассмотрим таблицу СОТРУДНИКИ, полученную после приведения исходной таблицы ко второй нормальной форме. Для этой таблицы существует функцио-

нальная связь между полями Код сотрудника и Зарплата. Однако эта функциональная связь является *транзитивной*.

ПРИМЕЧАНИЕ

Функциональная зависимость атрибутов X и Y отношения R называется транзитивной, если существует такой атрибут Z , что имеются функциональные зависимости $X \rightarrow Z$ и $Z \rightarrow Y$, но отсутствует функциональная зависимость $Z \rightarrow X$.

Транзитивность зависимости полей Код сотрудника и Зарплата означает, что заработная плата на самом деле является характеристикой не сотрудника, а должности, которую он занимает. В результате мы не сможем занести в базу данных информацию, характеризующую заработную плату должности, до тех пор пока не появится хотя бы один сотрудник, эту должность занимающий (так как первичный ключ не может содержать неопределенное значение). При удалении кортежа, описывающего последнего сотрудника, занимающего данную должность, мы лишимся информации о заработной плате, соответствующей этой должности. Кроме того, чтобы согласованным образом изменить заработную плату, соответствующую должности, будет необходимо предварительно найти все записи, описывающие сотрудников, занимающих данную должность. Таким образом, в таблице СОТРУДНИКИ по-прежнему существуют аномалии. Их можно устранить путем дальнейшей нормализации — приведения базы данных к третьей нормальной форме.

Отношение R находится в третьей нормальной форме в том и только том случае, если оно находится во второй нормальной форме и каждый неключевой атрибут нетранзитивно зависит от первичного ключа.

Чтобы перейти от второй нормальной формы к третьей, нужно выполнить следующие шаги.

1. Определить все поля (или группы полей), от которых зависят другие поля.
2. Создать новую таблицу для каждого такого поля (или группы полей) и группы зависящих от него полей и переместить их в эту таблицу. Поле (или группа полей), от которого зависят все остальные перемещенные поля, станет при этом первичным ключом новой таблицы.
3. Удалить перемещенные поля из исходной таблицы, оставив лишь те из них, которые станут внешними ключами.

Приведем рассматриваемую в качестве примера базу данных к третьей нормальной форме. Для этого разделим таблицу СОТРУДНИКИ на две — СОТРУДНИКИ и ДОЛЖНОСТИ (рис. 4.4).

ПРИМЕЧАНИЕ

Обратите внимание, что мы опять добавили новый атрибут — Код должности, который является первичным ключом для отношения ДОЛЖНОСТИ и внешним ключом для отношения СОТРУДНИКИ. Добавление новых атрибутов при нормализации позволяет получить таблицы с простыми первичными ключами, что облегчает выполнение операции связывания таблиц. Такие первичные ключи, как правило, являются искусственными.

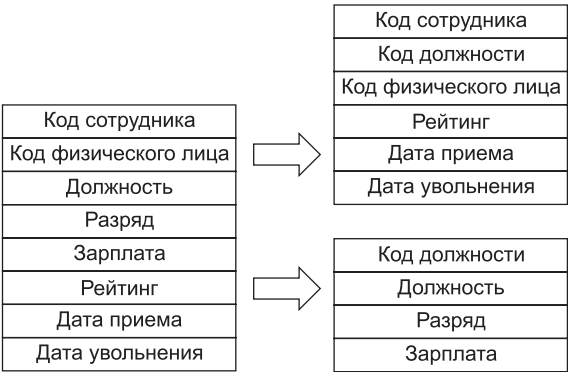


Рис. 4.4. Приведение базы данных к третьей нормальной форме

На практике третья нормальная форма схем отношений в большинстве случаев достаточна, и приведением к третьей нормальной форме процесс проектирования реляционной базы данных обычно заканчивается. Поэтому мы не будем рассматривать другие нормальные формы, тем более что в работе они используются сравнительно редко.

В заключение приведем схему базы данных, рассматриваемой в качестве примера и приведенной к третьей нормальной форме (рис. 4.5.).

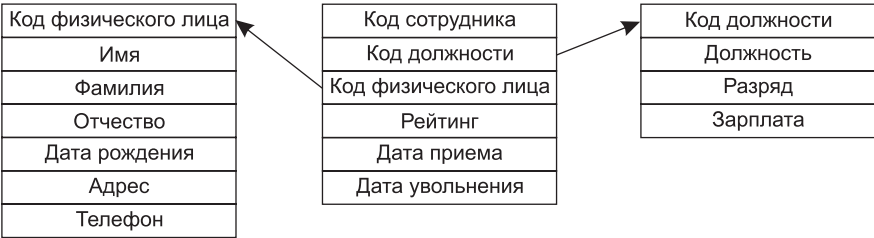


Рис. 4.5. Структура базы данных, приведенной к третьей нормальной форме

Глава 5

Управление реляционными базами данных

Использование реляционных баз данных возможно только при наличии эффективных средств управления ими. Поэтому после публикаций статей Кодда, предлагающих реляционную модель данных, стали активно проводиться исследования по созданию языков управления реляционными данными. В результате этих исследований был предложен ряд таких языков, среди которых следует отметить три:

- ❑ SQL — Structured Query Language (структурированный язык запросов);
- ❑ QBE — Query By Example (запрос по образцу);
- ❑ QUEL — Query Language (язык запросов).

Сейчас наибольшее распространение получил язык SQL, который является единственным языком реляционных баз данных, принятым в качестве стандарта ANSI.

ПРИМЕЧАНИЕ

Хотя SQL и называется языком *запросов*, он включает в себя, кроме средств запросов, и все необходимые средства по управлению базами данных.

В данной главе мы рассмотрим возможности языка SQL по управлению объектами реляционной базы данных и администрированию баз данных.

Краткая история языка SQL

Язык реляционных баз данных SQL был разработан в середине 1970-х годов в рамках исследовательского проекта экспериментальной реляционной СУБД System R компании IBM. Данный проект включал в себя разработку реляционной системы управления базами данных и языка SEQUEL (Structured English Query Language). Исходное название SEQUEL только частично отражало суть этого языка. Несмотря на то что язык был ориентирован главным образом на

удобную и понятную пользователям формулировку запросов к реляционной базе данных, он уже являлся полноценным языком реляционной базы данных, содержащим, помимо операторов формулирования запросов и манипулирования базой данных, следующие элементы:

- ❑ средства определения схемы базы данных и манипулирования ей;
- ❑ средства определения ограничений целостности и триггеров;
- ❑ средства создания представлений базы данных;
- ❑ возможности определения структур физического уровня, поддерживающих эффективное выполнение запросов;
- ❑ средства авторизации доступа к отношениям и их полям;
- ❑ средства поддержки точек сохранения транзакции и откатов.

В конце 1970-х годов модифицированный вариант языка SEQUEL, получивший название SQL, был выпущен корпорацией Oracle в качестве языка коммерческой системы управления базами данных. В 1983 году компания IBM выпустила SQL в качестве языка управления СУБД DB2.

Американский национальный институт стандартов (ANSI) принял язык SQL в качестве стандарта в 1986 году. С тех пор этот стандарт пересматривался два раза — в 1989 году были внесены некоторые незначительные изменения, а в 1992 году стандарт SQL был довольно существенно расширен и в настоящее время известен под названием ANSI SQL-92 или SQL/92.

ПРИМЕЧАНИЕ

Следует понимать, что ANSI SQL — всего лишь стандарт, не являющийся реальным языком. Каждый производитель систем управления базами данных, как правило, предлагает собственную реализацию языка SQL. Причем в таких реализациях могут быть как расширения существующего стандарта, так и отклонения от него, в том числе отсутствие некоторых стандартных элементов языка. Тем не менее, независимо от реализации, основа SQL сохраняется, поэтому при изучении языка SQL главным является понимание базовых концепций и команд ANSI SQL-92.

Типы команд SQL

Команды языка SQL обычно подразделяются на несколько групп. Основные типы команд следующие:

- ❑ DDL (Data Definition Language) — язык определения данных. Команды данной группы используются для создания и изменения структуры объектов базы данных (например, для создания и удаления таблиц);
- ❑ DML (Data Manipulation Language) — язык манипулирования данными. Команды DML используются для манипулирования информацией, содержащейся в объектах базы данных;
- ❑ DCL (Data Control Language) — язык управления данными. Соответствующие команды предназначены для управления доступом к информации, хранящейся в базе данных;

- ❑ DQL (Data Query Language) — язык запросов. Это наиболее часто используемые команды, предназначенные для формирования запросов к базе данных (*запрос* — это обращение к базе данных для получения соответствующей информации);
- ❑ команды администрирования базы данных предназначены для осуществления контроля за выполняемыми действиями и анализа производимых операций;
- ❑ команды управления транзакциями.

ПРИМЕЧАНИЕ

Язык запросов в данной главе рассматриваться не будет. Применение команды языка запросов будут подробно обсуждаться далее, в главе 11, «Выборка данных», на примере использования в приложениях Delphi.

Типы данных SQL/92

Типы данных, используемые в стандартном SQL, можно подразделить на следующие группы:

- ❑ строковые типы;
- ❑ числовые типы;
- ❑ типы для представления даты и времени.

Рассмотрим эти типы данных более подробно.

Строковые типы

В SQL/92 определены два строковых типа:

- ❑ символьные строки фиксированной длины;
- ❑ символьные строки переменной длины.

Символьные строки фиксированной длины

Данные, хранящиеся в виде символьных строк фиксированной длины, всегда занимают один и тот же объем памяти, определяемый при объявлении поля, независимо от реального размера строки, занесенной в поле. Объявление строки фиксированной длины согласно ANSI SQL-92 имеет вид:

CHARACTER(n)

где n — длина строки, определяющая размер поля, к которому это объявление относится.

При использовании строк фиксированной длины пустые места обычно заполняются пробелами. Например, если размер поля задан равным 10, а в него введена строка, состоящая из 3 символов, то оставшиеся 7 символов заполняются пробелами.

ПРИМЕЧАНИЕ

Не следует использовать тип CHARACTER для полей, предназначенных для хранения длинных строк, длина которых может сильно варьироваться, — это приведет к неоправданному расходу доступной внешней памяти (дискового пространства).

Символьные строки переменной длины

Длина строк переменной длины не является постоянной для всех данных, а зависит от реального размера строки, хранящейся в поле таблицы базы данных. Объявление строки переменной длины имеет вид:

`VARCHAR(n)`

где n — число, определяющее максимально возможную длину строки.

В отличие от типа `CHARACTER`, использование `VARCHAR` обеспечивает более экономное расходование дискового пространства. Независимо от того, какой размер строки указан в объявлении, поле будет занимать столько места, сколько необходимо для хранения занесенной в него информации. Например, если объявлено поле `VARCHAR(10)` и в него занесена строка длиной 3 символа, то для хранения этой строки будет использовано только три байта, а не 10, как в случае строки фиксированной длины.

Числовые типы

Числовые типы подразделяются на:

- ❑ целочисленные типы;
- ❑ вещественные типы с фиксированной точкой;
- ❑ вещественные типы с плавающей точкой;
- ❑ двоичные строки фиксированной и переменной длины.

Целочисленные типы

Стандартом ANSI SQL-92 устанавливаются два целочисленных типа:

- ❑ `INTEGER` — целое число со знаком, использующее 4 байта. Может представлять числа в диапазоне от $-2\,147\,483\,648$ до $2\,147\,483\,647$;
- ❑ `SMALLINT` — короткое целое число со знаком, использующее 2 байта. Может представлять целые числа в диапазоне от $-32\,768$ до $32\,767$.

Вещественные типы с фиксированной точкой

Вещественные типы с фиксированной точкой предназначены для точного представления дробных чисел. Наиболее часто эти типы используются в том случае, когда недопустимы погрешности, неизбежные при представлении вещественных чисел с плавающей запятой в двоичной форме (например, при хранении значений денежных величин). Вещественные типы с фиксированной запятой по сути являются целочисленными типами, в которых отображается десятичная точка.

Синтаксис объявления типа с фиксированной запятой следующий:

`DECIMAL(n,m)`

где n — точность;

m — масштаб.

Точность — это общая длина числового значения.

Масштаб — количество знаков, расположенных справа от десятичной точки.

Вещественные типы с плавающей точкой

Типы с плавающей точкой обычно используются в научных и инженерных расчетах. При их использовании следует учитывать, что в процессе занесения в базу данных некоторого числа при его преобразовании в двоичную форму с плавающей точкой всегда вносится некоторая погрешность. И хотя эта погрешность очень мала, в некоторых случаях она является недопустимой и может привести к серьезной ошибке, например при суммировании большого количества значений. Поэтому типы с плавающей точкой неприменимы для хранения значений денежных величин.

Наиболее часто используются два вещественных типа с плавающей точкой:

- FLOAT — числа с одинарной точностью;
- DOUBLE — числа с двойной точностью.

Двоичные строки

Двоичные строки используются сравнительно редко. Обычно поля такого типа применяются в качестве флагов или двоичных масок.

Так же как и символьные строки, двоичные строки бывают фиксированной и переменной длины. Двоичные строки фиксированной длины объявляются следующим образом:

BIT(*n*)

где *n* — длина строки в байтах.

Объявление строк переменной длины выглядит так:

BIT VARYING(*n*)

где *n* — максимальная длина строки в байтах.

Типы для представления даты и времени

Очевидно, что данные типы используются для хранения информации, относящейся к датам и времени.

ПРИМЕЧАНИЕ

Иногда типы данных, предназначенные для хранения времени и даты, называются *темпоральными*.

В стандарте SQL определены следующие типы данных для хранения информации о дате и времени:

- DATE — используется для хранения даты;
- TIME — используется для хранения времени;
- TIMESTAMP — хранит дату и время;
- INTERVAL — хранит промежуток времени между двумя датами или между двумя моментами времени.

ПРИМЕЧАНИЕ

Следует заметить, что в большинстве реализаций SQL поддерживаются некоторые дополнительные типы данных. Кроме того, синтаксические и семантические свойства типов, определенных в стандарте ANSI SQL-92, могут различаться в отдельных реализациях SQL.

ПРИМЕЧАНИЕ

Необходимо всегда иметь в виду, что хотя с использованием языка SQL можно определить схему базы данных, содержащую данные любого из рассмотренных типов, возможность использования этих данных в прикладных системах зависит от применяемого языка программирования.

Управление объектами базы данных

Объект базы данных — это любой объект, определенный в базе данных и используемый для хранения информации или для обращения к информации. Примерами объектов базы данных могут служить таблицы, представления и индексы.

Для управления объектами базы данных используется подмножество команд DDL языка SQL.

Создание, модификация и удаление таблиц

Таблица является основным объектом для хранения информации в реляционной базе данных. При создании таблицы обязательно указываются имена полей, содержащихся в таблице, и типы данных, соответствующие полям. Кроме того, при создании таблицы для полей могут оговариваться ограничительные условия и значения, задаваемые по умолчанию.

Ограничительные условия — это правила, ограничивающие значения величин в поле таблицы базы данных.

Значение по умолчанию — значение, которое автоматически вводится в поле таблицы базы данных при добавлении новой записи, если пользователь не указал значение этого поля.

Оператор CREATE TABLE

Для создания таблицы используется оператор CREATE TABLE. Его синтаксис имеет следующий вид:

```
CREATE TABLE имя_таблицы (  
    имя_поля_1 тип_данных,  
    имя_поля_2 тип_данных,  
    ...  
    имя_поля_N тип_данных)
```

Для примера рассмотрим оператор, создающий таблицу ФИЗИЧЕСКИЕ ЛИЦА, рассмотренную в предыдущей главе:

```
CREATE TABLE Физические лица (  
    Код_физического_лица INTEGER,  
    Имя VARCHAR(25),  
    Фамилия VARCHAR(25),  
    Отчество VARCHAR(25),  
    Дата_рождения DATE,  
    Адрес VARCHAR(50),  
    Телефон VARCHAR(25))
```

ПРИМЕЧАНИЕ

В приводимом примере, чтобы избежать путаницы и неоднозначности, мы использовали русские имена таблицы и полей. При создании же реальных таблиц реальной базы данных следует иметь в виду, что далеко не все СУБД допускают использование символов кириллицы в именах полей и таблиц. Более того, даже если СУБД позволяет использовать русские буквы, желательно все же применять в именах объектов базы данных только латинские символы, особенно если для создания интерфейсной части информационной системы предполагается использовать средства разработки третьих фирм. Это позволит избежать целого ряда трудноразрешимых проблем.

Оператор ALTER TABLE

Созданная таблица может быть модифицирована с использованием оператора ALTER TABLE. С его помощью можно добавлять и удалять поля таблицы, изменять тип данных полей, добавлять и удалять ограничения.

ПРИМЕЧАНИЕ

Оператор ALTER TABLE не определен в стандарте ANSI. Однако он поддерживается в большинстве реализаций SQL, обеспечивая существенно большую гибкость управления структурой базы данных. Если же используемая СУБД не поддерживает ALTER TABLE, то можно просто создать новую таблицу с измененной структурой и затем перенести в нее данные из старой таблицы, после чего старую таблицу можно будет удалить.

В общем виде синтаксис оператора ALTER TABLE выглядит следующим образом:

```
ALTER TABLE имя_таблицы [MODIFY] [имя_поля тип_данных]
[ADD] [имя_поля тип_данных]
[DROP] [имя_поля]
```

Действие, выполняемое оператором ALTER TABLE, определяется ключевым словом, указываемым после имени таблицы:

- ❑ MODIFY — изменяет определение поля;
- ❑ ADD — добавляет новое поле в таблицу;
- ❑ DROP — удаляет поле из таблицы.

Для изменения типа данных поля используется следующий синтаксис оператора ALTER TABLE:

```
ALTER TABLE имя_таблицы MODIFY (имя_поля тип_данных)
```

Например, для того чтобы добавить в таблицу ФИЗИЧЕСКИЕ ЛИЦА поле, в котором будет содержаться адрес электронной почты сотрудника, следует использовать следующий оператор:

```
ALTER TABLE Физические_лица ADD (Email CHARACTER(25))
```

Если же требуется изменить тип данных существующего поля, то следует использовать оператор ALTER TABLE в паре с ключевым словом MODIFY:

```
ALTER TABLE имя_таблицы MODIFY (имя_поля тип_данных)
```

Пусть, например, после того как мы добавили в таблицу ФИЗИЧЕСКИЕ ЛИЦА поле Email, выяснилось, что использование типа CHARACTER для этого поля неэффективно — у многих сотрудников нет электронной почты и, следовательно, часть дискового пространства расходуется впустую. Целесообразнее применить для этого поля тип данных VARCHAR. Для изменения типа данных вызовем оператор ALTER TABLE:

```
ALTER TABLE Физические_лица MODIFY (Email VARCHAR(25))
```

Удаление существующего поля выполняется вызовом оператора ALTER TABLE с ключевым словом DROP:

```
ALTER TABLE имя_таблицы DROP (имя_поля)
```

ПРИМЕЧАНИЕ

Следует быть очень осторожным при использовании оператора ALTER TABLE. Непродуманное внесение изменений в таблицы уже работающей базы данных может привести к нарушению работы всей системы в целом.

Оператор DROP TABLE

Для удаления таблиц используется оператор DROP TABLE. Его синтаксис имеет следующий вид:

```
DROP TABLE имя_таблицы [RESTRICT | CASCADE]
```

Если при вызове оператора DROP TABLE используется ключевое слово RESTRICT, а на удаляемую таблицу ссылается какое-либо представление или ограничение, то при выполнении оператора удаления таблицы будет сгенерировано сообщение об ошибке. Если же использовать ключевое слово CASCADE, то удаление таблицы будет выполнено и вместе с таблицей будут удалены все ссылающиеся на нее представления и ограничения.

Задание ограничений

Ограничения используются для того, чтобы обеспечить достоверность и непротиворечивость информации в базе данных. Существует достаточно большое количество различного рода ограничений, из которых мы рассмотрим лишь основные:

- ☐ ограничение NOT NULL;
- ☐ ограничение первичного ключа;
- ☐ ограничение UNIQUE;
- ☐ ограничение внешнего ключа;
- ☐ ограничение CHECK.

Ограничение NOT NULL

Ограничение NOT NULL может быть установлено для любого поля реляционной таблицы. При наличии этого ограничения запрещается ввод значений NULL в поле, для которого это ограничение установлено.

ПРИМЕЧАНИЕ

Следует понимать, что значение NULL не эквивалентно ни нулевому значению для числовых полей, ни пробелу для полей текстовых — если в поле занесено значение «0» (или « »), то поле не пустое, а содержит число 0 (или строку, состоящую из одного пробела). Если же значение поля равно NULL, то это означает, что поле содержит неопределенное значение (поле пустое), то есть в него не была занесена никакая информация.

Ограничение NOT NULL устанавливается при создании таблицы с помощью оператора CREATE TABLE. Чтобы задать ограничение NOT NULL для некоторого поля, следует просто указать NOT NULL после указания типа поля:

```
CREATE TABLE имя_таблицы (  
    имя_поля_1 тип_данных NOT NULL,  
    имя_поля_2 тип_данных NULL,  
    ...  
    имя_поля_N тип_данных NOT NULL)
```

Если же после задания типа данных поля следует слово NULL, то данное поле может содержать пустые значения. Однако атрибут NULL обычно устанавливается по умолчанию, поэтому указывать его явно нет необходимости.

Ограничение NOT NULL устанавливается для тех полей, в которые при занесении данных в таблицу обязательно должна быть введена какая-либо информация. Например, в таблице, содержащей личные данные о сотрудниках организации, можно задать ограничение NOT NULL для полей, в которых будут содержаться имя и фамилия сотрудника. Поэтому оператор создания таблицы ФИЗИЧЕСКИЕ ЛИЦА следует видоизменить следующим образом:

```
CREATE TABLE Физические лица (  
    Код_физического_лица INTEGER,  
    Имя VARCHAR(25) NOT NULL,  
    Фамилия VARCHAR(25) NOT NULL,  
    Отчество VARCHAR(25),  
    Дата_рождения DATE,  
    Адрес VARCHAR(50),  
    Телефон VARCHAR(25))
```

ПРИМЕЧАНИЕ

При добавлении нового поля в непустую таблицу с использованием оператора ALTER TABLE нельзя устанавливать ограничение NOT NULL для добавляемого поля. Это вполне очевидно — уже существующие записи в таблице не могут иметь в новом столбце непустые значения. Однако это ограничение можно преодолеть следующим образом.

1. Добавить в таблицу поле без ограничения NOT NULL.
 2. Заполнить значения нового поля для всех существующих записей.
 3. Изменить определение нового поля с помощью команды ALTER TABLE, задав ему ограничение NOT NULL.
-

Ограничение первичного ключа

Первичные ключи указываются при создании таблицы. Так как поля, входящие в состав первичного ключа, не могут принимать значение NULL, то для них

обязательным является ограничение NOT NULL. Ограничение первичного ключа может быть задано двумя путями.

1. В том случае, когда первичный ключ состоит только из одного поля, он может быть задан с помощью ключевых слов PRIMARY KEY, указываемых при описании поля в операторе CREATE TABLE:

```
CREATE TABLE имя_таблицы (  
    имя_поля_1 тип_данных NOT NULL PRIMARY KEY,  
    имя_поля_2 тип_данных,  
    ...  
    имя_поля_N тип_данных NOT NULL)
```

Обратите внимание на то, что указание ограничения NOT NULL для поля, являющегося первичным ключом, обязательно.

2. Первичный ключ может быть также задан в конце описания таблицы, после определений всех полей. Для этого также используется ключевая фраза PRIMARY KEY, после которой в круглых скобках указывается имя поля, составляющего первичный ключ:

```
CREATE TABLE имя_таблицы (  
    имя_поля_1 тип_данных NOT NULL,  
    имя_поля_2 тип_данных,  
    ...  
    имя_поля_N тип_данных NOT NULL,  
    PRIMARY KEY (имя_поля_1))
```

Второй способ особенно удобен для задания составных первичных ключей. В этом случае в скобках следует указать через запятую все поля, составляющие первичный ключ:

```
CREATE TABLE имя_таблицы (  
    имя_поля_1 тип_данных NOT NULL,  
    имя_поля_2 тип_данных,  
    имя_поля_3 тип_данных NOT NULL,  
    ...  
    имя_поля_N тип_данных NOT NULL,  
    PRIMARY KEY (имя_поля_1, имя_поля_3))
```

ПРИМЕЧАНИЕ

При использовании составного первичного ключа ограничение NOT NULL должно быть задано для всех полей, входящих в его состав.

Ограничение UNIQUE

Ограничение UNIQUE похоже на ограничение первичного ключа, так как при наличии этого ограничения для некоторого поля все значения, содержащиеся в этом поле, должны быть уникальными. Однако, в отличие от первичного ключа, ограничение UNIQUE допускает наличие пустых значений поля (если, конечно, для этого поля не установлено ограничение NOT NULL).

Ограничение UNIQUE задается при создании таблицы с помощью ключевого слова UNIQUE, указываемого при описании поля:

```
CREATE TABLE имя_таблицы (  
    имя_поля_1 тип_данных NOT NULL PRIMARY KEY,  
    ...  
    имя_поля_N тип_данных UNIQUE)
```

```
имя_поля_2 тип_данных UNIQUE,  
имя_поля_3 тип_данных NOT NULL,  
...  
имя_поля_N тип_данных NOT NULL UNIQUE)
```

Можно также задать ограничение UNIQUE не для одного поля, а для группы полей. Объявление уникальной группы полей отличается от объявления уникальных индивидуальных полей, так как именно комбинация значений, а не просто индивидуальные значения, обязана быть уникальной. То есть значение каждого поля, входящего в группу, не обязательно должно быть уникальным, а комбинация значений полей всегда должна быть уникальной.

Ограничение UNIQUE для группы полей, так же как и составной первичный ключ, задается после описания всех полей таблицы:

```
CREATE TABLE имя_таблицы (  
    имя_поля_1 тип_данных NOT NULL PRIMARY KEY,  
    имя_поля_2 тип_данных,  
    имя_поля_3 тип_данных NOT NULL,  
    ...  
    имя_поля_N тип_данных NOT NULL UNIQUE,  
    UNIQUE (имя_поля_2, имя_поля_3))
```

Ограничение внешнего ключа

Ограничение внешнего ключа является основным механизмом поддержания ссылочной целостности базы данных. Поле, определяемое в качестве внешнего ключа, используется для ссылки на поле другой таблицы, обычно называемое *родительским ключом*, а таблица, на которую внешний ключ ссылается, называется *родительской таблицей* (родительский ключ часто является первичным ключом родительской таблицы).

Типы полей внешнего и родительского ключа обязательно должны быть идентичны, а вот их имена могут быть разными. Однако во избежание путаницы желательно и имена полей для внешнего и родительского ключей задавать одинаковыми.

Внешний ключ не обязательно должен состоять только из одного поля. Подобно первичному ключу, внешний ключ может состоять из любого числа полей, которые обрабатываются как единый объект. Поля родительского ключа, на который ссылается составной внешний ключ, должны следовать в том же порядке, что и во внешнем ключе.

Когда поле таблицы является внешним ключом, оно определенным образом связано с таблицей, на которую этот ключ ссылается. Это фактически означает, что каждое значение внешнего ключа непосредственно привязано к значению в родительском ключе.

В качестве иллюстрации использования ограничения внешнего ключа возьмем пример из предыдущей главы — базу данных по учету сотрудников некоторой организации (рис. 5.1). Эта база данных состоит из трех таблиц:

- ❑ СОТРУДНИКИ — содержит информацию о профессиональных данных сотрудников;
- ❑ ФИЗИЧЕСКИЕ ЛИЦА — содержит информацию о личных данных сотрудников;
- ❑ ДОЛЖНОСТИ — содержит информацию о должностях организации.

Основной таблицей в этой базе данных является таблица СОТРУДНИКИ, которая ссылается на две другие таблицы и соответственно должна иметь два внешних ключа. В качестве родительских ключей в таблицах ФИЗИЧЕСКИЕ ЛИЦА и ДОЛЖНОСТИ используются первичные ключи.

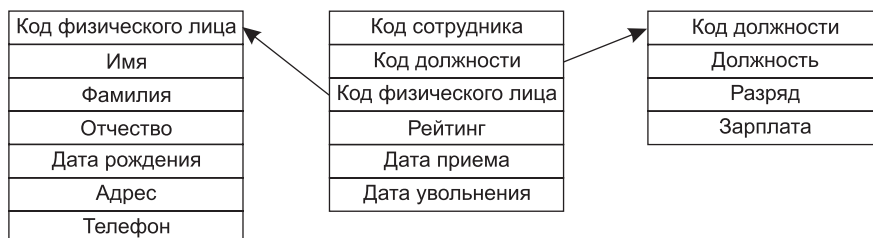


Рис. 5.1. База данных сотрудников организации

Ограничение внешнего ключа (FOREIGN KEY) может быть задано либо в операторе CREATE TABLE, либо с помощью оператора ALTER TABLE. Синтаксис ограничения FOREIGN KEY имеет следующий вид:

```
FOREIGN KEY имя_внешнего_ключа(список полей внешнего ключа)
REFERENCES имя_родительской_таблицы (список полей родительского ключа)
```

Первый список полей — это список из одного или нескольких полей таблицы, разделенных запятыми. Второй список полей представляет собой родительский ключ. Списки полей, указываемые в качестве внешнего и родительского ключей, должны быть совместимы:

- они должны иметь одинаковое число полей;
- порядок следования полей в списках должен совпадать. Причем совпадение определяется не именами полей, которые могут быть различны, а типами данных и размером полей.

Рассмотрим пример создания базы данных со связанными таблицами:

```
CREATE TABLE Физические лица (
    Код_физического_лица INTEGER NOT NULL PRIMARY KEY,
    Имя VARCHAR(25) NOT NULL,
    Фамилия VARCHAR(25) NOT NULL,
    Отчество VARCHAR(25),
    Дата_рождения DATE,
    Адрес VARCHAR(50),
    Телефон VARCHAR(25))
```

```
CREATE TABLE Должности (
    Код_должности INTEGER NOT NULL PRIMARY KEY,
    Должность VARCHAR(50) NOT NULL UNIQUE,
    Разряд INTEGER NOT NULL,
    Зарплата DECIMAL(7,2) NOT NULL)
```

```
CREATE TABLE Сотрудники (
    Код_сотрудника INTEGER NOT NULL PRIMARY KEY,
    Код_должности INTEGER,
    Код_физического_лица INTEGER NOT NULL,
```

```
Рейтинг DECIMAL(4,2),  
Дата_приема DATE NOT NULL,  
Дата_увольнения DATE,  
FOREIGN KEY Физ_ВК (Код_физического_лица)  
REFERENCES Физические_лица (Код_физического_лица),  
FOREIGN KEY Должн_ВК (Код_должности)  
REFERENCES Должности (Код_должности))
```

Внешний ключ может быть добавлен и после создания таблицы — с помощью оператора `ALTER TABLE` (естественно, только в том случае, если используемая реализация SQL поддерживает данный оператор). Синтаксис оператора `ALTER TABLE`, используемый для создания внешнего ключа, имеет следующий вид:

```
ALTER TABLE имя_таблицы  
ADD CONSTRAINT имя_внешнего_ключа FOREIGN KEY (список полей внешнего ключа)  
REFERENCE имя_родительской_таблицы (список полей родительского ключа)
```

ПРИМЕЧАНИЕ

Следует иметь в виду, что при использовании оператора `ALTER TABLE` для создания связи между таблицами необходимо, чтобы связываемые таблицы находились в состоянии ссылочной целостности. Иначе при попытке выполнения оператора будет выдано сообщение об ошибке.

Внешний ключ ограничивает значения, которые можно ввести в таблицу. Чтобы в поля, составляющие внешний ключ, можно было ввести некоторое значение, необходимо, чтобы это значение уже было введено в родительской таблице. Например, чтобы занести в таблицу `СОТРУДНИКИ` нового сотрудника, необходимо, чтобы в таблице `ФИЗИЧЕСКИЕ ЛИЦА` уже существовала запись о его личных данных — иначе невозможно будет заполнить обязательное поле `Код_физического_лица`.

Для внешнего ключа может быть задано ограничение `NOT NULL`, но это необязательно, а в некоторых случаях даже нежелательно. Например, предположим, что в организацию принимается на работу новый сотрудник, но еще не определена однозначно должность, которую он займет. В этом случае можно занести все необходимые данные о нем в таблицу `ФИЗИЧЕСКИЕ ЛИЦА` и в таблицу `СОТРУДНИКИ`, ничего не указывая в поле `Код_должности`, которое будет заполнено позже.

Ограничение внешнего ключа также оказывает влияние на удаление и модификацию записей родительской таблицы. Никакое значение родительского ключа, на которое ссылается какой-либо внешний ключ, не может быть удалено или изменено. Это означает, например, что нельзя удалить из таблицы `ФИЗИЧЕСКИЕ ЛИЦА` запись о сотруднике, если она связана с записью в таблице `СОТРУДНИКИ`. Это вполне понятно — если в таблице `СОТРУДНИКИ` присутствует запись о сотруднике фирмы, а из таблицы `ФИЗИЧЕСКИЕ ЛИЦА` запись об этом сотруднике удалена, то информация о его личных данных будет потеряна. Если же сотрудник уволился и запись о нем из таблицы `СОТРУДНИКИ` удалена, то нет необходимости хранить информацию о его личных данных и соответствующая запись из таблицы `ФИЗИЧЕСКИЕ ЛИЦА` также может быть удалена.

Аналогично, нельзя изменять значение родительского ключа, на который ссылается какой-либо внешний ключ, — это также приведет к потере информации и нарушению ссылочной целостности базы данных.

ПРИМЕЧАНИЕ

В некоторых реализациях SQL имеется возможность задавать для внешних ключей каскадное удаление и каскадное обновление. Это означает, что при попытке удалить или модифицировать значение родительского ключа, на которое ссылается внешний ключ, соответствующие записи внешнего ключа также будут удалены (каскадное удаление) или изменены (каскадное обновление). Данные возможности отсутствуют в стандарте ANSI SQL-92.

Одним из синтаксических вариантов задания каскадного обновления и удаления является следующий:

```
UPDATE OF имя_родительской_таблицы CASCADES  
DELETE OF имя_родительской_таблицы CASCADES
```

Ключевые фразы UPDATE OF и DELETE OF указываются в операторе CREATE TABLE. Вместо ключевого слова CASCADES можно указать слово RESTRICTED — в этом случае изменение и удаление значений родительского ключа, на которые ссылается внешний ключ из данной таблицы, будет запрещено.

```
CREATE TABLE Сотрудники (  
    Код_сотрудника INTEGER NOT NULL PRIMARY KEY,  
    Код_должности INTEGER,  
    Код_физического_лица INTEGER NOT NULL,  
    Рейтинг DECIMAL(4,2),  
    Дата_приема DATE NOT NULL,  
    Дата_увольнения DATE,  
    FOREIGN KEY Физ_ВК (Код_физического_лица)  
    REFERENCES Физические_лица (Код_физического_лица),  
    FOREIGN KEY Должн_ВК (Код_должности)  
    REFERENCES Должности (Код_должности),  
    UPDATE OF Физические_лица CASCADES  
    DELETE OF Физические_лица RESTRICTED)
```

Ограничение CHECK

Ограничение CHECK используется для проверки допустимости данных, вводимых в поле таблицы.

Ограничение CHECK состоит из ключевого слова CHECK, сопровождаемого предложением предиката, который использует указанное поле. Любая попытка модифицировать или вставить значение поля, которое могло бы сделать этот предикат неверным, будет отклонена.

ПРИМЕЧАНИЕ

Проверка корректности значений, заносимых в базу данных, может также выполняться в пользовательских приложениях. Однако использование ограничения CHECK обеспечивает дополнительный уровень защиты от ошибок.

Задание ограничения CHECK производится при создании таблицы. Для этого после описания полей таблицы указывается ключевая фраза:

```
CONSTRAINT имя_ограничения CHECK (ограничение)
```

В рассматриваемом нами примере базы данных сотрудников организации ограничение может быть задано, например, для поля Разряд таблицы ДОЛЖНОСТИ.

Допустим, разряд не может превышать 20. Тогда оператор создания таблицы ДОЛЖНОСТИ, в котором задано это ограничение, будет иметь следующий вид:

```
CREATE TABLE Должности (  
    Код_должности INTEGER NOT NULL PRIMARY KEY,  
    Должность VARCHAR(50) NOT NULL UNIQUE,  
    Разряд INTEGER NOT NULL,  
    Зарплата DECIMAL(7,2) NOT NULL,  
    CONSTRAINT CHK_RATE CHECK (Разряд<=20))
```

Можно задавать ограничение и для нескольких полей. Для этого следует просто включить их в ограничительное условие. Для формирования сложного ограничения, включающего несколько условий, используются логические операторы AND и OR.

В таблице ДОЛЖНОСТИ можно, например, ввести еще ограничение на минимальную зарплату:

```
...  
CONSTRAINT CHK_RATE CHECK (Разряд<=20 AND Зарплата>=1000))  
...
```

Задание значений по умолчанию

Для полей таблицы можно задавать *значения по умолчанию*, которые будут заноситься в поля при добавлении новой записи в таблицу, если значения этих полей не определены.

ПРИМЕЧАНИЕ

Значение NULL фактически является значением по умолчанию, принятым для каждого поля таблицы, для которого не задано ограничение NOT NULL и которое не имеет другого значения по умолчанию.

Для задания значения по умолчанию используется директива DEFAULT, которая указывается в команде CREATE TABLE при описании поля, для которого устанавливается значение по умолчанию:

```
CREATE TABLE (  
...  
имя_поля_N тип_данных DEFAULT =значение_по_умолчанию  
...)
```

В рассматриваемом примере значение по умолчанию может быть, например, установлено для поля Рейтинг таблицы СОТРУДНИКИ:

```
CREATE TABLE Сотрудники (  
    Код_сотрудника INTEGER NOT NULL PRIMARY KEY,  
    Код_должности INTEGER,  
    Код_физического_лица INTEGER NOT NULL,  
    Рейтинг DECIMAL(4,2) DEFAULT=0,  
    Дата_приема DATE NOT NULL,  
...)
```

Создание и удаление индексов

Стандарт ANSI в настоящее время не поддерживает индексы. Тем не менее индексы широко применяются практически во всех базах данных, поэтому работу с ними нельзя обойти вниманием.

Синтаксис оператора создания индекса может существенно различаться в зависимости от используемой реализации SQL. Наиболее часто встречается следующая синтаксическая форма команды создания индекса:

```
CREATE INDEX имя_индекса  
ON имя_таблицы (имя_поля_1, [имя_поля_2, ...])
```

ПРИМЕЧАНИЕ

Приведенная форма оператора CREATE INDEX может быть дополнена рядом многочисленных параметров, которые сильно различаются в разных реализациях SQL. Эти параметры используются, например, для упорядочивания информации по возрастанию или убыванию (параметры ASC и DESC).

Создание простого индекса

Простой индекс является простейшей и вместе с тем распространенной разновидностью индексов. Простой индекс состоит только из одного поля (столбца) таблицы, поэтому он часто также называется *одно столбцовым индексом*.

Наиболее типичный синтаксис команды создания простого индекса имеет вид:

```
CREATE INDEX имя_индекса  
ON имя_таблицы (имя_столбца)
```

Например, для таблицы ФИЗИЧЕСКИЕ ЛИЦА можно было бы создать индекс по полю, содержащему фамилии сотрудников, с помощью следующего оператора:

```
CREATE INDEX NAME_IDX  
ON Физические_лица (Фамилия)
```

Уникальные индексы

Уникальный индекс не допускает введения в таблицу дублирующихся значений. Таким образом, уникальные индексы используются не только с целью повышения производительности, но и для поддержания целостности данных.

Типичный синтаксис оператора создания уникального индекса имеет следующий вид:

```
CREATE UNIQUE INDEX имя_индекса  
ON имя_таблицы (имя_поля)
```

Например, для таблицы ДОЛЖНОСТИ можно создать уникальный индекс по полю Должность с помощью следующей команды:

```
CREATE UNIQUE INDEX POST_IDX  
ON Должности (Должность)
```

ПРИМЕЧАНИЕ

Создать уникальный индекс для существующей таблицы можно только в том случае, если в индексируемом поле не содержится повторяющихся значений.

Составные индексы

Составными называются индексы, построенные по двум и более полям. При создании составного индекса необходимо учитывать, что порядок следования полей в составном индексе оказывает существенное влияние на скорость поиска

данных. В общем случае поля в индексе следует располагать в порядке уменьшения ограничивающих значений.

Синтаксис задания составного индекса имеет следующий вид:

```
CREATE INDEX имя_индекса  
ON имя_таблицы (имя_поля_1, имя_поля_2, ...)
```

В нашем примере имеет смысл создать составной индекс для полей Фамилия и Имя таблицы ФИЗИЧЕСКИЕ ЛИЦА. Оператор создания такого индекса имеет следующий вид:

```
CREATE INDEX FULLNAME_IDX  
ON Физические_лица (Фамилия,Имя)
```

ПРИМЕЧАНИЕ

Обратите внимание на то, что порядок следования полей в последнем примере должен быть именно таким, так как поле Фамилия накладывает более сильное ограничение, чем поле Имя — вероятность того, что у нескольких сотрудников будут одинаковые имена, выше, чем вероятность совпадения фамилий.

Удаление индексов

Удаление индексов не вызывает никаких проблем. Для удаления необходимо знать только имя индекса (и, разумеется, обладать соответствующими правами). Синтаксис оператор удаления индекса имеет следующий вид:

```
DROP INDEX имя_индекса
```

Удаление индекса никак не влияет на информацию, содержащуюся в индексированных полях. После удаления индекс может быть создан вновь.

ПРИМЕЧАНИЕ

Типичной причиной удаления индексов является попытка увеличения производительности базы данных. Для достижения оптимальной производительности часто требуется проведение длительных экспериментов с индексами — их создание и удаление с возможным последующим воссозданием в прежнем или измененном виде.

Работа с представлениями

Представление (View) является объектом базы данных, работа с которым ничем не отличается от работы с обычной таблицей. Отличие представлений от таблиц заключается в следующем. Обычные таблицы баз данных содержат данные. Представления же данных не содержат, а их содержимое выбирается из других таблиц (или других представлений). Таблицы (или представления), на основе которых формируются представления, принято называть *базовыми таблицами* (или *базовыми представлениями*).

Фактически представление является запросом, который выполняется всякий раз, когда происходит обращение к представлению. Результат выполнения этого запроса в каждый момент времени является *содержанием представления*. При изменении данных в базовых таблицах представления изменяется и содержание представления. Изменение данных в представлении может приводить к изменению данных в таблицах, на основе которых это представление создано. На рис. 5.2 схематично поясняется процесс формирования представления.

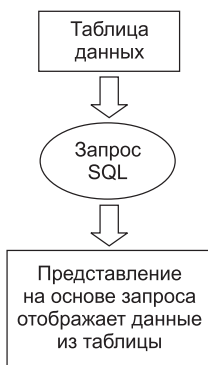


Рис. 5.2. Схема формирования представления

Использование представлений незначительно отличается от использования таблиц. Выборка данных из представления выполняется точно так же, как и из обычной таблицы. Допускаются также операции манипулирования данными представления, хотя здесь имеются некоторые ограничения.

Представления, в отличие от таблиц, не занимают дискового пространства (или, точнее, дисковое пространство, занимаемое представлениями, очень мало — только то, что требуется для хранения запроса).

Области применения представлений

Представления в основном применяются в двух случаях:

- ❑ с целью защиты данных;
- ❑ для формирования итоговых данных.

В первом случае представления применяются для того, чтобы предоставить пользователю информацию не из всей таблицы, а лишь из некоторых ее полей.

Рассмотрим следующий пример. Пусть, например, информация о рейтингах сотрудников, хранящаяся в поле Рейтинг таблицы СОТРУДНИКИ, считается конфиденциальной, и право доступа к ней имеют лишь руководители организации. Но часть информации, хранящейся в этой же таблице, необходима работникам отдела кадров — данные об именах сотрудников и датах их приема на работу. В этом случае для разграничения доступа к одной таблице удобно использовать представление, отобрав для него только ту информацию, к которой должны иметь доступ служащие отдела кадров. При этом они смогут выполнять свои служебные обязанности в полном объеме и не будут иметь доступа к конфиденциальной информации.

Представления могут быть использованы для ограничения доступа не только к полям, но и к записям таблицы. Для этого достаточно в запросе на выборку данных, на основе которого создается представление, указать соответствующее ограничительное условие. Например, в рассмотренном выше примере с работниками отдела кадров можно при создании представления задать условие, которое будет исключать из представления сотрудников, занимающих определенные должности.

ПРИМЕЧАНИЕ

Как уже отмечалось, представления строятся на основе *запросов* к базе данных, которые будут рассмотрены далее в главе 11, «Выборка данных», посвященной подмножеству команд DQL языка SQL. Поэтому в этом разделе мы приведем лишь общие сведения о представлениях, не вдаваясь в подробности формирования запросов.

Представления также используются для формирования итоговых результатов при формировании отчетов. В том случае, когда требуется часто распечатывать отчет, формируемый на основе таблиц с часто изменяемой информацией, удобно использовать представления. Так как представление может быть создано на основе запроса, содержащего предложения группировки, то можно создать представление, получающее информацию из ряда базовых таблиц и группирующих ее необходимым образом, а при выводе отчета обращаться к этому представлению как к обычной таблице. В этом случае не нужно будет каждый раз при выводе отчета формировать сложный SQL-запрос. Кроме того, в этом случае часть логики окажется вынесенной на сторону сервера базы данных, так как формирование отчета не будет зависеть от клиентского приложения.

Создание представлений

Для создания представлений используется оператор CREATE VIEW. Представление может быть создано на основе одной или нескольких таблиц и/или других представлений. Наиболее типичный синтаксис оператора создания представлений имеет следующий вид:

```
CREATE VIEW имя_представления AS  
{оператор выборки данных}
```

Оператор выборки может быть любой сложности, он может содержать любые условия отбора и предложения группировки.

ПРИМЕЧАНИЕ

Для получения подробной информации об операторе выборки SELECT обращайтесь к главе 11, «Выборка данных».

После создания представления с ним можно работать как с обычной таблицей, имеющей имя, заданное в качестве имени представления. Некоторым исключением являются представления, содержащие предложения группировки. Для таких представлений нет никаких ограничений по выборке данных, но применение к ним операторов манипулирования данными (подмножества команд DDL) недопустимо.

Удаление представлений

Удаление представлений выполняется с помощью оператора DROP VIEW, при вызове которого могут указываться параметры RESTRICT или CASCADE. Данные параметры определяют действия при удалении представления, на которое ссылаются другие представления и/или ограничения. При использовании варианта RESTRICT в этом случае будет выдано сообщение об ошибке, и удаление не будет

выполнено. Если же используется режим CASCADE, то выполнение оператора DROP VIEW приведет к удалению всех базовых представлений и ограничений.

Типовой синтаксис оператора DROP VIEW имеет следующий вид:

```
DROP VIEW имя_представления [RESTRICT | CASCADE]
```

ПРИМЕЧАНИЕ

Удаление представления (в отличие от удаления таблицы) не приводит к удалению данных, на которые это представление ссылается. Удаляется лишь запрос на выборку данных, на основе которого было создано представление.

Хранимые процедуры

Хранимые процедуры (Stored Procedure) представляют собой группы связанных операторов SQL. Использование хранимых процедур обеспечивает дополнительную гибкость при работе с базой данных, так как выполнить хранимую процедуру обычно гораздо проще, чем последовательность отдельных операторов SQL.

Хранимые процедуры находятся в базе данных в откомпилированном виде, что обеспечивает более высокую скорость их выполнения.

Хранимые процедуры могут получать входные параметры, возвращать значения приложению и могут быть вызваны явно из приложения или подстановкой вместо имени таблицы в инструкции SELECT.

Основные преимущества, которые дает использование хранимых процедур, заключаются в следующем:

- ❑ хранимые процедуры позволяют вынести часть логики на сервер базы данных. Это ослабляет зависимость базы данных информационной системы от клиентской части;
- ❑ хранимые процедуры обеспечивают модульность проекта: они могут быть общими для клиентских приложений, которые обращаются к одной и той же базе данных, что позволяет избегать повторяющегося кода и уменьшает размер приложений;
- ❑ хранимые процедуры упрощают сопровождение приложений: при обновлении процедур изменения автоматически отражаются во всех приложениях, которые их используют, без необходимости повторной компиляции и сборки;
- ❑ хранимые процедуры повышают эффективность работы информационной системы: они выполняются сервером, а не клиентом, что снижает сетевой трафик;
- ❑ скорость выполнения хранимых процедур выше, чем для последовательности отдельных операторов SQL. Это связано с тем, что хранимые процедуры хранятся на сервере в откомпилированном виде.

Различают два вида хранимых процедур:

- ❑ *процедуры выбора*, которые приложения могут использовать вместо таблиц или представлений в операторе выборки данных. Процедура выбора должна

возвращать одно или несколько значений, иначе результатом выполнения процедуры будет ошибка;

- ❑ *выполняемые процедуры*, которые вызываются явно с использованием специального оператора. Выполняемая процедура может не возвращать результата вызываемой программе.

Создание хранимых процедур

Для создания хранимых процедур используется оператор `CREATE PROCEDURE`. Синтаксис этого оператора сильно зависит от используемой реализации SQL, поэтому мы не будем его подробно рассматривать.

Язык процедур, как правило, включает все инструкции SQL для манипулирования данными и ряд расширений, включающих:

- ❑ условные операторы;
- ❑ различные виды операторов цикла;
- ❑ возможности обработки исключительных ситуаций.

Хранимые процедуры состоят из заголовка и тела. Заголовок процедуры содержит:

- ❑ имя процедуры, которое должно быть уникальным среди имен процедур и таблиц в базе данных;
- ❑ список входных параметров и их типов данных, которые процедура принимает из вызывающей программы (может отсутствовать);
- ❑ список выходных параметров и их типов данных, если процедура возвращает значения в вызывающую программу.

Тело процедуры содержит:

- ❑ список локальных переменных и их типов данных (если они используются в коде процедуры);
- ❑ блок инструкций на языке процедур и триггеров, заключенный между ключевыми словами `BEGIN` и `END`. Блок может включать в себя другие блоки, реализуя несколько уровней вложенности.

Выполнение хранимых процедур

Оператор, запускающий хранимую процедуру на выполнение, зависит от типа процедуры. Процедуры выбора выполняются при обращении к ним с помощью оператора выборки данных `SELECT` (данный оператор подробно описывается далее в главе 11, «Выборка данных»).

Для вызова выполняемой процедуры следует использовать специальный оператор `EXECUTE`. Синтаксис этого оператора зависит от используемой реализации SQL.

Удаление хранимых процедур

Для удаления хранимых процедур используется оператор `DROP PROCEDURE`. Синтаксис этого оператора является достаточно общим для различных реализаций SQL и имеет следующий вид:

```
DROP PROCEDURE имя_хранимой_процедуры
```

Триггеры

Триггеры представляют собой разновидность хранимых процедур. Однако в отличие от последних выполнение триггера происходит не в результате явного вызова некоторого оператора SQL, а при выполнении одного из операторов манипулирования данными, вносящими изменения в базу данных. При этом триггеры могут исполняться как до, так и после выполнения оператора манипулирования данными.

ПРИМЕЧАНИЕ

В определенном смысле триггеры являются аналогами обработчиков событий языка Object Pascal (и ряда других языков).

Триггеры используются для обеспечения ссылочной целостности данных в базе. Они предоставляют следующие возможности:

- ❑ возможность контроля вводимых данных, осуществляя гарантию того, что пользователь ввел в поля таблицы только допустимые значения;
- ❑ упрощение сопровождения приложений, так как изменение в триггере автоматически отражается во всех приложениях, которые используют таблицы со связанными с ними триггерами;
- ❑ автоматическое документирование изменений таблицы. Приложение может управлять журналом изменений с помощью триггеров, которые выполняют-ся всякий раз, когда происходит изменение таблицы.

Создание триггера

Для создания триггера используется оператор `CREATE TRIGGER`. Синтаксис этого оператора существенно зависит от используемой реализации SQL, поэтому мы не будем рассматривать его подробно и поговорим лишь об общих особенностях создания триггеров.

Так же как и хранимые процедуры, триггеры состоят из заголовка и тела. Заголовков триггера содержит:

- ❑ имя триггера, уникальное внутри базы данных;
- ❑ имя таблицы, с которой связан триггер;
- ❑ инструкции, которые определяют, когда триггер будет выполняться (при выполнении какого оператора манипулирования данными и в какой момент времени — до или после выполнения оператора).

Тело триггера содержит:

- ❑ список локальных переменных и их типов данных (если они используются в коде триггера);
- ❑ блок инструкций на языке процедур и триггеров, заключенный между ключевыми словами `BEGIN` и `END`. Блок может содержать в себе другой блок, реализуя несколько уровней вложенности.

Таким образом, отличие триггера от хранимой процедуры заключается только в заголовке.

Триггер связан с таблицей. Владелец таблицы и любой пользователь, наделенный привилегиями на таблицу, автоматически имеют право выполнять связанные с ней триггеры.

ПРИМЕЧАНИЕ

После создания триггера в него нельзя внести изменения. Чтобы внести изменения в уже созданный триггер, необходимо удалить его и создать заново. Некоторые реализации SQL также допускают замену триггера (в том случае если триггер с указанным именем уже существует) при выполнении оператора CREATE TRIGGER (при этом нет необходимости явно удалять ранее созданный триггер).

Удаление триггера

Для удаления триггера используется оператор DROP TRIGGER. Синтаксис этого оператора является достаточно общим для различных реализаций SQL и имеет следующий вид:

```
DROP TRIGGER имя_триггера
```

Манипулирование данными

Для манипулирования данными, хранящимися в базе данных, используется группа операторов SQL, выделяемая в качестве отдельного типа команд, называемых языком манипулирования данными (DML — Data Manipulation Language). С помощью операторов DML пользователь может загружать в таблицы новые данные, модифицировать и удалять существующие данные.

В языке SQL определены только три основных оператора DML:

- ☐ INSERT;
- ☐ UPDATE;
- ☐ DELETE.

Добавление в таблицу новой информации

Процесс ввода в таблицу базы данных новой информации обычно называется *загрузкой данных*. Для загрузки данных используется оператор INSERT.

Добавление к таблице новой записи

Для добавления к таблице новой записи используется следующая синтаксическая форма оператора INSERT:

```
INSERT INTO имя_таблицы  
VALUES (значение_1, значение_2, ..., значение_N)
```

При использовании данной формы оператора INSERT список VALUES должен содержать количество значений, равное количеству полей таблицы. Причем тип данных каждого из значений, указываемых в списке VALUES, должен совпадать с типом данных поля, соответствующего этому значению.

ПРИМЕЧАНИЕ

Последовательность полей определяется последовательностью их описания в операторе CREATE TABLE, с помощью которого таблица была создана.

Значения, относящиеся к символьным типам и датам, должны быть заключены в апострофы. В списке значений может также использоваться значение NULL.

Рассмотрим пример. Таблица ДОЛЖНОСТИ была создана с использованием следующего оператора:

```
CREATE TABLE Должности (  
    Код_должности INTEGER NOT NULL PRIMARY KEY,  
    Должность VARCHAR(50) NOT NULL UNIQUE,  
    Разряд INTEGER NOT NULL,  
    Зарплата DECIMAL(7,2) NOT NULL)
```

Для добавления новой записи в эту таблицу следует использовать следующий оператор INSERT:

```
INSERT INTO Должности  
VALUES (12, 'Ведущий программист', 12, 2000.00)
```

Ввод данных в отдельные поля таблицы

При добавлении данных в таблицу можно заполнять не все поля, а лишь некоторые из них. В этом случае используется следующая синтаксическая форма оператора INSERT:

```
INSERT INTO имя_таблицы (имя_поля_1, имя_поля_2, ..., имя_поля_N)  
VALUES (значение_1, значение_2, ..., значение_N)
```

Например, при добавлении информации о новом сотруднике в таблицу ФИЗИЧЕСКИЕ ЛИЦА необходимо указать только информацию о полном имени сотрудника. В этом случае можно использовать следующий оператор:

```
INSERT INTO Физические_лица (Код_физического_лица, Фамилия, Имя, Отчество)  
VALUES (234, 'Иванов', 'Федор', 'Михайлович')
```

При выполнении данного оператора во все остальные поля будет занесено значение NULL. Естественно, что поля, которые не указываются в круглых скобках после имени таблицы, не должны иметь ограничения NOT NULL, иначе попытка выполнения оператора INSERT окажется неудачной.

ПРИМЕЧАНИЕ

Список полей в операторе INSERT может иметь произвольный порядок, не зависящий от порядка задания полей при создании таблицы. Однако список значений должен соответствовать порядку, в котором указаны поля, связанные с этими значениями.

Занесение в таблицу данных, содержащихся в другой таблице

Иногда требуется перенести часть информации из одной таблицы в другую. Такого рода операцию можно выполнить с помощью комбинации оператора INSERT с оператором выборки данных SELECT.

ПРИМЕЧАНИЕ

С помощью оператора выборки данных SELECT формируется *запрос* — обращение к базе данных с целью получения определенной информации. Вопросы выборки данных подробно описываются далее, в главе 11, «Выборка данных».

Объединяя операторы INSERT и SELECT, можно добавить в таблицу данные, полученные в результате выполнения запроса из другой таблицы (таблиц). Синтаксис оператора INSERT в этом случае будет иметь следующий вид:

```
INSERT INTO имя_таблицы (имя_поля_1, имя_поля_2, ..., имя_поля_N)
SELECT [* | имя_поля_1, имя_поля_2, ..., имя_поля_N]
FROM имя_таблицы
[WHERE условие]
```

В данном операторе вместо предложения VALUES используется оператор SELECT. Кратко поясним синтаксис этого оператора. После слова SELECT указывается список полей, значения которых включаются в выборку (если после SELECT указать символ *, то в выборку будут включены все поля). Предложение FROM используется для указания имени таблицы, из которой производится выборка данных. Предложение WHERE является необязательным и используется для наложения ограничений на данные, включаемые в выборку.

Количество полей, указываемых в круглых скобках после имени таблицы в операторе INSERT, должно быть равно количеству полей, включаемых в выборку. Соответствие полей определяется порядком их следования: первому полю в списке оператора INSERT соответствует первое поле в списке оператора SELECT и т. д.

Изменение данных, хранящихся в таблице

Для изменения данных, уже занесенных в таблицу, используется оператор UPDATE. Данный оператор не добавляет новых записей в таблицу, а заменяет существующие данные на новые. Оператор UPDATE может быть применен как к одному полю таблицы (наиболее часто используемый случай), так и к нескольким полям. Количество изменяемых записей зависит от потребностей пользователя — с помощью UPDATE можно изменить как одну, так и несколько записей (вплоть до изменения значения всех записей, содержащихся в таблице).

Модификация данных в одном поле таблицы

Для изменения данных только в одном из полей таблицы используется наиболее простая форма оператора UPDATE, имеющая следующий вид:

```
UPDATE имя_таблицы
SET имя_поля = значение
[WHERE условие]
```

Смысл отдельных синтаксических элементов оператора UPDATE достаточно очевиден: после ключевого слова UPDATE указывается имя таблицы, в которой модифицируются данные, после ключевого слова SET выполняется присвоение полю с заданным именем нового значения. Условие, задаваемое с помощью необязательного предложения WHERE, определяет количество записей, которые будут модифицированы.

ПРИМЕЧАНИЕ

Условие, указываемое в предложении WHERE оператора UPDATE, формируется по тем же правилам, что и условие, задаваемое в предложении WHERE оператора SELECT, который будет подробно рассмотрен в главе 11, «Выборка данных».

Рассмотрим пример. Допустим, требуется изменить номер телефона сотрудника организации, хранящийся в таблице ФИЗИЧЕСКИЕ ЛИЦА (такая необходимость может возникнуть либо при смене номера телефона, либо в случае корректировки ошибочно занесенных данных). В этом случае оператор UPDATE должен изменить значение только одного поля и только в одной записи. Поэтому в предложении WHERE необходимо указать такое условие, которое бы выбирало необходимую нам запись. Наиболее простым решением будет использовать для отбора нужной записи поле первичного ключа Код_физического_лица. Значения, хранящиеся в этом поле, уникальны и однозначно определяют сотрудника. Тогда оператор UPDATE, выполняющий изменение номера телефона, будет иметь следующий вид:

```
UPDATE Физические_лица  
SET Телефон = '(095) 2347890'  
WHERE Код_физического_лица = 16
```

Данный оператор изменит значение номера телефона только для записи, соответствующей сотруднику, зарегистрированному в базе данных под номером 16. Если бы мы не задали ограничительного условия в приведенном выше операторе, то значение номера телефона было бы изменено для всех записей в таблице.

ПРИМЕЧАНИЕ

При использовании оператора UPDATE необходимо быть очень внимательным и правильно формулировать ограничительные условия. В противном случае выполнение оператора UPDATE может привести к потере информации, хранящейся в базе данных.

Изменение значений в нескольких полях таблицы

С помощью оператора UPDATE можно одновременно изменять значения в нескольких полях таблицы. Для этого следует указать после ключевого слова SET не одно, а несколько полей:

```
UPDATE имя_таблицы  
SET имя_поля_1 = значение_1,  
    имя_поля_2 = значение_2,  
    ...  
    имя_поля_N = значение_N  
[WHERE условие]
```

Использование оператора в данной форме ничем не отличается от рассмотренного ранее. Здесь точно так же нужно быть очень осторожным при формировании условия.

Удаление данных из таблицы

Удаление данных из таблицы выполняется с помощью оператора DELETE. Данный оператор полностью удаляет всю запись, а не данные из отдельных полей. Синтаксис оператора DELETE имеет следующий вид:

```
DELETE FROM имя_таблицы  
[WHERE условие]
```

Удаляемые записи определяются в соответствии с условием, заданным с помощью необязательного предложения WHERE. При отсутствии предложения WHERE в операторе DELETE данные будут удалены из всей таблицы.

Управление безопасностью базы данных

Одной из важнейших задач управления базами данных является обеспечение безопасности данных, то есть защиты данных от их несанкционированного использования.

ПРИМЕЧАНИЕ

Под несанкционированным использованием обычно понимается доступ к данным со стороны пользователей, не имеющих на это права.

Обеспечение безопасности данных является очень серьезным вопросом, детальное рассмотрение которого требует отдельной объемной книги. Поэтому здесь мы обсудим лишь один из аспектов обеспечения безопасности, а именно — управление доступом к базе данных.

Привилегии пользователей

Привилегиями называются уровни полномочий пользователей. Разграничение доступа к информации, хранящейся в базе данных, регулируется с помощью привилегий.

Различают привилегии двух типов:

- ☐ системные привилегии;
- ☐ объектные привилегии.

Рассмотрим каждый из типов более подробно.

Системные привилегии

Системные привилегии дают пользователям базы данных возможность выполнять действия, связанные с ее администрированием: создавать, удалять и изменять структуру как самой базы данных, так и отдельных ее объектов. Кроме того, системные привилегии дают право на изменение состояния базы данных и ее отдельных объектов.

Возможные системные привилегии существенно зависят от используемой СУБД. Но в любом случае они включают следующие привилегии:

- ☐ создание таблицы;
- ☐ создание представления;
- ☐ создание хранимой процедуры;
- ☐ удаление таблицы;
- ☐ удаление представления;
- ☐ удаление хранимой процедуры.

Этот список может быть расширен. Кроме того, каждая из привилегий имеет свои особенности в различных СУБД.

Объектные привилегии

Объектные привилегии представляют собой уровни полномочий пользователей, распространяемые на объекты базы данных. Это означает, что для того, чтобы выполнять определенные действия над объектами базы данных, пользователь должен иметь соответствующие права.

Стандартом ANSI предусмотрены следующие объектные привилегии:

- ❑ SELECT — разрешает производить выборку данных из указанной таблицы (представления);
- ❑ INSERT(имя_поля) — разрешает выполнять добавление данных в определенное поле указанной таблицы (представления);
- ❑ INSERT — разрешает добавление данных во все поля указанной таблицы (представления);
- ❑ UPDATE(имя_поля) — разрешает модифицировать данные в заданном поле указанной таблицы (представления);
- ❑ UPDATE — разрешает модифицировать данные во всех полях указанной таблицы (представления);
- ❑ REFERENCE(имя_поля) — разрешает ссылаться на заданное поле указанной таблицы (эта привилегия требуется при установке любых ограничений целостности);
- ❑ REFERENCE — позволяет ссылаться на все поля указанной таблицы.

ПРИМЕЧАНИЕ

Кроме указанных, существует целый ряд объектных привилегий, доступных в различных СУБД.

Управление доступом к базе данных

Для управления доступом пользователей к базе данных в языке SQL существуют два оператора:

- ❑ GRANT;
- ❑ REVOKE.

Как правило, эти операторы используются администратором базы данных или его помощником по безопасности.

Оператор GRANT

Оператор GRANT используется для предоставления пользователю как системных, так и объектных привилегий. Синтаксис данного оператора имеет следующий вид:

```
GRANT привилегия_1 [. привилегия_2]  
ON имя_объекта  
TO имя_пользователя [WITH GRANT OPTION]
```

Предоставление пользователю с именем USER права на выбор данных из таблицы СОТРУДНИКИ выполняется с помощью следующего оператора:

```
GRANT SELECT  
ON Сотрудники  
TO USER
```

С помощью одного оператора GRANT можно задавать сразу несколько привилегий. Например, следующий оператор предоставит пользователю USER право как просматривать, так и добавлять данные в таблицу СОТРУДНИКИ:

```
GRANT SELECT, INSERT  
ON Сотрудники  
TO USER
```

При вызове оператора GRANT может использоваться необязательное предложение WITH GRANT OPTION. Данное предложение означает, что пользователь, для которого предоставляются привилегии, также получает право предоставлять привилегии на данный объект. Например, если вызвать рассмотренный выше оператор с предложением WITH GRANT OPTION, то пользователь с именем USER, кроме права просматривать и добавлять данные в таблицу СОТРУДНИКИ, получит также право предоставлять эти привилегии другим пользователям:

```
GRANT SELECT, INSERT  
ON Сотрудники  
TO USER  
WITH GRANT OPTION
```

Оператор REVOKE

Оператор REVOKE используется для отмены предоставленных пользователю привилегий. Данный оператор может вызываться с одним из двух параметров — RESTRICT или CASCADE. При использовании варианта RESTRICT оператор REVOKE будет успешно выполнен только в том случае, если его выполнение не приведет к появлению так называемых *оставленных* привилегий.

ПРИМЕЧАНИЕ

Оставленными называются привилегии, оставшиеся у пользователя, которому они были предоставлены с помощью предложения WITH GRANT OPTION оператора GRANT.

При использовании режима CASCADE удаляются все привилегии, которые могли бы остаться у других пользователей. Это означает, что если пользователю USER1 были предоставлены привилегии с помощью параметра WITH GRANT OPTION, а он, в свою очередь, предоставил эти привилегии пользователю USER2, то отмена привилегий пользователю USER1 в режиме CASCADE приведет к отмене привилегий и для пользователя USER2.

Синтаксис оператора REVOKE имеет следующий вид:

```
REVOKE привилегия_1 [, привилегия_2]  
ON имя_объекта  
FROM имя_пользователя [RESTRICT | CASCADE]
```

Например, для отмены права добавления данных в таблицу СОТРУДНИКИ для пользователя USER следует использовать следующий оператор:

```
REVOKE INSERT  
ON Сотрудники  
FROM USER
```

Глава 6

Проектирование структуры базы данных

В предыдущей главе было рассмотрено создание базы данных и входящих в нее таблиц с помощью SQL-конструкций. Этот подход можно применить при конструировании небольших информационных систем, но создание больших баз данных, содержащих сотни и тысячи таблиц и сложные связи между ними, возможно только при использовании CASE-средств. Вручную очень трудно разработать и графически представить структуру системы, проверить ее на полноту и непротиворечивость, отслеживать версии и выполнять модификации.

В данной главе мы рассмотрим создание концептуальной и физической моделей, а затем их использование для создания и модификации структуры базы данных с помощью современных CASE-средств.

Концептуальное моделирование структуры данных

Широкое распространение реляционных СУБД и их использование в самых разнообразных приложениях показывает, что реляционная модель данных достаточна для моделирования предметных областей. Однако проектирование реляционной базы данных в терминах отношений на основе рассмотренного нами ранее механизма нормализации (см. главу 4 «Реляционные базы данных») часто представляет собой очень сложный и неудобный для проектировщика процесс. Это обусловлено некоторой ограниченностью реляционной модели данных, которая особенно ярко проявляется в следующих аспектах:

- ❑ реляционная модель не предоставляет достаточных средств для представления смысла данных. Проектировщик должен независимым от модели способом представлять семантику реальной предметной области. Примером данного ограничения может служить представление ограничений целостности;
- ❑ в ряде случаев предметную область трудно моделировать на основе плоских таблиц. Сложности могут возникнуть на начальной стадии проектирования

при описании предметной области в виде одной (возможно, даже ненормализованной) таблицы;

- ❑ хотя весь процесс проектирования происходит на основе учета зависимостей, реляционная модель не содержит никаких средств для представления этих зависимостей;
- ❑ несмотря на то что процесс проектирования начинается с выделения некоторых объектов (сущностей) предметной области, существенных для приложения, и выявления связей между этими сущностями, реляционная модель данных не предлагает какого-либо аппарата для разделения сущностей и связей.

Концептуальные модели данных

Для преодоления ограничений реляционной модели и обеспечения потребности проектировщиков баз данных в более удобных и мощных средствах моделирования предметной области проектирование баз данных обычно выполняется не в терминах реляционной модели, а с использованием *концептуальных моделей* предметной области.

Обычно различают концептуальные модели двух видов:

- ❑ *объектно-ориентированные модели*, в которых сущности реального мира представляются в виде объектов, а не записей реляционных таблиц;
- ❑ *семантические модели*, отражающие значения реальных сущностей и отношений.

Объектно-ориентированную модель можно рассматривать как результат объединения семантической модели данных и объектно-ориентированного языка программирования.

Несмотря на то что в последнее время все большее распространение получают объектно-ориентированные модели, не снижается и значение семантических моделей. Концептуальное моделирование баз данных на основе семантических моделей поддерживается во всех известных CASE-средствах (например, таких как ERWin и Power Designer). Кроме того, семантические модели более просты для понимания, особенно при проектировании сравнительно небольших баз данных.

Как и реляционная модель, любая развитая семантическая модель данных включает структурную, манипуляционную и целостную части. Главным назначением семантических моделей является обеспечение возможности выражения семантики данных.

Цель *семантического моделирования* — обеспечение наиболее естественных для человека способов сбора и представления той информации, которую предполагается хранить в создаваемой базе данных. Поэтому семантическую модель данных пытаются строить по аналогии с естественным языком (последний не может быть использован в чистом виде из-за сложности компьютерной обработки текстов и неоднозначности любого естественного языка). Основными конструктивными элементами семантических моделей являются сущности, связи между ними и их свойства (атрибуты).

Модель «сущность–связь»

Одной из наиболее популярных семантических моделей данных является модель «сущность–связь» (часто называемая также ER-моделью — по первым буквам английских слов Entity (сущность) и Relation (связь)).

На использовании разновидностей ER-модели основано большинство современных подходов к проектированию баз данных (главным образом, реляционных). Модель была предложена Питером Ченом в 1976 году. Моделирование предметной области базируется на использовании графических диаграмм, включающих небольшое число разнородных компонентов. В связи с наглядностью представления концептуальных схем баз данных ER-модели получили широкое распространение в CASE-средствах, предназначенных для автоматизированного проектирования реляционных баз данных.

Для моделирования структуры данных используются ER-диаграммы (диаграммы «сущность–связь»), которые в наглядной форме представляют связи между сущностями. В соответствии с этим ER-диаграммы получили распространение в CASE-системах, поддерживающих автоматизированное проектирование реляционных баз данных. Наиболее распространенными являются диаграммы, выполненные в соответствии со стандартом IDEF1X, который используют наиболее популярные CASE-системы (в частности ERwin, Design/ IDEF, Power Designer, DBDesigner).

Основными понятиями ER-диаграммы являются *сущность*, *связь* и *атрибут*.

Сущность

Сущность — это реальный или виртуальный объект, имеющий существенное значение для рассматриваемой предметной области, информация о котором подлежит хранению. Если не вдаваться в подробности, то можно считать, что сущности соответствуют таблицам реляционной модели. Каждая сущность должна обладать следующими свойствами:

- ☐ иметь уникальный идентификатор;
- ☐ содержать один или несколько атрибутов, которые либо принадлежат сущности, либо наследуются через связь с другими сущностями;
- ☐ содержать совокупность атрибутов, однозначно идентифицирующих каждый экземпляр сущности.

Любая сущность может иметь произвольное количество связей с другими сущностями.

В диаграммах ER-модели сущность представляется в виде прямоугольника, содержащего имя сущности (рис. 6.1).

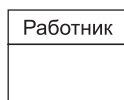


Рис. 6.1. Отображение сущности

Связь

Связь — это соединение двух сущностей, при котором, как правило, каждый экземпляр одной сущности, называемой родительской сущностью, ассоциирован с произвольным (в том числе нулевым) количеством экземпляров второй сущности, называемой сущностью-потомком, а каждый экземпляр сущности-потомка ассоциирован в точности с одним экземпляром сущности-родителя.

Связь представляется в виде линии, связывающей две сущности или идущей от сущности к ней же самой (рис. 6.2). Для каждой связи между сущностями указываются правила, обеспечивающие ее поддержание.



Рис. 6.2. Связь между двумя сущностями

Атрибут

Атрибут является характеристикой сущности, значимой для рассматриваемой предметной области. В ER-диаграммах список атрибутов сущности отображается в виде строк внутри прямоугольника с изображением сущности (рис. 6.3). В реляционных базах данных аналогом атрибута является поле таблицы.

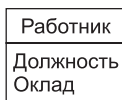


Рис. 6.3. Атрибуты сущности

Создание физической модели

Концептуальная модель позволяет понять суть создаваемой информационной системы, но она не подходит для создания непосредственно структуры базы данных. Для генерации структуры базы данных необходимо преобразовать концептуальную базу данных в физическую.

Рассмотрим общие принципы преобразования:

- ☐ каждая сущность преобразуется в таблицу. Имя сущности становится именем таблицы;
- ☐ каждый атрибут становится столбцом таблицы (полем) с тем же именем, уточняется тип данных, выбирается более точный формат;
- ☐ идентифицирующие атрибуты сущности превращаются в первичный ключ таблицы (ключевое поле). Если для данной сущности имеются зависимые связи, к числу столбцов первичного ключа добавляется копия уникального идентификатора сущности, находящегося на другом конце связи (этот процесс может продолжаться рекурсивно);

- связи многие-к-одному и один-к-одному становятся внешними ключами. Для них создается копия уникального идентификатора с одиночного конца связи, и соответствующие столбцы составляют внешний ключ;
- для первичного ключа (уникальный индекс) и внешних ключей создаются индексы;
- для связей многие-ко-многим создается таблица, столбцами которой являются уникальные идентификаторы связываемых сущностей (они составляют первичный ключ).

Общие сведения о CASE-средствах

За последнее десятилетие в области технических средств программирования сформировалось новое направление — *CASE-технология* (Computer-Aided Software/System Engineering). *CASE-технология* представляет собой совокупность методологий анализа, проектирования, разработки и сопровождения сложных систем и поддерживается комплексом взаимосвязанных средств автоматизации.

При использовании методологий структурного анализа появился ряд ограничений (сложность понимания, большая трудоемкость и стоимость использования, неудобство внесения изменений в проектные спецификации и т. д.). CASE-технологии с самого начала развивались именно с целью преодоления этих ограничений путем автоматизации процессов анализа и интеграции поддерживающих средств.

Основные возможности CASE-средств

Современные CASE-средства охватывают обширную область поддержки многочисленных технологий проектирования информационных систем: от простых средств анализа и документирования до полномасштабных средств автоматизации, покрывающих весь жизненный цикл программного обеспечения.

Наиболее трудоемкими этапами разработки информационной системы являются этапы анализа и проектирования, в процессе которых CASE-средства обеспечивают качество принимаемых технических решений и подготовку проектной документации. При этом большую роль играют методы визуального представления информации. Это предполагает построение структурных или иных диаграмм в реальном масштабе времени, использование многообразной цветовой палитры, сквозную проверку синтаксических правил. Графические средства моделирования предметной области позволяют разработчикам в наглядном виде изучать существующую информационную систему, перестраивать ее в соответствии с поставленными целями и имеющимися ограничениями.

В наиболее полном виде CASE-средства обладают следующими характеристиками:

- *единый графический язык*. CASE-технологии обеспечивают всех участников проекта, включая заказчиков, единым строгим, наглядным и интуитивно понятным графическим языком, позволяющим получать обозримые компоненты

с простой и ясной структурой. При этом программы представляются двумерными схемами (которые проще в использовании, чем многостраничные описания), позволяющими заказчику участвовать в процессе разработки, а разработчикам — общаться с экспертами предметной области, разделять деятельность системных аналитиков, проектировщиков и программистов, облегчая им защиту проекта перед руководством, а также обеспечивая легкость сопровождения и внесения изменений в систему;

- *единая база данных проекта.* Основа CASE-технологии — использование базы данных проекта (репозитория) для хранения всей информации о проекте, которая может совместно использоваться разработчиками в соответствии с их правами доступа. Содержимое репозитория включает не только информационные объекты различных типов, но и отношения между их компонентами, а также правила использования или обработки этих компонентов. Репозиторий может хранить объекты различных типов: структурные диаграммы, определения экранов и меню, проекты отчетов, описания данных, логику обработки, модели данных, их организации и обработки, исходные коды, элементы данных и т. п.;
- *интеграция средств.* На основе репозитория осуществляются интеграция CASE-средств и разделение системной информации между разработчиками. При этом возможности репозитория обеспечивают несколько уровней интеграции: общий пользовательский интерфейс по всем средствам, передачу данных между средствами, интеграцию этапов разработки через единую систему представления фаз жизненного цикла, передачу данных и средств между различными платформами;
- *поддержка коллективной разработки и управления проектом.* CASE-технология поддерживает групповую работу над проектом, обеспечивая возможность работы в сети, экспорт-импорт любых фрагментов проекта для их развития и/или модификации, а также планирование, контроль, руководство и взаимодействие, то есть функции, необходимые в процессе разработки и сопровождения проектов. Эти функции также реализуются на основе репозитория. В частности, через репозиторий могут осуществляться контроль безопасности (ограничения и привилегии доступа), контроль версий и изменений и т. п.;
- *макетирование.* CASE-технология дает возможность быстро строить макеты (прототипы) будущей системы, что позволяет заказчику на ранних этапах разработки оценить, насколько она устраивает его и приемлема для будущих пользователей;
- *генерация документации.* Вся документация по проекту генерируется автоматически на базе репозитория (как правило, в соответствии с требованиями действующих стандартов). Несомненное достоинство CASE-технологии заключается в том, что документация всегда отвечает текущему состоянию дел, поскольку любые изменения в проекте автоматически отражаются в репозитории (известно, что при традиционных подходах к разработке программного обеспечения документация в лучшем случае запаздывает, а ряд модификаций вообще не находит в ней отражения);

- ❑ *верификация проекта.* CASE-технология обеспечивает автоматическую верификацию и контроль проекта на полноту и состоятельность на ранних этапах разработки, что влияет на успех разработки в целом;
- ❑ *автоматическая генерация программного кода.* Генерация программного кода осуществляется на основе репозитория и позволяет автоматически построить до 85–90 % текстов на языках высокого уровня;
- ❑ *сопровождение и реинжиниринг.* Сопровождение системы в рамках CASE-технологии характеризуется сопровождением проекта, а не программных кодов. Средства реинжиниринга и обратного инжиниринга позволяют создавать модель системы из ее кодов и интегрировать полученные модели в проект, автоматически обновлять документацию при изменении кодов, автоматически изменять спецификации при редактировании кодов и т. п.

Далеко не все CASE-средства поддерживают все указанные выше возможности. Поэтому обычно к CASE-средствам относят любой программный продукт, автоматизирующий ту или иную совокупность процессов жизненного цикла программного обеспечения и обладающий следующими основными характеристиками:

- ❑ наличие мощных графических средств для описания и документирования информационной системы, обеспечивающих удобный интерфейс с разработчиком и развивающих его творческие возможности;
- ❑ интеграция отдельных компонентов CASE-средств, обеспечивающая управляемость процесса разработки информационной системы;
- ❑ использование специальным образом организованного хранилища проектных метаданных (репозитория).

Создание модели информационной системы

Рассмотрим создание модели информационной системы на примере базы данных Премьер. В качестве CASE-средства будем использовать свободно распространяемую систему моделирования данных — DBDesigner 4 фирмы fabFORCE. Данную систему отличает оптимальное сочетание простоты и функциональности. С ее помощью можно не только строить модели данных и их документировать, но и анализировать структуры существующих баз данных и вносить в них исправления, синхронизируя измененные модели с действующими базами данных.

База данных «Премьер»

Строительная компания «Премьер» возводит различные здания. Для всех зданий требуются разнообразные материалы в различных количествах. На разных этапах проекта работают разные бригады. Например, есть бригады арматурщиков, каменщиков, штукатуров и т. д. Составляя график работ, фирма «Премьер» варьирует состав бригад. Рабочие назначаются в разные бригады в соответствии с квалификацией. Один и тот же рабочий может выполнять разные работы

(например, работать как плотником, так и каменщиком), поэтому его могут включать в разные бригады.

Численность бригады меняется в зависимости от размера здания и предъявляемых к нему требований. Следовательно, бригады составляются исходя из требований конкретного здания. Кроме того, для каждой бригады, работающей на строительстве данного здания, назначается бригадир. Рабочий может возглавлять одну бригаду и работать в другой простым рабочим.

В базе данных должна содержаться информация о том, кто из рабочих фирмы в какую бригаду назначен на разных зданиях, какие материалы используются при возведении разных зданий, а также график работ по каждому зданию.

На рис. 6.4, а представлено отношение между зданиями и материалами. Объектное множество ЗДАНИЕ содержит элементы, соответствующие зданиям. Объектное множество ТИП МАТЕРИАЛА представляет типы материалов. Для каждого здания требуется несколько типов материалов, и каждый тип материалов используется в нескольких зданиях. Обратите внимание, что атрибут Адрес относится только к множеству ЗДАНИЕ. Атрибут Адрес является уникальным для конкретного здания и может использоваться в качестве ключа для отношения ЗДАНИЕ.

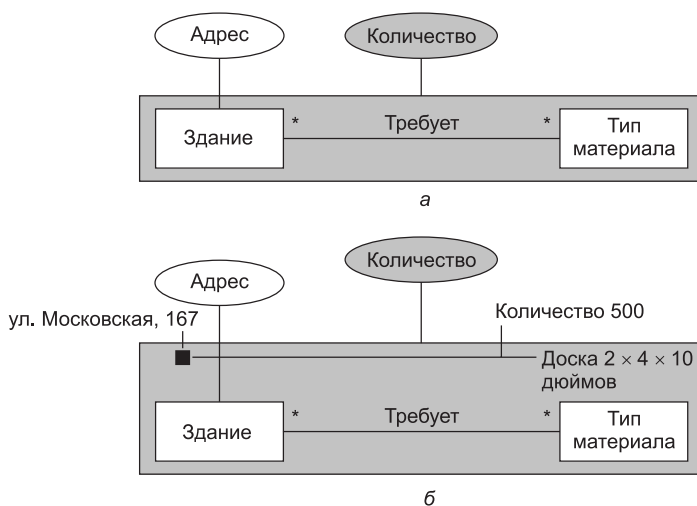


Рис. 6.4. Моделирование отношений между зданиями и материалами: а) отношение между зданиями и материалами; б) количество материала данного типа, использованного в здании

ПРИМЕЧАНИЕ

Важно отметить, что в этом примере объектное множество ТИП МАТЕРИАЛА представляет собой скорее концептуальный, чем физический объект. Это означает, что каждый элемент множества ТИП МАТЕРИАЛА обозначает именно тип, а не физический «кусоч» материала. Такое понятие концептуального объекта, противопоставляемого физическому объекту, часто применяется при концептуальном моделировании данных. В некоторых случаях требуется моделировать отдельные объектные множества для физических объектов.

Теперь мы покажем, как отразить формирование бригад и назначение рабочих и бригадиров. На рис. 6.5 представлено отношение между объектными множествами ТИП БРИГАДЫ и ЗДАНИЕ. ТИП БРИГАДЫ — еще один пример концептуального объектного множества; то есть элементы множества ТИП БРИГАДЫ соответствуют не *конкретным* бригадам, а *типам* бригад, таким как бригада арматурщиков или бригада каменщиков. Отношение между зданием и типом бригады представляет конкретную бригаду — бригаду, назначенную выполнять на данном здании данный тип работ. Таким образом, мы можем рассматривать это отношение как объект, назвав его БРИГАДА.

Для каждой бригады как элемента объектного множества БРИГАДА выбирают дни работы. Например, бригаде штукатуров требуется несколько дней для того, чтобы оштукатурить данное здание. Таким образом, у нас есть отношение много-ко-многим, РАБОТАЕТ-ТОГДА-ТО, между объектами БРИГАДА и ДАТА.

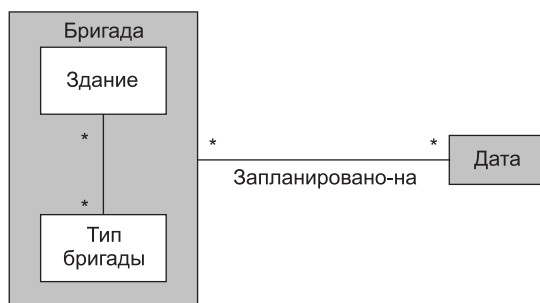


Рис. 6.5. Модель формирования бригад

На рис. 6.6 представлено распределение рабочих и бригадиров по бригадам. Обратите внимание, что отношение ЯВЛЯЕТСЯ-БРИГАДИРОМ имеет мощность один-ко-многим. Это связано с тем, что у бригады может быть только один бригадир, но при этом один и тот же человек может возглавлять несколько бригад.



Рис. 6.6. Назначение рабочих в бригады

На рис. 6.7 представлена объединенная диаграмма, представляющая полную модель данных для строительной компании «Премьер».

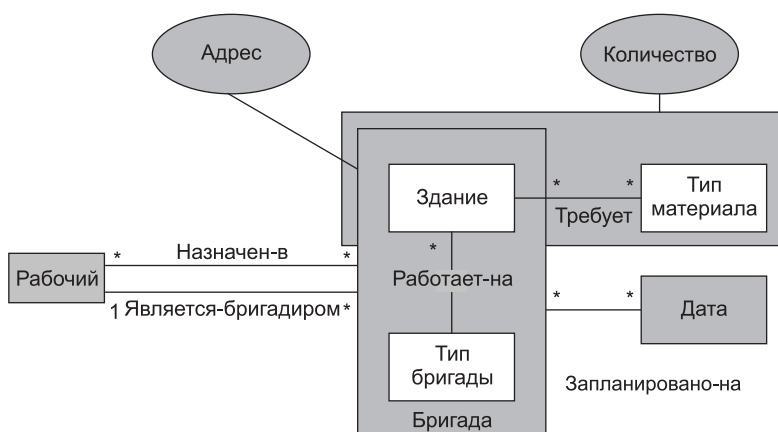


Рис. 6.7. Модель данных для строительной компании «Премьер»

Создание новой модели базы данных в DBDesigner

Для создания физической модели базы данных запустите программу DBDesigner и выберите из меню File команду New. Откроется основное окно программы, которое содержит область отображения модели, меню, панель инструментов и несколько окон для быстрого доступа к часто используемым опциям (рис. 6.8).

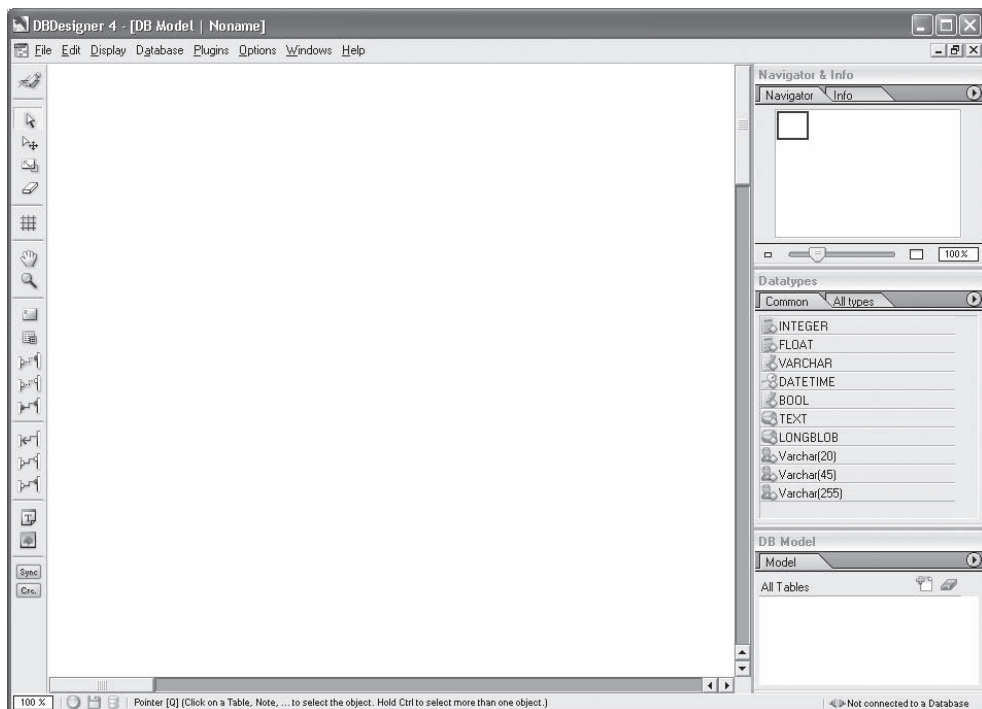


Рис. 6.8. Основное окно программы DBDesigner

Любая создаваемая в DBDesigner модель содержит два основных компонента: таблицы и связи. Для таблиц задаются имя, поля, индексы и различные опции. Связи устанавливают определенные отношения между таблицами. Кроме этого, вы можете использовать такие объекты, как примечания, рисунки и зоны, которые помогут вам более наглядно представить структуру модели.

Прежде всего определим свойства создаваемой модели, которые используются для ее идентификации, описания и отображения в отчетах по модели. Для этого выполните команду **Options ► Model Options**. Откроется окно диалога **Model Options** (рис. 6.9). Задайте в нем наименование модели, ее версию и введите описание.

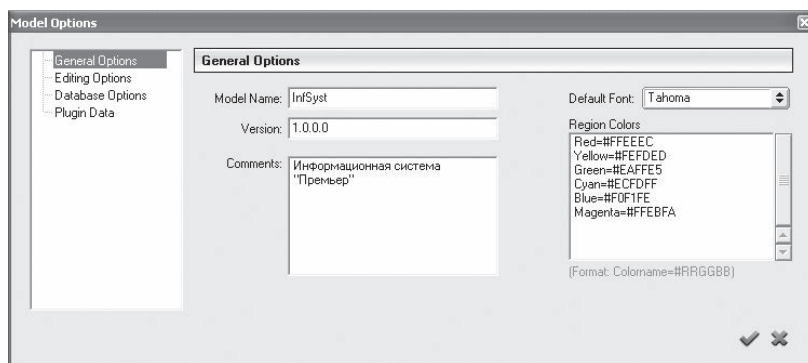


Рис. 6.9. Определение свойств модели

Перейдите на вкладку **Editing Options** и выберите строку **InnoDB** в разделе типов таблиц по умолчанию (**Default Table Type**). Данный тип таблиц предназначен для получения максимальной производительности при обработке больших объемов данных. По эффективности использования процессора этот тип намного превосходит другие типы таблиц реляционных баз данных.

Создание таблицы

Для создания таблицы выберите на панели инструментов значок с изображением прямоугольника **New Table** или воспользуйтесь сочетанием клавиш **[Ctrl+T]**. Щелчком левой кнопки мыши поместите таблицу в область модели. Создастся прямоугольник для новой таблицы, который пока содержит только наименование **Table_01** (рис. 6.10).

Для определения свойств таблицы сделайте двойной щелчок на изображении прямоугольника. Откроется окно диалога **Table Editor** (рис. 6.11).

Данное окно редактора таблиц позволяет задать имя таблицы, ее свойства, перечислить поля таблицы, задать их тип, определить индексы и установить некоторые другие опции.

Начнем создание модели информационной системы с описания таблицы **WORKER** (работник). Введите имя таблицы и нажмите клавишу **[Enter]**. Вы перейдете к области ввода названий полей таблицы. По умолчанию первым создается ключевое поле. Введите его название **WORKER_ID** (идентификатор работника). Поле **WORKER_ID** является идентифицирующим, так как однозначно определяет работника. Обратите внимание, что при этом автоматически устанавливается

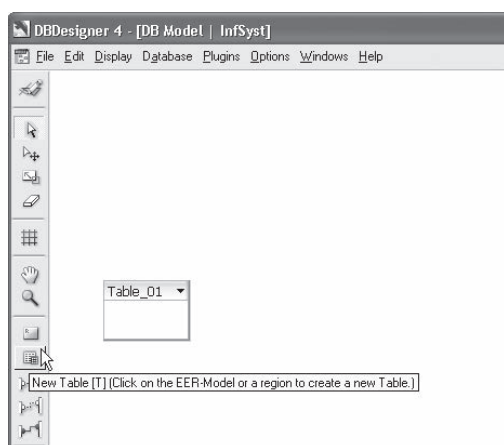


Рис. 6.10. Создание новой таблицы

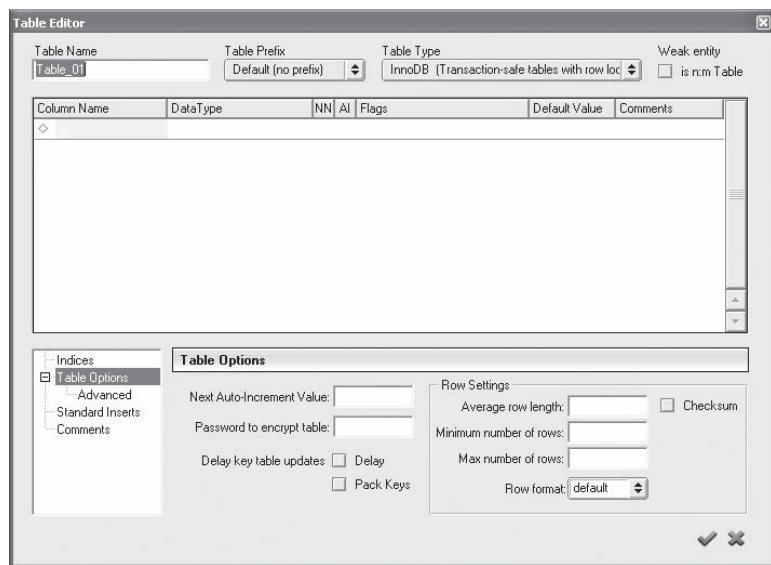


Рис. 6.11. Окно редактора таблиц

флаг запрета пустых значений в строке NN (Not Null). Флаг в строке AI (Auto Increment) означает автоматическое увеличение значения в этом поле при переходе к следующей записи.

Задайте еще два поля таблицы: `WORKER_NAME` (имя работника) и `HRLY_RATE` (почасовая ставка). Тип данных при этом будет определяться заданным по умолчанию. Чтобы изменить тип данных, либо дважды щелкните левой кнопкой мыши на типе данных поля и выберите необходимый, либо перетащите его мышью из расположенного справа окна `Datatypes`. Для поля `WORKER_NAME` выберите символьный тип данных `Varchar` длиной 50. Поле `HRLY_RATE` будет содержать денежные данные, поэтому задайте для него тип `Decimal(6,2)`, где первая цифра 6 представляет собой общее количество значащих десятичных знаков, а цифра 2 задает количество десятичных знаков после запятой.

ПРИМЕЧАНИЕ

Двойной щелчок мыши на одном из типов данных из окна Datatypes вызовет окно редактора типов данных, в котором вы не только сможете посмотреть описание типа, но и изменить его.

Заполненное окно редактора таблицы WORKER представлено на рис. 6.12.

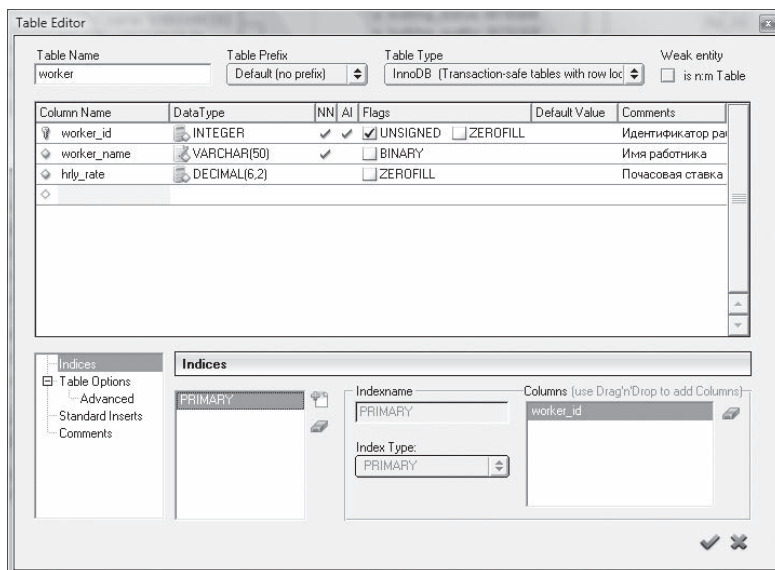


Рис. 6.12. Окно редактора таблицы WORKER

Закройте окно редактора таблицы. В области модели появится прямоугольник, соответствующий таблице WORKER (рис. 6.13). В прямоугольнике представлены наименования трех полей таблицы и указаны типы их данных. Ключевое поле WORKER_ID помечено символом ключа слева от названия.

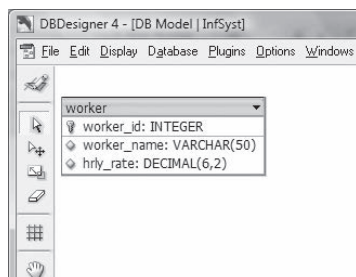


Рис. 6.13. Таблица WORKER

Аналогичным способом опишите структуры остальных таблиц: SKILL (навык), BUILDING (здание), ASSIGNMENT (назначение), MATERIAL (материал). Для отображения факта о возможности выполнения одним работником нескольких работ создайте таблицу WORKER_SKILL без полей. Она будет служить для связи таблиц WORKER и SKILL и содержать идентификаторы работников и соответствующих им навыков. Чтобы показать состав различных бригад в зависимости от их

назначения, создайте таблицу TEAM_WORKER. Она будет содержать идентификаторы назначений и работников. Кроме этого, необходимо создать таблицу MATERIAL_QUANTITY для отображения информации о необходимом количестве материала каждого типа в зависимости от строящегося здания. В результате у вас должно получиться так, как показано на рис. 6.14.

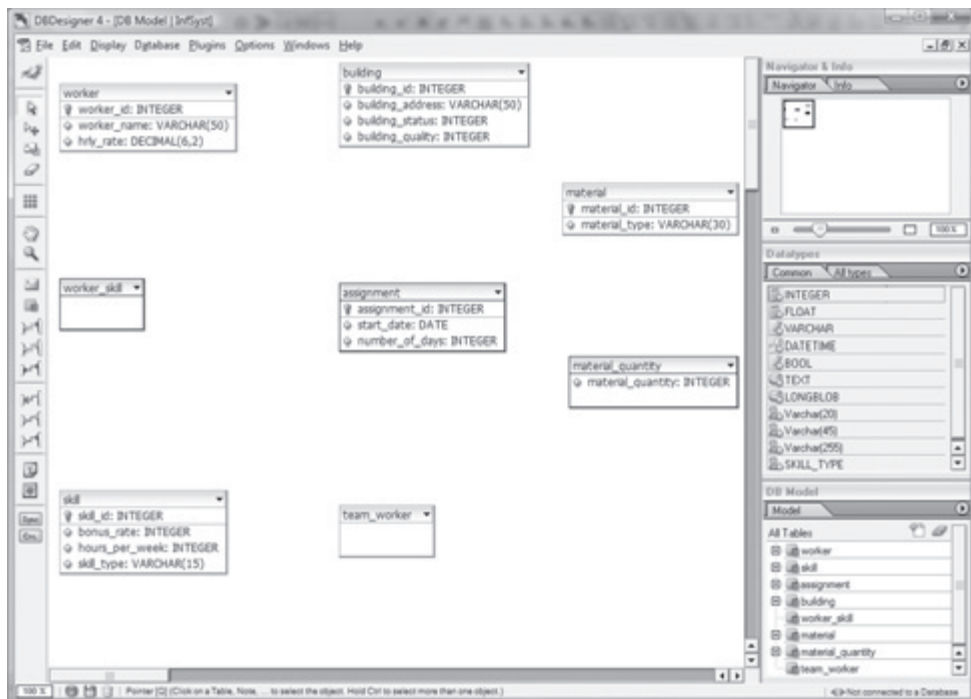


Рис. 6.14. Создание таблиц модели

ПРИМЕЧАНИЕ

Обратите внимание, что в окне DB Model (рис. 6.15) отражаются все созданные таблицы, их поля и связи. Двойной щелчок мыши на таблице вызовет окно редактора этой таблицы.

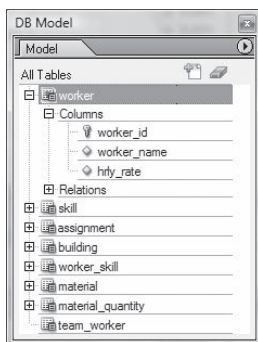


Рис. 6.15. Окно DB Model

Определение связей между таблицами

Для создания связи между двумя таблицами выполните следующие действия.

1. Выберите на панели инструментов кнопку **New 1:n Relation**, на которой показаны два прямоугольника, соединенные линией (рис. 6.16).

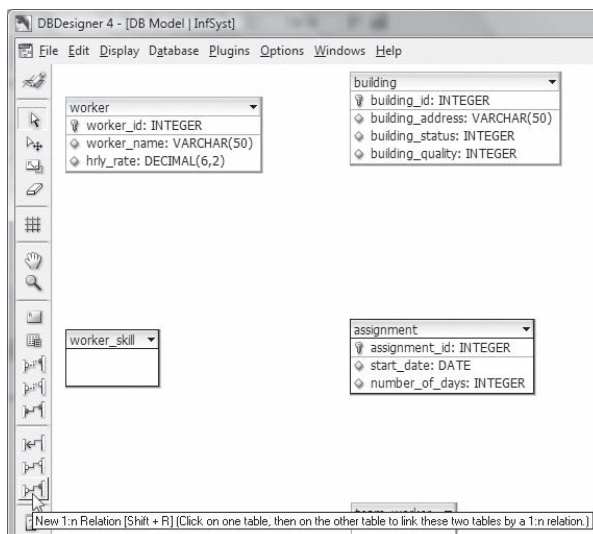


Рис. 6.16. Выбор инструмента для создания связи один-ко-многим

2. Щелкните левой кнопкой мыши на таблице **WORKER**.
3. Затем щелкните на таблице **WORKER_SKILL**.

В модели возникнет связь между выбранными таблицами, которой по умолчанию присваивается имя **Rel_n**, где **n** — порядковый номер создаваемой связи (рис. 6.17). Обратите внимание, что в таблице **WORKER_SKILL** появилось ключевое

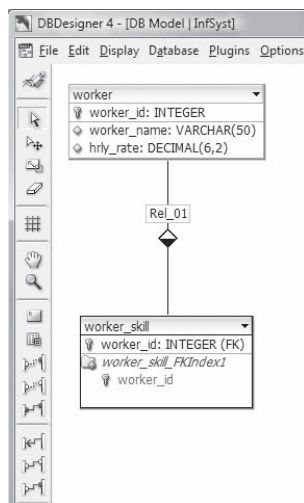


Рис. 6.17. Создание связи типа один-ко-многим между таблицами **WORKER** и **WORKER_SKILL**

поле `WORKER_ID`, отмеченное значком внешнего ключа FK справа от его типа данных. Можно сказать, что внешний ключ `WORKER_ID` дочерней таблицы `WORKER_SKILL` фактически служит ссылкой на потенциальный ключ `WORKER_ID` родительской таблицы `WORKER`. Отсюда вытекает правило ссылочной целостности: для каждого значения внешнего ключа должно существовать соответствующее значение потенциального ключа в родительской таблице.

Для определения свойств созданной связи сделайте на ней двойной щелчок мышью. Откроется окно редактора связи **Relation Editor** (рис. 6.18). Данное окно диалога позволяет переименовать связь, изменить тип связи, имя внешнего ключа дочерней таблицы, установить механизмы поддержания ссылочной целостности между таблицами.

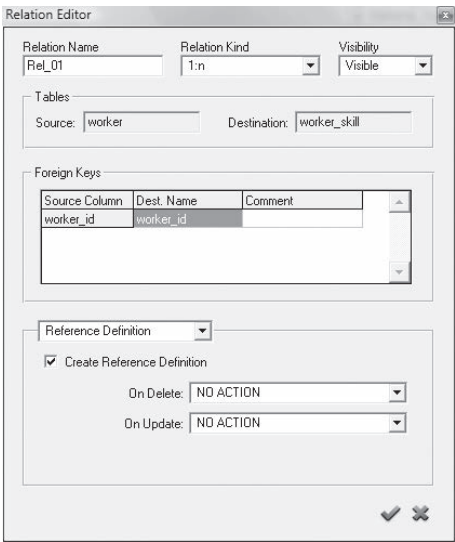



Рис. 6.18. Определение свойств связи

Рассмотрим подробнее типы связи между таблицами, реализуемые в DB-Designer. Откройте список **Relation Kind** в окне редактора связи. Ниже в таблице приведено описание всех представленных в списке типов связи и показаны соответствующие им кнопки на панели инструментов.

Тип связи	Описание	Кнопка
1:1	Связь один-к-одному. Одна запись родительской таблицы соответствует одной записи дочерней таблицы	
1:n	Связь один-ко-многим. Одна запись родительской таблицы соответствует нескольким записям дочерней таблицы	
1:n (Non Identifying)	Одна запись родительской таблицы соответствует нескольким записям дочерней таблицы, но внешний ключ не является компонентом потенциального ключа дочерней таблицы	
n:m	Связь много-ко-многим реализуется посредством использования двух связей один-ко-многим	
1:1 (Descendent Obj.)	Связь один-к-одному. Имеет собственное графическое отображение	

Тип связи	Описание	Кнопка
1:1 (Non Identifying)	Одна запись родительской таблицы соответствует одной записи дочерней таблицы, но внешний ключ не является компонентом потенциального ключа дочерней таблицы	

Мы определили связь между таблицами WORKER и WORKER_SKILL как связь один-ко-многим, так как один и тот же работник может иметь несколько специальностей и соответственно выполнять разные работы при возведении зданий. Пусть, к примеру, работник с идентификационным номером 13 может работать и столяром и плотником. Что произойдет в информационной системе, если такой ценный работник уйдет из строительной компании «Премьер»? Зададим вопрос точнее. Как отразится факт удаления записи о работнике под номером 13 из таблицы WORKER на данных таблицы WORKER_SKILL? Давайте разберемся. При удалении записи из родительской таблицы удаляется значение потенциального ключа. Значит, значения внешних ключей двух записей дочерней таблицы WORKER_SKILL, ссылающиеся на удаленный потенциальный ключ таблицы WORKER, станут некорректными. Удаление записей из родительской таблицы может привести к нарушению ссылочной целостности. DBDesigner предлагает пять основных механизмов поддержания ссылочной целостности на случай удаления или обновления значений полей. Их можно увидеть, раскрыв в разделе Reference Definition окна редактора связи Relation Editor список On Delete или On Update. Ниже в таблице приведены их описания.

Механизм	Описание
Restrict (Ограничить)	Не разрешается выполнение операций, приводящих к нарушению ссылочной целостности
Cascade (Каскадировать)	Разрешается выполнение требуемых операций, но вносятся каскадные изменения в подчиненные таблицы так, чтобы не допустить нарушения ссылочной целостности
Set Null (Установить Null)	Разрешается выполнение требуемых операций, но все некорректные значения внешних ключей изменяются на NULL-значения
No Action (Бездействовать)	Операции выполняются, не обращая внимания на нарушения ссылочной целостности
Set Default (Установить по умолчанию)	Разрешается выполнение требуемых операций, но все некорректные значения внешних ключей изменяются на некоторое значение, принятое по умолчанию

Если, к примеру, мы выберем в списке On Delete строку Cascade, то удаление записи о работнике с идентификационным номером 13 из таблицы WORKER приведет к каскадному удалению всех записей таблицы WORKER_SKILL, имеющих в поле WORKER_ID значение 13.

Свяжите аналогичным способом таблицы SKILL и WORKER_SKILL, WORKER и TEAM_WORKER, ASSIGNMENT и TEAM_WORKER, BUILDING и MATERIAL_QUANTITY, MATERIAL и MATERIAL_QUANTITY. Обратите внимание, что во всех дочерних таблицах WORKER_SKILL, TEAM_WORKER и MATERIAL_QUANTITY появились внешние ключи, являющиеся ссылками на потенциальные ключи соответствующих родительских таблиц. Причем каждый внешний ключ входит в состав потенциального ключа своей таблицы.

Теперь выберите на панели инструментов кнопку New 1:n Non-Identifying-Relation. Свяжите таблицы WORKER и ASSIGNMENT для того, чтобы иметь возмож-

ность выбирать бригадира в зависимости от назначения. Переименуйте внешний ключ таблицы ASSIGNMENT из WORKER_ID в SUPV_ID (идентификатор бригадира), используя окно редактора связи (рис. 6.19). Обратите внимание, что при таком типе связи внешний ключ уже не является компонентом потенциального ключа.

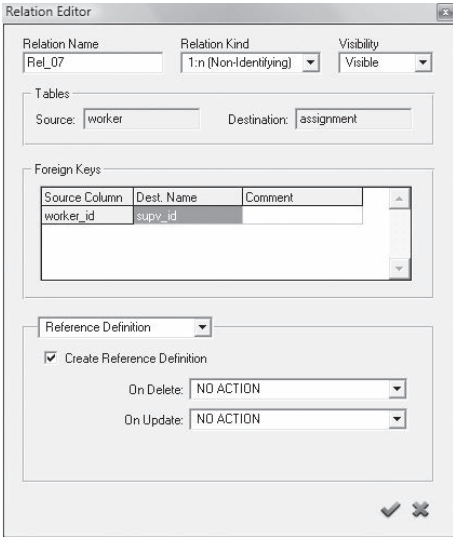


Рис. 6.19. Создание связи между таблицами WORKER и ASSIGNMENT

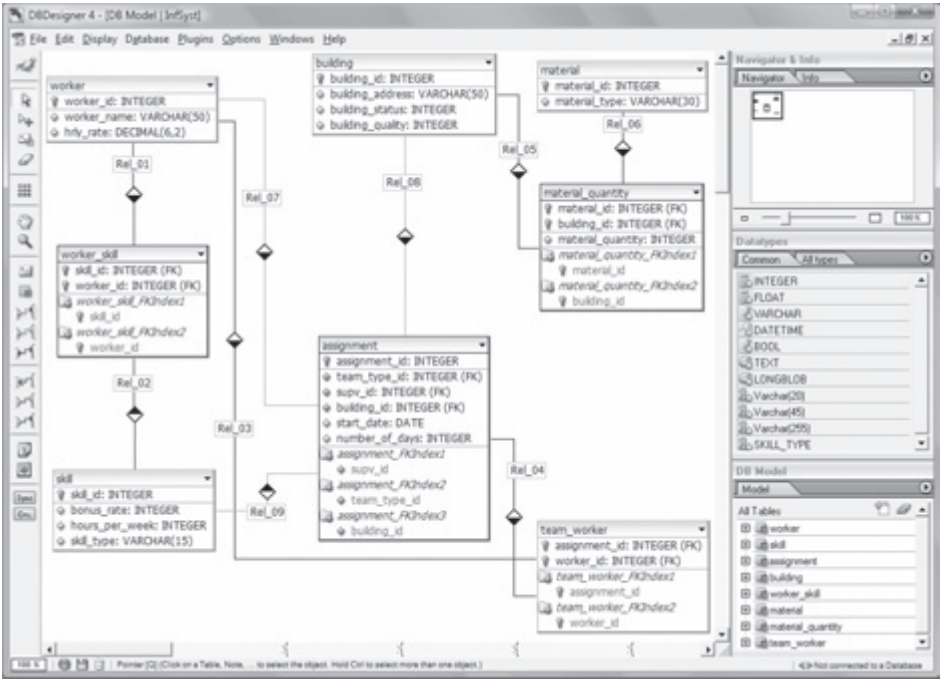


Рис. 6.20. Модель информационной системы «Премьер»

Подобную связь установите между таблицами BUILDING и ASSIGNMENT, SKILL и ASSIGNMENT. В окне редактора последней связи переименуйте внешний ключ таблицы ASSIGNMENT из SKILL_ID в TEAM_TYPE_ID (идентификатор типа бригады).

На рис. 6.20 представлен результат создания модели информационной системы «Премьер» в системе DBDesigner.

Документирование модели базы данных

CASE-средства содержат прекрасные возможности для создания описания модели базы данных. Во-первых, вы можете распечатать модель в графическом виде. Для этого в DBDesigner необходимо выполнить команду File ► Print и указать в появившемся окне диалога печатаемые страницы, число копий и при необходимости изменить некоторые параметры страницы.

Вы также можете сохранить модель базы данных в виде изображения. Выполнив команду File ► Export ► Export Model as Image, можно получить PNG- или BMP-файл с изображением всей модели данных в таком же виде, в каком вы ее видите на рабочем поле DBDesigner.

К примеру, для быстрой публикации в сети можно представить модель в виде HTML-документа. В DBDesigner для данной цели используется плагин HTML Report. Выполните команду Plugins ► HTML Report. Откроется окно диалога HTML Report Plugin (рис. 6.21). Укажите в нем список таблиц, условие сортировки и некоторые другие опции и, щелкнув на кнопке Execute, получите довольно подробный документ (рис. 6.22), в котором можно увидеть список таблиц,

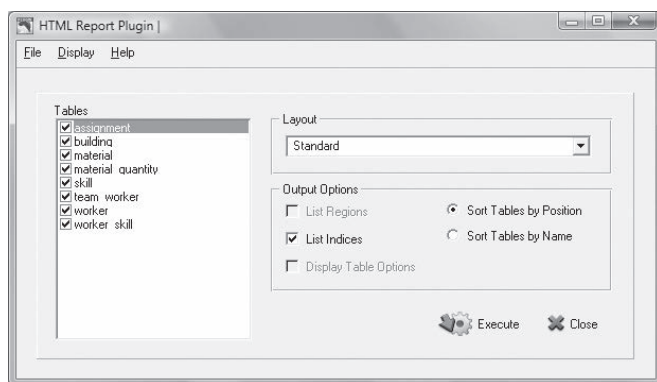


Рис. 6.21. Окно диалога HTML Report Plugin

assignment							
ColumnName	DataType	PrimaryKey	NotNull	Flags	Default Value	Comment	AutoInc
assignment_id	INTEGER	PK	NN	UNSIGNED			A1
team_type_id	INTEGER		NN	UNSIGNED			
supv_id	INTEGER		NN	UNSIGNED			
building_id	INTEGER		NN	UNSIGNED			
start_date	DATE		NN				
number_of_days	INTEGER			UNSIGNED			
IndexName	IndexType		Columns				
PRIMARY	PRIMARY		assignment_id				
assignment_FKIndex1	Index		supv_id				
assignment_FKIndex2	Index		team_type_id				
assignment_FKIndex3	Index		building_id				

Рис. 6.22. Фрагмент HTML-документа

их функциональное назначение, состав полей и индексов для каждой таблицы, а также комментарии к ней.

В случае когда модель базы данных нужно открыть в другой программе, например в MS Access, можно сохранить модель в MDB- или XML-формате командой File ► Export ► Export Model As MDB XML File.

Создание структуры базы данных

После описания физической модели базы данных вы можете создать SQL-скрипт с помощью команды File ► Export ► SQL Create Script, который затем запустить на выполнение средствами сервера базы данных. В открывшемся окне диалога SQL Export Script установите флажки создания таблиц, индексов, комментариев и т. п. (рис. 6.23).

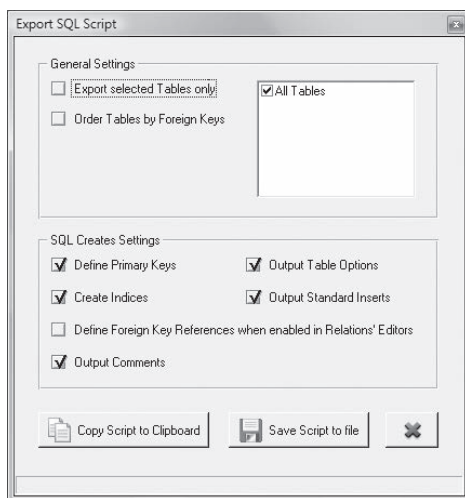


Рис. 6.23. Определение параметров создания SQL-скрипта

Щелчком на кнопке **Save Script to file** запишите скрипт в файл. Ниже приведен текст скрипта создания таблицы ASSIGNMENT.

```
CREATE TABLE assignment (  
assignment_id INTEGER UNSIGNED NOT NULL AUTO_INCREMENT,  
team_type_id INTEGER UNSIGNED NOT NULL,  
supv_id INTEGER UNSIGNED NOT NULL,  
building_id INTEGER UNSIGNED NOT NULL,  
start_date DATE NOT NULL,  
number_of_days INTEGER UNSIGNED NULL,  
PRIMARY KEY (assignment_id),  
INDEX assignment_FKIndex1 (supv_id),  
INDEX assignment_FKIndex2 (team_type_id),  
INDEX assignment_FKIndex3 (building_id)  
)  
TYPE=InnoDB;
```

Подобного результата можно добиться, используя контекстное меню таблицы. Щелкните правой кнопкой мыши на заголовке таблицы и выберите команду

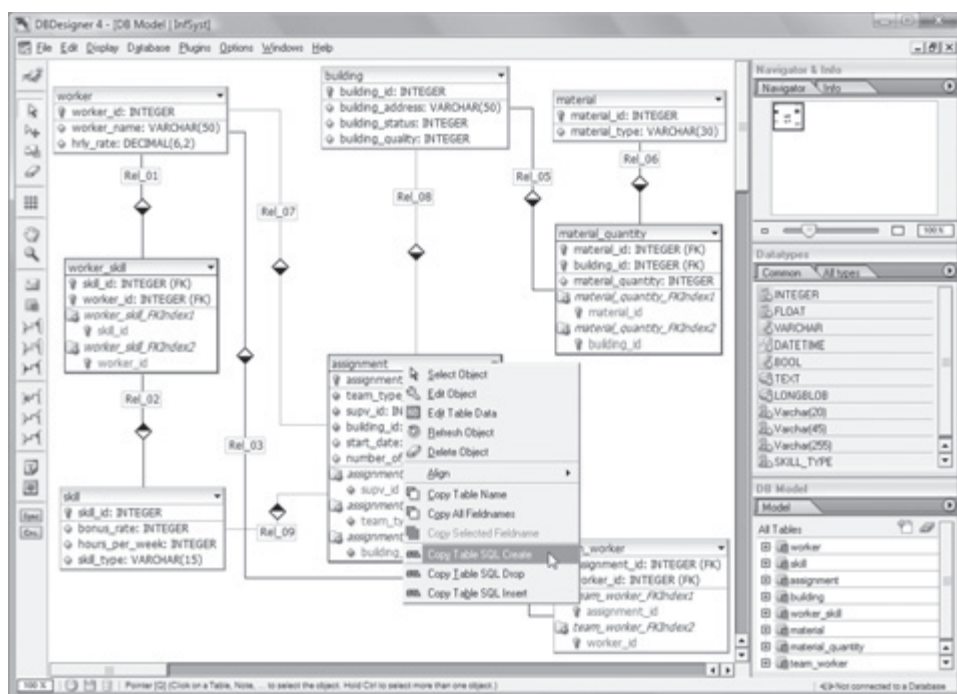


Рис. 6.24. Контекстное меню таблицы

Copy Table SQL Create (рис. 6.24). В буфер обмена скопируется сгенерированный SQL-скрипт CREATE TABLE assignment.

Контекстное меню можно использовать и для других целей. К примеру, команда Copy Table SQL Insert сгенерирует в буфер обмена заготовку для запроса INSERT вида:

```
INSERT INTO assignment (assignment id, team_type_id, supv_id, building_id,
start_date, number_of_days) VALUES
```

DBDesigner также позволяет создавать базу данных на сервере непосредственно из программы. Это происходит за счет подключения DBDesigner к серверу,

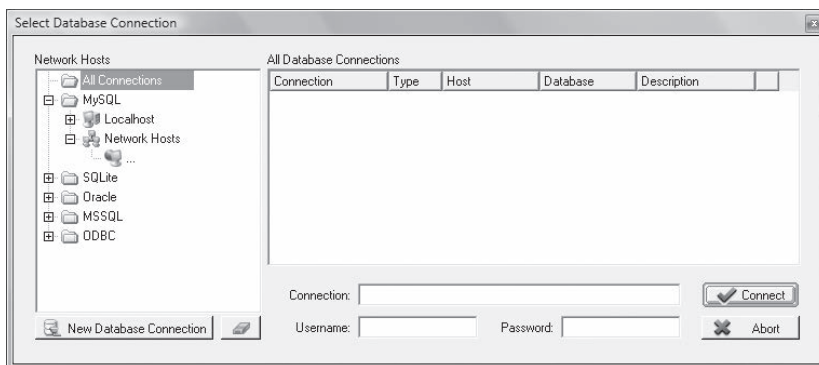


Рис. 6.25. Окно диалога установления соединения

созданию на нем базы данных и установлению синхронизации между базой на сервере и моделью.

Для начала необходимо установить соединение с сервером. Для этого выполните команду **Database ► Connect to Database**. Появится окно диалога **Select Database Connection**, в котором можно выбрать нужное соединение или создать новое (рис. 6.25).

Используя окно диалога **Select Database Connection**, создадим новое соединение с сервером MySQL. Щелкните на кнопке **New Database Connection**. Появится окно диалога **Database Connection Editor**, в котором введем все необходимые данные для доступа к новой базе данных, а именно имя соединения, хост, имя базы данных, имя пользователя и при необходимости пароль, как показано на рис. 6.26.

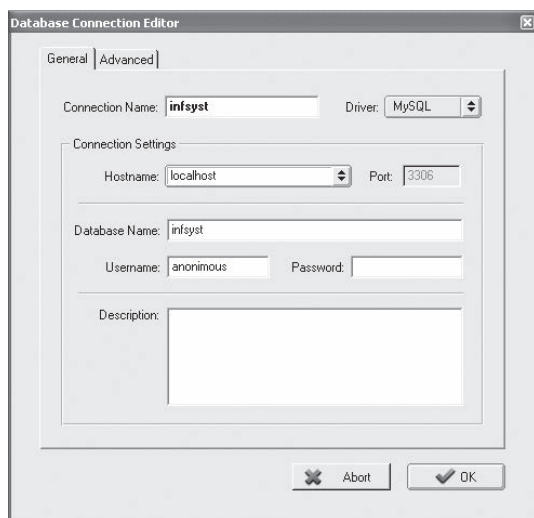


Рис. 6.26. Окно диалога редактирования соединения

Щелкните на кнопке **OK**, а затем на кнопке **Connect** окна диалога **Select Database Connection**, после чего соединение с новой базой данных будет установлено.

Модификация структуры базы данных

Жизненный цикл создания и сопровождения информационной системы имеет вид спирали. Это означает, что модификация структуры базы данных практически неизбежна. Использование CASE-средств позволяет облегчить выполнение данной задачи.

Используя возможности DBDesigner, выполним следующие действия:

- ☐ импортируем из существующей базы данных ее модель;
- ☐ внесем в нее необходимые изменения;
- ☐ синхронизируем модифицированную модель с существующей базой данных.

Прежде чем импортировать модель базы данных, необходимо установить с ней соединение, как это было рассмотрено выше. После установки соедине-

ния с базой данных и запуска команды Database ► Reverse Engineering программа выдаст окно диалога Reverse Engineering с опциями импорта модели данных (рис. 6.27). Здесь стоит обратить внимание на способ выявления связей между таблицами, раздел Build Relations. При выборе пункта Build Relations based on Primary Keys связи в модели будут строиться на основании первичных ключей, существующих в таблицах.

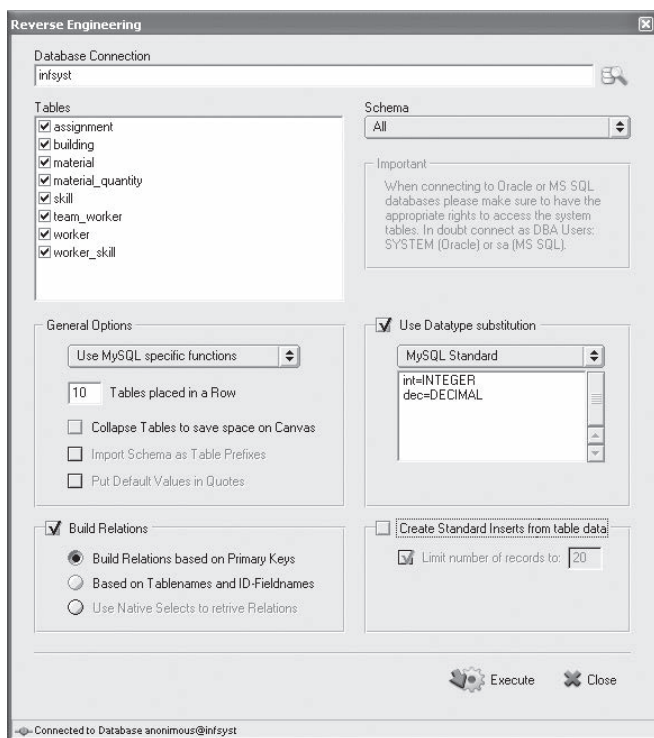


Рис. 6.27. Окно диалога Reverse Engineering

По щелчку на кнопке Execute на рабочем поле программы появятся все выбранные для импорта таблицы в виде ER-диаграммы (рис. 6.28).

Появившиеся таблицы и связи (надо отметить, что не все связи определены автоматически) являются рабочими объектами программы. А значит, можно не только проводить подробный анализ деталей таблиц и связей, но и вносить необходимые изменения в модель.

Допустим, вы внесли в некоторые таблицы новые поля или добавили к модели еще одну новую таблицу. Чтобы новую модель применить к изначальной базе данных, ее нужно синхронизировать. Выполните команду Database ► Database Synchronisation. Откроется окно диалога Database Synchronisation, где можно задать различные параметры синхронизации (рис. 6.29). К примеру, вы можете указать, чтобы при синхронизации не удалялись существующие таблицы базы данных, если вдруг они не вошли в модифицированную модель (опция Don't delete existing Tables) и др.

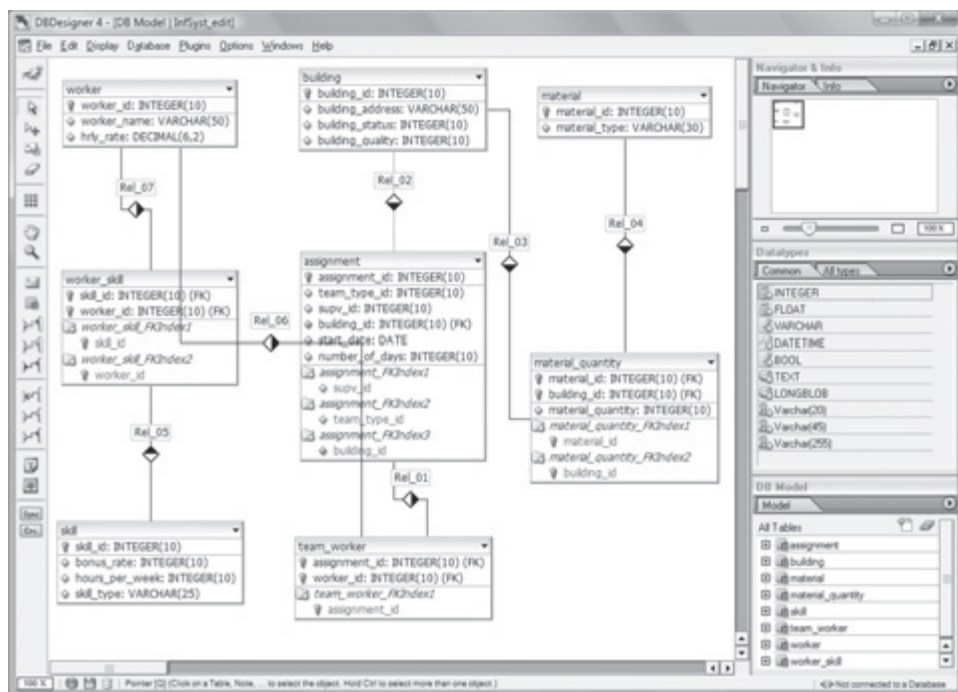


Рис. 6.28. Импортированная модель

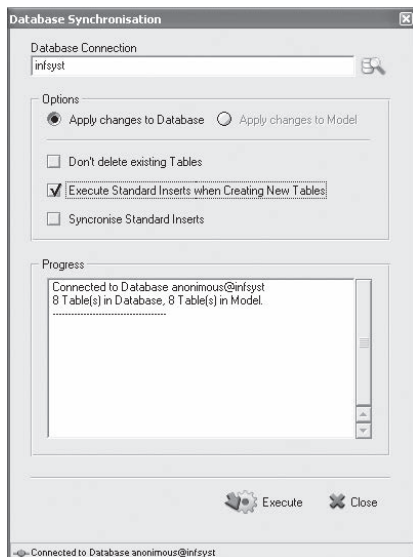


Рис. 6.29. Окно диалога установления параметров синхронизации

ПРИМЕЧАНИЕ

Синхронизация происходит безопасно для данных базы, если, конечно, не удалять из модели таблицы или поля с этими данными.

Часть II

Delphi — система быстрой разработки приложений

Глава 7

Delphi и объектно-ориентированное программирование

Существуют две основные модели построения программ. Первая называется *процедурно-ориентированной* и в ней программа представляется как ряд последовательно выполняемых операций (процедур). Программы, построенные с использованием процедурно-ориентированной модели, можно рассматривать как код, воздействующий на данные. Языки программирования, в которых реализован указанный подход к построению программ, называются процедурными. До определенного времени процедурно-ориентированный подход успешно применялся при разработке программ. Однако по мере увеличения объема и усложнения программ при использовании данного подхода возникают существенные проблемы.

В *объектно-ориентированной* модели программа рассматривается как совокупность объектов — отдельных фрагментов кода, обеспечивающих выполнение определенных действий и объединяющих данные и методы управления ими. Взаимодействие между объектами производится через определенные интерфейсы. Объектно-ориентированные программы можно характеризовать как данные, управляющие доступом к коду. Объектно-ориентированный подход повышает надежность разрабатываемых программ и обеспечивает возможность многократного использования кода.

Следует отметить, что объектно-ориентированное программирование является довольно возрастной концепцией — первые объектно-ориентированные языки (Simula, Smalltalk) появились около 30 лет назад.

Практически все современные алгоритмические языки поддерживают принципы объектно-ориентированного программирования. Наибольшее распространение в последнее время получили такие объектно-ориентированные языки, как C++, язык Delphi, Visual Basic, Java и др.

Основы языка Delphi

В данной главе рассматривается язык Delphi, используемый в системе визуального программирования Turbo Delphi фирмы Borland.

Язык Delphi является строгим языком, что во многом обусловлено учебным характером его предшественника языка Pascal.

Структура программы в Delphi

Программа, написанная на языке Delphi, состоит из ряда разделов (или блоков). Начало каждого раздела указывается с помощью специальных зарезервированных слов. В общем виде программа Delphi имеет следующий вид:

```
// Заголовок программы
Program имя_программы;

// Раздел объявления используемых модулей
Uses
    Модуль_1, Модуль_2, Модуль_3;

// Раздел объявления используемых меток
Label
    Метка_1, Метка_2;

// Раздел описания констант
Const
    идентификатор_константы_1 = значение_1;
    идентификатор_константы_2 = значение_2;
    идентификатор_константы_3 = выражение_1;

// Раздел описания пользовательских типов
Type
    идентификатор_типа_1 = определение_типа_1;
    идентификатор_типа_2 = определение_типа_2;

// Раздел объявления переменных
Var
    идентификатор_переменной_1 : определение_переменной_1;
    идентификатор_переменной_2;
    идентификатор_переменной_3 : идентификатор_типа_2;

// Раздел объявления процедур и функций программы
Procedure процедура_1;
    // текст процедуры
Function функция_1 : определение_типа_1;
    // текст функции

begin
    // текст программы
end
```

Заголовок программы

В заголовке после служебного слова `Program` указывается имя программы. Хотя заголовок программы не является обязательным разделом, при написании программы в среде Delphi имя программы надо указывать. При этом имя основного файла проекта должно совпадать с именем программы, указанным в заголовке.

ПРИМЕЧАНИЕ

В Delphi при сохранении файла проекта под новым именем изменение имени программы в файле производится автоматически.

Заголовок программы может быть только один; он обязательно должен быть первой строкой программы.

Раздел объявления модулей

Начало раздела объявления модулей указывается с помощью директивы `Uses`. Имена используемых модулей просто перечисляются через запятую (модули Delphi будут обсуждаться далее). Программа может содержать только один блок `Uses`, причем он должен следовать сразу за заголовком программы.

Разделов объявления меток, типов, констант и переменных может быть несколько, и они могут следовать в любом порядке.

Раздел объявления меток

Начало раздела объявления меток указывается с помощью директивы `Label`. В данном разделе через запятую перечисляются имена используемых в программе меток. Идентификатор метки может состоять из символов латинского алфавита и/или цифр, а также знаков подчеркивания (`_`).

ПРИМЕЧАНИЕ

В структурированных и объектно-ориентированных языках, к которым относится и язык Delphi, использование меток считается плохим стилем программирования. Поэтому меток в программах, разработанных в среде Delphi, практически никогда нет.

Раздел описания типов

В Delphi существует довольно большое количество стандартных типов и множество типов, описанных в стандартных модулях. Однако при разработке программ, особенно объектно-ориентированных, программисту необходима возможность создавать свои пользовательские типы данных, которые носят название «типы данных, определяемые пользователем». Для описания пользовательских типов используется раздел объявления типов, начинающийся с директивы `Type`. При создании типа указывается его идентификатор и после знака равенства приводится описание типа. Самым простым способом объявления собственного типа является просто объявление типа, аналогичного уже существующему, например:

Type

```
идентификатор_типа_1 = integer;
```

Наиболее часто программисту приходится определять так называемые структурные типы, назначение которых будет обсуждаться несколько позже.

Идентификатор типа может содержать буквы латинского алфавита, цифры и знак подчеркивания. Первым символом идентификатора обязательно должна быть либо латинская буква, либо символ подчеркивания.

Раздел переменных

Начало раздела переменных объявляется с помощью служебного слова `Var`. В данном разделе должны быть описаны все переменные программы. Компилятор Delphi не допускает использования переменных, не объявленных в разделе `Var`. Объявление переменной, не задействованной в программе, не приводит к ошибке компиляции, однако компилятор выдаст предупреждение о том, что переменная объявлена, но никогда не используется. Например, если объявить неиспользуемую переменную `A`, то компилятор выдаст следующее сообщение:
[Pascal Hint] Project1.dpr(8): Variable 'A' is declared but never used in 'Project1'

Число в скобках после имени файла идентифицирует номер строки, в которой описана неиспользуемая переменная.

При объявлении переменной указывается идентификатор и через двоеточие — тип переменной, например:

```
var  
идентификатор_переменной_1 : integer;
```

Идентификаторы переменных одного типа можно перечислять через запятую:

```
var  
идентификатор_переменной_1, идентификатор_переменной_2,  
идентификатор_переменной_3 : integer;
```

Для нестандартных типов имя типа должно быть описано в разделе `Type`, находящемся выше раздела `Var`, в котором оно используется.

Идентификатор переменной может состоять из символов латинского алфавита, цифр и символов подчеркивания. Первым символом идентификатора обязательно должна быть латинская буква или символ подчеркивания.

Раздел констант

Раздел констант содержит объявления констант и начинается с директивы `Const`. Константа фактически является переменной, значение которой устанавливается не в процессе выполнения программы, а на этапе компиляции. Значение константы не может изменяться программно, при попытке присвоить константе какое-либо значение компилятор выдает сообщение об ошибке. В объявлении константы можно использовать не только конкретные значения, но и выражения (которые могут быть вычислены на стадии компиляции, то есть не должны содержать переменных и вызовов функций пользователя). При объявлении константы указывается идентификатор и через знак равенства — значение

или выражение. Тип константы определяется присваиваемым ей значением или типом результата, получаемого при вычислении выражения.

Идентификатор константы может состоять из символов латинского алфавита, цифр и символов подчеркивания. Первым символом идентификатора обязательно должна быть латинская буква или символ подчеркивания.

Помимо обычных констант, которые принято называть истинными константами, в Delphi можно использовать так называемые типизированные константы, или константы-переменные, которые также объявляются в разделе `Const`. Как и обычным константам, на этапе компиляции константам-переменным также присваивается какое-либо значение. Однако значения констант-переменных можно изменять в процессе выполнения программы, то есть работать с ними как с обычными переменными.

ПРИМЕЧАНИЕ

Типизированные константы не могут иметь файловый тип и тип `Variant`.

Константы-переменные фактически являются обычными переменными, которым при запуске программы присваиваются некоторые значения. Объявление константы-переменной отличается от объявления обычной константы тем, что после идентификатора константы через двоеточие указывается ее тип, а затем после знака равенства — значение. Пример:

```
const
  // константа целого типа:
  идентификатор_константы_1 = 100;
  // константа вещественного типа:
  идентификатор_константы_2 = 100.0;
  // константа строкового типа:
  идентификатор_константы_3 = '100';
  // константа, заданная выражением:
  идентификатор_константы_4 = (2.5+1)/(2.5-1)
  // константа-переменная целого типа:
  идентификатор_константы_5 : integer = 20;
  // константа-переменная действительного типа:
  идентификатор_константы_6 : real = 3.14;
```

Как типизированную константу вы можете объявить также и константу-массив, константу-запись, константу-процедуру и константу-указатель. При объявлении константы-массива значения массива должны быть отделены друг от друга запятыми и весь список значений должен быть заключен в круглые скобки. При объявлении константы-записи необходимо определить значения для всех полей записи, указывая их одной строкой в круглых скобках и отделяя друг от друга точкой с запятой. Поля должны перечисляться в соответствии с порядком их объявления в типе записи. Константа-процедура предполагает обязательное определение функции или процедуры, которая является совместимой с объявленным типом константы. Константа-указатель требует, чтобы при ее объявлении указатель ссылался на конкретный участок памяти, который не будет изменен при компиляции. Этого можно достичь одним из трех способов: явным указанием ссылки на адрес памяти, с помощью оператора `@`, с помо-

щью использования служебного слова `nil`, а в том случае если константа имеет тип `PChar`, указанием на строковый литерал.

ПРИМЕЧАНИЕ

Константы-указатели не могут ссылаться на адреса локальных и динамических переменных.

Пример:

```
type
    TPoint = record
        X, Y: Single;
    end;
    TVector = array [0..1] of TPoint;
function Calc(X, Y: Integer): Integer;
begin
    ...
end;
type TFunction = function(X, Y: Integer): Integer;
// константа-массив:
идентификатор_константы_1: array [0..8] of Char = ('A', 'Б', 'В', 'Г', 'Д');
// константа-запись:
Origin: TPoint = (X: 0.0; Y: 0.0);
Line: TVector = ((X: -3.1; Y: 1.5), (X: 5.8; Y: 3.0));
// константа-процедура:
MyFunction: TFunction = Calc;
// константа-указатель:
WarningStr: PChar = 'Warning!'
```

Типы данных в Delphi

Язык Delphi отличается строгой типизацией данных. При присваивании переменной какого-либо значения компилятор всегда проверяет соответствие типов. Поэтому все переменные, используемые в программе, обязательно должны быть описаны в разделе объявления переменных.

Типы данных, используемые в Delphi, можно разделить на две группы: *простые* типы и *структурные* типы.

Несколько особняком стоят *указательные* типы, которые нельзя отнести ни к простым, ни к структурным. Указательные типы предназначены для работы с большими структурами данных и памятью.

Также в Delphi добавлена возможность объявления так называемых *вариантных* типов. Вариантные переменные могут динамически изменять свой тип в процессе выполнения программы.

Простые типы

Простыми являются типы данных, которыми напрямую может манипулировать процессор (или математический сопроцессор). Простые типы делятся на две группы: *порядковые* и *действительные*. Различие между ними заключается в следующем: *порядковые* типы представляют собой счетные множества чисел, лежащих

в определенном диапазоне; *действительные* типы не могут быть представлены в виде счетного множества чисел (если не принимать во внимание конечную точность представления действительных чисел при использовании цифровой вычислительной техники).

Порядковые типы

Порядковые типы подразделяются на целые, символьные, логические, перечисляемые и диапазонные.

- *Целые типы.* В переменных целого типа отсутствует дробная часть. В Delphi определено довольно большое количество стандартных целых типов, различающихся наличием или отсутствием знака, а также занимаемым объемом памяти. Диапазон значений каждого типа однозначно определяется этими двумя факторами: для n -разрядного числа без знака диапазон значений от 0 до 2^n , для числа со знаком — от -2^{n-1} до $2^{n-1} - 1$. Информация по всем целочисленным типам Delphi приведена в табл. 7.1.

Таблица 7.1. Целые типы Delphi

Обозначение типа	Диапазон значений	Занимаемый объем памяти
Без знака		
Byte	0...255	8
Word	0...65 535	16
LongWord	0...4 294 967 295	32
Cardinal	0...4 294 967 295	32
Со знаком		
ShortInt	-128...127	8
SmallInt	-32 768...32 767	16
Integer	-2 147 483 648...2 147 483 647	32
LongInt	-2 147 483 648...2 147 483 647	32
Int64	$-2^{63} \dots 2^{63} - 1$	64

- *Символьные типы.* Классическим методом представления символьной информации является использование 7-разрядной кодировки ASCII (American Standard Code for Information Interchange — Американский стандартный код для обмена информацией). Однако информация обычно хранится в 8-разрядном участке памяти. С помощью 8 бит можно закодировать 256 символов. Кодировка первых 128 символов является стандартной и используется для представления букв латинского алфавита, цифр, символов арифметических действий и ряда специальных символов, которые не могут быть введены с клавиатуры и применяются в качестве управляющих, например при выводе информации на принтер. Отнесение управляющих кодов к символному типу несколько условно, так как многие из них вообще не отображаются в виде каких-либо символов. Следующие 128 символов (с кодами от 128 до 255) называются расширенным набором ASCII. Существует несколько вариантов расширенного набора символов, которые используются для ото-

бражения символов русского алфавита, символов псевдографики и т. п. В Delphi применяется расширенный набор символов ANSI.

ПРИМЕЧАНИЕ

В последнее время широкое распространение получила 16-разрядная кодировка символьной информации, называемая UNICODE. Данная кодировка позволяет применять гораздо более обширный набор символов и получает все большее распространение. Следует отметить, что кодировка первых 256 символов UNICODE совпадает с кодировкой ANSI.

В Delphi поддерживается как кодировка ANSI (8-разрядная), так и кодировка UNICODE (16-разрядная). Соответственно определены два символьных типа:

- AnsiChar, или Char, — символьный тип с 8-разрядной кодировкой ANSI;
 - WideChar — символьный тип с 16-разрядной кодировкой UNICODE.
- *Логические типы.* Переменные логического типа могут принимать только два значения — true (истина) или false (ложь). В классическом языке Pascal был определен только один логический тип Boolean. Переменные данного типа занимали в памяти 1 байт. В последних версиях языка Delphi для совместимости с другими языками определены три логических типа, различающиеся занимаемым объемом памяти:
- Boolean, или ByteBool, — 1 байт;
 - WordBool — 2 байта;
 - LongBool — 4 байта.

ПРИМЕЧАНИЕ

Слова true и false в языке Delphi являются зарезервированными для работы с логическими переменными. При присваивании значения логической переменной можно использовать только true или false (или присваивать одной логической переменной значение другой логической переменной). В отличие от языка C/C++, в Delphi не допускается присваивание логической переменной значения целочисленного типа.

- *Перечисляемые типы.* Этот тип определяется перечислением соответствующих идентификаторов, разделяемых запятыми и заключаемых в круглые скобки. Переменные данного типа содержат дискретные значения, представляемые не числами, а *именами*:

```
type  
    перечисляемый_тип = (first, second, third);
```

В данном примере перечисляемый_тип представляет идентификатор перечисляемого типа, а идентификаторы first, second и third — возможные значения переменной типа перечисляемый_тип. Если в разделе var объявить переменную типа перечисляемый_тип, то этой переменной можно будет присваивать только значения first, second и third. Значения перечисляемых типов не являются числами и им нельзя присваивать числовые значения.

- *Диапазонные типы.* Переменные диапазонного типа содержат значения, соответствующие некоторому заданному диапазону любого порядкового типа. Определение диапазонного типа имеет следующий вид:

```
type  
    диапазонный_тип = минимум..максимум;
```

Диапазонные типы сохраняют все особенности исходного типа и совместимы с ним.

Действительные типы

Переменные *действительного* типа используются для представления чисел, имеющих дробную часть. В современной вычислительной технике действительные числа представляются в форме с плавающей точкой. Для работы с ними применяется математический сопроцессор, который сейчас имеется практически на каждом компьютере. В математическом сопроцессоре используются форматы представления чисел с плавающей точкой, стандартизованные Американским институтом инженеров электротехники и электроники (Institute of Electrical and Electronic Engineers, IEEE). Данные форматы различаются объемом занимаемой памяти и количеством значащих цифр мантиисы (с увеличением количества значащих цифр объем занимаемой памяти возрастает).

В Delphi используются три формата IEEE: Single, Double и Extended, предназначенные соответственно для хранения чисел с разрядностью 32, 64 и 80 бит. В более ранних версиях Delphi был определен тип Real, в котором для представления чисел с плавающей точкой использовались 48 бит. Этот формат был несовместим с форматами математического сопроцессора и требовал дополнительного времени на преобразование в стандартный вид. В последних версиях Delphi тип Real аналогичен типу Double, а для совместимости со старыми версиями введен дополнительный тип Real48, использующий 48 бит.

ПРИМЕЧАНИЕ

При использовании действительных типов необходимо учитывать, что на самом деле представляемые с помощью них числа все равно являются дискретными вследствие конечной точности их представления в цифровых вычислительных системах.

Помимо форматов с плавающей точкой в Delphi определены два вещественных формата с фиксированной точкой: Comp и Currency. Тип Comp представляется с помощью 64 бит и содержит только целые числа в диапазоне от $-2^{63} + 1$ до $2^{63} - 1$. Однако этот тип относится к вещественным и не совместим с целыми типами. Тип Currency также использует 64 бита, но содержит дробную часть, под которую отводится 4 десятичных знака. Данные форматы применяются для программирования операций с денежными единицами. Использование типов с фиксированной точкой позволяет уменьшить ошибку округления.

В табл. 7.2 приведена полная информация о действительных типах данных языка Delphi.

Таблица 7.2. Действительные типы Delphi

Обозначение типа	Диапазон значений	Количество значащих цифр	Объем памяти
Single	$\pm(1,5 \times 10^{-45} \dots 3,4 \times 10^{38})$	7...8	4 байта
Real48	$\pm(2,9 \times 10^{-39} \dots 1,7 \times 10^{38})$	11...12	6 байт
Real, Double	$\pm(5 \times 10^{-324} \dots 1,7 \times 10^{308})$	15...16	8 байт
Extended	$\pm(3,4 \times 10^{-4932} \dots 1,1 \times 10^{4932})$	19...20	10 байт
Comp	$-2^{63} + 1 \dots 2^{63} - 1$	19...20	8 байт
Currency	-922 337 203 685 477.5808... 922 337 203 685 477.5807	19...20	8 байт

Структурные типы

Структурные типы данных позволяют использовать переменные, содержащие несколько значений. Элементами структурных типов можно манипулировать и по отдельности, и как единым целым. Элементы структурного типа могут быть как простыми, так и структурными.

Данные в переменной структурного типа по умолчанию выравниваются по границе слова или двойного слова для обеспечения наиболее быстрого доступа к данным. Для отключения выравнивания при объявлении переменной структурного типа применяется ключевое слово `packed`:

```
type
идентификатор_типа_1 = packed array[0..100] of byte;
var
идентификатор_переменной_1 : идентификатор_типа_1;
идентификатор_переменной_2 : packed array[0..200] of char;
```

В Delphi определены следующие структурные типы:

- ☐ строки;
- ☐ массивы;
- ☐ множества;
- ☐ записи;
- ☐ файлы;
- ☐ классы.

Строковые типы

В Delphi определены три типа для представления текстовых строк.

- ☐ **ShortString.** Данный тип аналогичен типу `String` ранних версий языка Pascal. Его переменные могут содержать строку длиной до 255 символов с фиксированным размером 256 байт. Фактически тип `ShortString` представляет собой массив символов, индексированный от 0 до 255. Под хранение символов строки выделяются байты с 1-го по 255-й. Байт с нулевым номером используется для хранения длины строки.
- ☐ **AnsiString.** Переменные этого типа могут хранить строку практически неограниченной длины. Максимальное количество символов в такой строке ограничено только адресным пространством компьютера (например, на компьютерах

IBM PC число символов в строке может достигать величины 2^{32}). Переменные данного типа занимают в памяти 4 байта и представляют собой адрес первого символа строки.

- WideString. Этот тип аналогичен типу AnsiString, но в отличие от последнего символы строки WideChar представляются в кодировке UNICODE, то есть занимают два байта.

ПРИМЕЧАНИЕ

В языке Delphi при объявлении строковых переменных можно использовать тип String. При этом тип переменной, объявленной как String, будет зависеть от директивы компилятора \$H. Если задано {\$H+} (установка по умолчанию), то тип String аналогичен AnsiString, если {\$H-} — ShortString.

Массивы

В языке Delphi, используемом в системе Delphi, определены два типа массивов — *статические* и *динамические*.

Статические массивы идентичны обычным массивам, которые использовались еще в классическом языке Pascal. Данные массивы объявляются с помощью ключевого слова array, после которого в квадратных скобках указывается диапазон изменения индексов массива, а затем через слово of — тип элементов массива. Например, следующее объявление задает переменную типа «массив десяти целых чисел с индексами от 1 до 10»:

```
var  
  A : array[1..10] of integer;
```

Индексы массива, указываемые в квадратных скобках, фактически являются диапазоном типом. Индексация массивов может быть произвольной, однако общепринято начинать ее с нуля.

Массивы могут быть *многомерными*. В этом случае в объявлении массива в квадратных скобках через запятую указываются несколько диапазонов изменения индексов. Например, для определения матрицы вещественных чисел размером 10×5 можно использовать следующее объявление:

```
var  
  A : array[0..9,0..4] of double;
```

Для обращения к элементу массива указываются его имя и индекс элемента в квадратных скобках. Если массив многомерный, то в квадратных скобках указывается соответствующее количество индексов через запятую. При этом обращение к элементу массива выполняется как обращение к обычной переменной соответствующего типа.

С массивами одного типа можно выполнять операцию присваивания. Переменные-массивы относятся к одному типу в двух случаях.

- Если при объявлении в разделе var идентификаторы переменных перечислялись через запятую.
- Если применялся пользовательский тип массива.

Для примера рассмотрим следующий фрагмент кода:

```
type
  Мой_массив = array[0..9] of integer;
var
  A1, A2 : array[0..9] of integer;
  A3 : array[0..9] of integer;
  A5 : Мой_массив;
  A6 : Мой_массив;
```

Отметьте, что переменные A1, A3 и A5 имеют разный тип! Поэтому операции присваивания между ними недопустимы, так как приведут к ошибке компиляции. Однотипными здесь являются только переменные A1 и A2, а также A5 и A6.

Динамический массив представляет собой указатель на первый элемент массива. При объявлении динамического массива не указывается его размер, то есть диапазон изменения индекса:

```
var
  A : array of char;
```

Хотя переменная динамического массива фактически является указателем, работа с динамическим массивом почти идентична работе со статическим массивом. Отличие наблюдается только при выполнении операции присваивания переменных. Например, если объявлены два динамических массива A1 и A2, то после выполнения операции присваивания A1 := A2 обе переменных будут ссылаться на один и тот же фрагмент памяти, то есть фактически будут являться одним массивом. Изменение элементов массива A1 будет приводить к такому же изменению тех же элементов массива A2.

Нумерация элементов динамических массивов всегда начинается с нуля.

Размер динамических массивов определяется во время выполнения программы, для чего используется процедура:

```
SetLength (var S; NewLength: Integer)
```

Здесь S — строка или динамический массив; NewLength — размер строки или массива.

Для строк типа ShortString процедура SetLength просто устанавливает индикатор длины строки (символ с нулевым номером) в значение, заданное параметром NewLength. В этом случае величина NewLength не должна превышать 255.

ПРИМЕЧАНИЕ

В случае длинных строк или динамических массивов процедура SetLength выделяет объем памяти, равный параметру NewLength, умноженному на объем памяти, занимаемый одним элементом массива. Если SetLength применяется к строке или массиву, которые уже содержат какие-либо данные, то содержимое строки или массива сохраняется, но содержимое дополнительно выделенного объема памяти оказывается неопределенным. При недостатке памяти для размещения массива или строки заданного объема возникает исключительная ситуация EOutOfMemory.

Освобождение выделенной памяти производится с помощью процедуры Finalize(var S) или присваиванием переменной динамического массива значения nil.

Динамические массивы могут быть *многомерными*. При этом их объявление выглядит следующим образом:

```
var
  A : array of array of word;
```

При выделении памяти под многомерный массив в процедуре `SetLength` задаются все размерности массива, например `SetLength(A,10,5)`.

Можно также создавать динамические массивы с различной длиной по разным индексам. Для этого используется объявление массива как динамического многомерного (как в последнем примере) и сначала задается его размерность по первому индексу (например, процедура `SetLength(A,10)` задает массив `A`, состоящий из 10 строк). После этого можно отдельно задавать длину каждой строки:

```
SetLength(A[0],10);
SetLength(A[1],5);
...
SetLength(A[9],8);
...
```

Для освобождения памяти, занимаемой таким массивом, достаточно просто вызвать процедуру `Finalize(A)`.

Множества

Множество представляет собой набор значений какого-либо порядкового типа. Для объявления переменной типа множества используется ключевое слово `set`:

```
type
  NumSet = set of AnsiChar;
var
  A1 : NumSet;
  A2 : set of 0..100;
```

Минимальный и максимальный порядковые номера типа, на основе которого создается множество, должны лежать в пределах от 0 до 255.

Множеству можно присваивать произвольное подмножество:

```
A1:=['A','B','C','D','E'];
```

Записи

Записи представляют собой структурный тип, объединяющий элементы различных типов. Объявление записи выглядит следующим образом:

```
type
  MyRecType = record
    field1 : integer;
    field2,field3 : real;
    field4 : array[0..4] of char;
  end;
var
  RecVar1 : MyRecType;
  RecVar2 : record
    field1 : byte;
    field2,field3 : extended;
  end;
```

Элементы записи называются *полями*. Для обращения к отдельному полю используется идентификатор переменной записи и через точку указывается

идентификатор поля: `RecVar1.field1`. Кроме того, существует специальный оператор `with... do`, предназначенный для работы с записями. Использование данного оператора выглядит следующим образом:

```
with RecVar1 do field1:=10;
```

Запись может содержать вариантную часть, задаваемую с помощью оператора `case`. Например, запись следующего вида содержит вариантные поля `num_int` и `byte1`, `byte2`, `byte3`, `byte4`:

```
var
  RecVar = record
    field1 : real;
    case byte of
      1: num_int : integer;
      2: byte1,byte2,byte3,byte4 : byte
    end;
```

Все варианты занимают в памяти одно и то же место. Например, поле `byte1` в рассмотренном примере будет содержать первый байт переменной типа `integer`, хранящейся в поле `num_int`, поле `byte2` — второй байт и т. д.

ПРИМЕЧАНИЕ

Вариантные поля записей часто бывает удобно использовать для преобразования типов.

Запись может содержать только одну вариантную часть, которая должна находиться после всех фиксированных полей.

Файлы

Файловый тип данных используется для организации операций файлового ввода-вывода данных. Файловые переменные подразделяются на *типизированные* и *нетипизированные*.

Объявление переменной файлового типа подобно объявлению массива только без указания числа элементов. При этом вместо слова `array` используется ключевое слово `file`. Для *типизированных* файлов после слова `file` через `of` указывается тип элементов файла. Этот тип может быть любым, кроме `file` и `class`. Объявление *нетипизированной* файловой переменной отличается только тем, что тип элементов файла не указывается. Для работы с текстовыми файлами используется специальный тип `Text` или `TextFile`. Пример:

```
var
  F1 : file of real; // переменная файла вещественных чисел
  F2 : file;         // нетипизированная файловая переменная
  F2 : TextFile;     // переменная текстового файла
```

Более подробно работа с файлами будет обсуждаться позднее.

Классы

Классы являются структурным типом, похожим на тип `record`. Однако они позволяют объединять в одной структуре не только данные, но и методы их

обработки — процедуры и функции. Более подробно классы обсуждаются в разделе «Основы объектно-ориентированного программирования».

Указательные типы

Переменная указательного типа представляет собой адрес, по которому расположено значение переменной. Переменные указательного типа всегда занимают в памяти фиксированный объем — 4 байта (при этом они могут ссылаться на объемы данных любого размера, вплоть до 2^{32}).

Указатель может ссылаться на данные конкретного типа (типизированный указатель) или просто на область памяти (нетипизированный указатель). Для объявления нетипизированного указателя используется ключевое слово `pointer`. Для объявления типизированного указателя используется имя соответствующего типа, перед которым ставится знак разыменования (^):

```
var
  P1 : pointer; // нетипизированный указатель
  P2 : ^real;   // указатель на переменную типа real
  P3 : ^MyType; // указатель на переменную типа MyType,
               // определенного пользователем
```

Работа с переменными указательного типа имеет ряд особенностей. Необходимо помнить, что переменная-указатель является адресом. Чтобы обратиться к значению, на которое данный указатель ссылается, необходимо использовать перед именем переменной знак разыменования (^). Например, чтобы работать с переменной `P1` как с переменной вещественного типа, используется обращение `^P1`. Обращение к указателю без знака разыменования будет являться обращением не к значению переменной, а к ее адресу!

ПРИМЕЧАНИЕ

Переменные указательного типа часто называются динамическими переменными.

При работе с динамическими переменными им можно присваивать значение `nil`. Слово `nil` является зарезервированным, и присвоение данного значения указателю означает, что он ни на что не ссылается.

Объявление переменной указательного типа не означает, что при запуске программы для нее будет выделен необходимый объем памяти. Поэтому перед обращением к динамической переменной необходимо выделить для нее память, а по окончании работы с указателем эту память освободить. Для этого используются специальные процедуры языка Delphi, перечисленные ниже.

```
procedure New(var P: Pointer);
```

Выделяет необходимый объем памяти для хранения значения переменной, на которую ссылается указатель `P`. Объем выделяемой памяти определяется типом указателя. Если недостаточно свободной памяти, генерируется исключительная ситуация `EOutOfMemory`.

```
procedure Dispose(var P: Pointer);
```

Освобождает область памяти, на которую ссылается указатель P.

```
procedure GetMem(var P: Pointer; Size: Integer);
```

Выделяет Size байт для размещения данных, на которые ссылается указатель P. Если памяти недостаточно, генерируется исключительная ситуация EOutOfMemory.

```
procedure FreeMem(var P: Pointer [; Size: Integer]);
```

Освобождает область памяти, на которую ссылался указатель P. Необязательный параметр Size определяет, сколько байтов памяти будет освобождено. Если параметр Size указывается, то его значение должно точно соответствовать объему памяти, выделенному переменной P процедурой GetMem.

```
procedure FreeAndNil(var P);
```

Освобождает память, на которую ссылается указатель P, и присваивает указателю P значение nil.

Для типизированных указателей выделение и освобождение памяти следует производить с помощью процедур New и Dispose.

ПРИМЕЧАНИЕ

После освобождения памяти с помощью процедур Dispose и FreeMem значение указателя не будет равно nil. Если необходимо проверять, ссылается ли что-нибудь на указатель или нет, то после освобождения памяти ему следует явно присвоить значение nil или использовать процедуру FreeAndNil, которая, однако, работает только в том случае, если память выделялась процедурой GetMem.

Вариантные типы

Вариантные типы используются в тех случаях, когда необходимо передавать значение, тип которого заранее неизвестен.

Для объявления переменной вариантного типа используется зарезервированное слово *variant*. Под переменную данного типа отводится 16 байт. В них содержится код типа, а также значение переменной или указатель на значение.

Тип *variant* позволяет хранить все простые типы данных (кроме *Int64*), а также динамические массивы.

В Delphi определены два особых значения переменных типа *variant*:

- ❑ *Unassigned* — означает, что переменной пока не присвоено значение какого бы то ни было типа, то есть к вариантной переменной ни разу не обращались (данное значение присваивается вариантной переменной при ее инициализации);
- ❑ *Null* — означает, что вариантная переменная содержит данные неизвестного типа.

Для получения информации о типе данных, хранимых в вариантной переменной, используется специальная функция *VarType*. Коды, возвращаемые данной функцией, приведены в табл. 7.3.

Таблица 7.3. Коды типов вариантных переменных

Код типа	Значение	Тип
varEmpty	\$0000	Значение Unassigned
varNull	\$0001	Тип отсутствует
varSmallInt	\$0002	SmallInt
varInteger	\$0003	Integer
varSingle	\$0004	Single
varDouble	\$0005	Double
varCurrency	\$0006	Currency
varDate	\$0007	Дата и время
varOleStr	\$0008	WideString
varDispatch	\$0009	Указатель на объект OLE Automation, интерфейс Dispatch
varError	\$000A	Код системной ошибки
varBoolean	\$000B	Boolean
varVariant	\$000C	Variant
varUnknown	\$000D	Указатель на объект OLE Automation, интерфейс Unknown
varByte	\$0011	Byte
varString	\$0100	AnsiString
varArray	\$2000	Массив
varByRef	\$4000	Указатель

Для работы с вариантными массивами в Delphi существует ряд специальных функций:

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

Эта функция предназначена для создания вариантного массива с границами, заданными параметром Bounds, и типом элементов, заданных параметром VarType. Значение VarType должно быть одним из кодов, приведенных в табл. 7.3. Например, следующая строка создает массив вещественных чисел, состоящий из 10 элементов (переменная A должна быть описана как variant):

```
A:=VarArrayCreate([0,9],varDouble);
```

Элементы массива могут иметь вариантный тип и, следовательно, содержать данные различных типов. Например, пусть имеется вызов:

```
A:=VarArrayCreate([0,4],varVariant);
```

После выполнения этого вызова будут допустимы следующие присваивания:

```
A[0]:= 1;
A[1]:= 1234.5678;
A[2]:= 'Hello world';
A[3]:= True;
```



```
function VarArrayOf (const Values:array of variant):Variant;
```

Данная функция возвращает одномерный вариантный массив с элементами, представленными в переменной Values. Нижний индекс массива всегда равен 0, а верхний определяется количеством элементов массива, переданного в Values. Например, после выполнения следующего оператора переменная A будет являться вариантным массивом, состоящим из 3 элементов вариант-ного типа:

```
A:=VarArrayOf([10, 3.14, 'Text']);
```

```
procedure VarArrayRedim (var A: Variant; HighBound: Integer);
```

Применяется для изменения верхнего предела HighBound вариантного массива A. Значения элементов массива, определенные перед изменением предела, сохраняются.

```
function VarArrayLock (var A: Variant): Pointer;
```

Используется для фиксирования вариантного массива A и возвращения указателя на первый элемент массива. Пока массив зафиксирован, его размерность не может быть изменена, и вызов функции VarArrayRedim будет безрезультатным. Для отмены фиксации используется процедура:

```
VarArrayUnlock(var A: Variant)
```

Операторы языка Delphi

Операторы предназначены для контроля порядка вычисления выражений и количеством вычислений. Операторы, используемые в языке Delphi, условно можно разделить на две группы: простые операторы и структурные операторы. К простым операторам следует отнести операторы присваивания и безусловного перехода. Группу структурных операторов составляют условные операторы, операторы циклов и составной оператор.

Оператор присваивания

Оператор присваивания предназначен для изменения значения переменных. Результат выражения, расположенного справа от оператора присваивания (: =), заносится в переменную, расположенную слева. При использовании оператора присваивания всегда необходимо строго соблюдать правила соответствия типов.

Оператор безусловного перехода

Оператор безусловного перехода goto передает управление оператору, помеченному указанной меткой:

```
goto label5;
```

Метка должна быть описана в разделе label того блока, в котором выполняется оператор goto.

В тексте программы идентификатор метки отделяется двоеточием от оператора, на который метка указывает:

```
...  
label5: A:=expression;  
...
```

ПРИМЕЧАНИЕ

В настоящее время оператор goto считается анахронизмом и практически никогда не используется.

Условный оператор

Условный оператор обеспечивает выполнение или невыполнение определенного оператора в зависимости от выполнения или невыполнения заданного условия. В Delphi определены два условных оператора: if и case... of.

Оператор if

Оператор if обеспечивает выбор из двух вариантов:

```
if логическое_выражение  
then оператор_1           // выполняется, если  
                           // логическое_выражение = true  
else оператор_2;          // выполняется, если  
                           // логическое_выражение = false
```

В операторе if часть else необязательна:

```
if логическое_выражение  
then оператор;            // выполняется, если значение  
                           // логическое_выражение = true
```

В Delphi определены следующие операции сравнения двух значений:

- ❑ $A > B$ — больше;
- ❑ $A < B$ — меньше;
- ❑ $A \geq B$ — больше или равно;
- ❑ $A \leq B$ — меньше или равно;
- ❑ $A = B$ — равно;
- ❑ $A \neq B$ — не равно.

В логическом выражении, приводимом после слова if, допускается использование следующих логических операций:

- ❑ not — инверсия;
- ❑ and — логическое умножение;
- ❑ or — логическое сложение;
- ❑ xor — исключающее или.

Например, логическое выражение может быть сформулировано следующим образом:

$(A=0) \text{ and } (B<0) \text{ or } (C\geq 0)$

ПРИМЕЧАНИЕ

Так как все логические операции имеют одинаковый приоритет, то операции сравнения чисел следует заключать в скобки.

Оператор case

Оператор `case... of` применяется для выполнения одного оператора из нескольких в зависимости от значения переменной (или результата вычисления выражения), указываемой между словами `case` и `of`. Данная переменная называется *селектором*. Селектор обязательно должен иметь порядковый тип. После описания селектора следует список операторов, каждому из которых предшествует одна или несколько меток, отделяемых от оператора двоеточием. Заканчивается оператор ключевым словом `end`. Метки представляют значения, которые может принимать селектор. При обращении к оператору `case` выполняется оператор, метка которого соответствует значению селектора:

```
case селектор of
  1 : оператор_1; // выполняется, если селектор = 1
  2,3 : оператор_2 // выполняется, если селектор = 2
                        // или если селектор = 3
end;
```

Метки в операторе `case` могут задаваться в виде диапазонов. Оператор `case` может иметь блок `else`. Оператор, расположенный после `else`, выполняется в том случае, если значение селектора не соответствует ни одной из указанных меток.

```
case селектор of
  'A' : оператор_1; // выполняется, если селектор = 'A'
  'D'..'H' : оператор_3; // выполняется, если селектор лежит
                        // в диапазоне от 'D' до 'H'
  else оператор_4; // выполняется во всех остальных
                  // случаях
end;
```

Операторы цикла

Операторы цикла обеспечивают возможность многократного повторения одного или нескольких операторов. В Delphi определены три оператора цикла: `for... do`, `while... do` и `repeat... until`.

Цикл for

Цикл `for... do` предназначен для выполнения строго заданного количества повторений. Между словами `for` и `do` для некоторой переменной целого типа (называемой переменной цикла) указывается диапазон изменения:

```
for i:=начальное_значение to конечное_значение do оператор;
for i:= начальное_значение downto конечное_значение do оператор;
```

При первом обращении к оператору `for` переменная цикла всегда принимает значение `начальное_значение`.

Если при указании диапазона изменения переменной цикла используется зарезервированное слово `to`, то для того чтобы оператор, следующий после слова `do`, был выполнен, необходимо, чтобы текущее значение переменной цикла было не больше `конечное_значение`. При каждом выполнении оператора, следую-

щего за `do`, значение переменной цикла увеличивается на 1. Если используется ключевое слово `downto`, то оператор, указанный после `do`, выполняется в том случае, если начальное_значение не меньше, чем конечное_значение; при каждом проходе значение переменной цикла уменьшается на 1.

Цикл `for` используется в тех случаях, когда заранее известно, сколько раз надо выполнить оператор.

Цикл `while...do`

Оператор `while...do` обеспечивает выход из цикла по условию. Условие указывается между словами `while` и `do` и представляет собой логическое выражение:

`while` логическое_выражение `do` оператор;

Оператор, следующий за `do`, выполняется до тех пор, пока результат логического выражения не станет равным `false`. Значение логического выражения вычисляется в первую очередь, и если оно изначально равно `false`, то оператор, указанный после `do`, не будет выполнен ни разу.

Цикл `repeat...until`

Конструкция `repeat...until` также обеспечивает выход из цикла по условию:

```
repeat
  оператор_1;
  оператор_2;
  ...
  оператор_n;
until логическое_выражение;
```

Операторы, расположенные между ключевыми словами `repeat` и `until`, будут выполняться до тех пор, пока логическое выражение, указанное после `until`, не примет значение `true`.

В отличие от цикла `while`, логическое выражение, обуславливающее выход из цикла, вычисляется после выполнения операторов тела цикла. Поэтому даже если оно изначально равно `true`, операторы все равно будут выполнены один раз.

ПРИМЕЧАНИЕ

При использовании циклов `while` и `repeat` возможно заикливание. Если логическое выражение никогда не будет принимать значение `false` для цикла `while` или `true` для цикла `repeat`, то программа никогда не выйдет из этого цикла. Поэтому будьте внимательны при использовании конструкций `while...do` и `repeat...until`.

В Delphi для работы с циклами имеются два дополнительных оператора `break` и `continue`, которые работают со всеми видами циклов:

- ❑ оператор `break` прерывает выполнение цикла и передает управление оператору, следующему за оператором цикла;
- ❑ оператор `continue` прерывает текущую итерацию и производит переход к первому оператору тела цикла, при этом в цикле `for` происходит изменение переменной цикла.

Составной оператор

Составной оператор позволяет интерпретировать группу операторов как один оператор. Это необходимо для работы практически со всеми рассмотренными

операторами. Например, условные операторы и операторы цикла `for` и `while` обеспечивают переход по условию или выполнение цикла только для одного оператора. Если же заключить группу операторов между словами `begin` и `end`, то они будут восприниматься как один оператор:

```
...
if a<b
  then begin
    c:=выражение_1;
    d:=выражение_2;
  end
  else begin
    c:=выражение_3;
    d:=выражение_4;
  end;
...
case i of
  1: begin
    оператор_1;
    оператор_2;
  end;
  2,3: оператор_3;
end;
...
for i:=0 to 10 do begin
  a:=выражение_1;
  b:=выражение_2;
end;
...
while a>b do
begin
  a:=выражение_1;
  b:=выражение_2;
end;
...
```

Процедуры и функции

Процедуры и функции представляют собой блоки программного кода, имеющие точно такую же структуру, как и программа (единственное отличие заключается в том, что процедуры и функции не могут содержать раздел `uses`).

Процедуры

Ниже приведен пример программной реализации процедуры:

```
procedure proc_id(<список параметров>); // заголовок процедуры
const // раздел описания локальных констант
  const1 = value1;
type // раздел описания локальных типов
  type_id1 = type_def1;
var // раздел описания локальных переменных
  var_id1 : type_id1;
  var_id2, var_id3 : type_def2;
begin
... // текст процедуры
end;
```

Заголовок процедуры состоит из зарезервированного слова `procedure`, идентификатора процедуры и списка параметров, заключенного в круглые скобки (список параметров не обязателен, можно создавать процедуры без параметров). Параметры, указываемые в заголовке процедуры, называются *формальными* и предназначены для обмена данными между процедурой и основной программой. В списке указывается идентификатор параметра и через двоеточие — его тип. Друг от друга параметры отделяются точкой с запятой:

```
procedure proc_id(param1:integer;param2:real);
```

Отметим основные свойства процедуры.

- ❑ Количество передаваемых параметров не ограничено.
- ❑ Внутри процедуры формальные параметры представляют собой обычные переменные или константы.
- ❑ Вызов процедуры осуществляется с помощью оператора вызова, состоящего из идентификатора процедуры и списка параметров, перечисляемых через запятую:

```
...  
proc_id(A,B);  
...
```

- ❑ Параметры, указываемые при вызове процедуры, называются *фактически-ми*. Они представляют собой переменные или константы, описанные в программе.
- ❑ Передача параметров производится через стек и может выполняться либо по значению, либо по ссылке.

При передаче параметра *по значению* в стек заносится значение переменной, соответствующей фактическому параметру. По умолчанию считается, что передача производится по значению, то есть при указании в заголовке процедуры параметра, передаваемого по значению, не требуется никаких дополнительных ключевых слов. Значение параметра, переданного по значению, можно изменять внутри процедуры, однако в программу это изменение передаваться не будет. Например, при выполнении следующего фрагмента кода значение глобальной переменной `A` после вызова процедуры `MyProc` не изменится и останется равным 5:

```
...  
var  
  A : integer;  
...  
procedure MyProc(B:integer);  
begin  
  B:=10;  
end;  
...  
A:=5;  
MyProc(A);  
...
```

ВНИМАНИЕ

При передаче параметров по значению необходимо соблюдать осторожность, так как при передаче больших структур данных возможно переполнение стека.

Второй способ передачи параметров — передача *по ссылке*. В этом случае в стек заносится не значение параметра, а его адрес. Таким образом, независимо от объема памяти, занимаемого переменной, в стеке будет занято только 4 байта. При передаче параметра по ссылке перед его идентификатором в списке формальных параметров указывается одно из ключевых слов: `var` или `const`. В первом случае параметр называется *параметром-переменной*, во втором — *параметром-константой*. Значения параметров-переменных в тексте процедуры можно изменять, и эти изменения будут передаваться в программу. Таким образом, параметры-переменные используются для передачи данных из процедуры в вызывающую программу. Например, при выполнении следующего фрагмента кода значение глобальной переменной `A` изменится и станет равным 10:

```
...  
var  
  A : integer;  
...  
procedure MyProc(var B:integer);  
begin  
  B:=10;  
end;  
...  
A:=5;  
MyProc(A);  
...
```

Значения параметров-констант внутри процедуры изменять нельзя. Попытка присвоить параметру-константе какое-нибудь значение приведет к ошибке компиляции:

```
[Error] Unit2.pas(10): Left side cannot be assigned to
```

- ❑ Тип формального параметра в заголовке процедуры можно не указывать, например:

```
procedure MyProc(var A);
```

В этом случае в качестве фактического параметра может быть использована переменная любого типа. Нетипизированные параметры передаются только по ссылке. При использовании нетипизированных параметров без типа компилятор не может проверить соответствие типов в тексте процедуры. Поэтому программист должен сам следить за соответствием типов во избежание возникновения ошибок во время выполнения программы (runtime error).

- ❑ Формальные параметры могут быть открытыми массивами. В этом случае процедуре можно передавать в качестве фактического параметра массивы разной длины. Параметр, имеющий тип открытого массива, объявляется следующим образом:

```
procedure proc_id(var param:array of type_id);
```

К открытому массиву внутри процедуры можно обращаться только поэлементно. Для определения количества элементов открытого массива используются следующие стандартные функции:

- `high(x)` — возвращает максимальный индекс, который на 1 меньше количества элементов;
- `low(x)` — возвращает минимальный индекс, всегда равный 0;
- `SizeOf(x)` — возвращает объем массива в байтах.

ВНИМАНИЕ

Открытые массивы можно передавать как по ссылке, так и по значению. Однако при передаче по значению фактический массив записывается в стек, что может привести к ошибке переполнения стека.

Функции

Функции отличаются от процедур только тем, что их идентификатор возвращает некоторое значение. Поэтому при описании функции необходимо через двоеточие указать тип возвращаемого значения:

```
function MyFunc(A:integer):single;  
begin  
...  
end;
```

Идентификатор функции может быть использован в выражении:

```
var_id1:=10*MyFunc(var_id2)/var_id3;
```

В теле функции для возвращения значения используется либо идентификатор функции, либо неявно определенный идентификатор `result`. Например, функция, вычисляющая величину, обратную заданному параметру, может быть описана двумя эквивалентными способами.

❑ **Первый способ:**

```
function reverse(a:double):double;  
begin  
    reverse:=1/double  
end;
```

❑ **Второй способ:**

```
function reverse(a:double):double;  
begin  
    result:=1/double  
end;
```

С переменной `result` внутри функции можно работать как с обычной переменной.

ПРИМЕЧАНИЕ

Процедуры и функции могут содержать не только разделы объявления констант, типов и переменных, но и включать в себя объявления функций и процедур. Все эти идентификаторы, называемые локальными, видны только внутри тех процедур и функций, в которых они объявлены.

Модули Delphi

При разработке программ в среде Delphi широко используются так называемые *модули*. Они позволяют объединить логически связанные типы данных, переменные, процедуры и функции в один программный блок. Причем все идентификаторы, описанные в модуле, могут быть использованы в других программных блоках. Фактически модуль представляет собой нечто вроде библиотеки подпрограмм, типов данных, переменных и констант. Для использования идентификаторов, описанных в модуле в программе (или другом модуле), достаточно объявить имя модуля в разделе `uses`.

Структура модуля Delphi имеет следующий вид:

```
unit имя_модуля;      // заголовок модуля
interface            // блок интерфейса
uses
  модуль_1, модуль_2;
const
  константа_1 = значение_1;
  константа_2 = выражение_1;
type
  тип_1 = определение_типа_1;
var
  идентификатор_переменной_1 : определение_типа_1;
  идентификатор_переменной_2 : определение_типа_2;
procedure идентификатор_процедуры_1;
function идентификатор_функции_1 : определение_типа_3;
implementation        // блок реализации
uses
  модуль_3, модуль_4;
const
  константа_3 = значение_2;
type
  тип_2 = определение_типа_4;
var
  идентификатор_переменной_3,
  идентификатор_переменной_4 : определение_типа_5;
procedure процедура_1;
begin
  ...
end;
function функция_1 : определение_типа_6;
begin
  ...
end;
procedure идентификатор_процедуры_1;
begin
  ...
end;
function идентификатор_функции_1 : определение_типа_7;
begin
  ...
end;
initialization        // блок инициализации
  оператор_1;
  оператор_2;
```

```
finalization           // блок завершения
  оператор_3:
  оператор_4:
end.
```

В приведенном примере модуля можно выделить структурные единицы.

- ❑ *Заголовок модуля* состоит из ключевого слова `unit` и имени модуля, которое обязательно должно совпадать с именем файла. В отличие от заголовка программы, заголовок модуля является обязательным.
- ❑ *Блок интерфейса* содержит описание констант, типов, переменных, процедур и функций, которые будут доступны в других программах и модулях. Интерфейсная часть начинается с зарезервированного слова `interface` и всегда должна следовать сразу за заголовком. Блок интерфейса может содержать раздел `uses`, который должен быть описан в первую очередь, до всех других объявлений. В блоке интерфейса описываются только заголовки процедур и функций. Их текст приводится в разделе реализации.
- ❑ *Блок реализации* начинается с зарезервированного слова `implementation` и может содержать объявления констант, типов, переменных, процедур и функций. Все эти объявления доступны только в данном модуле. Кроме того, в разделе `implementation` размещается реализация всех процедур и функций, заголовки которых объявлены в блоке интерфейса. Все идентификаторы, описанные в разделе интерфейса, доступны в разделе реализации.

ПРИМЕЧАНИЕ

Блок реализации может содержать раздел `uses`, который должен следовать сразу за словом `implementation`. Причем используемые модули лучше указывать именно в блоке реализации.

- ❑ *Блок инициализации* содержит операторы, выполняемые только один раз при запуске программы, в которой используется данный модуль. Блок инициализации необязателен. Начало блока указывается зарезервированным словом `initialization`. Инициализационные разделы модулей выполняются в том порядке, в котором модули указаны в разделе `uses`.
- ❑ *Блок завершения* начинается с ключевого слова `finalization` и включается в программу только в том случае, если модуль содержит блок инициализации. Блок завершения содержит операторы, которые выполняются только один раз при завершении работы программы, использующей данный модуль. Разделы завершения выполняются в порядке, обратном порядку перечисления модулей в блоке `uses`.

Основы объектно-ориентированного программирования

Концепция объектно-ориентированного программирования (ООП) позволяет упростить разработку сложных программ и повысить их надежность. Однако объектно-ориентированная модель построения программ принципиально отли-

чается от процедурно-ориентированной. Ее основу составляет не алгоритм, а иерархия объектов, из которых состоит программа (хотя разработка отдельных объектов все равно требует алгоритмического подхода). Поэтому для эффективного использования ООП требуется иной взгляд на проблему, иначе даже применение объектно-ориентированных языков не поможет добиться объектно-ориентированного стиля программирования.

В данном разделе описываются основные принципы ООП, которые иллюстрируются примерами на языке Delphi.

Основные понятия и отличительные черты ООП

В основе объектно-ориентированного программирования лежит идея объединения данных и действий, которые производятся над этими данными, в одной структуре.

Каждая используемая в программе переменная имеет смысл только тогда, когда может принимать какие-либо значения. Множество значений, которые может принимать переменная, является определяющей характеристикой переменной и называется ее *типом*. Тип переменной, в свою очередь, определяет набор операций, которые можно к ней применять.

В объектно-ориентированном программировании базовыми единицами программ и данных являются *классы*.

Классы

Класс — это структура данных, которая может содержать в своем составе переменные, функции и процедуры. Переменные, в зависимости от назначения, называются *полями (field)*, или *свойствами*. Процедуры и функции, входящие в состав класса, называются *методами*.

ПРИМЕЧАНИЕ

Классы также называются объектными типами.

В Delphi определен структурный тип `class`. Объявление типа `class` похоже на объявление типа `record`, однако в нем могут содержаться не только поля-переменные, но и методы. Кроме того, в объявлении класса используется ряд специальных зарезервированных слов, определяющих область видимости полей и методов. В отличие от всех остальных типов, тип `class` обязательно должен быть описан как пользовательский тип в разделе `type`, например:

```
type
  TMyClass = class
    field1 : type_definition1;
    field2 : type_definition2;
    procedure method1;
    function method2 : type_definition3;
  end;
```

Затем в разделе `var` может быть объявлена переменная объектного типа:

```
var
  Object1 : TMyClass;
```

ПРИМЕЧАНИЕ

Имена типов в Delphi принято начинать с большой буквы Т. Желательно следовать этому правилу для удобочитаемости программы.

При объявлении класса вначале описываются поля, а затем — методы. Поля класса являются переменными, входящими в состав его структуры. Они предназначены для использования внутри класса. В описании объектного типа присутствуют только заголовки методов. Сами методы описываются в разделе реализации того модуля, в котором объявляется новый объектный тип.

Объекты

Объектом, или *экземпляр*ом класса, называется переменная объектного типа.

Чтобы объект мог обмениваться данными с другими объектами, используются свойства. *Свойства объекта* определяют его состояние. Технология ООП запрещает работать с объектом иначе, чем через методы, то есть изменение состояния объекта производится только через вызов методов этого объекта. Этим существенно ограничивается возможность приведения объекта в недопустимое состояние и/или несанкционированного разрушения объекта.

Взаимодействие между объектами осуществляется с помощью сообщений. Объект может посылать сообщения другим объектам и принимать сообщения от них. *Сообщение* является совокупностью данных определенного типа, передаваемых объектом-отправителем объекту-получателю, имя которого указывается в сообщении. Получатель реагирует на сообщение выполнением некоторого метода, имя которого также может быть указано в сообщении, или никак не реагирует на него.

Объект можно интерпретировать как модель некоторого реального объекта или процесса, которая обладает следующими свойствами:

- ❑ поддается хранению и обработке;
- ❑ способна взаимодействовать с другими объектами и вычислительной средой, посылая сообщения и реагируя на принимаемые сообщения.

В системе ООП совокупность объектов образует среду, в которой вычисления выполняются путем обмена сообщениями между объектами.

ПРИМЕЧАНИЕ

В ООП состояние вычислительной среды разделяется на состояния объектов, что в принципе отличает объектно-ориентированные вычисления от вычислений, заданных в процедурных языках. Процедуры выполняются в общей памяти, в то время как объекты выполняют свои операции с учетом данных, получаемых из сообщений, и собственного состояния.

Основные концепции ООП

Объектно-ориентированное программирование базируется на трех основных принципах: *инкапсуляции*, *наследовании* и *полиморфизме*.

Инкапсуляция

Инкапсуляция — это комбинирование данных с процедурами и функциями, которые манипулируют этими данными. Данные и методы используются для определения содержания и возможностей объекта. Например, окружность описывается координатами центра и радиусом (данные). Кроме того, над окружностью можно проделывать различные действия (методы): вычислять ее длину и площадь ограниченного ею круга, проверять, находится ли некоторая точка внутри данной окружности и т. п.

Класс, описывающий объект «окружность», может выглядеть следующим образом:

```
type
  TCircle = class
    x,y : double;
    r : double;
    function area : double;
    function circumference : double;
    function inside(x,y:double) : boolean;
  end;
```

ПРИМЕЧАНИЕ

Поля и методы, входящие в состав класса, называются членами класса.

Для работы с классом необходимо создать его экземпляр, то есть описать в разделе `var` переменную данного объектного типа:

```
var
  Circle : TCircle;
```

Доступ к полям класса производится точно так же, как доступ к полям записи, с помощью одного из двух способов:

- ☐ указанием имени соответствующего поля после имени экземпляра класса через точку;
- ☐ использованием оператора `with`.

Например, для того чтобы задать координаты центра окружности и ее радиус, можно использовать следующие фрагменты кода:

```
...
Circle.x:=5;
Circle.y:=20;
Circle.r:=10;
...
```

или:

```
...
with Circle do begin
  x:=5;
  y:=20;
  r:=10;
end;
...
```

Аналогичным образом производится и вызов методов. Например, чтобы рассчитать площадь окружности, требуется следующая строка:

```
A:=Circle.area;
```

ПРИМЕЧАНИЕ

Обратите внимание, что методу `area` не нужно передавать никаких данных. Подразумевается, что метод применяется к экземпляру класса, внутри которого он определен. Таким образом, для расчета площади метод `area` использует данные, содержащиеся в поле `r` данного экземпляра класса.

Инкапсуляция позволяет обеспечить защиту данных от внешнего вмешательства или неправильного использования. Такая возможность обеспечивается разделением доступа к данным и методам объекта, которые могут обладать различной степенью доступности: от общедоступных до таких, которые доступны только из методов самого объекта. Обычно открытые члены класса используются для того, чтобы обеспечить контролируемый интерфейс с его закрытой частью.

Наследование

Наследование — это возможность использования уже определенных классов для построения иерархии классов, производных от них. Новый, или *производный*, класс может быть определен на основе уже имеющегося (базового) класса. При этом новый класс сохраняет все свойства старого: данные объекта базового класса включаются в данные производного объекта, а методы базового класса могут быть вызваны для объекта производного класса, причем они будут выполняться над данными включенного в него объекта базового класса. Иначе говоря, новый класс наследует как данные старого класса, так и методы их обработки.

Например, на основе класса, описывающего объект «окружность», можно создать класс, описывающий объект «кольцо». Причем часть свойств и методов у этих объектов будут общими: координаты центра, радиус внешней окружности, метод расчета длины внешней окружности. Поэтому при объявлении класса «кольцо» не нужно заново описывать эти свойства и методы:

```
TRing = class(TCircle)
  r2 : double;
  function area : double;
  function circumference2 : double;
  function inside(x,y:double) : boolean;
end;
```

В объявлении класса `TRing` указываются функции расчета площади кольца и определения попадания некоторой точки с заданными координатами внутрь кольца. Хотя имена этих методов совпадают с именами соответствующих методов для класса `TCircle`, их реализация должна быть иной, так как они применяются к разным геометрическим фигурам. Если имена методов, объявляемых в дочернем классе, совпадают с именами полей или методов родительского класса, то говорят, что они *перекрываются*. В зависимости от типа методов

результаты перекрытия методов будут различными. Тип метода определяется служебным словом после объявления метода.

В Delphi при объявлении объектного типа имя наследуемого класса указывается в круглых скобках после слова `class`. По умолчанию считается, что класс, определяемый пользователем, является наследником от класса `TObject`, поэтому следующие объявления идентичны:

```
TMyClass = class
TMyClass = class(TObject)
```

Для поддержки классов .NET с запретом наследования было добавлено ключевое слово `sealed`. Класс, объявленный как `sealed`, не может иметь наследников. Например:

```
TMyClass = class [abstract | sealed] (TObject)
```

ПРИМЕЧАНИЕ

Сведения о новой платформе Microsoft .NET приведены в следующей главе этой книги.

Полиморфизм

Полиморфизм дает возможность определения единственного имени для действия (процедуры или функции), применимого одновременно ко всем объектам иерархии наследования, причем для каждого объекта учитываются особенности реализации данного действия.

На практике это означает способность объектов выбирать метод, исходя из типа данных. Например, ранее мы рассмотрели пример класса `TCircle` и дочернего от него класса `TRing`. Согласно правилу соответствия типов, принятому в Delphi, переменные дочернего класса всегда совместимы с переменными класса-предка, причем совместимость односторонняя: переменной класса-предка можно присвоить значение переменной дочернего класса, но не наоборот. Таким образом, если имеется какая-либо процедура, для которой формальным параметром является переменная класса `TCircle`, то в данную процедуру можно передать в качестве фактического параметра переменную типа `TRing`. Причем классы `TCircle` и `TRing` имеют методы с одинаковым названием, но по-разному выполняемые. Концепция полиморфизма подразумевает, что внутри процедуры будут вызываться методы, соответствующие не типу формальной переменной, а типу реально переданной переменной.

Реализация концепции полиморфизма означает, что можно создать общий интерфейс для группы близких по смыслу действий. Достоинством полиморфизма является то, что он помогает снижать сложность программ, разрешая использование единого интерфейса для единого класса действий.

Поля, свойства и методы

Класс является сложной структурой данных, объединяющей переменные, функции и процедуры в одном типе данных. Переменные, входящие в состав класса, называются полями. Процедуры и функции класса обычно называются методами. Свойства класса представляют собой поля, обращение к которым производится

через специальные методы. Свойства позволяют реализовать важный принцип объектно-ориентированного программирования, называемый скрытием данных.

Поля

Поля класса представляют собой переменные, объявленные внутри класса. Фактически поля класса аналогичны полям записи. Объявление полей класса должно предшествовать объявлению методов и свойств. Например, класс, содержащий одно поле и один метод, описывается следующим образом:

```
TSampleClass = class(TObject) // Объявление нового класса
  FSample : integer;          // Поле класса
  procedure SampleMethod;     // Метод класса
end;
```

Свойства

Прямое обращение к полям, определяющим состояние объекта, противоречит принципам объектно-ориентированного программирования. Поэтому для обмена данными с другими объектами используются *свойства*, обращение к которым может выполняться не напрямую, а только через соответствующие методы. В этом и заключается отличие свойств от полей, к которым можно обращаться непосредственно.

Для объявления *свойств* используется служебное слово `property`. Так как свойство может обмениваться данными только через соответствующие методы, то при объявлении свойства обычно указываются три элемента: свойство и два метода, обеспечивающие обращение к нему (чтение и запись):

```
TSampleClass = class(TObject) // Класс со свойством
  FSample : integer;
  procedure SetProp : TPropType; // Метод записи
  function GetProp(NewValue : TPropType); // Метод чтения
  // Объявление свойства
  property SampleProp : TPropType read GetProp write SetProp;
end;
```

Прокомментируем особенности использования свойств.

- В объявлении свойства после служебного слова `read` указывается имя метода, обеспечивающего чтение значения свойства, а после директивы `write` — имя метода, изменяющего значение свойства. Чтение и запись значения свойства могут производиться только через некоторое промежуточное поле. Например, метод, обеспечивающий чтение, может выглядеть следующим образом:

```
function TSampleClass.GetProp : TPropType;
begin
  Result:=SampleField;
end;
```

А метод, обеспечивающий запись:

```
procedure TMyClass.SetProp(Value : TPropType);
begin
```



```
SampleField:=Value;
end;
```

ВНИМАНИЕ

Обращения к свойствам в методах GetProp и SetProp приведет к ошибке переполнения стека.

- ❑ При обращении к свойству класса нет необходимости в явном виде вызывать методы, обеспечивающие чтение значения свойства и/или изменение его значения. Синтаксически обращение к свойству может выглядеть точно так же, как и обращение к полю (но при этом следует помнить, что эта операция будет выполняться через соответствующие методы):

```
...
SampleObject.SampleProp:=NewValue;
...
Value:= SampleObject.SampleProp;
```

- ❑ Для обращения к свойству не обязательно использовать методы. Вместо имен методов после слов read и write в объявлении свойства можно указать просто имя поля (тип поля обязательно должен соответствовать типу свойства):

```
TSampleClass = class(TObject)
  FSample : integer;
  property SampleProp : TPropType read FSample
    write FSample;
end;
```

ПРИМЕЧАНИЕ

Для записи предпочтительнее использовать метод, так как это позволит контролировать корректность изменения значения свойства (например, попадание величины в допустимый диапазон).

- ❑ Если в объявлении свойства указан только метод (или поле), обеспечивающий чтение, то данное свойство предназначено *только для чтения* (read only). Поэтому изменить его значение в процессе выполнения программы нельзя. Аналогично, если указан только метод, обеспечивающий запись, то значение свойства при выполнении программы нельзя считывать; данное свойство является свойством *только для записи* (write only).
- ❑ Свойству может быть присвоено значение по умолчанию. Для этого используется служебное слово default. Значение, указанное после слова default, присваивается свойству при создании экземпляра объекта:

```
TSampleClass = class(TObject)
  SFld : integer;
  property SProp : integer read SFld write SFld; default 10;
end;
```

- ❑ Свойство может быть *векторным*. В этом случае объявление свойства напоминает объявление массива:

```
property VProp[index : integer] :
  TPropType read GetProp write SetProp;
```

Методы, читающие и изменяющие значения такого свойства, обязательно должны иметь параметр с тем же именем и того же типа, что и индекс свойства. В методе, обеспечивающем чтение, данный параметр должен быть единственным; в методе, изменяющем значение свойства, индекс должен быть первым передаваемым параметром:

```
function GetProp(index : integer) : TPropType;  
procedure SetProp(index : integer; Value : TPropType);
```

Доступ к векторным полям может производиться только посредством методов — использовать в описании свойства векторных полей после ключевых слов `read` и `write` недопустимо.

- Для векторных свойств директива `default` имеет иной смысл, чем для скалярных. Если в объявлении свойства присутствует директива `default`, то это означает, что при обращении к данному векторному свойству индекс можно писать сразу после имени объекта, не указывая имя свойства в явном виде. Например, пусть в описании класса указано свойство `VProp` с директивой `default`:

```
type  
  TSampleClass = class(TObject)  
  ...  
  property VProp[index:integer] :  
    TPropType read GetP write SetP; default;  
  ...  
end;  
var SampleObject : TSampleClass;  
...
```

Тогда обращение к этому свойству может быть выполнено двумя путями:

```
SampleObject.VProp[k]:=NewValue;  
SampleObject[k]:=NewValue;
```

Методы

Методы предназначены для манипулирования данными, входящими в состав класса. Фактически методы представляют собой обычные процедуры и функции, которым разрешен доступ ко всем полям класса.

Методы объявляются в описании класса после объявления полей. Существуют несколько типов методов, различающихся по механизму наследования.

Статические методы

Статические методы при их переопределении в классах-потомках полностью перекрываются. Для статических методов можно полностью изменить объявление метода. По умолчанию методы считаются статическими, поэтому для их объявления не требуется никаких дополнительных команд. Определение адресов статических методов производится на этапе компиляции. При вызове статического метода выполняется процедура или функция, определяющаяся только типом объектной переменной. Тип самого объекта, на который данная переменная ссылается, не принимается во внимание. Поэтому использование статических методов не позволяет реализовать концепцию полиморфизма.

ПРИМЕЧАНИЕ

В версии Delphi 8 for .NET статическими могут быть не только методы, но также поля и свойства.

Виртуальные и динамические методы

При обращении к виртуальным и динамическим методам вызываемая процедура или функция определяется только в момент обращения. Такой механизм называется *поздним связыванием*. Именно виртуальные и динамические методы позволяют в полной мере реализовать концепцию полиморфизма. При объявлении виртуальных и динамических методов используются директивы `virtual` и `dynamic` соответственно. Остановимся на основных возможностях этих методов.

- ❑ При вызове виртуальных и динамических методов выполняемая процедура или функция определяется по типу фактического параметра. Для этого используется таблица виртуальных методов (Virtual Method Table, VMT) в случае виртуальных методов и таблица динамических методов (Dynamic Method Table, DMT) в случае динамических.
- ❑ Таблица виртуальных методов создается для каждого объектного типа. В ней содержатся адреса виртуальных методов этого объектного типа. Независимо от количества переменных данного объектного типа для него создается только одна таблица VMT. При вызове виртуального метода каким-либо экземпляром местонахождение кода реализации данного метода определяется по таблице VMT для типа данного экземпляра. Взаимосвязь между VMT и экземпляром класса устанавливается при инициализации объекта. Так как адреса виртуальных методов при их вызове определяются через VMT объекта, то гарантированно будут использоваться методы, соответствующие типу объекта.
- ❑ В таблице VMT содержатся адреса всех виртуальных методов класса — как унаследованных от предков, так и переопределенных в данном классе. Поэтому виртуальные методы вызываются достаточно быстро, но требуют большого объема памяти. В отличие от виртуальных методов, динамические методы вызываются медленнее, но зато занимают меньше памяти. Это объясняется тем, что в таблице динамических методов класса хранятся адреса только тех динамических методов, которые определены в данном классе. При вызове динамического метода адрес кода его реализации сначала ищется в таблице DMT, относящейся к типу данного экземпляра. Если адрес не найден, производится поиск в таблицах DMT всех классов-предков в порядке иерархии.
- ❑ Для перекрытия виртуальных и динамических методов используется служебное слово `override`.

Рассмотрим пример использования виртуального метода для реализации концепции полиморфизма. Вернемся к рассмотренным ранее классам «окружность» и «кольцо». С целью реализации концепции полиморфизма объявим

методы класса «окружность», которые переопределяются в классе «кольцо», как виртуальные:

```
TCircle = class
  x,y : double;
  r : double;
  function area : double; virtual;
  function circumference : double;
  function inside(x,y:double) : boolean; virtual;
end;

TRing = class(TCircle)
  r2 : double;
  function area : double; override;
  function circumference2 : double;
  function inside(x,y:double) : boolean; override;
end;
```

При таком определении эти классы являются полиморфными.

Виртуальный метод может быть объявлен с модификатором `final`. Такие методы не могут быть перекрыты в потомках класса.

Абстрактные методы

Абстрактные методы определяются в классе, но не выполняют никаких действий. Абстрактные методы должны быть переопределены в потомках класса. Для объявления абстрактного метода используется директива `abstract`. При этом не требуется написание кода реализации, достаточно только лишь объявления в описании класса.

Поскольку абстрактные методы обязательно должны быть переопределены, то абстрактными можно объявлять только виртуальные и динамические методы. Попытка вызвать абстрактный метод приведет к возникновению исключительной ситуации `EAbstractError` (исключительные ситуации будут обсуждаться позже).

Абстрактные методы используются при построении иерархии объектов, позволяя задавать на верхних уровнях иерархии методы, не привязанные к конкретным типам данных. Например, класс «окружность» может быть создан на основе класса «точка». Для объекта «точка» методы определения площади и длины не имеют никакого смысла. Однако данные методы можно описать в классе `TPoint` как абстрактные, при этом будет удобнее использовать его в качестве базового типа, совместимого со всеми дочерними типами:

```
TPoint = class
  x,y : double;
  function area : double; virtual; abstract;
  function circumference : double; virtual; abstract;
end;

TCircle = class(TPoint)
  r : double;
  function area : double; override;
  function circumference : double; override;
end;
```

Перекрытие методов

Если в классе-потомке объявляются методы, имена которых совпадают с именами методов в родительском классе, то говорят, что эти методы *перекрываются*.

Статические методы полностью перекрываются в классе-потомке при переопределении. При этом можно изменять количество и типы параметров в заголовке метода.

При перекрытии виртуальных и динамических методов следует сохранять количество и тип параметров в заголовках методов.

Перекрываемый метод из родительского класса может быть вызван внутри методов класса-потомка. Для этого используется специальная директива `inherited`.

Конструкторы и деструкторы

Объекты в Delphi в действительности являются указателями (то есть представляют собой переменные, содержащие адреса областей памяти, в которых находятся объекты). Причем ресурсы под объект выделяются динамически. Поэтому перед использованием объекта необходимо выполнить его *инициализацию* — выделить необходимую память под объект. Для инициализации объекта используются специальные методы, называемые *конструкторами*. Для объявления конструктора указывается специальное зарезервированное слово `constructor`. В остальном конструктор ничем не отличается от обычного метода. Как правило, конструктор имеет имя `Create`. По завершении работы с объектом следует освободить занятые им ресурсы. Для этого используется *деструктор*. Для объявления деструктора указывается зарезервированное слово `destructor`.

Перегружаемые методы

В Delphi допускается определение в одном классе нескольких методов с одинаковыми именами, но разными списками параметров. Такие методы называются *перегружаемыми*, они объявляются при помощи директивы `overload`. С их помощью можно присваивать одинаковые имена родственным методам. Выбор конкретной версии метода, применимой в данных обстоятельствах, осуществляется компилятором.

При перегрузке метода каждая его версия может выполнять любые действия. Перегружаемые методы вовсе не обязательно должны быть связаны между собой. Однако с точки зрения хорошего стиля программирования перегрузка методов подразумевает взаимосвязь между ними.

Вложенные типы данных

Язык Delphi (для версии 8) поддерживает вложенные типы данных. Синтаксис объявления вложенного типа:

```
type
  className = class [abstract | sealed] (ancestorType)
    memberList
  type
    nestedTypeDeclaration
    memberList
  end;
```

Области видимости

В классах языка Delphi существует возможность разграничивать области видимости полей и методов. Область видимости задается специальным зарезервированным словом. Различаются пять вариантов областей видимости.

- ❑ *Общая область видимости* задается директивой `public`. Она не накладывает никаких ограничений на видимость. Поля и методы категории `public` доступны для других объектов в любом модуле, который ссылается на модуль, содержащий описание класса.
- ❑ *Личная область видимости* задается директивой `private`. С ее помощью реализуется минимальная область видимости. Вне модуля, в котором определен класс, элементы категории `private` недоступны. Использование области видимости `private` позволяет полностью скрыть особенности внутренней реализации класса.
- ❑ *Защищенная область видимости* задается директивой `protected`. Элементы категории `protected` помимо модуля, в котором определен класс, доступны в классах, являющихся потомками данного, даже если они определяются в других модулях.
- ❑ *Опубликованная область видимости* задается директивой `published`. Элементы категории `published` предназначены для создания интерфейса визуального программирования. Во время выполнения программы свойства, указанные в секции `published`, аналогичны свойствам категории `public`, то есть не имеют ограничений на видимость. Кроме того, свойства, объявленные в разделе `published`, видны из среды разработки Delphi.

Объектная модель Delphi 7 и более ранних версий позволяет получать доступ к защищенным и личным членам класса из других классов, объявленных в том же модуле. Директива `strict`, появившаяся в версии 8, запрещает такое поведение. Например, объявим класс, чтобы процедура `Dispose` не могла быть вызвана другими классами, даже если они будут объявлены в том же модуле:

```
TWinForm = class(System.Windows.Forms.Form)
    strict private
    strict protected
        procedure Dispose(Disposing: Boolean); override;
end;
```

- ❑ *Область автоматизации* задается директивой `automated`. Элементы с данной областью видимости используются для создания интерфейсов COM (Component Object Model).

Области видимости указываются не для каждого элемента класса — директива задает область видимости для всех следующих за ней элементов, пока не будет указана другая директива.

В классах-потомках можно изменять области видимости методов и свойств, причем только в сторону расширения. При описании дочернего класса для изменения области видимости метода или свойства достаточно упомянуть их

в соответствующей секции. Пусть, например, в классе `TSample1` имеется защищенное свойство `SampleProp`:

```
TSample1 = class(TObject)
private
  Field : TPropType;
protected
  property Sample Prop : TPropType read Field;
end;.
```

Тогда в потомке данного класса `TSample2` можно расширить область видимости этого свойства и сделать его общим, просто упомянув в секции `public`:

```
TSample2 = class(TSample1)
public
  property SampleProp;
end;
```

Обработка исключительных ситуаций

Исключительная ситуация — это событие, прерывающее нормальное выполнение программы. Иначе говоря, исключительная ситуация является ошибкой, возникающей во время выполнения программы. В языке Delphi существуют специальные средства для обработки исключительных ситуаций.

Исключительные ситуации, возникающие во время выполнения программы, описываются в языке Delphi с помощью специального объектного типа `Exception`. На базе этого типа определен ряд дочерних классов, соответствующих наиболее типичным исключительным ситуациям. Имена классов-потомков `Exception` начинаются с буквы `E`.

В Delphi определены две конструкции для работы с исключительными ситуациями: `try... except` и `try... finally`.

Блок `try... except`

Блок `try... except` применяется для реакции на конкретную исключительную ситуацию. Использование блока `try... except` выглядит следующим образом:

```
try
  оператор_1;
  оператор_2;
  ...
except
  on исключительная_ситуация_1 do оператор_3;
  on исключительная_ситуация_2 do оператор_4;
  ...
  else оператор_N;
end;
```

Если при выполнении операторов, расположенных в разделе `try`, не возникает исключительная ситуация, то обращения к разделу `except` вообще не происходит. Если же в разделе `try` возникает исключительная ситуация, то управление сразу передается разделу `except`. Раздел `except` содержит набор операторов `on... do`, определяющих реакцию на исключительные ситуации. Между ключевыми словами `on` и `do` указывается имя класса исключительной ситуации. Оператор, расположенный после слова `do`, предназначен для ее обработки.

ПРИМЕЧАНИЕ

После обработки исключительной ситуации управление не передается назад в раздел try.

Блок try... finally

Блок try... finally используется в тех случаях, когда необходимо выполнить некоторые действия даже в случае возникновения исключительной ситуации (например, освободить занятую память). Блок try... finally имеет следующий вид:

```
try
    оператор_1;
    оператор_2;
    ...
finally
    оператор_3;
    оператор_4;
    ...
    оператор_N;
end;
```

В данной конструкции сначала выполняются операторы, расположенные в разделе try. Если при их выполнении не возникло исключительной ситуации, то выполняются операторы, расположенные в разделе finally. Если же при выполнении операторов в разделе try возникает исключительная ситуация, то управление сразу передается первому оператору раздела finally.

Эта конструкция не обрабатывает исключительную ситуацию, а лишь служит для защиты выделенных ресурсов, позволяя освободить их даже в случае возникновения исключительной ситуации.

Исключительную ситуацию можно программно вызвать с помощью специального оператора raise.

Глава 8

Средство быстрой разработки приложений Delphi

Быстрая разработка приложений (RAD — Rapid Application Development) основывается на визуализации процесса создания программного кода. Рассматриваемая технология является инструментальным программным обеспечением, которое предоставляет программистам средства, ускоряющие разработку необходимого прикладного процесса, сокращающие работу по модификации уже готовой прикладной программы и внесению в нее необходимых дополнений или изменений. С целью максимального упрощения перечисленных действий используются графические инструментальные средства.

Не следует сводить RAD только к визуальной генерации пользовательского интерфейса. Возможности этой технологии гораздо шире «простого» набора процедур, включающих помещение управляющих элементов на «формы» с последующей установкой их свойств. Средства быстрой разработки приложений основываются на *компонентной* архитектуре. При этом *компоненты* являются объектами, объединяющими данные и методы, а также свойства. Свойства, с одной стороны, позволяют работать с данными так же, как с членами классов, а с другой стороны, скрывают за операциями чтения/записи вызовы методов, переводя операции над объектами на более высокий уровень абстракции.

Компоненты могут быть как визуальными, так и невидимыми; атомарными и контейнерными (содержащими другие компоненты); низкоуровневыми (системными) и высокоуровневыми.

Средства визуального программирования

Визуальное проектирование пользовательского интерфейса предоставляет возможность выбора отдельных компонентов из палитры с последующим размещением их в нужном месте. Для обозначения инструментов визуального проектирования интерфейса используется широкий набор терминов, включающих: *конструктор компоновки, конструктор форм, визуальный композиционный редактор,*

визуальный редактор, проектировщик экрана, экранный редактор, проектировщик форм, конструктор графического пользовательского интерфейса.

Процедура разработки интерфейса средствами RAD сводится к набору последовательных операций, включающих:

- ☐ размещение компонентов интерфейса в нужном месте;
- ☐ задание моментов времени их появления на экране;
- ☐ настройку связанных с ними атрибутов и событий.

В идеале среда визуальной разработки должна позволять быстро перетаскивать компоненты с помощью мыши и задавать значения изменяемых параметров. Если на выполнение каждой операции будет уходить ощутимое количество времени, то проектирование сложного интерфейса превратится в очень длительную процедуру. Эффективность визуального программирования определяется не столько наличием визуальных компонентов, сколько их взаимосвязью и взаимодействием с традиционными средствами.

Интегрированная среда разработки является средством, с помощью которого выполняются проектирование, отладка, тестирование и дальнейшее распространение прикладных программ. Для повышения эффективности данного процесса каждое из средств (конструкторы, отладчики и т. д.) должно быть реализовано на очень высоком уровне.

Даже если среда не содержит достаточного количества требуемых компонентов, она все равно будет востребована, если позволяет использовать имеющиеся на рынке средства, альтернативные отсутствующим в ней.

В настоящее время существует большое количество средств визуального программирования, основанных на различных алгоритмических языках. В нашей стране они широко представлены фирмами Microsoft и Borland. Каждая из них предлагает несколько сред визуального программирования.

Среда разработки Delphi

Система визуального программирования Delphi (в данной книге рассматривается версия Turbo Delphi) фирмы Borland позволяет в полной мере реализовать современные концепции программирования, включая:

- ☐ объектно-ориентированный подход;
- ☐ визуальные средства быстрой разработки приложений (Rapid Application Development), основанные на компонентной архитектуре;
- ☐ использование компиляции, а не интерпретации (компилируемые приложения обладают меньшей ресурсоемкостью, кроме того, скорость вычислений в компилируемых приложениях часто значительно больше по сравнению с интерпретируемыми);
- ☐ возможность работы с базами данных универсальными методами.

В среде Delphi используется язык Delphi (ранее Object Pascal), предоставляющий возможность полной реализации основных принципов ООП (инкапсуляция, наследование, полиморфизм) и обладающий встроенной обработкой исключительных ситуаций. Компонентная архитектура Delphi является пря-

мым развитием поддерживаемой объектной модели. Все компоненты являются объектными типами (классами), обладающими возможностью неограниченного наследования. Компоненты Delphi поддерживают РМЕ-модель (Property, Method, Events — свойства, методы, события), позволяющую изменять поведение компонентов без необходимости создания новых классов.

В поставку Turbo Delphi входят три механизма взаимодействия с базами данных:

- ❑ dbGo — обеспечивает программный объектно-ориентированный механизм работы с базами данных на основе технологии Microsoft ADO;
- ❑ dbExpress — набор драйверов к базам данных, который обеспечивает быстрый доступ к различным SQL-серверам;
- ❑ Borland Database Engine (BDE) — обеспечивает единообразную работу с локальными данными (Paradox, dBase, FoxPro) и серверами БД (Oracle, Sybase, MS SQL Server, InterBase и т. д.).

ПРИМЕЧАНИЕ

Актуальным механизмом работы с базами данных в настоящее время является dbGo.

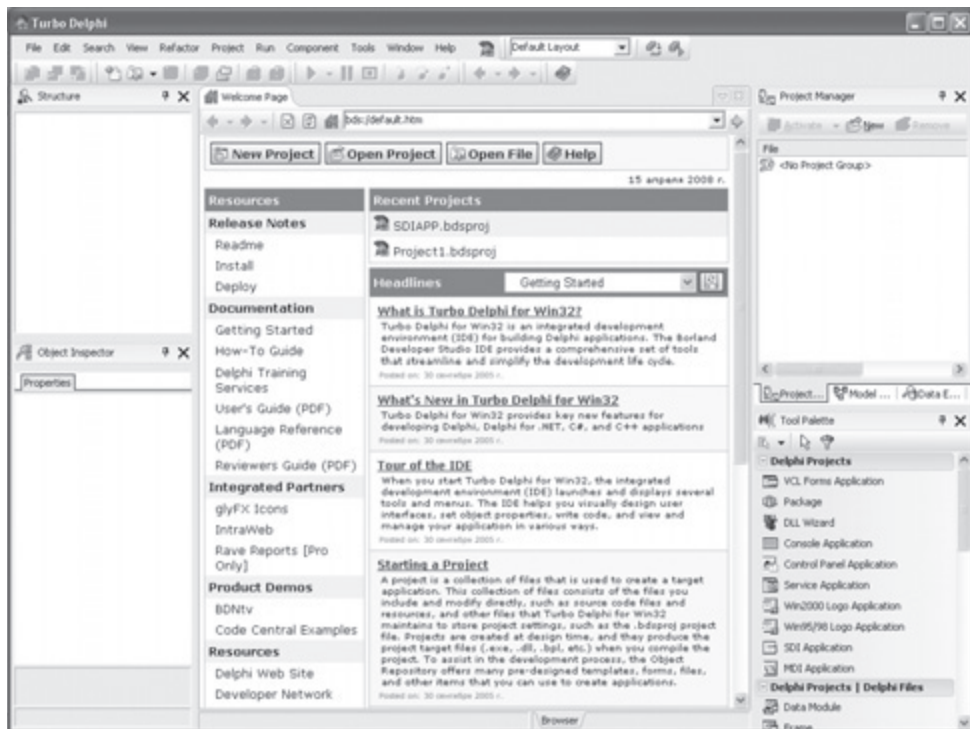


Рис. 8.1. Внешний вид основных элементов интегрированной среды разработки Delphi

После запуска интегрированной среды разработки Turbo Delphi вы увидите окно, показанное на рис. 8.1. Большую часть окна занимает Страница приветствия (Welcome Page). Вы можете:

- ❑ открыть различные разделы документации по Delphi, воспользовавшись разделом **Documentation**;
- ❑ перейти на сайты фирмы Borland, посвященные Delphi с помощью разделов **Resources** и **Product Demos**;
- ❑ перейти к различным разделам справочной системы Delphi, воспользовавшись выпадающим списком **Headlines**;
- ❑ открыть один из проектов, с которыми вы работали за последнее время, с помощью списка **Recent Projects**;
- ❑ или создать новый проект, открыть существующий проект, открыть файл или перейти к окну справочной системы, воспользовавшись соответственно кнопками **New Project**, **Open Project**, **Open File**, **Help**.

Главное окно Delphi IDE

Для начала работы щелкнем на кнопке **New Project**, чтобы начать работу с новым проектом. В появившемся диалоговом окне **New Items** в списке **Item Categories** выберем сначала элемент **Delphi Projects**, а затем в списке справа выберем элемент **VCL Forms Application** и нажмем кнопку **OK**. Перед нами теперь будет главное окно Delphi IDE, в котором открыт новый проект.

Главное окно Delphi состоит из семи основных разделов:

- ❑ *главное меню* располагается непосредственно под заголовком главного окна и позволяет получить доступ ко всем функциям, обеспечиваемым IDE;
- ❑ *панель инструментов* представляет собой группу кнопок, используемых для быстрого доступа к ряду команд главного меню. Настройка панели инструментов может производиться пользователем в соответствии с личными предпочтениями;
- ❑ *панель структуры* (**Structure**) предназначена для отображения структуры исходного кода или HTML-кода, который открыт в данный момент в редакторе кода (**Code Editor**). В этой панели структура проекта отображается в виде иерархической структуры, и двойной щелчок на элементе структуры приведет к тому, что в редакторе кода будет осуществлен переход к объявлению этого элемента. Можно изменить внешний вид и содержимое панели **Structure** с помощью диалогового окна **Options** в разделе **Environment Options** ▶ **Explorer**;
- ❑ *панель инспектора объектов* (**Object Inspector**) предназначена для задания свойств компонентов и создания обработчиков событий в режиме проектирования. Более подробно данная панель будет рассмотрена ниже;
- ❑ *панель менеджера проекта* (**Project Manager**) позволяет организовать и просматривать файлы проекта, такие как: формы, исполняемые файлы, объекты и файлы библиотек. Также с помощью менеджера проекта вы можете объ-

единить несколько связанных проектов в одну группу и компилировать их одновременно;

- *панель палитры инструментов* (Tool Palette) используется при разработке приложения. Содержимое этой панели зависит от текущего режима. Например, если происходит редактирование формы, то панель отображает компоненты, которые можно на ней разместить. А в режиме просмотра кода в редакторе кода панель палитры инструментов отображает различные сегменты кода, которые можно добавить к приложению;
- *редактор кода* представляет собой текстовый редактор, специально приспособленный для редактирования исходных текстов программ. Он поддерживает возможность автоматической генерации части текстов программ, исходя из визуально спроектированной части приложения. В процессе проектирования доступен либо режим визуального проектирования (редактор форм), либо редактирования текста. Также поддерживаются подсветка синтаксиса языка Delphi, минимизации блоков текста, автоматического выбора свойств компонента. Для реализации последней возможности достаточно правильно указать имя компонента и поставить точку, через короткий промежуток времени редактор выдаст список доступных свойств и методов объекта.

Главное меню

Главное меню содержит полный набор команд, необходимых для работы в Delphi. Однако вследствие того, что Delphi является средой визуального программирования, частого обращения к командам главного меню, как правило, не требуется. Полный перечень всех команд главного меню занял бы слишком много места, поэтому здесь приводится описание только некоторых основных команд.

Меню File

Меню File содержит команды, предназначенные для работы с файлами. Часть команд, содержащихся в данном меню, — обычные команды для работы с файлами: Open, Save, Save As, Close, Exit, назначение которых не требует пояснений.

Остальные команды имеют некоторые особенности, поэтому остановимся на них подробнее.

Команды Open Project и Save Project As используются для открытия проектов и их сохранения под другим именем. Под *проектом* в Delphi понимается набор файлов, которые необходимы для создания исполняемого приложения или динамически связываемой библиотеки.

Команды Save All и Close All предназначены для сохранения и закрытия всех открытых файлов, относящихся к активному проекту.

Для создания новых проектов и отдельных файлов используются команды подменю New. Для создания новых проектов и отдельных файлов используются четыре команды: VCL Forms Application — Delphi for Win32, Package — Delphi for Win32, Form — Delphi for Win32, Unit — Delphi for Win32, Other — Delphi for Win32, Customize — Delphi for Win32. Первые четыре команды применяются для создания новых файлов только определенного типа — *приложения* (Application), *пакета* (Package), *формы* (Form) или *модуля* (Unit).

При выполнении команды **File ► New ► Other** открывается окно диалога **New Items** (рис. 8.2), которое позволяет создать файл любого доступного типа.

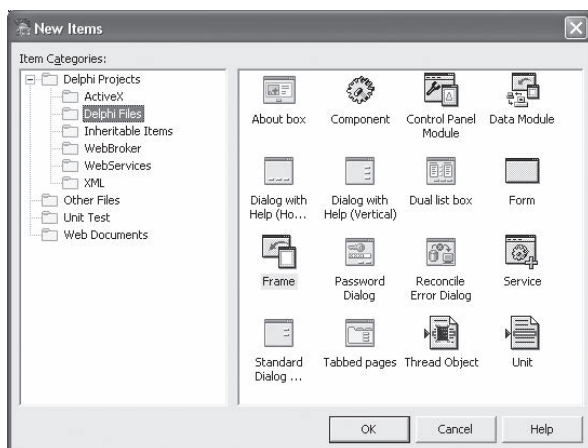


Рис. 8.2. Окно диалога **New Items** обеспечивает доступ к хранилищу объектов

В этом окне диалога предлагается в том числе набор объектов, содержащихся в *хранилище объектов* (Object Repository). Система визуального программирования Delphi позволяет разрабатывать различные проекты. Среди них — приложения, динамические библиотеки, компоненты Delphi, элементы ActiveX и т. п. Хранилище объектов содержит шаблоны кода, которые используются в качестве основы при разработке сложных объектов или приложений.

Шаблоны объектов, содержащиеся в хранилище, разделены на несколько групп, которые отображаются на разных страницах окна **New Items**. При выборе элемента **Inheritable Items** активизируется один из трех переключателей, расположенных в нижней части окна. Положение переключателя задает способ использования выбранного шаблона.

- ❑ **Copy** — в проект добавляется копия выбранного объекта. В этом варианте изменения, вносимые в объект, содержащийся в проекте, не отражаются на объекте, расположенном в хранилище. Последующие изменения, вносимые в содержащийся в хранилище шаблон объекта, не влияют на объект, скопированный в проект.
- ❑ **Inherited** — в проект добавляется копия выбранного объекта. Изменения, вносимые во встроенный в проект объект, не отражаются на шаблоне, находящемся в хранилище. Однако изменения, вносимые в объект, содержащийся в хранилище, изменяют и объект, используемый в проекте.
- ❑ **Use** — используется для модификации объектов, находящихся в хранилище. Изменения объекта в проекте сразу вносятся в шаблон объекта из хранилища (и отражаются во всех других проектах, в которых используется данный шаблон).

Продолжим обсуждение команд меню **File**.

Команда **Use Unit** используется для подключения к проекту дополнительных модулей, открытых в редакторе кода.

Команда Print предназначена для печати формы или программы. Конкретный вариант ее выполнения зависит от типа активного окна — редактора форм или редактора кода.

Окно диалога Print Form (Печать формы) содержит переключатель, предоставляющий выбор одного из трех вариантов:

- ☐ Proportional (based on PixelsPerInch) — при печати масштаб формы изменяется в зависимости от свойства формы PixelsPerInch;
- ☐ Print to fit page — масштаб формы подстраивается под размеры страницы;
- ☐ No scaling — форма при печати не масштабируется.

Окно диалога Print Selection печати программного кода содержит следующие установки:

- ☐ Print selection block — при выборе данного параметра печатается не весь файл, а только выделенный блок;
- ☐ Header/page number — вверху каждой страницы печатаются имя файла, текущая дата и номер страницы;
- ☐ Line numbers — при печати нумеруются строки программы;
- ☐ Syntax print — слова, выделяемые в редакторе кода, выделяются и при печати;
- ☐ Use color — использование цвета для отображения выделяемых слов;
- ☐ Wrap lines — перенос длинных строк;
- ☐ Left margin — задает количество символов, определяющее ширину левого поля.

Меню Edit

Меню Edit (рис. 8.3) содержит ряд стандартных команд, используемых в приложениях ОС Windows для редактирования: Cut (Вырезать), Copy (Копировать), Paste (Вставить), Delete (Удалить), Select All (Выделить все), Undo (Отменить), Redo (Повторить). Остальные команды используются для разработки форм и дублируют команды контекстного меню редактора форм.

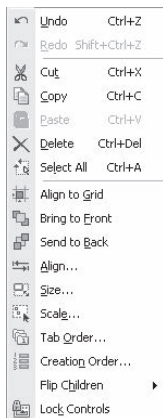


Рис. 8.3. Меню Edit

Меню Search

Данное меню содержит некоторые распространенные команды, предназначенные для работы с текстами: Find (Найти), Replace (Заменить), Search Again (Повторный поиск), Go to Line Number (Перейти на строку с номером...), а также несколько специальных команд:

Команда Find in Files предназначена для поиска текстовых строк в нескольких файлах. При выборе этой команды открывается окно диалога Find Text (рис. 8.4).

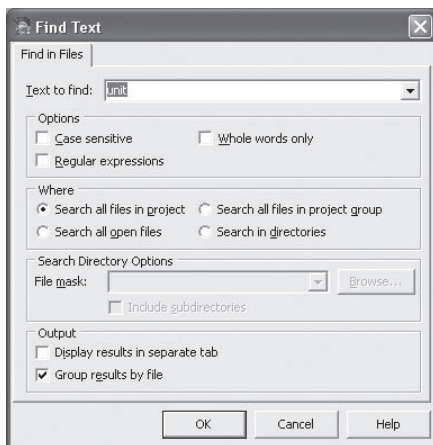


Рис. 8.4. Окно диалога Find Text

Данное окно диалога содержит ряд установок, определяющих параметры поиска. Часть из них, расположенных в разделе Options, включает три флажка:

- ☐ Case sensitive — определяет, различать ли верхний/нижний регистр при поиске;
- ☐ Whole words only — задает поиск целого слова;
- ☐ Regular expressions — в строке, заданной для поиска, могут использоваться специальные символы подстановки (например, символ «*» означает, что на его месте может находиться любое количество произвольных символов).

Другая часть настроек, представленная в разделе Where, предназначена для выбора одного из трех альтернативных вариантов:

- ☐ Search all files in project — устанавливает поиск в файлах, относящихся только к текущему проекту;
- ☐ Search all open files — задает поиск во всех открытых файлах;
- ☐ Search in directories — ведет поиск только в текущей директории;
- ☐ Search all files in project group — устанавливает поиск в файлах, относящихся только к текущей группе проектов.

Раскрывающийся список File mask позволяет задать шаблон имен файлов, в которых будет производиться поиск (эта возможность доступна, если выбран

вариант Search in directories). Установка флажка Include subdirectories дает возможность ведения поиска во всех вложенных папках (эта возможность также доступна, только если выбран вариант Search in directories).

Меню View

Команды меню View (рис. 8.5) предназначены для управления отображением информации. Остановимся на назначении наиболее важных из них.

Команда Project Manager открывает панель менеджера проекта (Project Manager), показанную на рис. 8.6, в которой отображаются все файлы, входящие в текущую группу проектов. Менеджер проекта позволяет добавлять проекты в группу, удалять проекты из группы, выбирать активный проект, сохранять отдельные проекты или всю группу в целом. Это окно диалога также может быть использовано для добавления, удаления, открытия и сохранения отдельных файлов, входящих в состав проекта. Все перечисленные действия могут выполняться с помощью кнопок, расположенных в верхней части окна менеджера проектов, либо с помощью контекстного меню, активизирующегося при нажатии правой кнопки мыши на каком-либо модуле, входящем в состав группы проектов или отдельного проекта.

Команды Data Explorer и Model View (рис. 8.7) переключают окно менеджера проектов соответственно в режим управления связями с источниками данных и режим представления проекта в виде диаграмм классов UML. Окно менеджера проектов в режиме Data Explorer позволяет отслеживать взаимодействие между процессором баз данных dbExpress и клиентской базой данных, а также позволяет контролировать подключения к базе данных, транзакции, запросы и т. п.

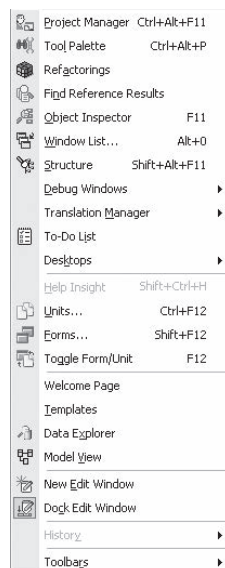


Рис. 8.5. Меню View

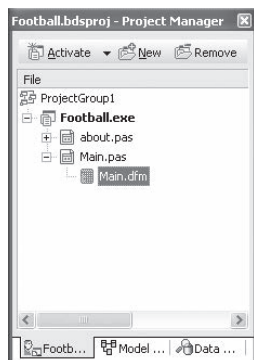


Рис. 8.6. Окно Project Manager

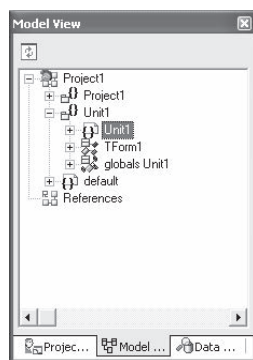


Рис. 8.7. Окно Model View и Data Explorer



ПРИМЕЧАНИЕ

Возможность объединять несколько проектов в одну группу удобно использовать, например, в тех случаях, когда одновременно с приложением разрабатываются динамические библиотеки.

Подменю команды Translation Manager предназначены для разработки приложений с поддержкой нескольких языков.

Команда Object Inspector применяется для активизации инспектора объектов, который используется при разработке форм. Более подробно его назначение будет рассмотрено далее.

Команда To-Do List используется для управления списком задач, которые могут относиться как ко всему проекту в целом, так и к отдельному его модулю. Все текущие задачи отображаются в окне диалога To-Do List (рис. 8.8). Отметьте, что задачи, относящиеся к проекту и к модулям, обозначаются разными пиктограммами. Используя команды контекстного меню, вы можете добавлять задачи в список окна To Do Items, а также удалять и редактировать их. Слева от пиктограммы выполненной задачи устанавливается флажок.

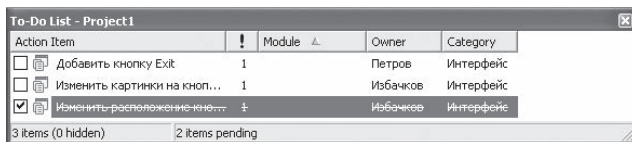


Рис. 8.8. Окно To-Do List

Способ включения задачи в список зависит от ее типа. В случае если задача относится ко всему проекту, то используется команда Add контекстного меню окна диалога To-Do List. Для отдельного модуля применяется команда Add To-Do Item контекстного меню редактора кода. При выполнении любой из этих команд открывается окно диалога Edit To-Do Item (рис. 8.9). Здесь в поле Text вводится описание назначения задачи, а в трех других полях может быть задан ряд дополнительных параметров: приоритет (Priority), имя исполнителя (Owner), тип задачи (Category).

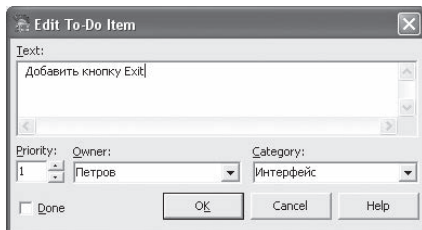


Рис. 8.9. Окно диалога Edit To-Do Items

ПРИМЕЧАНИЕ

Список задач никак не влияет на компиляцию проекта и используется просто как записная книжка, помогая планировать разработку программы и разделять задачи между разными программистами.

Продолжим обзор команд меню View.

Команда **Toolbars ▶ Align** открывает панель инструментов Align, содержащую ряд команд, используемых в редакторе форм при разработке интерфейса.

Команда **Window List** открывает окно, содержащее список всех открытых окон Delphi и позволяющее сделать активным любое из них.

Команды подменю **Debug Windows** используются для отладки приложения.

Группа команд **Desktops** позволяет сохранять текущую конфигурацию среды разработки: информацию о том, какие окна открыты и где они расположены. Кроме того, с помощью команды **View ▶ Desktops ▶ Set Debug Desktop** можно создать специальную конфигурацию **Debug desktop**, которая будет загружаться в режиме отладки проекта и выгружаться при выходе из этого режима.

Команда **Toggle Form/Unit** используется для переключения между окнами редактора кода и редактора форм.

Команда **Units** открывает окно, содержащее список всех модулей, входящих в состав текущего проекта.

Команда **Forms** позволяет открыть список всех форм, содержащихся в проекте.

Команда **New Edit Window** открывает новое окно редактора кода.

Команды подменю **Toolbars** используются для настройки панелей инструментов.

Меню Project

В этом меню содержатся команды для создания и редактирования проектов. Ниже приводится краткое разъяснение их назначения.

Команда **Add to Project** используется для добавления к проекту какого-либо файла (модуля, файла ресурсов и т. п.). Она полностью дублирует команду **Add** контекстного меню менеджера проектов.

Команда **Remove from Project** позволяет удалять из проекта ненужные модули. При выполнении данной команды открывается окно со списком всех используемых модулей, в котором можно выбрать те из них, которые следует удалить из проекта.

Команда **Add to Repository** добавляет текущий проект в хранилище объектов в качестве шаблона.

Команда **View Source** открывает в редакторе кода главный файл проекта.

Команды подменю **Languages** используются при разработке приложений, поддерживающих несколько языков.

Команда **Add New Project** добавляет в текущую группу проектов новый проект. Она является аналогом соответствующей команды контекстного меню менеджера проектов.

Команда **Add Existing Project** добавляет существующий проект в текущую группу проектов. Ее назначение аналогично команде контекстного меню менеджера проектов.

Команда **Information** открывает одноименное окно диалога (рис. 8.10), в котором отображается информация о текущем проекте:

- ❑ **Source compiled** — количество строк программного кода в проекте;
- ❑ **Code size** — размер исполняемого файла или динамической библиотеки (DLL) без отладочной информации;

- ❑ Data size — объем памяти, занимаемый глобальными переменными;
- ❑ Initial stack size — объем памяти, отведенной для хранения локальных переменных;
- ❑ File size — размер выходного файла;
- ❑ Package Used — список всех пакетов, используемых проектом.

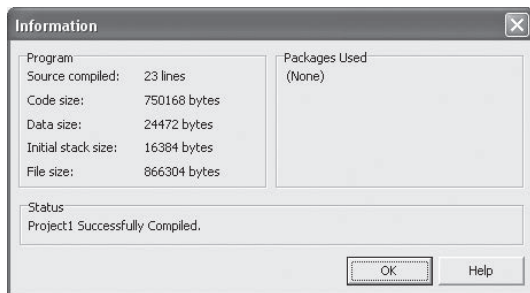


Рис. 8.10. Окно диалога Information

Меню Run

Меню Run содержит ряд команд, предназначенных для запуска и отладки приложения. Часть этих команд будет рассмотрена в главе 15, «Справочная система приложения».

Команда Attach to Process открывает одноименное окно диалога со списком всех программ, работающих в текущий момент (рис. 8.11). Выбранная из этого списка программа загружается в среду Delphi в режиме отладки. При этом она может быть любой (необязательно написанной в Delphi), поскольку ее текст отображается в отладчике в командах ассемблера.

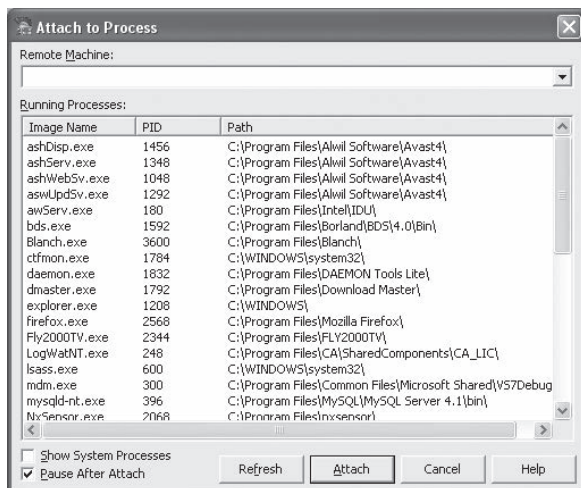


Рис. 8.11. Окно диалога Attach to Process

Команда **Show Execution Point** открывает окно редактора кода и показывает строку программы, выполняемую в данный момент. Эта команда доступна только в режиме пошаговой отладки программы.

Команда **Program Pause** останавливает выполнение запущенной программы, но не закрывает ее.

Команда **Program Reset** завершает работу запущенной программы.

Команды **Inspect**, **Evaluate/Modify** и **Add Watch** используются для просмотра и изменения значений переменных в режиме отладки.

Группа команд **Add Breakpoint** предназначена для установки и снятия точек останова в программе.

Меню Component

Команды меню **Component** используются для создания новых компонентов.

Команда **New VCL Component** используется при разработке нового компонента.

Меню Tools

Меню **Tools** содержит команды, вызывающие диалоговые окна настроек среды Delphi, редактора кода, встроенного отладчика и хранилища объектов. Кроме того, в данном меню содержатся команды, позволяющие запускать некоторые внешние программы (утилиты), причем имеется возможность настройки списка вызываемых программ.

Панели инструментов

Панели инструментов (рис. 8.12) содержат кнопки, обеспечивающие быстрый доступ к командам главного меню.

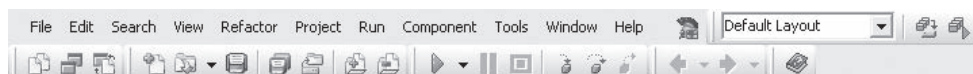


Рис. 8.12. Панели инструментов Delphi IDE

Главное окно Delphi включает семь панелей инструментов: **Standard**, **View**, **Debug**, **Custom**, **Desktops**, **Personality** и **Browser**, каждая из которых может закрываться и открываться с помощью команды главного меню **View** ► **Toolbars** ► <Название панели> или с помощью контекстного меню панели инструментов.

Содержание каждой панели инструментов может настраиваться пользователем с помощью окна диалога **Customize** (рис. 8.13), открываемого командой **View** ► **Toolbars** ► **Customize** главного меню или командой **Customize** контекстного меню.

Окно диалога **Customize** содержит три вкладки: **Toolbars**, **Commands** и **Options**:

- ☐ вкладка **Toolbars** содержит набор флажков, используемых для включения и выключения отображения соответствующих панелей инструментов;
- ☐ вкладка **Commands** используется для настройки содержания панели инструментов. На ней имеются два списка: **Categories** и **Commands**. Первый из них содержит перечень пунктов главного меню. Во втором отображаются все команды меню, выделенного в списке **Categories**. Команды помечаются соответ-

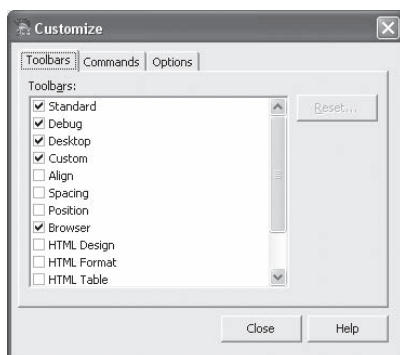


Рис. 8.13. Окно диалога Customize

ствующими им значками. Настройка производится путем перетаскивания (drag-and-drop). Например, для удаления кнопки достаточно перетащить ее за пределы панели инструментов. Чтобы добавить кнопку в панель инструментов, выбирается соответствующий пункт меню в списке **Categories**, затем находится нужная команда в списке **Commands**, после чего соответствующий ей значок перетаскивается на панель инструментов;

ПРИМЕЧАНИЕ

Кнопки можно размещать на любых панелях инструментов, независимо от того, какие команды они выполняют. Delphi IDE не проводит никакой проверки на соответствие названия панели инструментов размещенным на ней командам.

- ❑ вкладка **Options** содержит два флажка, управляющих выводом подсказок, отображаемых при наведении указателя мыши на кнопку панели инструментов. Флажок **Show tooltips** включает/выключает отображение подсказок, флажок **Show shortcut keys on tooltips** позволяет включать в текст подсказки информацию об оперативных клавишах для выбранной команды.

Палитра инструментов

Палитра компонентов (Tool Palette, рис. 8.14) используется для отображения компонентов, содержащихся в библиотеке компонентов Delphi.

В соответствии с выполняемыми ими функциями все расположенные в палитре компоненты разделены на группы, каждая из которых размещается на отдельной странице палитры.

Палитра компонентов полностью конфигурируется пользователем. Можно создавать новые страницы и удалять существующие, добавлять и удалять компоненты, перемещать компоненты между страницами. Настройка производится с помощью страницы Tool Palette окна диалога Options (рис. 8.15), открываемой командой **Properties** контекстного меню палитры компонентов.

Стандартная конфигурация палитры инструментов Turbo Delphi содержит 14 страниц, каждая из которых предоставляет разнообразные компоненты и элементы управления:

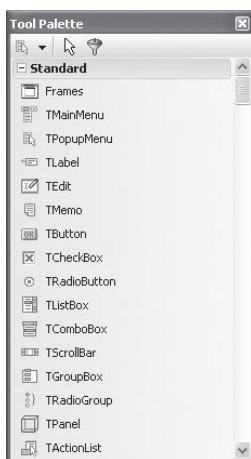


Рис. 8.14. Палитра инструментов Delphi

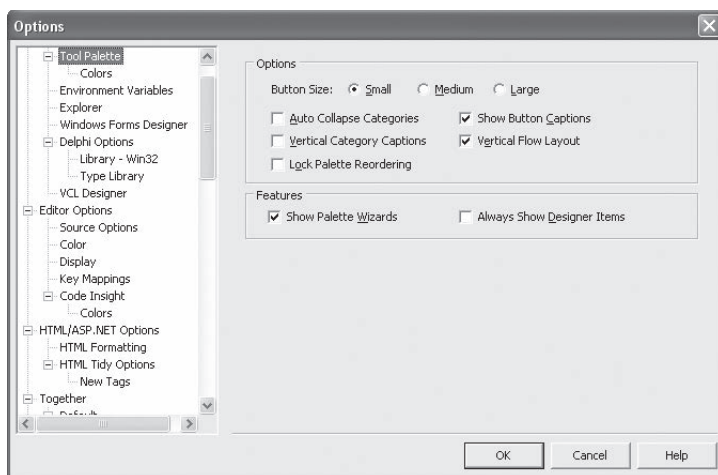


Рис. 8.15. Страница Tool Palette окна диалога Options предназначена для настройки палитры компонентов

- ❑ **Standard** — стандартные элементы управления оконного интерфейса Windows;
- ❑ **Additional** — специализированные элементы управления интерфейса Windows;
- ❑ **Win32** — элементы интерфейса, содержащиеся в 32-битных системных библиотеках Windows95 и Win32s;
- ❑ **System** — специализированные системные элементы управления;
- ❑ **Data Access** — компоненты, обеспечивающие доступ к информации, хранящейся в базах данных, и использующие процессор баз данных BDE (Borland Database Engine);

- ❑ Data Controls — компоненты для отображения и редактирования информации, хранящейся в базах данных;
- ❑ dbGo — компоненты, обеспечивающие доступ к информации, хранящейся в базах данных, с использованием технологии Microsoft ADO;
- ❑ dbExpress — компоненты для работы с SQL-серверами;
- ❑ BDE — компоненты, обеспечивающие доступ к информации, хранящейся в базах данных, с использованием технологии BDE;
- ❑ WebServices — компоненты для разработки приложений веб-серверов и клиентов многоуровневых приложений управления базами данных;
- ❑ Internet — компоненты для создания веб-приложений;
- ❑ Dialogs — стандартные диалоги открытия файла, сохранения файла, печати и т. п.;
- ❑ Win 3.1 — элементы управления оконного интерфейса Windows 3.1 (используются для совместимости с Delphi 1);
- ❑ Servers — компоненты для организации взаимодействия с приложениями Microsoft Office.

Инспектор объектов

Инспектор объектов (Object Inspector) является одним из важнейших инструментов разработки приложения и используется для настройки опубликованных свойств компонента. Окно Object Inspector показано на рис. 8.16.

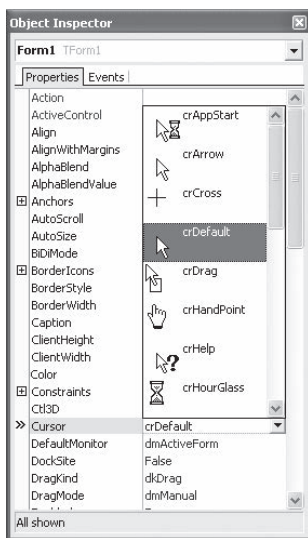


Рис. 8.16. Окно инспектора объектов с примером выбора значения графического свойства

Окно инспектора объектов содержит выпадающий список и две вкладки — Properties и Events. На первой из них отображается список свойств выделенного

объекта. На второй — список событий, на которые реагирует объект. Выпадающий список содержит перечень всех компонентов, размещенных на активной в данный момент форме (включая и саму форму).

Каждая вкладка разделена на две колонки. В левой колонке перечислены имена свойств, а в правой — их значения. Значения свойств можно редактировать. Некоторые свойства имеют в поле значений собственный выпадающий список, из которого можно выбрать необходимый параметр.

Настройки инспектора объектов выполняются с помощью команд контекстного меню:

- ☐ команда **View** определяет, какие категории свойств будут отображаться (все свойства компонентов делятся на ряд категорий, отображением которых можно управлять);
- ☐ команда **Arrange** определяет способ сортировки отображаемых свойств — по алфавиту (**by Name**) или по категориям (**by Category**);
- ☐ команда **Stay on Top** располагает окно инспектора объектов поверх всех других окон;
- ☐ команда **Hide** скрывает окно инспектора объектов.

Редактор форм

Редактор форм (**Form Designer**) представляет собой инструмент визуальной разработки интерфейса приложения. Он автоматически отображается в момент начала работы с формой. Чтобы переключиться в Редактор форм, щелкните на заголовке вкладки **Design**, расположенном в нижней части главного окна.

С помощью Редактора форм можно реализовать выполнение следующих функций:

- ☐ размещение компонентов на форме;
- ☐ модификацию свойств компонентов и самой формы;
- ☐ установку обработчиков событий.

Более подробно работа с редактором форм рассматривается в главе 10, «Создание форм для ввода и редактирования данных», а здесь приведем только методику установки обработчиков событий.

Установка обработчиков событий

Обработчик события — это процедура, предназначенная для создания реакции на какое-либо воздействие. События, на которые компонент может реагировать, перечисляются на вкладке **Events** инспектора объектов.

Чтобы задать обработчик для какого-либо события, выполните двойной щелчок мышью на поле значения события в инспекторе объектов. При этом происходит переключение в редактор кода, в котором автоматически генерируется код заголовка процедуры-обработчика события.

ПРИМЕЧАНИЕ

Delphi генерирует только заголовок обработчика события. Реакцию на событие требуется описывать вручную в теле процедуры-обработчика, которое ничем не отличается от обычной процедуры языка **Object Pascal**.

Редактор кода

Редактор кода является обычным текстовым редактором, ориентированным на написание текстов программ. Переключиться в Редактор кода вы можете аналогично переключению в Редактор форм, щелкнув на вкладке Code.

Основные компоненты Delphi. Построение простых приложений

Среда визуального программирования Delphi содержит базовый набор используемых в приложениях стандартных элементов управления, доступ к которым осуществляется через палитру инструментов. Используя готовые компоненты, разработчик может создавать приложения высокого уровня сложности. В платных версиях Delphi предусмотрена также возможность разработки собственных компонентов (или иерархии компонентов) или использования компонентов, разработанных другими программистами.

Библиотека визуальных компонентов

Delphi содержит разветвленную иерархию классов, объединенных в библиотеку, называемую библиотекой визуальных компонентов (Visual Component Library, VCL). Значительная часть входящих в нее классов реализована в виде компонентов, то есть классов, доступных для использования в палитре компонентов.

Пакеты

В Delphi библиотека визуальных компонентов состоит из файлов, содержащих откомпилированный код классов. Эти файлы обычно называются *пакетами*. Пакет представляет собой библиотеку динамической компоновки (DLL), содержащую, кроме кода классов, дополнительную информацию, которая позволяет использовать этот код совместно с несколькими приложениями.

Существует два вида пакетов:

- ☐ *пакеты времени разработки* (design-time packages);
- ☐ *пакеты времени выполнения* (runtime packages).

Пакеты времени разработки содержат код, который используется самой средой Delphi только во время разработки приложения и не нужен при выполнении программы (например, пакеты времени разработки могут содержать специальные редакторы свойств компонентов). В стандартную поставку Turbo Delphi Explorer входит более 10 пакетов времени разработки (файлы пакетов времени разработки имеют расширение BPL и расположены в каталоге .../BDS/4.0/Bin).

Пакеты времени выполнения содержат код, который выполняется только во время работы приложения. При компиляции проекта код, содержащийся в пакетах времени выполнения, может либо включаться в исполняемый файл, либо нет. В первом случае исполняемый файл будет иметь больший размер, но для его работы не потребуются никаких дополнительных файлов. Во втором случае размер исполняемого файла получается небольшим, но для работы программы

потребуется файлы пакетов времени выполнения, в которых содержится код компонентов, используемых в программе. Настройка способа компиляции производится на странице Packages окна диалога Project Options (рис. 8.17).

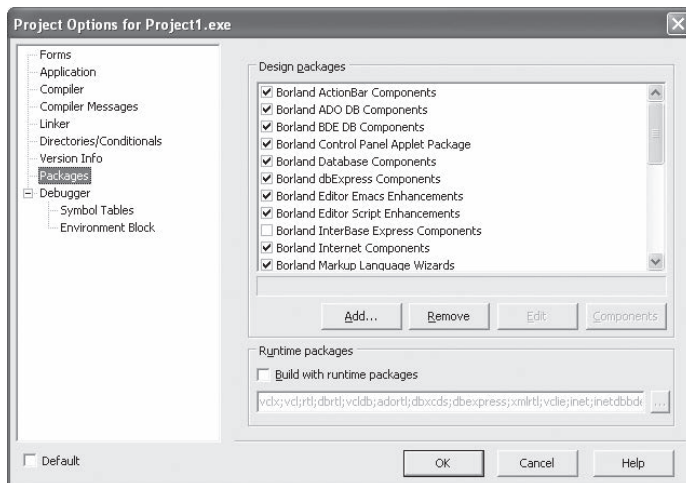


Рис. 8.17. Окно диалога Project Options используется для настройки пакетов

В случае установки флажка **Build with runtime packages** код компонентов, содержащихся в пакетах, перечисленных в строке ввода под флажком, не будет включаться в исполняемый файл.

Основные компоненты для построения простых приложений

Условно все компоненты Delphi можно разделить на две группы: *визуальные* (или компоненты интерфейса) и *невизуальные* (или системные компоненты).

Визуальные компоненты видны как во время разработки, так и во время выполнения программы. Они позволяют отображать какую-либо информацию, вводить текст, выбирать элементы из списка и т. п. В основном визуальные компоненты используются для создания интерфейса пользователя.

Невизуальные компоненты видны только во время разработки. Они в основном предназначены для разработки логической структуры приложения. В то же время некоторые невидимые компоненты используются и для построения интерфейса. Например, окна диалога не видны в процессе разработки приложения и относятся к невидимым компонентам, но в то же время они являются типичным элементом интерфейса.

Формы

Форма представляет собой окно приложения на этапе разработки. Любое приложение Windows должно иметь по крайней мере одно окно, поэтому проект Delphi также должен содержать хотя бы одну форму. Формы обеспечивают

создание интерфейса пользователя, являясь своего рода контейнером, содержащим другие элементы интерфейса.

ПРИМЕЧАНИЕ

В принципе приложение Windows может и не иметь окна — такие приложения называются консольными и используются сравнительно редко. В данной книге мы ограничимся рассмотрением оконных приложений.

Хотя форма и является стандартным компонентом Delphi, входящим в состав VCL, на палитре компонентов ее нет. Для создания новой формы используются команды главного меню.

При создании нового приложения всегда автоматически создается форма, соответствующая главному окну приложения. Сразу после создания приложения можно откомпилировать и запустить. При этом на экране появится пустое окно, с которым можно проделывать достаточно сложные манипуляции, например перемещать по экрану, изменять его размеры, сворачивать и т. п. (не написав ни одной строки кода!). Все эти функции обеспечиваются классом формы `TForm`, который подробно рассматривается в главе 10, «Создание форм для ввода и редактирования данных».

Стандартные элементы интерфейса

В библиотеке VCL Delphi содержится ряд компонентов, которые предназначены для создания стандартных элементов интерфейса приложений Windows. Все эти компоненты доступны в палитре компонентов и могут размещаться на формах или фреймах. Все компоненты Delphi (включая и формы) являются потомками одного класса (`TComponent`) и имеют большое количество общих свойств и событий.

Кнопки

Кнопки (`Button`) являются одним из наиболее распространенных элементов управления Windows. Свойства и методы компонента «кнопка» инкапсулированы в классе `TButton` (табл. 8.1).

Таблица 8.1. Основные свойства компонента `TButton`

Свойство	Тип	Описание
<code>Anchors</code>	<code>TAnchors = set of (akTop, akLeft, akRight, akBottom)</code>	<p>Задаёт привязку кнопки:</p> <ul style="list-style-type: none"> • <code>akTop</code> — к верхней границе контейнера; • <code>akLeft</code> — к левой границе контейнера; • <code>akRight</code> — к правой границе контейнера; • <code>akBottom</code> — к нижней границе контейнера. <p>Если заданы все четыре точки привязки, то при изменении размеров формы пропорционально будут изменяться и размеры кнопки</p>
<code>Caption</code>	<code>TCaption</code>	Строка текста, отображаемого на кнопке
<code>Cancel</code>	<code>Boolean</code>	Если это свойство установлено равным <code>true</code> , то при нажатии на клавишу <code>Escape</code> вызывается обработчик события <code>OnClick</code> данной кнопки. На форме может быть только одна кнопка со свойством <code>Cancel</code> , установленным в <code>true</code>

Свойство	Тип	Описание
Default	Boolean	Если это свойство установлено равным true, то при нажатии на клавишу Enter вызывается обработчик события OnClick данной кнопки. На форме может быть только одна кнопка со свойством Default, установленным в true
ModalResult	TmodalResult	При нажатии на кнопку значение свойства ModalResult родительской формы устанавливается равным свойству ModalResult кнопки. Используется при создании окон диалога
Enable	Boolean	При установке данного свойства равным false кнопка становится недоступной, отображаясь серым цветом
Visible	Boolean	При установке данного свойства в false кнопка становится невидимой и недоступной
Font	TFont	Определяет шрифт, которым отображается надпись на кнопке
TabOrder	TTabOrder	Определяет порядок перебора элементов управления, расположенных на форме, при нажатии на клавишу Tab
TabStop	Boolean	Если данное свойство установлено в false, то фокус ввода никогда не будет передаваться данной кнопке при переборе элементов клавишей Tab. Однако такую кнопку все равно можно нажать мышью
Top	Integer	Положение верхней границы кнопки (в пикселах) относительно верхней границы контейнера, содержащего кнопку
Left	Integer	Положение левой границы кнопки (в пикселах) относительно левой границы контейнера, содержащего кнопку

Класс TButton содержит также ряд методов. Однако при работе с кнопками они используются редко.

Основное событие кнопки — OnClick, оно вызывается при нажатии на кнопку и используется для программирования реакции на нажатие.

Рассмотрим пример использования кнопки. Напишем программу, в которой кнопки используются для изменения заголовка окна программы и для завершения программы.

1. С помощью команды **File ► New ► VCL Forms Application — Delphi for Win32** создайте новое приложение.
2. Разместите на форме две кнопки. Затем выберите в окне **Object Inspector** вкладку **Properties** и, используя свойство **Caption**, измените название первой кнопки на **Изменить подпись**, второй — на **Выход**.
3. С помощью мыши выделите кнопку **Изменить подпись** и перейдите в окне **Object Inspector** на вкладку **Events**. Дважды щелкните мышью на поле значения события **OnClick**. После этого окно редактора кода становится активным и в него автоматически добавляется заголовок процедуры-обработчика события **OnClick**:

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

4. В теле процедуры `Button1Click` напишите код, выполняемый при возникновении события `OnClick`. Для изменения заголовка формы следует использовать свойство `Caption` объекта `Form1` (это идентификатор экземпляра `TForm`, задаваемый по умолчанию):

```
procedure TForm1.Button1Click(Sender: TObject);  
begin  
    Form1.Caption:='Пример использования кнопок';  
end;
```

5. Выделите с помощью мыши кнопку **Выход** и выберите в окне **Object Inspector** вкладку **Events**. Дважды щелкните мышью на поле значения события `OnClick`. Окно редактора кода при этом становится активным и в него автоматически добавляется заголовок процедуры-обработчика события `OnClick` для выделенной кнопки:

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

6. Напишите код завершения программы в теле процедуры `Button2Click`. Для этого воспользуйтесь методом `Terminate` класса `TApplication` (более подробно этот класс рассмотрен в главе 13, «Управление проектом и создание приложения»):

```
procedure TForm1.Button2Click(Sender: TObject);  
begin  
    Application.Terminate;  
end;
```

7. Выполните компиляцию программы. После ее запуска на экран будет выведена форма, содержащая две кнопки, примерный вид которой показан на

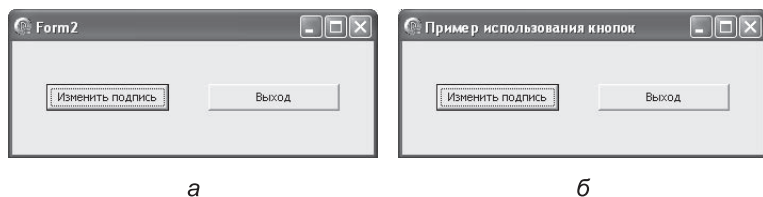


Рис. 8.18. Пример формы с кнопками

рис. 8.18, *а*. При щелчке на кнопке **Изменить подпись** заголовок окна принимает значение **Пример использования кнопок** (рис. 8.18, *б*). Щелчок на кнопке **Выход** приводит к завершению программы.

Надписи

Надписи (`Label`) используются для отображения на форме текста без возможностей редактирования. Чаще всего надписи применяются для создания подписей к другим элементам управления. Свойства и методы компонента «надпись» инкапсулированы в классе `TLabel` (табл. 8.2).

Таблица 8.2. Основные свойства компонента TLabel

Свойство	Тип	Описание
Alignment	TAlignment = (taLeftJustify, taRightJustify, taCenter)	Определяет способ выравнивания текста по горизонтали: <ul style="list-style-type: none"> • taLeftJustify — по левому краю; • taRightJustify — по правому краю; • taCenter — по центру
Layout	TTextLayout = (tlTop, tlCenter, tlBottom)	Определяет способ выравнивания текста по вертикали: <ul style="list-style-type: none"> • tlTop — по верхнему краю; • tlCenter — по центру; • tlBottom — по нижнему краю
Autosize	Boolean	Если данное свойство равно true, то ширина и высота объекта Label автоматически изменяются таким образом, чтобы отобразить весь текст, заданный в свойстве Caption
Transparent	Boolean	Если данное свойство равно true, то фон надписи будет прозрачным
WordWrap	Boolean	Если данное свойство установлено в true, то выполняется автоматический перенос текста на следующую строку

Основное свойство надписей — Caption, в котором задается выводимый текст. Изменять значение этого свойства можно как во время разработки программы (в окне Object Inspector), так и во время выполнения программы. Следует иметь в виду, что свойство Caption имеет строковый тип и ему естественно может быть присвоено только строковое значение. Для вывода числовых значений с использованием надписей необходимо воспользоваться функциями преобразования чисел в строки:

- ❑ IntToStr — преобразует целое число, заданное параметром, в строку;
- ❑ FloatToStr — преобразует действительное число в соответствующее ему строковое представление;
- ❑ FloatToStrF — преобразует действительное число в строку с возможностью форматирования.

Рассмотрим пример использования надписей для вывода текста. Создадим программу, подобную той, что была рассмотрена в предыдущем примере, добавим на форму компонент TLabel и будем изменять значение его свойства Caption при нажатии на одну из кнопок.

1. С помощью команды File ► New ► VCL Forms Application — Delphi for Win32 создайте новое приложение.
2. Разместите на форме две кнопки. Затем выберите в окне Object Inspector вкладку Properties и, используя свойство Caption, измените название первой кнопки на Изменить надпись, второй — на Выход.
3. Поместите на форму компонент Надпись (TLabel). С помощью инспектора объектов присвойте свойству Caption этого компонента значение Кнопка Изменить надпись «ни разу не была нажата».

4. Задайте следующий обработчик события `OnClick` для кнопки **Изменить надпись**:

```
procedure TForm2.Button1Click(Sender: TObject);
begin
    inc(i);
    // изменение надписи
    Label1.Caption:=''+Button1.Caption+'' нажата ''+
        IntToStr(i)+' раз';
end;
```

5. В разделе переменных объявите глобальную переменную `i` типа `integer` для подсчета количества нажатий на кнопку **Изменить надпись**.

6. Напишите код завершения программы в обработчике события `OnClick` кнопки **Выход**:

```
procedure TForm2.Button2Click(Sender: TObject);
begin
    Application.Terminate;
end;
```

7. Выполните компиляцию программы. После ее запуска на экран будет выведена форма, содержащая две кнопки, примерный вид которой показан на рис. 8.19, а. При первом щелчке на кнопке **Изменить надпись** надпись изменяется и принимает значение **Кнопка «Изменить надпись» нажата 1 раз**. При каждом последующем щелчке число в надписи увеличивается на 1. На рис. 8.19, б приведено окно программы после 5 щелчков на кнопке **Изменить надпись**. Щелчок на кнопке **Выход** приводит к завершению программы.

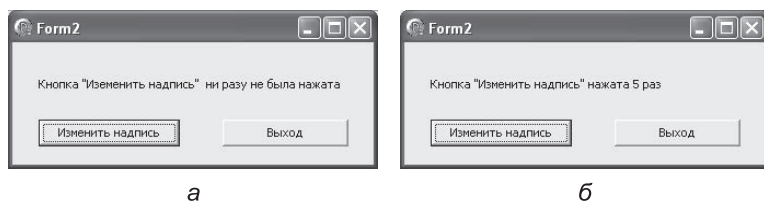


Рис. 8.19. Пример использования надписей

Флажки

Флажки (Check box) используются для выбора одного из двух вариантов. Элемент «флажок» может находиться в одном из двух состояний: установлен (включен) или снят (выключен). Установленный флажок помечается «галочкой». Возможно и третье состояние флажка: «установлен и закрашен серым». Это состояние используется для того, чтобы показать, что флажок имеет вложенные флажки, часть из которых установлена, а часть — нет. Третье состояние флажков можно наблюдать, например, в программе установки Windows при выборе устанавливаемых компонентов.

Компоненту «флажок» соответствует класс `TCheckBox`.

Флажки имеют три основных свойства:

- **Checked**: `boolean` — показывает, установлен флажок или нет. Если данное свойство имеет значение `true`, то флажок установлен; если `false`, то флажок снят или установлен в «третье состояние»;

- ❑ State: TCheckBoxState, тип данных определен так: TCheckBoxState = (cbUnchecked, cbChecked, cbGrayed) — определяет состояние флажка: установлен (cbChecked), снят (cbUnchecked) или установлен и закрашен серым (cbGrayed);
- ❑ AllowGrayed: Boolean — определяет, может (true) или нет (false) флажок иметь «третье» состояние.

Флажки также имеют свойство Caption, с помощью которого может задаваться поясняющая надпись (рис. 8.20).

Рассмотрим в качестве примера использование флажка для управления реакцией на нажатие кнопки.

1. С помощью команды File ► New ► VCL Forms Application — Delphi for Win32 создайте новое приложение.
2. Разместите на форме одну кнопку и один элемент TCheckBox. Затем выберите в окне Object Inspector вкладку Properties и, используя свойство Caption, измените название кнопки на OK. Аналогично задайте для флажка поясняющую надпись Завершение программы.
3. Установите следующий обработчик события OnClick кнопки OK:

```
procedure TForm2.Button1Click(Sender: TObject);  
begin  
    if CheckBox1.Checked  
    then Application.Terminate;  
end;
```

4. Выполните компиляцию программы. После ее запуска на экран будет выведена форма, содержащая одну кнопку и один снятый флажок (рис. 8.20). Если флажок снят, то при нажатии на кнопку OK ничего не происходит. Если же флажок установлен, то нажатие на кнопку приводит к завершению программы.



Рис. 8.20. Пример использования флажка

Переключатели

Переключатели (Radio Button) предназначены для выбора одного из нескольких альтернативных вариантов. Свойства и методы данного компонента инкапсулированы в классе TRadioButton. Основным свойством переключателя является свойство Checked типа Boolean, которое показывает, выбран данный переключатель или нет. Все переключатели, помещаемые в один контейнер (форма, фрейм и т. п.), считаются входящими в одну группу, из которой может быть выбран только один переключатель. Свойство Caption класса TRadioButton используется для задания поясняющей надписи.

ПРИМЕЧАНИЕ

Событие `OnClick` для компонента `TRadioButton` имеет одну особенность: оно вызывается только при выборе переключателя. Если переключатель уже выбран, то нажатие на нем левой кнопки мыши не вызовет события `OnClick`.

Рассмотрим пример, подобный предыдущему, где для управления реакцией на нажатие кнопки будем использовать переключатели.

1. С помощью команды `File ► New ► VCL Forms Application — Delphi for Win32` создайте новое приложение.
2. Разместите на форме одну кнопку и три элемента `TRadioButton`. Затем выберите в окне `Object Inspector` вкладку `Properties` и, используя свойство `Caption`, измените название кнопки на `OK`. Для переключателей аналогичным образом задайте следующие поясняющие надписи: `Ничего` — для первого, `Звуковой сигнал` — для второго и `Завершение программы` — для третьего (рис. 8.21).

3. Установите следующий обработчик события `OnClick` кнопки `OK`:

```
procedure TForm2.Button1Click(Sender: TObject);
begin
    if RadioButton2.Checked
    then Beep;
    if RadioButton3.Checked
    then Application.Terminate;
end;
```

4. Выполните компиляцию программы. После ее запуска на экран будет выведена форма, содержащая одну кнопку и три переключателя (см. рис. 8.21). Если выбран первый переключатель, то при нажатии на кнопку `OK` ничего не происходит. При выборе второго переключателя нажатие на кнопку вызывает звуковой сигнал. Если выбран третий переключатель, то нажатие на кнопку приводит к завершению программы.



Рис. 8.21. Пример использования переключателей

Текстовые поля

Текстовое поле (Edit box) — стандартное поле ввода, которое позволяет отображать и редактировать текст. Возможность ввода текста с клавиатуры реализована в Delphi в классе `TEdit`. Класс `TEdit` содержит ряд свойств, не имеющих аналогов в уже рассмотренных элементах управления (табл. 8.3).

Таблица 8.3. Основные свойства компонента TEdit

Свойство	Тип	Описание
AutoSelect	Boolean	Определяет, будет ли выделяться текст в поле ввода при получении полем фокуса ввода
CharCase	TEditCharCase = (ecNormal, ecUpperCase, ecLowerCase)	Определяет регистр вводимого текста: <ul style="list-style-type: none"> • ecNormal — текст может вводиться в разных регистрах; • ecUpperCase — вводимый текст преобразуется в символы верхнего регистра; • ecLowerCase — вводимый текст преобразуется в символы нижнего регистра
SelText	String	Содержит выделенный фрагмент текста
Text	String	Содержит текст, отображаемый в поле ввода
ReadOnly	Boolean	Если данное свойство равно true, то при выполнении программы отсутствует возможность редактирования текста, отображаемого в поле ввода

Элемент TEdit позволяет обрабатывать событие OnChange, вызываемое при изменении содержимого поля ввода. Например, если установить следующий обработчик события OnChange для текстового поля:

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
    Label1.Caption:=Edit1.Text;
end;
```

то при изменении текста в поле ввода Edit1 синхронно будет изменяться и текст надписи Label1.

С помощью полей ввода можно вводить только текстовую информацию. Если требуется производить ввод чисел, то следует воспользоваться следующими функциями преобразования:

- ❑ StrToInt(const S: string) — преобразует строку в число целого типа. Если такое преобразование осуществить невозможно (заданная строка не является строковым представлением целого числа), то вызывается исключительная ситуация EConvertError;
- ❑ StrToIntDef(const S: string; Default: Integer) — преобразует строку в целое число. Если преобразование нельзя выполнить, функция возвращает число, заданное параметром Default.
- ❑ StrToFloat(const S: string) — преобразует строку в действительное число. Если преобразование нельзя выполнить, то вызывается исключительная ситуация EConvertError.

Объединение элементов управления

При разработке интерфейса приложения несколько функционально связанных элементов управления часто объединяются в одну группу. Обычно для этого используются специальные контейнерные элементы TPanel и TGroupBox.

Для группированных элементов свойства контейнера Enable и Visible переносятся и на помещенные в них элементы. При изменении расположения контейнера во время разработки все элементы также перемещаются вместе с ним.

Класс TPanel

Компонент TPanel представляет собой прямоугольную область, которая может быть «вдавленной» или «выпуклой» относительно формы, на которой она размещена (это определяется свойствами BevelInner и BevelOuter класса TPanel).

Класс TGroupBox

Компонент TGroupBox используется для визуального выделения группы элементов управления. В отличие от панели может иметь заголовок. Других принципиальных отличий нет.

Глава 9

Компоненты для ввода и редактирования данных

Практически любая компьютерная программа предназначена для обработки данных. В процессе работы программа получает исходные данные и возвращает результат обработки. Наиболее часто используются два способа получения исходных данных: ввод данных пользователем (как правило, с помощью клавиатуры или мыши) или считывание данных, хранящихся на каком-либо носителе информации (чаще всего — магнитном диске). Обычно применяются оба способа получения данных. Результаты расчета могут отображаться на экране (в виде строк, таблиц, графиков, диаграмм и т. п.) либо записываться в файл.

Информация, с которой работают программы, часто хранится в базах данных. В этом случае программа должна обеспечивать возможности отображения и изменения данных, размещенных в базе данных.

Компоненты Delphi, применяемые для ввода и редактирования информации, условно можно разделить на две группы: ориентированные и не ориентированные на работу с базами данных. Кроме этого, Delphi содержит невизуальные компоненты, предназначенные для работы с файлами: окна диалога открытия и сохранения файлов.

Стандартные компоненты Delphi для ввода и редактирования данных

В предыдущей главе мы уже рассмотрели некоторые элементы управления, предназначенные для ввода данных в программу (флажки, переключатели, текстовые поля). Теперь рассмотрим ряд более сложных компонентов, предназначенных для обмена данными между пользователем и программой. Все компоненты, о которых пойдет речь, входят в стандартную поставку Delphi и представлены на вкладках **Standard** и **Additional** палитры компонентов.

Многострочные текстовые поля

Многострочное текстовое поле (компонент `TMemo`) предназначено для отображения и редактирования текста. Основные свойства этого элемента управления приведены в табл. 9.1.

Таблица 9.1. Основные свойства класса `TMemo`

Свойство	Тип	Описание
<code>Alignment</code>	<code>TAlignment</code>	Задаёт способ выравнивания текста в поле. Возможные значения этого свойства: <ul style="list-style-type: none"> • <code>taLeftJustify</code> — выравнивание текста по левому краю; • <code>taCenter</code> — выравнивание по центру; • <code>taRightJustify</code> — выравнивание по правому краю
<code>CaretPos</code>	<code>TPoint</code>	Определяет координаты текстового курсора в поле. Координаты указываются в строках (по вертикали) и символах (по горизонтали)
<code>Lines</code>	<code>TStrings</code>	Массив строк, содержащихся в поле

Текст в поле `TMemo` может быть выделен с помощью мыши или клавиатуры. Определить выделенный фрагмент текста можно с помощью свойства `SelectText`.

Списки

Список состоит из строк (пунктов списка). Пользователь может просмотреть список и выбрать одну или несколько строк для последующей обработки. Непосредственно редактировать содержимое списка нельзя. Если все строки не помещаются в списке, в него автоматически добавляется вертикальная полоса прокрутки.

Свойства и методы, обеспечивающие работу со списками, инкапсулированы в классе `TListBox`. Основные свойства этого класса приведены в табл. 9.2.

Таблица 9.2. Основные свойства класса `TListBox`

Свойство	Тип	Описание
<code>Columns</code>	<code>Integer</code>	Позволяет создавать списки, состоящие из нескольких столбцов. Если это свойство равно нулю, то список состоит из одного столбца, если больше нуля — из нескольких, причем значение <code>Columns</code> определяет количество столбцов
<code>ItemIndex</code>	<code>Integer</code>	Порядковый номер выделенного пункта списка
<code>Items</code>	<code>TStrings</code>	Массив пунктов списка. Используется для добавления, вставки, перемещения и удаления пунктов списка
<code>MultiSelect</code>	<code>Boolean</code>	Задаёт возможность выбора в списке нескольких пунктов
<code>SelCount</code>	<code>Boolean</code>	Количество выбранных пунктов списка
<code>Selected[Index: Integer]: Boolean</code>	<code>Boolean</code>	Определяет, выделен пункт списка с порядковым номером <code>Index</code> или нет
<code>Sorted</code>	<code>Boolean</code>	Позволяет задать сортировку пунктов списка по алфавиту

Комбинированные списки

Комбинированные списки объединяют возможности списков и текстовых полей. Данные элементы управления позволяют пользователю выбрать в списке существующую строку (пункт списка) или ввести в поле списка строку, которой нет в списке. Комбинированный список может быть раскрывающимся. Для описания комбинированных списков используется класс `TComboBox`. Его основные свойства перечислены в табл. 9.3.

Таблица 9.3. Основные свойства класса `TComboBox`

Свойство	Тип	Описание
<code>DropDownCount</code>	<code>Integer</code>	Количество строк, отображаемых в раскрывающейся части комбинированного списка
<code>DroppedDown</code>	<code>Boolean</code>	Определяет, раскрыт список или нет
<code>ItemIndex</code>	<code>Integer</code>	Определяет выделенный пункт в раскрывающемся списке
<code>Items</code>	<code>TStrings</code>	Массив строк раскрывающегося списка
<code>MaxLength</code>	<code>Integer</code>	Максимальное количество символов, которое пользователь может ввести в поле списка
<code>SelfText</code>	<code>String</code>	Выделенный фрагмент строки в поле списка
<code>Sorted</code>	<code>Boolean</code>	Определяет, отсортированы пункты списка по алфавиту или нет
<code>Style</code>	<code>TComboBoxStyle</code>	Задаёт стиль комбинированного списка

Изображения

Для вывода изображений используется элемент `TImage`. Данный элемент может отображать графические файлы форматов BMP, JPG, EMF, JPEG, GIF, PNG, TIF, TIFF, WMF и ICO. Основные свойства класса `TImage` приведены в табл. 9.4.

Таблица 9.4. Основные свойства класса `TImage`

Свойство	Тип	Описание
<code>AutoSize</code>	<code>Boolean</code>	Если данное свойство равно <code>true</code> , то размеры компонента изменяются в соответствии с размерами загружаемого изображения
<code>Center</code>	<code>Boolean</code>	Если данное свойство имеет значение <code>true</code> , то изображение центрируется относительно клиентской области компонента, если <code>false</code> — изображение располагается в левом верхнем углу клиентской области компонента
<code>Picture</code>	<code>TPicture</code>	Содержит изображение, отображаемое в компоненте <code>TImage</code>
<code>Stretch</code>	<code>Boolean</code>	Если данное свойство равно <code>true</code> , то изображение масштабируется таким образом, чтобы вписаться в размеры компонента <code>TImage</code>

Свойство `Picture` обеспечивает управление загрузкой и отображением графических файлов. Данное свойство имеет тип `TPicture` — класс, инкапсулирующий свойства и методы, предназначенные для работы с графическими файлами. Свойства `Width` и `Height` класса `TPicture` содержат размеры изображения.

Для загрузки и сохранения графических файлов используются следующие методы класса TPicture:

```
procedure LoadFromFile (const FileName: string);  
procedure SaveToFile (const FileName: string);
```

Например, фрагмент программы, обеспечивающий выбор файла и загрузку его в объект TImage, выглядит следующим образом:

```
if dlgOpenPic.Execute  
then  
  imgMyPic.Picture.LoadFromFile(dlgOpenPic.FileName);
```

Изображение можно также получить из системного буфера обмена (clipboard) и занести в буфер с использованием следующих методов:

```
procedure LoadFromClipboardFormat (AFormat: Word;  
  AData: THandle; APalette: HPALETTE);  
procedure SaveToClipboardFormat (var AFormat: Word;  
  var AData: THandle; var APalette: HPALETTE);
```

Параметр AFormat позволяет задать формат изображения. В Windows по умолчанию зарегистрированы три формата: один растровый (CF_BITMAP) и два векторных (CF_METAFILE и CF_ENHMETAFILE).

Доступ к предопределенным типам графических объектов может производиться через одно из трех свойств:

- ☐ Bitmap: TBitmap — растровое изображение;
- ☐ Metafile: TMetafile — векторное изображение;
- ☐ Icon: TIcon — изображение значка (в формате ICO).

Кроме того, для доступа к графическим объектам можно использовать свойство Graphic типа TGraphic, являющегося родительским для объектных типов TBitmap, TMetafile и TIcon.

Стандартные окна диалога Delphi

Одним из основных элементов интерфейса приложений Windows являются окна диалога. Выполнение ряда стандартных операций, используемых практически во всех приложениях, обеспечивается *стандартными* окнами диалога. К ним относятся окна диалога для открытия и сохранения файлов, печати документов, изменения параметров шрифта и т. п.

В Delphi входит ряд компонентов, предназначенных для создания стандартных окон диалога. Все они расположены на странице Dialogs палитры компонентов.

Стандартные окна диалога являются невидимыми компонентами и не отображаются во время разработки приложения.

Окна диалога для работы с файлами

Операции открытия и сохранения файлов необходимы практически во всех программах. Для реализации этих операций в Delphi имеются шесть компонентов:

- ☐ TOpenDialog — окно диалога открытия файла;
- ☐ TSaveDialog — окно диалога сохранения файла;

- ❑ TOpenPictureDialog — окно диалога открытия графического файла;
- ❑ TSavePictureDialog — окно диалога сохранения графического файла.
- ❑ TOpenTextFileDialog — окно диалога открытия текстового файла;
- ❑ TSaveTextFileDialog — окно диалога сохранения текстового файла;

Свойства всех этих классов идентичны (табл. 9.5).

Таблица 9.5. Свойства окон диалога для работы с файлами

Свойство	Тип	Описание
DefaultExt	String	Расширение, автоматически добавляемое к имени выбранного файла. Расширения длиной более трех символов не поддерживаются. При задании данного свойства не следует добавлять точку перед символами расширения
FileEditStyle	TFileEditStyle = (fsEdit, fsComboBox)	Вид диалогового окна выбора файла. Оставлено для совместимости с ранними версиями Delphi
FileName	TFileName	Имя последнего выделенного файла в окне диалога
Files	TStrings	Список файлов, выбранных в окне диалога
Filter	String	Задаёт фильтры для выбора файлов в окне диалога
FilterIndex	Integer	Фильтр для выбора файлов, задаваемый по умолчанию
HistoryList	TStrings	Список ранее выбранных файлов
InitialDir	String	Каталог, к которому при открытии обращается окно выбора файла
Title	String	Заголовок окна диалога
Options	TOpenOptions	Параметры окна диалога

При работе с окнами диалога обычно используется всего один метод, который осуществляет вызов окна диалога во время выполнения программы:

```
function Execute : boolean;
```

После выбора файла и последующего щелчка на одной из кнопок, подтверждающих выбор (OK, Open или Save), функция Execute возвращает значение true. При щелчке на кнопке Cancel данная функция возвращает false.

Окна диалога для работы с файлами могут реагировать на ряд событий:

- ❑ OnCanClose — вызывается, когда пользователь пытается закрыть окно диалога без отмены;
- ❑ OnFolderChange — вызывается при смене каталога;
- ❑ OnIncludeItem — вызывается, перед тем как в список файлов окна диалога будет добавлен файл;
- ❑ OnSelectionChange — вызывается при изменении списка выделенных файлов;
- ❑ OnTypeChange — вызывается при изменении фильтра;
- ❑ OnClose — вызывается при закрытии диалога и используется для проверки имени выбранного файла;
- ❑ OnShow — вызывается при открытии окна диалога.

После закрытия окна диалога щелчком на кнопке **Open** имя выбранного файла, включая полный путь, будет доступно через свойство `FileName`. В случае выделения нескольких файлов данное свойство будет содержать имя файла, выделенного последним. Список выделенных файлов доступен через свойство `Files`.

ПРИМЕЧАНИЕ

Заметьте, что рассмотренные окна диалога обеспечивают только выбор файла (или группы файлов). Все остальные действия (открытие файла, сохранение и т. п.) возлагаются на программиста.

При работе с окнами диалога часто бывает удобно пользоваться фильтрами, с помощью которых можно выбирать файлы с заданными расширениями. Фильтры, используемые в окне диалога, задаются свойством `Filter`, имеющим тип `String`. Определяющая фильтр строка состоит из двух частей, разделяемых вертикальной чертой: названия фильтра и шаблона для имени файла. Например, строка, задающая фильтр для отбора исполняемых файлов, может выглядеть следующим образом:

```
'Программы|*.exe'
```

В одном фильтре можно задать несколько шаблонов для отбора файлов, которые должны разделяться точкой с запятой. Например:

```
'Программы|*.exe;*.com'
```

Можно также задать несколько фильтров. В этом случае фильтры разделяются вертикальной чертой:

```
'Программы|*.exe;*.com|Динамические библиотеки|*.dll'
```

Проще всего задавать фильтры в специальном редакторе `Filter Editor` (рис. 9.1), окно которого открывается при двойном щелчке в поле значения свойства `Filter` в окне инспектора объектов.

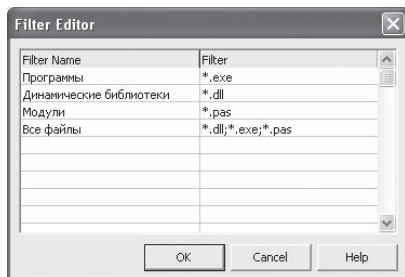


Рис. 9.1. Окно редактора фильтров

В левом списке редактора фильтров задается название фильтра, в правом — шаблоны для отбора файлов.

В правой части окна диалога для открытия и сохранения графических файлов имеется область предварительного просмотра, предназначенная для отображения текущего выделенного файла (рис. 9.2).

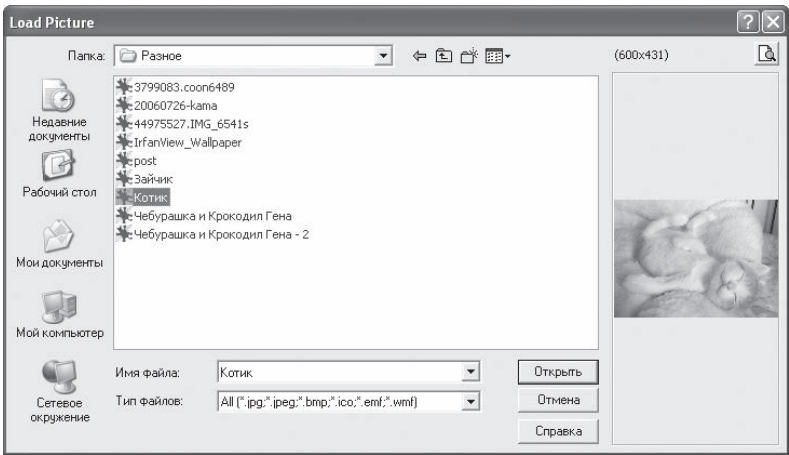


Рис. 9.2. Окно диалога для открытия графического файла с областью предварительного просмотра

Окно диалога для установки и настройки шрифтов

Для установки шрифтов и изменения их параметров используется компонент TFontDialog. Основные свойства данного компонента приведены в табл. 9.6.

Таблица 9.6. Свойства окна диалога для выбора шрифта

Свойство	Тип	Описание
Device	TFontDialogDevice = (fdScreen, fdPrinter, fdBoth)	Устройство с доступными для установки шрифтами
Font	TFont	Заданные в окне диалога параметры шрифта
MaxFontSize	Integer	Максимальный размер шрифта, который будет доступен в окне диалога
MinFontSize	Integer	Минимальный размер шрифта, который будет доступен в окне диалога
Options	TFontDialogOptions	Параметры окна диалога

Окно диалога для установки и настройки шрифтов может обрабатывать только три события:

- OnApply — вызывается при щелчке на кнопке Apply. Это событие используется для того, чтобы применить выбранный шрифт без закрытия окна диалога;
- OnClose — вызывается при закрытии окна диалога;
- OnShow — вызывается при открытии окна диалога.

Окно диалога для выбора цвета

Выбор цвета обеспечивается компонентом TColorDialog. Основные свойства класса TColorDialog приведены в табл. 9.7.

Таблица 9.7. Основные свойства класса TColorDialog

Свойство	Тип	Описание
Color	TColor	Выбранный цвет
CustomColors	TStrings	Дополнительные цвета в шестнадцатеричном коде в формате RGB. Каждый цвет определяется строкой вида ColorA = ff0000 (ColorB = 00ff00 и т. д.)
Options	TColorDialogOptions	Параметры окна диалога

Окно диалога для выбора цвета позволяет обрабатывать только два события: OnShow и OnClose.

Окна диалога для работы с принтером

Delphi содержит компоненты для создания окон диалога, позволяющих работать с принтером:

- ❑ TPrinterSetupDialog — окно диалога для настройки принтера;
- ❑ TPrintDialog — окно диалога для задания параметров печати.

Состав элементов управления в окне диалога настройки принтера зависит от установленного драйвера принтера. Никаких важных свойств класс TPrinterSetupDialog не имеет, поскольку используемые в окне параметры устанавливаются непосредственно драйвером принтера.

Основные свойства класса TPrintDialog приведены в табл. 9.8.

Таблица 9.8. Основные свойства класса TPrintDialog

Свойство	Тип	Описание
Collate	Boolean	Состояние флажка Разобрать по копиям
Copies	Integer	Количество копий, заданное в окне диалога
FromPage	Integer	Номер страницы документа, начиная с которой будет производиться печать
MaxPage	Integer	Максимальный номер страницы, который может быть задан в окне диалога
MinPage	Integer	Минимальный номер страницы, который может быть задан в окне диалога
Options	TPrintDialogOptions	Параметры окна диалога
PrintRange	TPrintRange	Определяет диапазон печатаемых страниц: весь документ (prAllPages), выделенный фрагмент (prSelection) или страницы с указанными номерами (prPageNums)
PrintToFile	Boolean	Состояние флажка Печать в файл
ToPage	Integer	Номер последней печатаемой страницы

Окно диалога печати позволяет обрабатывать только два события: OnShow и OnClose.

Работа с базами данных в Delphi

Использование баз данных является одним из приоритетных направлений развития прикладного программного обеспечения. Среда Delphi всегда отличалась богатыми возможностями по поддержке систем доступа к базам данных, причем по мере выхода новых версий Delphi эти возможности постоянно расширялись. Изначально доступ к базам данных в Delphi обеспечивается *процессором баз данных Borland* (Borland Database Engine, BDE). В настоящее время актуальной технологией является dbGo, которая основана на *технологии ADO.NET* (ActiveX Data Objects for .NET), разработанной и поддерживаемой фирмой Microsoft. Эта технология аналогична BDE по назначению и довольно близка по возможностям.

Для отображения и редактирования данных, хранящихся в базах данных, в Delphi реализован ряд специально ориентированных на это компонентов.

Любое приложение, работающее с базами данных, должно предоставлять ряд типовых функциональных возможностей, обеспечивающих подключение к базе данных, считывание информации из таблиц базы данных, редактирование данных и навигацию по набору данных.

Понятие набора данных

Набор данных представляет собой двухмерную таблицу. Строки таблицы называются *записями*, столбцы — *полями*.

Таблицы баз данных не загружаются в память полностью ввиду их большого размера. В память загружаются только значения полей, относящиеся к какой-либо записи таблицы. Запись, значения полей которой загружены в память, называется *текущей*. Перемещение по набору данных означает загрузку в выделенную для хранения текущей записи память новых значений полей из таблицы базы данных.

С текущей записью связано понятие *курсора* набора данных. Курсор представляет собой объект, который содержит значения текущей записи и инкапсулирует методы, позволяющие загружать новые записи и при этом сохранять изменения, внесенные в текущую запись.

Доступ к данным с использованием dbGo

Механизм dbGo позволяет получить доступ к данным с помощью технологии Microsoft ADO, данный механизм работы с данными является на настоящий момент самым актуальным при создании приложений в среде Delphi.

Иерархия классов компонентов dbGo показана на рис. 9.3.

Класс TADOConnection

Класс TADOConnection позволяет установить подключение к источнику данных ADO. Данный компонент удобно использовать, например, для различных наборов данных TADODataset, в качестве значения свойства Connection. Это избавляет нас от необходимости индивидуально для каждого компонента задавать значение Connection String. Компонент TADOConnection может использоваться и с другими

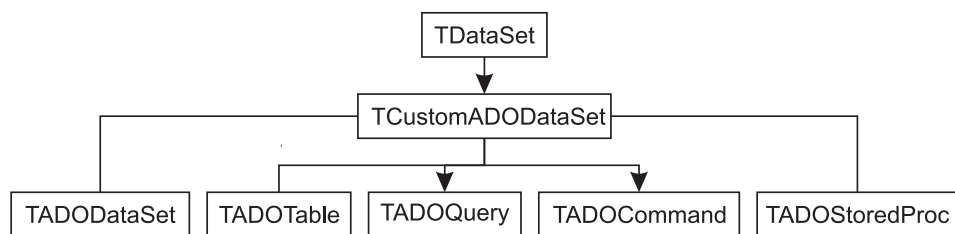


Рис. 9.3. Иерархия классов dbGo

компонентами **dbGo**, такими как **TADOCommand**, **TADOTable**, **TADOQuery**, **TADOStored-Procedure**.

Основные свойства класса **TADOConnection** приведены в таблице 9.9.

Таблица 9.9. Основные свойства класса **TADOConnection**

Свойство	Тип	Описание
Attributes	TXactAttribute	Определяет, открывать ли новую транзакцию автоматически: <ul style="list-style-type: none"> • xaCommitRetaining — при подтверждении транзакции; • xaAbortRetaining — при отмене транзакции
CommandCount	Integer	Показывает число командных (TADOCommand) компонентов, связанных с данным компонентом TADOConnection
Commands	TADOCommand	Содержит список команд (объектов TADOCommand), связанных с данным компонентом TADOConnection
CommandTimeout	Integer	Задает период времени в секундах, в течение которого будет производиться попытка выполнения команд. По умолчанию значение равно 30. Если команда не была выполнена по истечении времени, заданного данным свойством, то попытка выполнения команды прекращается
Connected	Boolean	Определяет, установлено ли подключение к источнику данных
ConnectionString	WideString	Задает строку подключения к источнику данных
ConnectionTimeout	Integer	Задает период времени в секундах (по умолчанию 15), в течение которого будет выполняться попытка установления подключения. Если в течение заданного времени подключение не будет установлено, то будет вызвано исключение
ConnectOptions	TConnectOption	Задает параметры подключения, т. е. определяет, будет подключение синхронным или асинхронным
CursorLocation	TCursorLocation	Определяет, где будет выполняться работа с набором данных на стороне клиента или на стороне сервера
DefaultDatabase	WideString	Задает базу данных, используемую компонентом TADOConnection по умолчанию. В случае если задано значение свойства ConnectionString , при открытии подключения значение свойства DefaultDatabase будет автоматически изменено на значение свойства ConnectionString
DataSets	TCustomADODataSet	Представляет собой массив, содержащий все активные наборы данных, связанных с данным компонентом TADOConnection

Свойство	Тип	Описание
Errors		Предоставляет доступ к коллекции Errors подключения ADO (то есть объекту ADO Errors Collection), который связан с данным компонентом TADOConnection
InTransaction	Boolean	Показывает, когда транзакция находится в процессе выполнения
IsolationLevel	TIsolationLevel	<p>Определяет тип транзакции.</p> <p>IIUnspecified — сервер самостоятельно определяет подходящий тип изоляции.</p> <p>IIChaos — транзакции с более высоким уровнем изоляции не могут изменять данные, измененные, но не подтвержденные в текущей транзакции.</p> <p>IIReadUncommitted — чтение данных измененных в не подтвержденных транзакциях. То есть изменения видны сразу после того, как другая транзакция передала их на сервер.</p> <p>IIBrowse — то же самое, что и IIReadUncommitted.</p> <p>IIReadCommitted — чтение данных, измененных подтвержденными транзакциями. То есть изменение данных будет видимо после выполнения Commit в другой транзакции.</p> <p>IICursorStability — то же самое, что и RealCommitted.</p> <p>IIRepeatableRead — изменения, сделанные другими транзакциями, не видимы, но при выполнении перезапроса транзакция может получать новый набор данных.</p> <p>IIIsolated — транзакция не видит изменений данных, произведенных другими транзакциями.</p> <p>IISerializable — то же самое, что и IIIsolated</p>
KeepConnection	Boolean	Определяет, будет ли соединение открыто при закрытии последнего набора данных
Mode	TConnectMode	Показывает режимы доступа, доступные для подключения
Provider	WideString	Задает провайдера данных для подключения
State	TObjectStates	Показывает текущее состояние подключения
Version	WideString	Показывает версию ADO

Класс TADOConnection имеет ряд методов, которые позволяют выполнять те или иные действия с подключением к источнику данных.

```
procedure Open(const UserID: WideString; const Password: WideString); overload;
```

Открывает подключение к источнику данных. UserID — имя пользователя, Password — пароль.

```
procedure Close;
```

Закрывает подключение к источнику данных.

```
procedure Cancel();
```

Прерывает процесс подключения к источнику данных. Метод Cancel может использоваться только с асинхронными подключениями (то есть свойство ConnectOptions должно иметь значение coAsyncConnect).

function BeginTrans(): Integer;

Начинает новую транзакцию для источника данных.

procedure CommitTrans();

Подтверждает открытую транзакцию.

procedure RollbackTrans();

Выполняет откат транзакции.

Класс TADOConnection имеет также ряд событий. Рассмотрим ниже некоторые из них.

property AfterConnect: TNotifyEvent;

Данное событие происходит после того, как установлено подключение к источнику данных.

property BeforeConnect: TNotifyEvent;

Данное событие происходит непосредственно перед тем, как будет установлено подключение к источнику данных.

property AfterDisconnect: TNotifyEvent;

Данное событие происходит после закрытия подключения к источнику данных.

property BeforeDisconnect: TNotifyEvent;

Данное событие происходит непосредственно перед закрытием подключения к источнику данных.

property OnDisconnect: TDisconnectEvent;

Данное событие происходит после того, как подключение к источнику данных закрыто. Оно происходит после вызова метода Close или после того, как свойству Connected будет присвоено значение false.

property OnConnectComplete: TConnectErrorEvent;

Данное событие происходит после того, как подключение к источнику данных было успешно установлено. Оно происходит после вызова метода Open или после того, как свойству Connected будет присвоено значение true.

Класс TADODataset

Класс TADODataset позволяет работать с набором данных, полученным из источника данных ADO. Основные свойства класса TADODataset приведены в табл. 9.10.

Таблица 9.10. Основные свойства класса TADODataset

Свойство	Тип	Описание
CacheSize	Integer	Задаёт размер кэша для набора данных
CommandText	WideString	Задаёт команду. Значение этого свойства представляет собой текст SQL-оператора, имени таблицы или имени хранимой процедуры
CommandTimeout	Integer	Задаёт период времени в секундах, в течение которого будет производиться попытка выполнения команд. По умолчанию значение равно 30. Если команда не была выполнена по истечении времени, заданного данным свойством, то попытка выполнения команды прекращается

Свойство	Тип	Описание
CommandType	TCommandType	Задаёт тип команды
Connection	TADOConnection	Определяет компонент TADOConnection, связанный с данным набором данных
ConnectionString	WideString	Задаёт строку подключения к источнику данных. Вы можете задать значение этого свойства вручную, введя все необходимые параметры, или в режиме редактирования формы с помощью инспектора объектов, вызвав соответствующее диалоговое окно
CursorLocation	TCursorLocation	Определяет, где будет выполняться работа с набором данных, на стороне клиента или на стороне сервера
CursorType	TCursorType	Определяет тип курсора, используемый набором данных ADO
EnableBCD	Boolean	Определяет, будут ли числовые поля обрабатываться как значения с плавающей точкой или значения типа BCD
ExecuteOptions	TExecuteOptions	Определяет параметры выполнения команды
Filter	String	Определяет текст текущего фильтра для набора данных
Filtered	Boolean	Определяет, будет ли фильтроваться набор данных или нет
IndexDefs	TIndexDefs	Содержит информацию об индексах таблицы
IndexFieldCount	Integer	Количество полей, относящихся к текущему индексу таблицы
LockType	TADOLockType	<p>Определяет тип блокировки записей, используемой при открытии набора данных. Возможные значения данного свойства:</p> <ul style="list-style-type: none"> • ltUnspecified — библиотека ADO сама определяет, какой тип будет использоваться; • ltReadOnly — только чтение, изменение данных невозможно; • ltPessimistic — пессимистическая блокировка. Запись блокируется сразу после начала редактирования и до сохранения записей; • ltOptimistic — оптимистическая блокировка. Запись блокируется, только когда изменения сохраняются; • ltBatchOptimistic — то же самое, что и ltOptimistic, но используется отложенное сохранение изменений записей. Более подробно она рассматривается в следующем пункте; • MarshalOptions — это свойство определяет, будут ли отправлены на сервер те поля, которые не были изменены. При значении moMarshalAll будут, а при moMarshalModifiedOnly не будут. <p>Тип блокировки должен быть задан перед открытием набора данных. Значением по умолчанию является значение ltOptimistic</p>
MarshalOptions	TMarshalOption	Определяет, будут ли отправлены на сервер те поля, которые не были изменены
MasterFields	WideString	Задаёт поля главной таблицы, используемые при организации связи с другими таблицами
MaxRecords	Integer	Определяет максимальное количество записей, которое компонент возвращает из источника данных
ParamCheck	Boolean	Определяет, обновлять ли список параметров при изменении текста SQL-оператора
Parameters	TParameters	Содержит коллекцию параметров SQL-оператора

Таблица 9.10 (продолжение)

Свойство	Тип	Описание
Prepared	Boolean	Определяет, будет ли запрос подготовлен заранее. Задание значения этого свойства равным true позволяет избежать подготовки запроса при каждом его открытии
StoreDefs	Boolean	Определяет, где будут сохраняться определения полей и индексов. Если значение равно true, то определения полей и индексов будут храниться в форме или модуле данных. В противном случае они будут храниться в файлах базы данных

Класс TADODataset содержит также ряд методов, которые позволяют обрабатывать набор данных во время выполнения программы. Основные из них перечислены ниже.

```
procedure CancelBatch(AffectRecords: TAffectRecords);
```

Отменяет внесенные, но не примененные (отложенные) изменения.

```
procedure UpdateBatch(AffectRecords: TAffectRecords);
```

Сохраняет отложенные изменения на диск.

Параметр AffectRecords может иметь одно из следующих значений:

- ☐ arCurrent — изменения применяются только к текущей записи;
- ☐ arFiltered — изменения применяются только к записям, которые отфильтрованы с помощью текущего фильтра;
- ☐ arAll — изменения применяются только ко всем записям набора данных.

ПРИМЕЧАНИЕ

Чтобы работать с отложенными изменениями, свойство CursorType должно иметь значение ctKeySet или ctStatic, а свойство LockType должно иметь значение ltBatchOptimistic.

```
procedure CancelUpdates();
```

Данный метод выполняет те же самые действия, что и метод CancelBatch с параметром AffectRecords равным arAll.

```
procedure DeleteRecords(AffectRecords: TAffectRecords);
```

Метод DeleteRecords удаляет одну или большее количество записей. Количество удаляемых записей зависит от значения параметра AffectRecords, который имеет одно из следующих значений:

- arCurrent — удаляется только текущая запись;
- arFiltered — удаляются только записи, которые отфильтрованы с помощью текущего фильтра;
- arAll — удаляются все записи набора данных;
- arAllChapters — удаляются записи во всех главах ADO (ADO chapters).

```
function Locate(const KeyFields: String;  
  const KeyValues: Variant; Options: TLocateOptions): Boolean;
```

Ищет запись, удовлетворяющую заданному критерию. KeyFields — список полей, по которым ведется поиск; KeyValues — поисковое значение; Options — условия поиска. При успешном поиске возвращает true и устанавливает курсор на найденную запись.

```
function Lookup(const KeyFields: string; var KeyValues: Variant; const  
  ResultFields: string): Variant;
```

Данный метод получает значения полей из записи, которая соответствует заданному критерию поиска. KeyFields — список полей, по которым ведется поиск; KeyValues — поисковое значение; ResultFields — представляет собой строку, содержащую список полей, разделенный точками с запятой, значения которых будут возвращены из найденной строки.

```
procedure Requery(Options: TExecuteOptions);
```

Данный метод выполняет обновление набора данных. Оно выполняется путем повторного выполнения SQL-оператора. Вызов данного метода аналогичен последовательному вызову двух методов Close и Open.

Класс TADODataSet имеет ряд событий.

```
property OnFieldChangeComplete: TFieldChangeCompleteEvent;
```

Данное событие происходит после того, как было изменено поле.

```
property OnRecordChangeComplete: TRecordChangeCompleteEvent;
```

Данное событие происходит после того, как была изменена одна или несколько записей в наборе данных. Таким образом, данное событие происходит после добавления, изменения и удаления строки набора данных, а также после отмены этих действий.

```
property OnRecordsetChangeComplete: TRecordsetErrorEvent;
```

Данное событие происходит после того, как изменяется набор записей, то есть после открытия, закрытия, обновления и синхронизации набора данных.

События

```
property OnWillChangeField: TWillChangeFieldEvent;
```

```
property OnWillChangeRecord: TWillChangeRecordEvent;
```

```
property OnWillChangeRecordset: TRecordsetReasonEvent;
```

аналогичны трем рассмотренным выше с той разницей, что они происходят до, а не после соответствующих операций с данными.

```
property OnMoveComplete: TRecordsetErrorEvent;
```

Данное событие происходит после изменения указателя на текущую запись. Вызов одного из методов: First, Last, Next, Prior, MoveBy, Insert, Delete, Post, Requery и Resynch приводит к данному событию.

```
property OnWillMove: TRecordsetReasonEvent;
```

Данное событие происходит до изменения указателя на текущую запись.

```
property OnEndOfRecordset: TEndOfRecordsetEvent;
```

Данное событие происходит при достижении конца набора данных. Его обработчик можно использовать, например, для добавления новой записи.

property OnFetchComplete: TRecordsetEvent;

Данное событие происходит после того, как записи были приняты в набор данных, то есть оно происходит после вызова метода Open или задания свойству Active значения true.

property OnFetchProgress: TFetchProgressEvent;

Данное событие происходит периодически, в процессе получения данных при асинхронных операциях.

Класс TADOQuery

Класс TADOQuery позволяет выполнять запросы к источнику данных. Основные свойства класса TADOQuery приведены в табл. 9.11.

Таблица 9.11. Основные свойства класса TADOQuery

Свойство	Тип	Описание
DataSource	TDataSource	Определяет компонент DataSource, из которого берется значение поля, используемое в качестве значения параметра с тем же именем, что и имя поля
RowsAffected	Integer	Показывает количество строк, которые были изменены или удалены в результате последнего выполнения запроса. Если значение этого свойства равно -1, то это означает, что в процессе выполнения запроса произошла ошибка
SQL	TWideStrings	Содержит строку SQL-оператора запроса

Класса TADOQuery имеет ряд методов:

function ExecSQL(): Integer;

Метод выполняет SQL-оператор, заданный значением свойства SQL. SQL-оператор должен быть из ряда INSERT, UPDATE, DELETE, CREATE TABLE.

Класс TADOQuery имеет те же события, что и класс TADODataset.

Класс TADOTable

Основные свойства класса TTable приведены в табл. 9.12.

Таблица 9.12. Основные свойства класса TTable

Свойство	Тип	Описание
Active	Boolean	Определяет, открыт набор данных или нет. После установки этого свойства в true можно производить запись и чтение информации, хранящейся в базе данных
BOF	Boolean	Если данное свойство равно true, то текущая позиция в наборе данных находится на первой записи
FieldCount	Integer	Количество полей в текущей записи
Fields	TFields	Список полей текущей записи. Используется для доступа к отдельным полям
IndexDefs	TIndexDefs	Содержит информацию об индексах таблицы
IndexFieldCount	Integer	Количество полей, относящихся к текущему индексу таблицы
IndexFieldNames	String	Список полей, используемых в качестве текущего индекса таблицы

Свойство	Тип	Описание
IndexFields: [Index: Integer]	TField	Список полей текущего индекса
IndexName	String	Вторичный индекс для таблицы
MasterFields	String	Задаёт поля главной таблицы, используемые при организации связи с другими таблицами
MasterSource	TDataSource	Источник данных, связанный с данным набором данных и являющийся главным в отношении «главный/подчиненный»
Modified	Boolean	Определяет, была ли изменена текущая запись
ReadOnly	Boolean	Включает и выключает режим «только для чтения»
RecordCount	LongInt	Количество записей в наборе данных
TableName	TFileName	Имя таблицы
TableDirect	Boolean	Определяет, каким образом осуществляется доступ к таблице: по имени таблицы или с использованием SQL-оператора. Не все провайдеры данных позволяют получить доступ к таблице просто через ее имя, некоторые требуют использования оператора SELECT. Если данное свойство имеет значение true, то данные запрашиваются с помощью фонового SQL-оператора. Если значение свойства равно false, то компонент создает оператор SELECT для выборки данных таблицы. По умолчанию свойство имеет значение false
TableName	WideString	Определяет имя таблицы, с которой ведется работа, посредством компонента. Если вы хотите сменить имя таблицы компонента во время выполнения программы, то вы должны сначала закрыть таблицу, например с помощью метода Close

Класс TADOTable имеет метод GetIndexNames:

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

Данный метод позволяет получить список индексов данной таблицы.

Класс TADOTable имеет те же события, что и класс TADODataset.

Класс TADOStoredProc

Класс TADOStoredProc позволяет работать с хранимыми процедурами с помощью технологии ADO.

Класс TADOStoredProc имеет основное свойство ProcedureName типа WideString. Данное свойство определяет хранимую процедуру, связанную с компонентом TADOStoredProc.

Основным методом класса TADOStoredProc является метод ExecProc, который запускает выполнение хранимой процедуры на сервере. Перед вызовом данного метода необходимо присвоить нужное значение свойству Parameters. Синтаксис метода ExecProc:

```
procedure ExecProc();
```

Класс TADOStoredProc имеет те же события, что и класс TADODataset.

Класс TADOCCommand

Класс TADOCCommand позволяет работать с командой ADO.
Основные свойства класса TADOCCommand приведены в табл. 9.13.

Таблица 9.13. Основные свойства класса TADOCCommand

Свойство	Тип	Описание
CommandText	WideString	Задает команду. Значение этого свойства представляет собой текст SQL-оператора, имени таблицы или имени хранимой процедуры
CommandTimeout	Integer	Задает период времени в секундах, в течение которого будет производиться попытка выполнения команд. По умолчанию значение равно 30. Если команда не была выполнена по истечении времени, заданного данным свойством, то попытка выполнения команды прекращается
CommandType	TCommandType	Задает тип команды
Connection	TADOConnection	Определяет, какой компонент TADOConnection будет использоваться для связи с источником данных
ConnectionString	WideString	Определяет строку подключения к источнику данных ADO
ExecuteOptions	TExecuteOptions	Определяет параметры выполнения команды
ParamCheck	Boolean	Определяет, обновлять ли список параметров при изменении текста SQL-оператора
Parameters	TParameters	Содержит коллекцию параметров SQL-оператора
Prepared	Boolean	Определяет, будет ли запрос подготовлен заранее. Задав значение этого свойства равным true, вы позволяете избежать подготовки запроса при каждом его открытии
States	TObjectStates	Показывает текущее состояние подключения

Класс TADOCCommand имеет ряд методов.
procedure Cancel();

Метод Cancel останавливает выполнение команды ADO. Может быть остановлена только команда, выполняемая асинхронно (то есть в свойстве TExecuteOption значение eoAsyncExecute должно быть true). Если команда выполняется синхронно, то вызывается исключение.

procedure Assign(Source: TPersistent); override;

Метод Assign делает один компонент TADOCCommand дубликатом другого.
function Execute(): _Recordset; overload;
function Execute(var Parameters: OleVariant): _Recordset; overload;
function Execute(var RecordsAffected: Integer; var Parameters: OleVariant): _Recordset; overload;

Данный метод позволяет выполнить команду ADO, которая задана свойством CommandText. Аргумент RecordsAffected задает количество записей, с которым работает команда, в том случае если команда ADO выполняется над данными. Аргумент Parameters представляет собой коллекцию параметров, в том случае если команда имеет параметры.

Доступ к данным с использованием dbExpress

Технология dbExpress представляет собой набор драйверов, которые предоставляют доступ к серверам баз данных. Для каждого сервера баз данных, поддерживаемого данной технологией, существует отдельный драйвер. Этот драйвер учитывает особенности конкретного сервера баз данных и адаптирует эти особенности к универсальному интерфейсу технологии dbExpress. Каждый драйвер представляет собой динамически подключаемую библиотеку.

Технология dbExpress предоставляет пять интерфейсов для выборки метаданных, выполнения операторов SQL и хранимых процедур. При этом данные читаются в одном направлении, то есть используются однонаправленные курсоры (unidirectional cursor).

Иерархия классов для работы с наборами данных компонентов dbExpress приведена на рис. 9.4.

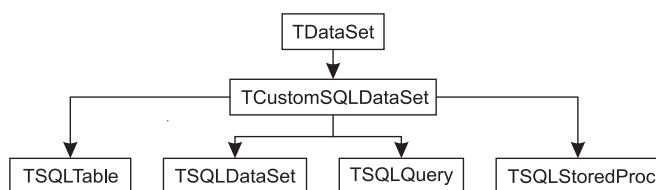


Рис. 9.4. Иерархия классов dbExpress

Класс TSQLConnection

Класс TSQLConnection позволяет работать с подключением к серверу баз данных посредством технологии dbExpress. Основные свойства класса TSQLConnection приведены в табл. 9.14.

Таблица 9.14. Основные свойства класса TSQLConnection

Свойство	Тип	Описание
Connected	Boolean	Задайте этому свойству значение true, чтобы установить подключение к базе данных. Чтобы закрыть подключение, задайте этому значению свойство false
ConnectionName	String	Данное свойство позволяет использовать именованную конфигурацию подключения. Задание свойства ConnectionName приводит к тому, что автоматически устанавливаются значения свойств DriverName и Params, чтобы соответствовать драйверу и параметрам подключения, сохраненным под именем, заданным ConnectionName в файле dbxconnections.ini
DriverName	String	Если вы задаете значение свойству ConnectionName, то значение свойства должно содержать имя одного из установленных драйверов dbExpress, таких как INTERBASE, MYSQL, INFORMIX, ORACLE или DB2. Установленные драйверы перечислены в файле dbxdrivers.ini, вы можете получить их список с помощью функции GetDriverNames
GetDriverFunc	String	Определяет имя функции в драйвере dbExpress, который задается свойством LibraryName. Обычно значение свойства GetDriverFunc представляет собой имя функции, которая хранится в файле dbxdrivers.ini

Таблица 9.14 (продолжение)

Свойство	Тип	Описание
KeepConnection	Boolean	Определяет, будет ли подключение открыто, если не открыто ни одного набора данных
LibraryName	String	Задаёт драйвер, который используется для подключения. Значение этого свойства означает библиотеку, ассоциированную с драйвером, заданным свойством <code>DriverName</code> . Большинству приложений не нужно задавать значение свойства <code>LibraryName</code> напрямую, т. к. это свойство автоматически назначается при задании значения свойства <code>DriverName</code>
LoadParamsOnConnect	Boolean	Определяет, когда компонент <code>TSQLConnection</code> загружает конфигурацию с именем, заданным свойством <code>ConnectionName</code> непосредственно перед подключением к серверу
Params	TWideStrings	Данное свойство представляет собой объект типа <code>TStrings</code> , содержащий список параметров подключения. Каждый элемент списка имеет вид <code>Name=Value</code> , где <code>Name</code> — это имя параметра, а <code>Value</code> — его значение
TableScope	TTableScopes	Показывает, какой тип таблиц будет возвращен при вызове метода <code>GetTableNames</code>
VendorLib	String	Задаёт динамически подключаемую библиотеку (DLL) поставляемую вендором СУБД. Обычно это свойство задавать не требуется

Класс `TSQLConnection` имеет ряд методов. Рассмотрим некоторые из них. Метод `CloseDataSets` позволяет закрыть все наборы данных, ассоциированные с текущим подключением, при этом не происходит отключения от сервера баз данных. Синтаксис метода `CloseDataSets`:

```
procedure CloseDataSets();
```

Метод `Commit` фиксирует открытую транзакцию. Синтаксис метода `Commit`:

```
procedure Commit(TransDesc: TTransactionDesc);
```

Метод `Rollback` отменяет все обновления, вставки и удаления для указанной транзакции и завершает ее. Синтаксис метода `Rollback`:

```
procedure Rollback(TransDesc: TTransactionDesc);
```

Метод `StartTransaction` начинает транзакцию в связанной с компонентом базе данных. Синтаксис метода `StartTransaction`:

```
procedure StartTransaction(TransDesc: TTransactionDesc);
```

Также класс `TSQLConnection` имеет, аналогично классу `TADOConnection`, методы `Open` и `Close`, которые соответственно открывают и закрывают подключение.

Класс `TSQLConnection` имеет событие `OnLogin`:

```
property OnLogin: TSQLConnectionLoginEvent;
```


Класс TSQLDataSet

Основные свойства TSQLDataSet представлены в табл. 9.15.

Таблица 9.15. Основные свойства класса TSQLDataSet

Свойство	Тип	Описание
CommandText	WideString	Задаёт текст команды, выполняемой данным компонентом. Значение этого свойства зависит от значения свойства CommandType: <ul style="list-style-type: none"> • если значение свойства CommandType равно ctQuery, то значение свойства CommandText является SQL-оператором; • если значение свойства CommandType равно ctStoredProc, то значение свойства CommandText является именем хранимой процедуры; • если значение свойства CommandType равно ctTable, то значение свойства CommandText является именем таблицы на сервере баз данных
CommandType	TSQLCommandType	Определяет смысл значения свойства CommandText. Значения свойства CommandType были приведены в предыдущей строке таблицы
DataSource	TDataSource	Определяет ссылку на другой набор данных, который предоставляет значения для параметров данного набора данных
GetMetadata	Boolean	Определяет, когда набор данных будет получать данные вместе с метаданными
MaxBlobSize	Integer	Определяет максимальное количество байт, которое получает набор данных для каждого поля типа BLOB
Params	TParams	Представляет собой параметры запроса или хранимой процедуры. Данное свойство содержит коллекцию объектов TParams, каждый из которых представляет собой параметр запроса или хранимой процедуры
SortFieldNames	WideString	Определяет порядок сортировки, когда набор данных имеет тип ctTable. Чтобы задать порядок сортировки, задайте значение данного свойства в виде списка имен полей, разделенных точками с запятой
SQLConnection	TSQLConnection	Задаёт компонент SQL Connection, который определяет подключение для данного набора данных

Метод ExecSQL является основным методом компонента TSQLDataSet. Синтаксис метода:

```
function ExecSQL(ExecDirect: Boolean): Integer; override;
```

Этот метод запускает выполнение запроса или хранимой процедуры, которые не возвращают набор данных. Набор данных не возвращают такие SQL-операторы, как INSERT, UPDATE, DELETE и CREATE TABLE. Параметр ExecDirect указывает, нужно или нет подготавливать оператор перед выполнением. Его обычно устанавливают в true, если запрос не имеет параметров. Если значение этого параметра равно false, то запрос будет подготовлен перед выполнением. Метод ExecSQL возвращает количество строк, задействованных при выполнении команды.

Компонент TSQLQuery

Основные свойства TSQLQuery представлены в табл. 9.16.

Таблица 9.16. Основные свойства класса TSQLQuery

Свойство	Тип	Описание
SQL	TwideStrings	Задаёт SQL-оператор запроса, который будет выполнен на сервере баз данных при открытии запроса. Если SQL-оператор представляет собой оператор SELECT, то запрос начнет выполнение оператора при присвоении свойству Active значения true или вызове метода Open. Если SQL-оператор не возвращает набор данных, то для выполнения этого SQL-оператора следует вызывать метод ExecSQL
Text	String	Данное свойство позволяет задать SQL-оператор как одиночную строку, в отличие от свойства SQL, которое позволяет разбить строку на несколько подстрок

Класс TSQLQuery имеет ряд методов (Open, Close, First, Last, Next и т. д.), которые позволяют выполнять те или иные действия с запросами. Эти методы наследованы в основном от класса TDataSet.

Компонент TSQLTable

Основные свойства TSQLTable представлены в табл. 9.17.

Таблица 9.17. Основные свойства класса TSQLTable

Свойство	Тип	Описание
IndexFieldNames	WideString	Представляет собой список полей, используемый для сортировки записей и для связи главный/подчиненный. Значение этого свойства должно представлять собой список имен полей, разделенный точками с запятой
IndexName	WideString	Представляет собой индекс для сортировки записей или для связи таблиц главный/подчиненный
MasterFields	WideString	Задаёт поля из главного набора данных для использования этой таблицы как подчиненной в отношении главный/подчиненный
MasterSource	TDataSource	Задаёт главный набор данных для использования данной таблицы как подчиненной в отношении главный/подчиненный
TableName	WideString	Задаёт имя таблицы на сервере баз данных для данного компонента

Компонент TSQLStoredProc

Основные свойства TSQLStoredProc представлены в табл. 9.18.

Таблица 9.18. Основные свойства класса TSQLStoredProc

Свойство	Тип	Описание
PackageName	WideString	Определяет имя пакета Oracle, которому принадлежит данная хранимая процедура. Это свойство следует применять только при использовании сервера баз данных Oracle
StoredProcName	WideString	Задаёт имя хранимой процедуры, которая будет выполнена на сервере баз данных

Компонент TSQLMonitor

Основные свойства TSQLMonitor представлены в табл. 9.19.

Таблица 9.19. Основные свойства класса TSQLMonitor

Свойство	Тип	Описание
Active	Boolean	Позволяет запустить или остановить мониторинг команд, переданных на сервер баз данных. При присваивании данному свойству значения true начинается мониторинг взаимодействия между компонентом SQLConnection и сервером баз данных. При задании значения false мониторинг останавливается
AutoSave	Boolean	Определяет, будут ли сообщения записываться в файл в процессе мониторинга
FileName	string	Задаёт имя файла, в который будут записываться сообщения, получаемые в процессе мониторинга
SQLConnection	TSQLConnection	Задаёт компонент TSQLConnection, мониторинг сообщений которого будет производиться
TraceList	TWideStrings	Список сообщений, которые были переданы между компонентом TSQLConnection и сервером баз данных. С помощью этого свойства вы можете получить доступ к отдельным командам в списке

Компонент TSimpleDataSet

Основные свойства TSimpleDataSet представлены в табл. 9.20.

Таблица 9.20. Основные свойства класса TSimpleDataSet

Свойство	Тип	Описание
Connection	TSQLConnection	Задаёт компонент TSQLConnection, который подключает данный набор данных к серверу баз данных
DataSet	TInternalSQLDataSet	Позволяет получить доступ к внутреннему набору данных, который получает данные от сервера баз данных

Доступ к данным с использованием BDE

Механизм BDE обеспечивает доступ как к локальным, так и к распределённым базам данных. Взаимодействие с ними осуществляется через драйверы. В поставку Delphi включены стандартные драйверы для работы с локальными базами данных dBase, Paradox и FoxPro, а также с SQL-серверами Oracle, Informix, Sybase, DB2 и InterBase.

Механизм BDE представляет собой программную прослойку между прикладной программой и базой данных. Запрос из приложения передается внутрь механизма BDE, который использует драйверы для непосредственной работы с соответствующей базой данных.

Реализация в 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 параметров BDE, практически не исправляя исходные тексты прикладной программы.

ПРИМЕЧАНИЕ

Механизм доступа к данным BDE оставлен в новых версиях Delphi в основном для совместимости с предыдущими версиями. При создании приложений с нуля в качестве инструмента для работы с базами данных рекомендуется использовать dbGo.

Для организации доступа к базе данных используются следующие компоненты:

- ❑ TTable — обеспечивает доступ к таблицам локальных баз данных и управление ими;
- ❑ TQuery — использует для доступа к базе данных SQL-запросы, поэтому позволяет работать как с локальными, так и с распределенными базами данных.

Данные компоненты расположены на вкладке Data Access палитры компонентов.

Классы TTable и TQuery являются потомками одного и того же класса TDataSet, инкапсулирующего абстрактный набор данных и наиболее общие методы работы с ним. Иерархия классов, используемых для работы с набором данных, изображена на рис. 9.5.

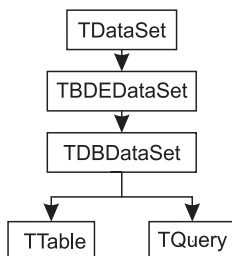


Рис. 9.5. Иерархия классов BDE

В данном разделе мы подробно рассмотрим класс TTable, предназначенный для работы с локальными базами данных. Класс TQuery будет рассмотрен в главе 11, посвященной языку SQL.

Основные свойства класса TTable приведены в табл. 9.21.

Таблица 9.21. Основные свойства класса TTable

Свойство	Тип	Описание
Active	Boolean	Определяет, открыт набор данных или нет. После установки этого свойства в true можно производить запись и чтение информации, хранящейся в базе данных
BOF	Boolean	Если данное свойство равно true, то текущая позиция в наборе данных находится на первой записи
DatabaseName	String	Путь к базе данных, связанной с компонентом TTable

Свойство	Тип	Описание
DefaultIndex	Boolean	Управляет сортировкой данных в таблице. Для значения true сортировка производится по первичному ключу, для значения false данные не сортируются
EOF	Boolean	Если данное свойство равно true, то текущая позиция в наборе данных находится на последней записи
Exclusive	Boolean	Ограничивает доступ к таблице. Если установлено значение true, то с таблицей может работать только одно приложение
Exists	Boolean	Определяет, существует ли связанная с компонентом TTable таблица (если true, то существует)
FieldCount	Integer	Количество полей в текущей записи
Fields	TFields	Список полей текущей записи. Используется для доступа к отдельным полям
IndexDefs	TIndexDefs	Содержит информацию об индексах таблицы
IndexFieldCount	Integer	Количество полей, относящихся к текущему индексу таблицы
IndexFieldNames	String	Список полей, используемых в качестве текущего индекса таблицы
IndexFields: [Index: Integer]	TField	Список полей текущего индекса
IndexFiles	TStrings	Индексные файлы, используемые для таблиц dBase
IndexName	String	Вторичный индекс для таблицы
KeyExclusive	Boolean	Определяет, как интерпретировать границы диапазона, задаваемые методом SetRange. Если установлено значение true, то записи, соответствующие границам, не включаются в диапазон
KeyFieldCount	Integer	Количество полей ключа, используемое при поиске. Если равно 0, то используется первое поле, если 1 — первое и второе и т. д.
MasterFields	String	Задаёт поля главной таблицы, используемые при организации связи с другими таблицами
MasterSource	TDataSource	Источник данных, связанный с данным набором и являющийся главным в отношении главный/подчиненный
Modified	Boolean	Определяет, была ли изменена текущая запись
ReadOnly	Boolean	Включает и выключает режим «только для чтения»
RecordCount	LongInt	Количество записей в наборе данных
TableLevel	Integer	Уровень таблицы, используемый в драйвере BDE
TableName	TFileName	Имя таблицы
TableType	TTableType = (ttDefault, ttParadox, ttDBase, ttASCII, ttFoxPro);	Тип таблицы

Класс TTable содержит также ряд методов, которые позволяют обрабатывать набор данных во время выполнения программы. Основные из них перечислены ниже.

```
procedure AddIndex(const Name, Fields : string;  
Options : TIndexOptions);
```

Создает новый индекс. Name — имя создаваемого индекса, Fields — список его полей (перечисляются через запятую), Options — тип индекса.

```
procedure ApplyRange;
```

Применяет заданный диапазон к набору данных.

```
procedure Cancel;
```

Отменяет изменения, внесенные в запись в режиме редактирования.

```
procedure CancelRange;
```

Удаляет текущий диапазон.

```
procedure DeleteTable;
```

Удаляет таблицу базы данных. Набор данных при этом должен быть закрыт.

```
procedure Edit;
```

Переводит текущую запись в режим редактирования.

```
procedure EditKey;
```

Включает режим редактирования буфера поиска.

```
procedure EditRangeEnd;
```

Позволяет изменить начало существующего диапазона. После вызова этого метода для изменения границы диапазона следует вызвать метод FieldByName. Для таблиц Paradox и dBase данный метод работает только с индексированными полями.

```
procedure EditRangeStart;
```

Позволяет изменить конец диапазона.

```
procedure EmptyTable;
```

Удаляет все записи из таблицы.

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

Предоставляет доступ к полю таблицы по его имени.

```
function FindKey(const KeyValues: array of const): Boolean;
```

Производит поиск записей, соответствующих значениям полей, заданных в параметре KeyValues. Значения разделяются запятыми. При успешном поиске возвращает true.

```
procedure FindNearest(const KeyValues: array of const);
```

Перемещает курсор на запись, которая наиболее точно соответствует значениям полей, заданных в параметре KeyValues.

```
procedure First;
```

Перемещает курсор на первую запись в таблице.

function GotoKey: Boolean;

Перемещает курсор на запись, соответствующую текущему значению в буфере поиска. При успешном выполнении возвращает true. Если соответствующая запись не найдена, положение курсора не изменяется и возвращается false.

procedure GotoNearest;

Перемещает курсор на запись, наиболее точно соответствующую значению в буфере поиска.

procedure Insert;

Создает пустую запись в наборе данных и переводит ее в режим редактирования.

procedure Last;

Перемещает курсор на последнюю запись в наборе данных.

function Locate(const KeyFields: String;
const KeyValues: Variant; Options: TLocateOptions): Boolean;

Ищет запись, удовлетворяющую заданному критерию. KeyFields — список полей, по которым ведется поиск; KeyValues — поисковое значение; Options — условия поиска. При успешном поиске возвращает true и устанавливает курсор на найденную запись.

procedure LockTable(LockType: TLockType);

Блокирует доступ к таблице Paradox или dBase из других приложений.

function MoveBy(Distance: Integer): Integer;

Перемещает курсор на количество записей, заданное параметром Distance. Возвращает число записей, на которое переместился курсор (оно отличается от заданного, если достигнут конец или начало таблицы).

procedure Next;

Перемещает курсор на следующую запись.

procedure RenameTable(const NewTableName: String);

Переименовывает таблицу Paradox или dBase, связанную с текущим набором данных.

procedure Post;

Сохраняет изменения, внесенные в запись в режиме редактирования.

procedure Prior;

Перемещает курсор на предыдущую запись.

procedure SetKey;

Освобождает буфер поиска и позволяет заносить в него новые значения полей с использованием метода FieldByName.

procedure SetRange(const StartValues, EndValues: array of const);

Задает диапазон отбора записей и применяет его. StartValues — значения полей для начала диапазона, EndValues — для конца диапазона.

procedure SetRangeEnd;

Позволяет задать конец диапазона с помощью метода FieldByName.

procedure SetRangeStart;

Позволяет задать начало диапазона с помощью метода FieldByName.

procedure UnlockTable (LockType: TLockType);

Отменяет блокировку таблицы, заданную методом LockTable.

Состояния набора данных

В зависимости от выполняемых над данными действий набор данных может находиться в различных состояниях, которые можно условно разделить на две группы:

- ❑ состояния, в которые набор данных переводится посредством изменения свойств или вызовов методов класса TTable;
- ❑ состояния, в которые набор данных переходит автоматически.

Текущее состояние набора данных можно определить с помощью свойства State. Данное свойство имеет тип TDataSetState, описываемый следующим образом:

```
TDataSetState = (dsInactive, dsBrowse, dsEdit, dsInsert,  
dsSetKey, dsCalcFields, dsFilter, dsNewValue, dsOldValue,  
dsCurValue, dsBlockRead, dsInternalCalc, dsOpening);
```

Пять из тринадцати состояний управляются из приложения. Перевод в эти состояния набора данных осуществляется вызовом определенных методов или изменением значений определенных свойств.

- ❑ dsInactive — набор данных закрыт, данные недоступны для просмотра и редактирования. Переход в это состояние производится вызовом метода Close или установкой свойства Active в значение false.
- ❑ dsBrowse — набор данных открыт и доступен для просмотра, но не для редактирования. Данное состояние устанавливается вызовом метода Open или установкой свойства Active в значение true.
- ❑ dsEdit — набор данных открыт и доступен для редактирования. Для установления этого состояния используется метод Edit. При перемещении курсора на другую запись набор данных автоматически переходит в состояние dsBrowse.
- ❑ dsInsert — в набор данных добавляется новая запись. Набор данных переходит в это состояние после вызова метода Insert. При этом к набору данных добавляется пустая запись в текущую позицию курсора. При переходе на другую запись набор данных автоматически переходит в состояние dsBrowse.
- ❑ dsSetKey — в наборе данных производится установка ключа для поиска. Данное состояние сохраняется до вызова метода FindKey.

Следующие восемь состояний устанавливаются автоматически:

- ❑ dsNewValue — при обращении к свойству NewValue;
- ❑ dsOldValue — при обращении к свойству OldValue;

- ❑ dsCurValue — при обращении к свойству CurValue;
- ❑ dsInternalCalc — при вычислении значений полей;
- ❑ dsCalcField — при обработке события OnCalcFields;
- ❑ dsBlockRead — при перемещении курсора;
- ❑ dsFilter — при обработке события OnFilterRecord;
- ❑ dsOpening — при открытии набора данных.

События класса TTable

В классе TTable определен ряд событий, возникающих при изменении состояния набора данных. Все эти события можно условно разделить на три группы:

- ❑ события, возникающие *перед* изменением *состояния* набора данных;
- ❑ события, возникающие *после* изменения *состояния* набора данных;
- ❑ события, возникающие *в момент* изменения набора данных.

Набор событий для первой и второй групп практически идентичен. Различие заключается только в моменте возникновения события и имени обработчика события. Данные события имеют следующие имена и вызываются при следующих действиях:

- ❑ BeforeCancel, AfterCancel — отмена изменений, внесенных в набор данных;
- ❑ BeforeClose, AfterClose — закрытие набора данных;
- ❑ BeforeDelete, AfterDelete — удаление записи;
- ❑ BeforeEdit, AfterEdit — переход в режим редактирования dsEdit;
- ❑ BeforeInsert, AfterInsert — добавление новой записи;
- ❑ BeforeOpen, AfterOpen — открытие набора данных;
- ❑ BeforePost, AfterPost — сохранение изменений в базе данных;
- ❑ BeforeScroll, AfterScroll — изменение положения курсора.

События, возникающие перед тем, как произойдут какие-либо изменения в базе данных, могут использоваться, например, для запроса у пользователя подтверждения на внесение этих изменений.

События, возникающие после внесения изменений в базу данных, обычно применяются для вывода сообщений о внесенных изменениях.

Из третьей группы рассмотрим только два события, которые используются наиболее часто:

- ❑ OnNewRecord — возникает при добавлении новой записи к набору данных (может использоваться для установки начальных значений полей записи или для добавления новых записей в другие таблицы, связанные с данной);
- ❑ OnCalcFields — возникает при пересчете значений вычисляемых полей (это происходит при открытии набора данных или при переходе набора данных в состояние dsEdit).

Работа с полями

Поле набора данных представляет собой описание типа данных, которому соответствует значение, находящееся в таблице базы данных. В наборе данных Delphi для представления полей используется специальный класс, который должен обеспечивать возможность работы с любыми типами данных.

Тип данных обычно однозначно связан с полем таблицы базы данных. Для работы с полями различных типов в Delphi определен ряд классов для представления типизированных полей. Причем все эти классы являются потомками базового класса TField, инкапсулирующего основные свойства и методы поля, не зависящие от типа данных, хранящихся в нем.

Класс TField

Объектный тип TField обеспечивает взаимодействие набора данных с компонентами отображения данных и правильную визуализацию хранящейся в базе данных информации. Кроме того, класс TField реализует возможности контроля вводимых значений при редактировании базы данных, а также позволяет преобразовывать хранящуюся в поле информацию к различным типам данных.

Основные свойства класса TField представлены в табл. 9.22.

Таблица 9.22. Основные свойства класса TField

Свойства	Тип	Описание
Alignment	TAlignment	Способ выравнивания при визуализации информации в компоненте отображения данных
AsBoolean	Boolean	Значение поля в формате логического типа
AsCurrency	Currency	Значение поля в формате типа Currency
AsDateTime	TDateTime	Значение поля в календарном формате
AsFloat	Double	Значение поля в формате вещественных чисел
AsInteger	Integer	Значение поля в целочисленном формате
AsString	String	Значение поля в строковом формате
AsVariant	Variant	Значение поля в формате вариантного типа
Calculated	Boolean	Определяет тип поля: вычисляемое (true) или нет (false)
CanModify	Boolean	Определяет, можно (true) или нет (false) редактировать данное поле
ConstraintErrorMessage	String	Сообщение, выдаваемое при вводе в поле недопустимого значения
CurValue	Variant	Текущее значение поля с учетом изменений, внесенных в набор данных другими пользователями (в многопользовательском режиме работы)
CustomConstraint	String	Строка на языке SQL, описывающая ограничения на значение поля
DataSet	TDataSet	Набор данных, которому принадлежит поле
DataSetSize	Word	Объем памяти, занимаемый текущим значением поля
DataType	TFieldType	Тип данных, хранящихся в поле

Свойства	Тип	Описание
DefaultExpression	String	Строка на языке SQL, описывающая значение поля по умолчанию
DisplayLabel	String	Заголовок поля
DisplayText	String	Значение, показываемое в компоненте отображения данных
FieldKind	TFieldKind	Определяет тип поля: поле данных, то есть физическое поле, содержащееся в таблице базы данных (fkData); вычисляемое поле (fkCalculated); поле перекрестного просмотра (fkLookup); внутреннее вычисляемое поле, то есть вычисляемое поле, содержащееся в таблице базы данных (fkInternalCalc); агрегатное поле (fkAggregate)
IsIndexField	Boolean	Определяет, является поле индексированным (true) или нет (false)
IsNull	Boolean	Определяет, содержит поле какое-либо значение (false) или нет (true)
KeyFields	String	Задаёт поля набора данных, для которых организован синхронный просмотр
Lookup	Boolean	Определяет, является поле полем синхронного просмотра (true) или нет (false)
NewValue	Variant	Текущее значение поля. Применяется в многопользовательском режиме
OldValue	Variant	Исходное значение поля
ReadOnly	Boolean	Задаёт режим «только для чтения»
ValidChars	TFieldChars	Набор символов, допустимых для задания значения поля
Value	Variant	Значение, содержащееся в поле
Visible	Boolean	Видимость поля

Класс TField содержит также ряд методов, которые иногда используются при работе с полями. Рассмотрим основные из них:

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

Присваивает полю значение свойства Value поля Source. При вызове данного метода следует следить за соответствием типов.

```
procedure AssignValue (const Value: TVarRec);
```

Присваивает полю значение Value.

```
procedure Clear;
```

Устанавливает значение поля в NULL.

```
function IsBlob: Boolean;
```

Определяет, содержит ли поле данные в формате BLOB.

```
function IsValidChar (InputChar: Char): Boolean;
```

Определяет, можно ли использовать в значении поля символы InputChar.

```
procedure SetFieldType (Value: TFieldType);
```

Задаёт тип данных поля.

Вычисляемые поля

Значения *вычисляемых полей* рассчитываются на основе существующих данных без изменения структуры таблицы данных. Расчет значений производится в методе-обработчике события OnCalcFields набора данных. Это событие возникает в следующих случаях:

- ☐ при открытии набора данных;
- ☐ при переходе набора данных в режим редактирования;
- ☐ при перемещении фокуса ввода между компонентами отображения данных и при удалении записей из набора данных (свойство набора данных AutoCalcFields при этом должно быть установлено в true).

Вычисляемые поля создаются с помощью редактора полей (рис. 9.6), который вызывается командой **Fields Editor** контекстного меню компонента TTable либо двойным щелчком на этом компоненте.

Для добавления вычисляемого поля используется команда **New Field** контекстного меню редактора полей. При выполнении данной команды открывается окно диалога, в котором задаются параметры создаваемого поля (рис. 9.7). Для создания вычисляемого поля установите в группе **Field type** переключатель **Calculated**.



Рис. 9.6. Окно редактора полей

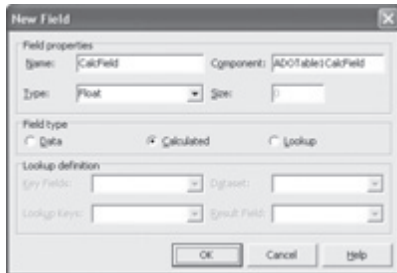


Рис. 9.7. Окно диалога New Field предназначено для создания нового поля

Типизированные поля

На основе класса TField определен ряд классов для представления типизированных полей. Кратко рассмотрим основные классы типизированных полей.

- ☐ TStringField — строковое поле. Длина строки не более 8192 символов.
- ☐ TSmallIntField — целочисленное поле, хранящее данные в формате ShortInt.
- ☐ TIntegerField — целочисленное поле, хранящее данные в формате Integer.
- ☐ TLargeField — целочисленное поле, хранящее данные в формате LongInt.
- ☐ TWordField — целочисленное поле, хранящее данные в формате Word.
- ☐ TBooleanField — логическое поле, тип данных Boolean.
- ☐ TFloatField — поле действительных чисел, тип данных Double.

- ❑ **TBLOBField** — поле, хранящее данные в виде большого двоичного объекта (BLOB). Может содержать любые данные, представимые в виде двоичного объекта. В базах данных двоичные объекты хранятся в отдельных файлах, а поля содержат только ссылки на них.
- ❑ **TGraphicField** — поле, хранящее изображения. Фактически аналогично типу BLOB: изображение хранится в отдельном файле, а поле содержит ссылку на него. Изображение должно иметь формат BMP.
- ❑ **TArrayField** — массив полей любого типа, кроме TArrayField.
- ❑ **TDataSetField** — поле, содержащее набор данных.
- ❑ **TMemoField** — поле, содержащее набор строк.

Подключение базы данных к BDE

Delphi позволяет работать с таблицами базы данных, не зарегистрированными в BDE. Для этого достаточно указывать полные пути поиска в свойствах соответствующих компонентов. Однако такой подход в корне неправилен. Он не позволяет работать с базой данных как с целостным объектом, напрямую обращаясь к ней по имени, точнее по псевдониму.

Псевдоним указывает на каталог, в котором размещена база данных, и представляет собой некоторое сокращенное наименование полного пути, например WORK, SALES и т. д. Удобство такого подхода очевидно — при переносе программы на другой компьютер достаточно переопределить псевдоним, не привязываясь к конкретному каталогу.

Подключение базы данных происходит через утилиту BDE Administrator, входящую в поставку Delphi. В левой части окна BDE Administrator находится список всех зарегистрированных в системе BDE баз данных, а в правой представлены свойства текущей базы данных, выбранной в этом списке.

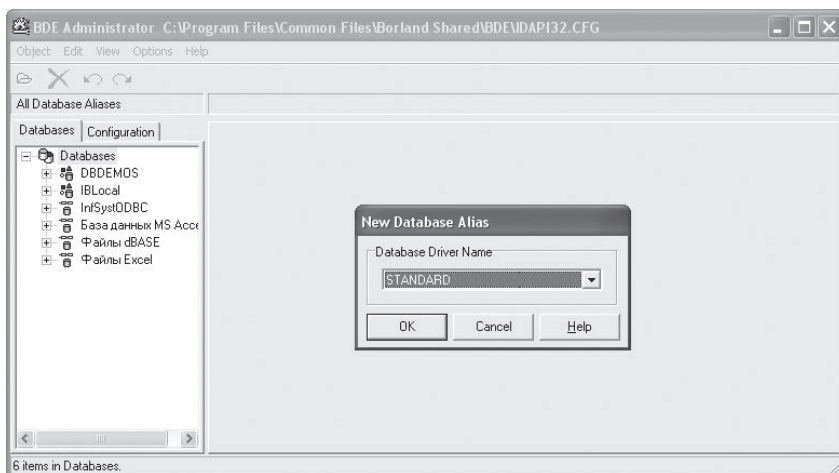


Рис. 9.8. Регистрация базы данных в BDE

Для добавления новой базы данных необходимо выполнить команду Object-New и в списке появившегося окна выбрать пункт STANDARD (рис. 9.8). После щелчка на кнопке OK в списке появится новый элемент, помеченный треугольником. Это означает, что регистрация базы данных не завершена. По умолчанию формируется имя базы данных Standard1; имеет смысл изменить его на значащее имя, например на Sales (продажи). Нужно убедиться, что свойства базы данных заданы правильно, то есть верно указан драйвер. Для свойства PATH указывается рабочий каталог базы данных или ее псевдоним. После задания свойств командой контекстного меню Apply необходимо завершить регистрацию базы данных.

Далее база данных в программах будет доступна под тем именем, которое вы зададите. Это имя выбирается в списке возможных значений свойства DatabaseName подключаемых таблиц данных.

Компоненты Delphi для отображения и редактирования данных

Для полноценной работы с базами данных недостаточно только обеспечить доступ к информации, хранящейся в базе. Необходимы также средства визуализации и редактирования этой информации. В Delphi для этого имеется целый ряд компонентов. Все они размещены на вкладке Data Controls палитры компонентов.

Для взаимодействия между набором данных и элементами отображения и редактирования данных используется специальный компонент TDataSource, расположенный на вкладке Data Access палитры компонентов.

Класс TDataSource

Класс TDataSource используется в качестве интерфейса для соединения набора данных с компонентами отображения данных. Он обеспечивает передачу в компоненты отображения значений полей из набора данных и внесение в набор данных изменений при редактировании этих значений.

ПРИМЕЧАНИЕ

Компонент TDataSource не задействует механизм BDE. Поэтому он способен взаимодействовать с компонентами доступа к данным, использующими как BDE, так и ADO.

Компонент TDataSource передает в элементы отображения данных значения полей текущей записи. При перемещении курсора набора данных на другую запись значения полей в компонентах отображения данных автоматически обновляются.

Класс TDataSource содержит небольшое количество свойств и методов. Наиболее часто используемые свойства класса TDataSource приведены в табл. 9.23.

Таблица 9.23. Основные свойства класса TDataSource

Свойство	Тип	Описание
AutoEdit	Boolean	Если значение данного свойства равно true, то при попытке пользователя изменить значение поля в элементе отображения данных набор данных автоматически переводится в состояние dsEdit
DataSet	TDataSet	Указывает на набор данных, с которым связан объект TDataSource
Enabled	Boolean	Определяет, отображать или нет данные в элементах отображения данных, связанных с данным объектом TdataSource
State	TDataSetState	Содержит текущее состояние набора данных, связанного с компонентом TDataSource

procedure Edit;

Проверяет состояние набора данных перед переводом его в состояние dsEdit.
function IsLinkedTo (DataSet: TDataSet): Boolean;

- Проверяет, связан ли компонент TDataSource с набором данных DataSet.
- В классе TDataSource определена обработка только трех событий:
- ❑ OnDataChange — вызывается при перемещении курсора набора данных, связанного с компонентом TDataSource, если в текущую запись были внесены изменения;
 - ❑ OnStateChange — вызывается при изменении состояния набора данных, связанного с компонентом TDataSource;
 - ❑ OnUpdateData — вызывается перед сохранением изменений в базе данных.

Модули данных

При работе с базами данных часто бывает удобно использовать модули данных. Чтобы создать новый модуль данных, сначала с помощью команды главного меню File ▶ New ▶ Other вызовите диалоговое окно New Items (рис. 9.9).

В открывшемся окне в списке Item Categories выберите элемент Delphi Files, а затем в списке, расположенном справа, выберите элемент Data Module.

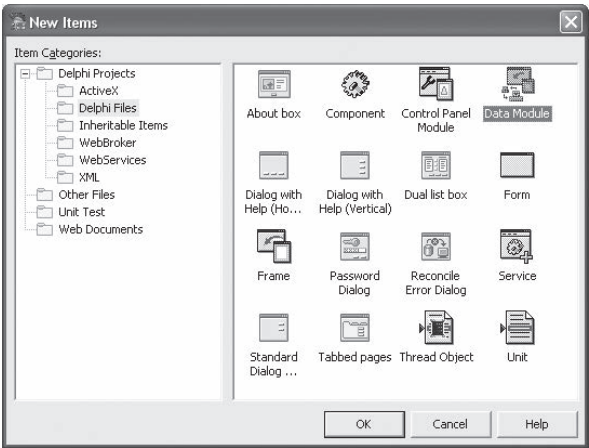


Рис. 9.9. Создание модуля данных

Модули данных представляют собой контейнеры, в которые можно помещать невидимые компоненты для доступа к данным (рис. 9.10): TADOTable, TADOQuery, TTable, TQuery, TDataSource и т. п., а также визуально устанавливать связи между ними.

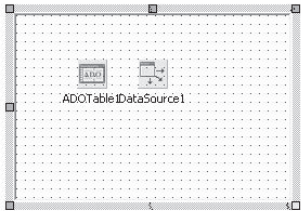


Рис. 9.10. Окно редактора модуля данных

Для использования модуля данных в приложении необходимо объявить его в разделе uses модуля приложения.

Класс TDBGrid

Компонент TDBGrid используется для представления набора данных в виде таблицы. Структура этой таблицы соответствует структуре набора данных: строки являются записями, а столбцы — полями. Основные свойства класса TDBGrid приведены в табл. 9.24.

Таблица 9.24. Основные свойства класса TDBGrid

Свойство	Тип	Описание
Columns	TDBGridColumns	Коллекция объектов, описывающих столбцы таблицы
DataSource	TDataSource	Источник данных, с которым связана таблица
DefaultDrawing	Boolean	Определяет способ отображения данных в таблице: автоматический (true) или нет (false). Если задано значение false, то для управления процессом отображения данных используются обработчики событий OnDrawColumnCell и OnDrawDataCell
FieldCount	Integer	Количество столбцов, отображаемых в таблице
Fields[Index:integer]	TField	Обеспечивает доступ к полям набора данных, отображаемых в таблице, по порядковому номеру столбца (Index)
Options	TDBGridOptions	Задаёт параметры отображения данных в таблице
ReadOnly	Boolean	Включает (true) или выключает (false) режим «только для чтения»
SelectedField	TField	Поле набора данных, которое является текущим полем таблицы, отображающей данные
SelecteedIndex	Integer	Номер текущего столбца

В таблице необязательно отображать все поля, содержащиеся в наборе данных. Нужные поля выбираются с помощью редактора столбцов (рис. 9.11), вызываемого двойным щелчком на компоненте TDBGrid.

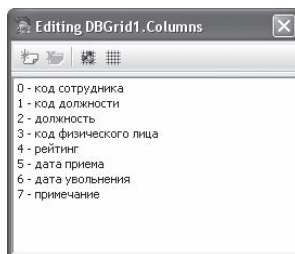


Рис. 9.11. Окно редактора столбцов

Новый столбец добавляется щелчком на кнопке **Add New** на панели инструментов редактора столбцов. Столбцы могут следовать в любом порядке, независимо от расположения соответствующих им полей в наборе данных.

После выделения столбца в инспекторе объектов отображаются его свойства, доступные для редактирования. Для представления столбца используется класс `TColumn`. Основные свойства этого класса приведены в табл. 9.25.

Таблица 9.25. Основные свойства класса `TColumn`

Свойство	Тип	Описание
Alignment	<code>TAlignment</code>	Способ выравнивания данных в столбце
Color	<code>TColor</code>	Цвет фона столбца
DropDownRows	<code>Cardinal</code>	Количество строк в раскрывающемся списке столбца
FieldName	<code>String</code>	Имя поля набора данных, связанного со столбцом
Font	<code>TFont</code>	Шрифт, используемый для отображения данных в столбце
PickList	<code>TStrings</code>	Раскрывающийся список столбца, используемый при редактировании данных
PopupMenu	<code>TPopupMenu</code>	Контекстное меню, связанное со столбцом
Showing	<code>Boolean</code>	Определяет, отображается (<code>true</code>) столбец или нет (<code>false</code>)
Title	<code>TColumnTitle</code>	Заголовок столбца и его параметры
Visible	<code>Boolean</code>	Видимость столбца
Width	<code>Integer</code>	Ширина столбца в пикселах

Для каждого столбца можно задать список, раскрывающийся при щелчке на расположенной в ячейке таблицы кнопке, относящейся к данному столбцу. Выбранное в списке значение заносится в ячейку. Для задания списка используется свойство столбца `PickList`. Количество отображаемых строк раскрывающегося списка задается свойством `DropDownRows`.

Компоненты для доступа к отдельным полям

Для доступа к отдельным полям базы данных в VCL Delphi имеется ряд элементов, аналогичных обычным элементам управления, рассмотренным нами ранее. Отличие заключается только в том, что элементы, работающие с базой данных, получают значения напрямую из набора данных и изменения этих значений заносятся в базу данных.

Все элементы отображения данных, отвечающие за доступ к отдельным полям набора данных, имеют два общих свойства:

- ❑ `DataSource: TDataSource` — указывает на источник данных, с которым связан компонент отображения данных;
- ❑ `DataField: String` — имя поля набора данных, из которого элемент отображения данных получает информацию.

TDBText

Компонент `TDBText` отображает текущее значение поля набора данных. Редактировать значение поля с помощью этого элемента нельзя. Компонент `TDBText` аналогичен компоненту `TLabel`.

TDBEdit

Компонент `TDBEdit` представляет собой обычное поле ввода, аналогичное `TEdit`. В отличие от `TDBText`, с помощью данного компонента можно не только просматривать базу данных, но и редактировать ее.

TDBMemo

Компонент `TDBMemo` предназначен для отображения и редактирования многострочных текстовых полей (обычно это поля типа `TMemoField` или `TBLOBField`). Компонент `TDBMemo` аналогичен компоненту `TMemo`.

TDBCheckBox

Данный компонент (аналог `TCheckBox`) предназначен для просмотра и редактирования данных, которые могут принимать только два значения. Состояние флажка определяется свойствами `ValueChecked` и `ValueUnChecked`, а также значением, содержащимся в поле, с которым связан компонент. По умолчанию `ValueChecked = true` и `ValueUnChecked = false`. Однако этим свойствам можно присваивать и строковые значения, причем одному свойству можно назначить несколько возможных значений, разделенных точкой с запятой.

Если значение поля соответствует одному из значений свойства `ValueChecked`, то флажок будет установлен; если одному из значений свойства `ValueUnChecked` — сброшен.

При изменении состояния флажка пользователем значение поля становится равным первому значению в списке `ValueChecked` (при установке флажка) или первому значению в списке `ValueUnChecked` (при сбросе флажка).

TDBRadioGroup

Компонент `TDBRadioGroup` представляет собой группу переключателей, состояние которых зависит от значения связанного с ним поля. Если текущее значение поля соответствует значению какого-либо переключателя, то он устанавливается. При изменении состояния переключателей пользователем в поле заносится значение установленного переключателя.

Свойство `Items` содержит список переключателей. Значения, на которые реагируют переключатели, определяется свойством `Values`. Каждому значению,

заданному в списке Values, соответствует один переключатель. Текущее значение поля содержится в свойстве Value.

TDBListBox

Компонент TDBListBox служит для отображения текущего значения поля данных и замены его любым выбранным в списке значением. При этом значение поля данных должно совпадать с одним из пунктов списка. В остальном компонент TDBListBox ничем не отличается от компонента TListBox.

TDBComboBox

Компонент TDBComboBox отображает значение поля данных, связанного с данным компонентом, в поле списка. Текущее значение поля данных можно изменять, выбирая новое значение в раскрывающемся списке либо редактируя текст в поле списка. Компонент TDBComboBox аналогичен компоненту TComboBox.

TDBImage

Компонент TDBImage используется для отображения графической информации, хранящейся в базе данных. Этот компонент похож на TImage, но содержит некоторые дополнительные свойства и методы.

AutoDisplay: Boolean;

Если для данного свойства установлено значение true, то изображение из связанного поля данных отображается автоматически, если значение false, то для загрузки изображения необходимо вызывать метод LoadPicture.

procedure LoadPicture;

Загружает изображение из связанного поля данных.

procedure CopyToClipboard;

Копирует изображение в буфер обмена.

procedure CutToClipboard;

Копирует изображение из буфера обмена и обнуляет текущее значение поля.

procedure PasteFromClipboard;

Загружает изображение из буфера обмена.

Навигация по набору данных

Компоненты, работающие с отдельными полями, не имеют встроенных средств для изменения положения курсора набора данных, добавления новых записей и удаления существующих записей. Поэтому при их использовании требуются дополнительные элементы управления, обеспечивающие навигацию по набору данных.

В Delphi имеется компонент TDBNavigator, позволяющий легко решить задачу такой навигации. Этот компонент представляет собой набор кнопок, выполняющих следующие функции:

- ☐ перемещение курсора набора данных на следующую запись;
- ☐ перемещение курсора на предыдущую запись;

- ❑ перемещение курсора на первую запись;
- ❑ перемещение курсора на последнюю запись;
- ❑ вставка новой пустой записи в текущую позицию курсора набора данных;
- ❑ удаление текущей записи;
- ❑ перевод набора данных в режим редактирования;
- ❑ запись изменений в набор данных;
- ❑ отмена изменений, внесенных в текущую запись;
- ❑ восстановление исходного значения записи.

Набор кнопок, содержащихся в компоненте `TDBNavigator`, определяется пользователем с помощью свойства `VisibleButtons`, имеющего тип `TButtonSet`.

Связь компонента `TDBNavigator` с набором данных устанавливается через свойство `DataSource`, указывающее на источник данных для требуемого набора.

Свойство `ConfirmDelete`, имеющее тип `Boolean`, позволяет включить запрос подтверждения при удалении записи.

Щелчки на кнопках компонента `TDBNavigator` можно производить программно при помощи следующего метода:

```
procedure BtnClick(index: TNavigateBtn);
```

Параметр `index` определяет кнопку, на которой был выполнен щелчок.

Компонент `TDBNavigator` обрабатывает два события:

- ❑ `BeforeAction` – вызывается после щелчка на кнопке компонента `TDBNavigator`, но *до* выполнения операции, связанной с этой кнопкой (обычно применяется для запроса у пользователя подтверждения на выполнение операции, приводящей к изменению данных);
- ❑ `OnClick` – вызывается после щелчка на кнопке и *после* выполнения операции, связанной с этой кнопкой.

Глава 10

Создание форм для ввода и редактирования данных

Любое приложение, разработанное в среде Delphi, должно содержать по крайней мере одну форму. Конструктор форм Delphi предоставляет разработчику все необходимые средства и инструменты для создания форм любой сложности. Кроме того, формы для работы с базами данных можно создавать с помощью специального мастера форм. Использование мастера облегчает создание форм для отображения и редактирования данных как для одной таблицы, так и для связанных таблиц.

Формы в Delphi

Форма представляет окно приложения на этапе разработки и обеспечивает создание интерфейса пользователя, являясь контейнером для размещения элементов интерфейса.

Различают два типа форм — *модальные* и *немодальные*. *Модальные* формы не позволяют передавать фокус ввода в другие окна приложения до тех пор, пока модальное окно не закрыто. Типичный пример модальных окон — окна диалога.

Немодальные формы могут передавать управление другим окнам приложения, оставаясь открытыми. Примером немодальных окон могут служить окна инспектора объектов, редактора кода, редактора форм среды Delphi.

Класс TForm позволяет создавать два типа оконного интерфейса — *однодокументный* (Single Document Interface, SDI) и *многодокументный* (Multi Document Interface, MDI). В обоих случаях программа содержит одно главное окно, создающееся и отображающееся после запуска программы. Различие между ними заключается в способе взаимодействия главного окна с дочерними окнами. В SDI-программах все дочерние окна могут перекрывать главное окно. Примером SDI-приложения является сама среда Delphi. В MDI-приложениях дочерние окна не могут перекрывать родительскую форму и отображаются только в ее клиентской области.

Свойства класса TForm

Все свойства класса TForm можно разделить на две группы — *опубликованные свойства*, то есть те свойства, которые отображаются в окне инспектора объектов во время разработки приложения, и свойства, которые можно изменять только в процессе выполнения программы.

В табл. 10.1 приведены основные опубликованные свойства формы.

Таблица 10.1. Основные опубликованные свойства класса TForm

Свойство	Тип	Описание
ActiveControl	TWinControl	Указывает на элемент управления, имеющий фокус ввода
AutoScroll	Boolean	Определяет необходимость создания автоматических полос прокрутки, если размер формы не позволяет отобразить все расположенные на ней элементы управления
AutoSize	Boolean	Если значение данного свойства равно true, то размеры формы автоматически изменяются таким образом, чтобы вместить все расположенные на ней элементы управления
BorderIcons	TBorderIcon = set of (biSystemMenu, biMinimize, biMaximize, biHelp)	Определяет набор кнопок в заголовке окна
BorderStyle	TBorderStyle = (bsNone, bsSingle, bsSizeable, bsDialog, bsToolWindow, bsSizeToolWindow)	Определяет тип границы окна и задает вид заголовка формы, отображение строки меню, наличие кнопок в заголовке окна
BorderWidth	TBorderWidth	Ширина рамки вокруг окна
Caption	TCaption	Заголовок окна
ClientHeight	Integer	Высота клиентской области в пикселах
ClientWidth	Integer	Ширина клиентской области в пикселах
Color	TColor	Цвет фона окна
Constraints	TSizeConstraints	Ограничения на изменения размера окна
Ct13D	Boolean	Определяет внешний вид окна — «трехмерное» или «плоское»
Cursor	TCursor	Вид указателя мыши над формой
FormStyle	TFormStyle = (fsNormal, fsMDIChild, fsMDIForm, fsStayOnTop)	Стиль окна: <ul style="list-style-type: none"> fsNormal — обычное окно SDI; fsStayOnTop — окно SDI, всегда остающееся поверх всех других окон; fsMDIForm — главная форма MDI; fsMDIChild — дочерняя форма MDI
Height	Integer	Высота окна в пикселах
Icon	TIcon	Значок формы, отображающийся в верхнем левом углу окна
Left	Integer	Левая граница формы в координатах экрана

Свойство	Тип	Описание
Menu	TMainMenu	Указатель на главное меню окна
Name	TComponentName	Идентификатор экземпляра TForm
PopupMenu	TPopupMenu	Указатель на контекстное меню окна
Top	Integer	Верхняя граница формы в координатах экрана
Visible	Boolean	Определяет видимость формы
Width	Integer	Ширина окна в пикселах
WindowState	TWindowState = (wsNormal, wsMinimized, wsMaximized)	Состояние окна: свернуто (wsMinimized), развернуто на весь экран (wsMaximized), обычное состояние (wsNormal)
Tag	LongInt	Не имеет никакого predefined значения

Тип границы окна и вид заголовка формы определяется свойством `BorderStyle`, которое может принимать одно из следующих значений:

- ☐ `bsSizeable` — обычное окно с изменяемыми размерами. Наличие кнопок в заголовке определяется свойством `BorderIcons`;
- ☐ `bsSingle` — окно, размеры которого не изменяются во время выполнения программы. Кнопки в заголовке определяются `BorderIcons`;
- ☐ `bsDialog` — форма для создания окон диалога. Не изменяет размеры во время выполнения программы. Независимо от значения свойства `BorderIcons` в заголовке выводится только кнопка для закрытия окна;
- ☐ `bsNone` — окно без границ и без заголовка;
- ☐ `bsToolWindow` — окно в стиле панели инструментов с фиксированным размером;
- ☐ `bsSizeToolWindow` — окно панели инструментов с изменяющимися размерами.

Все свойства, приведенные в табл. 10.1, доступны для изменений как в инспекторе объектов во время разработки приложения, так и в процессе выполнения программы.

Класс `TForm` содержит также ряд свойств, доступ к которым может осуществляться только во время выполнения программы (табл. 10.2). Часть этих свойств доступна для модификации, часть — нет (только для чтения).

Таблица 10.2. Основные свойства класса `TForm`, доступные во время выполнения программы

Свойство	Тип	Описание
Active	Boolean	Определяет, активна форма или нет (только для чтения)
ActiveMDIChild	Tform	Указывает на активную дочернюю форму
Canvas	TCanvas	Используется для «рисования» на форме (только для чтения)
ModalResult	TmodalResult	Величина, возвращаемая функцией <code>ShowModal</code> при закрытии модального окна
MDIChildCount	Integer	Количество дочерних окон (только для чтения)
MDIChildren[i : integer]	Tform	Указывает на <i>i</i> -е дочернее окно (только для чтения)

Кроме свойств, класс TForm включает ряд методов, которые могут быть полезны при разработке приложения. Основные методы TForm представлены в табл. 10.3.

Таблица 10.3. Основные методы класса TForm

Метод	Описание
procedure Close	Вызывает метод CloseQuery и, если он возвращает true, закрывает форму
function CloseQuery: Boolean	Используется для определения, может ли форма быть закрыта
procedure FocusControl (Control:TWinControl)	Устанавливает фокус ввода на элемент Control
function GetFormImage : TBitmap	Возвращает растровое изображение формы
procedure Hide	Скрывает форму, не уничтожая ее
procedure Print	Печатает изображение формы
procedure Release	Закрывает форму и освобождает занимаемые ею ресурсы
procedure Show	Отображает форму в немодальном режиме
function ShowModal : integer	Отображает форму в модальном режиме

В классе TForm определен ряд методов-обработчиков событий, которые позволяют задавать реакцию экземпляра класса TForm на определенные действия. Всего определено 39 событий, на которые может реагировать форма. Здесь мы рассмотрим только основные из них:

- ❑ OnActivate — вызывается при передаче форме фокуса ввода;
- ❑ OnClick — вызывается при одиночном щелчке на форме;
- ❑ OnDbClick — вызывается при двойном щелчке на форме;
- ❑ OnClose — вызывается при закрытии формы;
- ❑ OnCloseQuery — вызывается перед закрытием формы. Используется для задания параметра, возвращаемого методом CloseQuery;
- ❑ OnCreate — вызывается при создании формы;
- ❑ OnDeactivate — вызывается при потере формой фокуса ввода;
- ❑ OnDestroy — вызывается перед уничтожением формы;
- ❑ OnPaint — вызывается при перерисовке формы;
- ❑ OnShow — вызывается при отображении формы;
- ❑ OnKeyPress — вызывается при нажатии на клавишу;
- ❑ OnMouseDown — вызывается при нажатии левой кнопки мыши в области формы;
- ❑ OnMouseUp — вызывается при отпускании левой кнопки мыши в области формы;
- ❑ OnMouseMove — вызывается при движении указателя мыши над формой.

С помощью методов-обработчиков событий можно выполнить ряд действий, не прибегая к применению дополнительных элементов управления. Для иллю-

страции такой возможности рассмотрим пример программы, в которой при одиночном щелчке на форме к заголовку добавляется имя исполняемого файла, а при двойном щелчке на форме программа закрывается.

Для ее реализации выполните следующие действия.

1. Выберите команду **File ► New ► VCL Form Application — Delphi for Win32**.
2. Перейдите в окне **Object Inspector** вкладку **Events** и дважды щелкните на поле значения события **OnClick**. После этого станет активным окно редактора кода и в него будет автоматически добавлен заголовок процедуры-обработчика события **OnClick**:

```
procedure TForm1.FormClick(Sender: TObject);  
begin
```

```
end;
```

3. В теле процедуры **FormClick** напишите код, выполняемый при возникновении события **OnClick**. Для изменения заголовка формы будем использовать свойство **Caption** объекта **Form1** (это идентификатор экземпляра **TForm**, задаваемый по умолчанию). Чтобы определить имя исполняемого файла, воспользуемся свойством **ExeName** объекта **Application**:

```
procedure TForm1.FormClick(Sender: TObject);  
const first : boolean = true;  
// константа-переменная используется для того, чтобы заголовок  
// изменялся только один раз  
begin  
    if first then begin  
        Form1.Caption:=Form1.Caption+' - ' +Application.ExeName;  
        first:=false  
    end;  
end;
```

4. Выберите в окне **Object Inspector** вкладку **Events** и дважды щелкните на поле значения события **OnDbClick**. Окно редактора кода станет активным и в него будет автоматически добавлен заголовок процедуры-обработчика события **OnDbClick**:

```
procedure TForm1.FormDbClick(Sender: TObject);  
begin
```

```
end;
```

5. Напишите код завершения программы в теле процедуры **FormDbClick**. Для этого воспользуйтесь методом **Terminate** класса **TApplication**:

```
procedure TForm1.FormDbClick(Sender: TObject);  
begin  
    Application.Terminate;  
end;
```

6. Выполните компиляцию программы. После ее запуска на экране отобразится пустая форма с заголовком **Form1**. Щелчок мыши на этой форме приведет к изменению заголовка. При двойном щелчке выполнение программы будет завершено.

Фреймы

Фреймы представляют собой контейнеры, предназначенные для размещения на них элементов управления. В этом плане фреймы подобны формам, однако в отличие от них фреймы могут помещаться на формы и другие фреймы.

Перед размещением фрейма на форме его необходимо предварительно создать с помощью команды **File ► New ► Other**. В появившемся диалоговом окне **New Items** в списке **Items Categories** выберите элемент **Delphi Files**, а затем в списке, расположенном справа, выберите элемент **Frame** и щелкните на кнопке **OK**. Процедура размещения элементов управления на фрейме производится точно так же, как и на форме. Для помещения готового фрейма на форму используется компонент **Frames** палитры компонентов (страница **Standard**). При этом отображается список всех фреймов, существующих в текущем проекте. Выбранный из списка фрейм помещается на форму как обычный компонент.

Так как фреймы очень похожи на формы, то они обладают примерно теми же свойствами. Однако в отличие от форм фреймы содержат меньшее число методов и реагируют на меньшее количество событий. Так, например, у фреймов нет методов **Show** и **ShowModal**, так как они не могут отображаться вне форм.

Использование базовых классов для создания форм ввода

В Delphi при создании форм используется класс **TForm**, рассмотренный ранее в главе 9. Класс **TForm** фактически является контейнерным классом, позволяющим включать в себя любое количество элементов управления, а также другие контейнеры: **TGroup**, **TPanel** и т. п.

Размещение и удаление элементов управления

Для создания новой формы используется команда **File ► New ► Form** — **Delphi for Win32** главного меню. После создания формы на ней можно размещать элементы управления. Для этого щелкните на соответствующем значке компонента в палитре, а затем — в том месте формы, где предполагается разместить компонент. Положение курсора при этом задает место вставки левого верхнего угла элемента управления, а размер размещаемого компонента определяется установками, используемыми в Delphi по умолчанию. Однако можно одновременно с размещением элемента устанавливать и его размер. Для этого нажмите кнопку мыши на форме и, продолжая удерживать ее, перемещайте указатель по диагонали до получения прямоугольного контура нужного размера, в который будет «вписан» размещаемый компонент. После размещения компонента его размеры можно изменять либо с помощью инспектора объектов, либо с помощью мыши.

ПРИМЕЧАНИЕ

Для добавления на форму нескольких одинаковых компонентов при выборе компонента в палитре нажмите клавишу **Shift**. После этого при каждом нажатии левой кнопки мыши на форме будет добавляться экземпляр выбранного компонента.

Для удаления компонента с формы выделите его щелчком мыши и нажмите клавишу **Delete**. Можно также воспользоваться командой главного меню **Edit ▶ Delete**.

Выравнивание компонентов на форме

Для выравнивания элементов управления относительно формы, друг друга или заданной сетки в Delphi используется Редактор форм. Кроме того, имеется возможность установки одинаковых размеров для группы выделенных объектов. Доступ к командам выравнивания обеспечивается с помощью панели инструментов **Align**, открывающейся при выполнении команды **View ▶ Toolbars ▶ Align** главного меню, или через окно диалога **Alignment**, открывающееся при выполнении команды **Position ▶ Align** контекстного меню редактора форм.

Выделение группы элементов управления

Команды выравнивания, как правило, применяются к группе выделенных компонентов. Для выделения нескольких элементов можно воспользоваться двумя способами:

- ☐ удерживая клавишу **Shift**, последовательно щелкните на нужных компонентах;
- ☐ удерживая левую кнопку мыши нажатой, обведите область формы, на которой расположены выбираемые компоненты, контуром выделения (рис. 10.1).

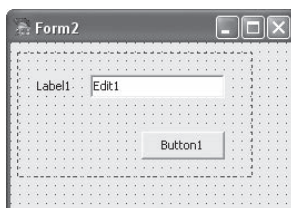


Рис. 10.1. Выделение нескольких элементов с помощью мыши

Команды выравнивания компонентов

Для выравнивания компонентов можно использовать либо окно диалога **Alignment**, открывающееся командой **Position ▶ Align** контекстного меню редактора форм (рис. 10.2), либо панель инструментов **Align** (рис. 10.3).

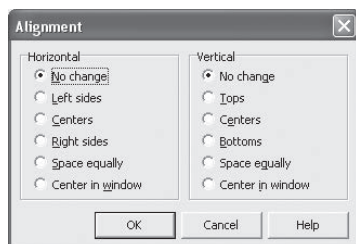


Рис. 10.2. Окно диалога Alignment

Окно диалога Alignment содержит две группы переключателей, управляющих выравниванием выделенных компонентов по горизонтали (Horizontal) и по вертикали (Vertical).



Рис. 10.3. Панель инструментов Align

Описание команд выравнивания приводится в табл. 10.4.

Таблица 10.4. Команды выравнивания компонентов в редакторе форм

Команда	Описание
No change	Не изменяет положение компонентов
Left sides	Устанавливает левую границу всех выделенных компонентов в соответствии с границей самого левого компонента
Right sides	Устанавливает правую границу всех выделенных компонентов в соответствии с границей самого правого компонента
Center	Центрирует компоненты относительно выделенной области
Tops	Устанавливает верхнюю границу всех выделенных компонентов в соответствии с границей самого верхнего компонента
Bottoms	Устанавливает нижнюю границу всех выделенных компонентов в соответствии с границей самого нижнего компонента
Space equally	Устанавливает равные интервалы между границами выделенных компонентов
Center in window	Центрирует выделенную область относительно клиентской области формы

ПРИМЕЧАНИЕ

Команда Space equally устанавливает равные интервалы для одноименных границ компонентов: между левыми (правыми) границами и между верхними (нижними) границами. Поэтому применение данной команды к элементам с разными размерами не приведет к установлению равных промежутков между ними.

Изменение размеров и перемещение компонентов

Каждый элемент управления обладает рядом свойств, определяющих его размеры и положение на форме. Все эти свойства измеряются в пикселах и доступны для редактирования в инспекторе объектов:

- ❑ Width — ширина элемента управления;
- ❑ Height — высота элемента управления;
- ❑ Top — верхняя граница элемента управления относительно верхней границы клиентской области формы;
- ❑ Left — левая граница компонента относительно левой границы клиентской области формы.

Обычно для изменения размеров и расположения элементов удобнее использовать мышь. Чтобы изменить размер компонента с помощью мыши, необходимо вначале его выделить. Для этого достаточно щелкнуть на нем, после

чего контур выделенного компонента показывается черными квадратными маркерами, расположенными по углам и по центру каждой стороны элемента. Для изменения размеров компонента перетащите мышью один из маркеров до установления нужного размера.

Выделенный компонент можно также перемещать с помощью клавиш со стрелками при одновременно нажатой клавишей Ctrl. Однократное нажатие любой из них приводит к перемещению компонента на один пиксел в соответствующем направлении.

Для одновременного изменения размеров группы выделенных компонентов используются команды окна диалога Size (рис. 10.4), открываемого с помощью команды Position ► Size из контекстного меню Редактора форм.

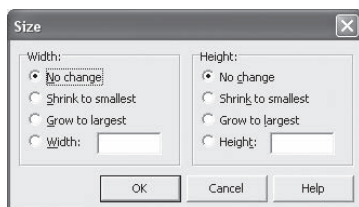


Рис. 10.4. Окно диалога Size

Данное окно диалога содержит две группы переключателей, используемых для одновременного изменения ширины (Width) и высоты (Height) группы выделенных элементов. Назначение входящих в него команд приведено в табл. 10.5.

Таблица 10.5. Команды окна диалога Size

Команда	Описание
No change	Размеры элемента не изменяются
Shrink to smallest	Ширина (или высота) всех выделенных элементов устанавливается равной минимальной ширине (или высоте)
Grow to largest	Ширина (или высота) всех выделенных элементов устанавливается равной максимальной ширине (или высоте)
Width	Для всех выделенных элементов ширина устанавливается равной величине, заданной в поле ввода
Height	Для всех выделенных элементов высота устанавливается равной величине, заданной в поле ввода

Порядок обхода элементов

Хотя основным инструментом при работе в среде Windows является мышь, иногда пользователю приходится использовать и клавиатуру. Для передачи фокуса ввода от одного элемента управления к другому с помощью клавиатуры используется клавиша Tab. По умолчанию порядок передачи фокуса ввода при нажатии на клавишу Tab определяется порядком помещения элементов на форму на стадии разработки приложения. Однако заранее довольно трудно определить предпочтительное направление обхода, чтобы в соответствии с ним размещать элементы. Поэтому в Delphi предусмотрена возможность изменения

порядка обхода после размещения элементов. Для этого используется свойство `TabOrder`, имеющееся у всех визуальных элементов управления. Значение, присвоенное этому свойству, определяет порядок передачи фокуса ввода. Компонент, для которого `TabOrder=0`, получит фокус ввода при открытии формы.

Свойство `TabStop` определяет, может элемент получить фокус ввода (`true`) или нет (`false`).

Настройка внешнего вида формы

Настройка внешнего вида формы производится установкой набора свойств, определяющих ее размеры, расположение на экране, заголовок, цвет фона и значок, отображаемый в левом верхнем углу формы. Все эти свойства можно настраивать с помощью инспектора объектов, работа с которым была рассмотрена ранее в главе 8.

Как уже отмечалось, размеры формы и ее местоположение на экране можно также задавать с помощью мыши.

Простые формы для ввода данных

При создании простых форм для ввода данных используются компоненты отображения и редактирования данных, работающие с отдельными полями базы данных (их функционирование было рассмотрено в предыдущей главе). Благодаря им на форме в каждый момент времени отображается информация только из одной записи. Поскольку в VCL Delphi имеются компоненты для визуализации полей различных типов (текстовых, числовых, многострочных, графических), то при использовании элементов редактирования, работающих с отдельными полями, легко организовать просмотр и редактирование информации практически любого типа.

Компоненты, связываемые с отдельными полями набора данных, можно размещать на форме произвольным образом. Это позволяет создавать очень наглядные и удобные в использовании формы для ввода данных.

При разработке простых форм, кроме элементов редактирования полей, на форму всегда следует помещать компонент `TDBNavigator`. Это обусловлено тем, что компоненты, работающие с отдельными полями, не имеют встроенных средств навигации по набору данных.

Пример создания простой формы

В качестве примера разработаем форму для просмотра и редактирования информации, содержащейся в таблице **Физические лица**. Данная таблица содержит следующие поля:

- ☐ Код физического лица — используется в качестве первичного ключа, тип `Integer`;
- ☐ Фамилия, Имя, Отчество, Телефон, Индекс, Страна, Город, Адрес — текстовые поля;
- ☐ дата рождения — поле типа `TDateField`;
- ☐ пол — поле типа `Boolean`.

Для отображения значений этих полей будем использовать компоненты TDBEdit. Кроме того, на форму поместим элемент TDBNavigator для обеспечения навигации по набору данных, а также несколько обычных элементов TLabel, с помощью которых будем пояснять назначение полей ввода.

Последовательность действий при создании простых форм будет примерно следующей.

1. Для создания нового приложения выполните команду **File ► New ► VCL Forms Application**. Так как в данном примере мы работаем только с одной таблицей, то использовать модуль не имеет смысла и компоненты доступа к данным можно поместить прямо на форму. Как было указано ранее, при работе с базами данных MS Access для доступа к данным удобнее всего использовать технологию ADO.
2. Используя раздел **dbGo** палитры инструментов, разместите на форме компонент TADOTable. Затем перейдите в палитре инструментов в раздел **Data Access** и установите на форму компонент TdataSource. Последний необходим для связи набора данных ADO с компонентами визуализации данных.
3. Теперь необходимо подключить к компоненту TADOTable таблицу **Физические лица** базы данных **employees.mdb**.
4. Выделите на форме компонент TADOTable и щелкните на кнопке с многоточием в поле ввода свойства **ConnectionString** в инспекторе объектов.
5. В открывшемся окне диалога **ConnectionString** (рис. 10.5) выберите переключатель **Use Connection String** и щелкните на кнопке **Build**.

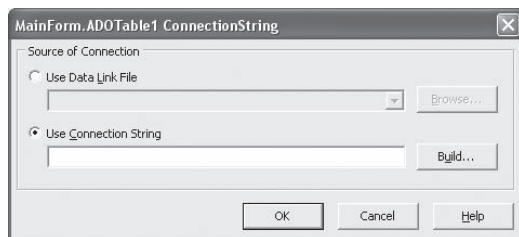


Рис. 10.5. Окно диалога **ConnectionString**

6. На вкладке **Поставщик данных (Provider)** открывшегося окна диалога **Свойства связи с данными (Data Link Properties)** задайте вид соединения с базой данных — **Microsoft Jet 4.0 OLE DB Provider** (рис. 10.6). Щелкните на кнопке **Далее**.
7. В открывшейся вкладке **Подключение (Connection)** окна диалога **Свойства связи с данными (Data Link Properties)** (рис. 10.7) щелкните на кнопке с троеточием, чтобы выбрать файл базы данных, к которому необходимо подключиться, а затем щелкните на кнопке **OK** (предварительно можно щелкнуть на кнопке **Проверить подключение**, чтобы убедиться, что база данных подключена корректно).

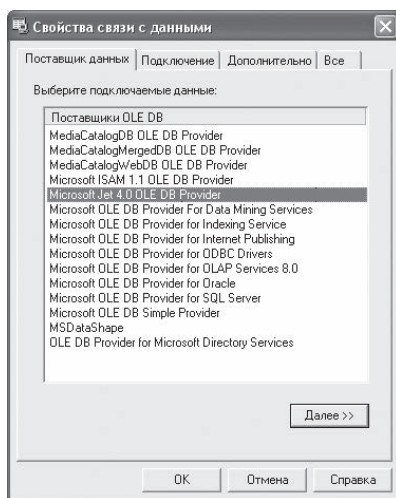


Рис. 10.6. Задание вида соединения с базой данных

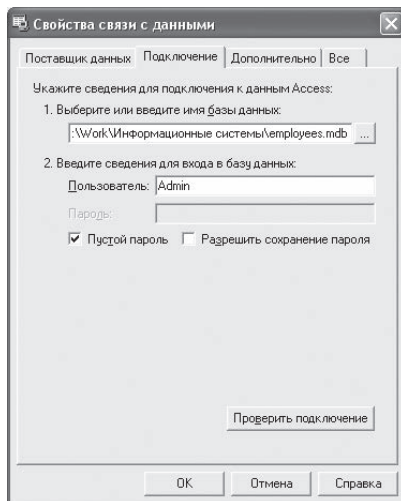


Рис. 10.7. Задание базы данных, с которой устанавливается связь

Теперь, после подключения базы данных, необходимо указать используемую таблицу.

8. Выделите на форме компонент `TADOTable` и затем в поле ввода свойства `TableName` в инспекторе объектов укажите имя используемой таблицы — физические лица.

Следующий этап — настройка источника данных `TDataSource`. Чтобы связать источник данных с набором данных, используйте свойство `DataSet`.

9. С помощью инспектора объектов укажите в свойстве `DataSet` имя объекта `TADOTable` (по умолчанию — `ADOTable1`).

10. Разместите на форме необходимые элементы управления и выполните их настройку. Примерный вариант размещения компонентов показан на рис. 10.8.

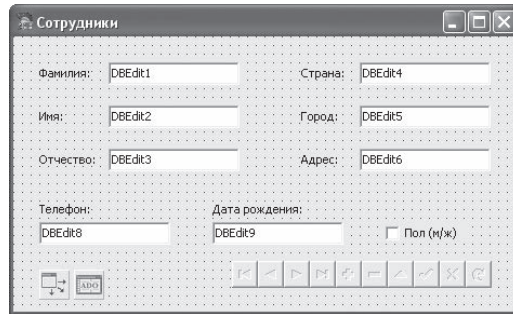


Рис. 10.8. Пример размещения элементов управления на простой форме

11. Для настройки элементов визуализации полей базы данных (три поля ввода TDBEdit) и элемента навигации по набору данных (TDBNavigator) отредактируйте в инспекторе объектов их свойство DataSource. Затем укажите имя источника данных (по умолчанию — DataSource1) и имя поля набора данных, с которым связывается элемент отображения и редактирования данных.

Осталось реализовать процедуры открытия и закрытия набора данных. Набор данных должен открываться при запуске приложения и закрываться при его завершении. Для открытия набора данных используется метод Open класса TADOTable, для закрытия — метод Close того же класса.

12. Вызовите метод Open в обработчике события OnShow главной формы, а метод Close — в обработчике OnClose.

Текст модуля разработанной формы приведен в листинге 10.1.

Листинг 10.1. Главный модуль приложения с простой формой для ввода данных

```
unit employees;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, DB, ADODB, ExtCtrls, DBCtrls, StdCtrls, Mask;

type
  TfrmEmployees = class(TForm)
    ADOTable1: TADOTable;
    DBEdit1: TDBEdit;
    DBEdit2: TDBEdit;
    DBEdit3: TDBEdit;
    DBEdit4: TDBEdit;
    DBEdit5: TDBEdit;
    DBEdit6: TDBEdit;
    DBEdit8: TDBEdit;
```

продолжение ➤

Листинг 10.1 (продолжение)

```

DBEdit9: TDBEdit;
DBCheckBox1: TDBCheckBox;
DBNavigator1: TDBNavigator;
DataSource1: TDataSource;
Label1: TLabel;
Label2: TLabel;
Label3: TLabel;
Label4: TLabel;
Label5: TLabel;
Label6: TLabel;
Label7: TLabel;
Label8: TLabel;
procedure FormClose(Sender: TObject; var Action: TCloseAction);
procedure FormShow(Sender: TObject);
private
{ Private declarations }
public
{ Public declarations }
end;

var
  frmEmployees: TfrmEmployees;

implementation

{$R *.dfm}

procedure TfrmEmployees.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  ADOTable1.Close
end;

procedure TfrmEmployees.FormShow(Sender: TObject);
begin
  ADOTable1.Open
end;

end.

```

13. Откомпилируйте и запустите программу. Внешний вид окна программы приведен на рис. 10.9.

Рис. 10.9. Вид простой формы в процессе выполнения программы

Табличные формы

Часто наиболее естественным способом представления информации при просмотре и редактировании является таблица. Для представления данных в табличной форме используется компонент `TDBGrid` (сетка), рассмотренный в предыдущей главе. С помощью данного компонента удобно отображать текстовые и числовые поля базы данных. Рассмотрим пример создания табличной формы. В качестве исходных данных будем использовать таблицу *Сотрудники*.

1. Создайте новое приложение с помощью команды **File ▶ New ▶ VCL Forms Application — Delphi for Win32**.
2. Поместите на форму компонент `TADOTable`. Затем для создания связи между компонентом доступа к данным и элементом `TDBGrid` поместите на форму элемент `TDataSource`. Настройка этих компонентов выполняется в соответствии с описанием, приведенным в предыдущей главе.
3. Поместите на форму компонент `TDBGrid`. С помощью инспектора объектов задайте значение свойства `Align` данного компонента равным `alClient`. При этом размеры компонента автоматически будут изменяться в соответствии с размерами клиентской области формы, на которой она размещена.
4. В свойстве `DataSource` задайте имя источника данных, через который набор данных подключается к `TDBGrid`.

Обратите внимание на то, что хотя компонент `TDBTable` подключен к набору данных, в нем не отображается никакой информации. Это обусловлено тем, что связь между набором данных и таблицей устанавливается только при открытии набора данных, для чего необходимо выполнить метод `Open` класса `TADOTable` или установить значение свойства `Active` данного объекта в значение `true`.

5. Измените заголовок формы. Для этого задайте в инспекторе объектов свойству формы `Caption` значение *Табличная форма*. Обратите внимание, что при изменении этого свойства сразу изменяется заголовок формы в редакторе форм.

Чтобы во время работы программы в таблице отображалась информация, хранящаяся в таблице физические лица, при запуске программы необходимо открыть набор данных, а при ее завершении — закрыть его. Для этого мы воспользуемся двумя методами-обработчиками событий `OnShow` и `OnClose`. Первый выполняется при отображении формы. В нем мы будем открывать набор данных. Второй выполняется при закрытии формы, в него мы занесем код закрытия набора данных.

6. Для создания обработчика события `OnShow` выберите в выпадающем списке в инспекторе объектов компонент `Form1` и перейдите на вкладку **Events**. Выполните двойной щелчок на поле ввода для события `OnShow`. При этом Delphi автоматически переключится в редактор кода и сгенерирует заголовок процедуры-обработчика выбранного события. В теле данной процеду-

ры напишите код открытия набора данных (см. листинг 10.2, процедура `FormShow`).

7. Аналогично задайте обработчик события `OnClose`, закрывающий набор данных (см. листинг 10.2, процедура `FormClose`).

Листинг 10.2. Модуль приложения табличной формы

```
unit gridform;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, Grids, DBGrids, DB, ADODB;

type
  TfrmGridForm = class(TForm)
    ADOTable1: TADOTable;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    procedure FormShow(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  frmGridForm: TfrmGridForm;

implementation

{$R *.dfm}

procedure TfrmGridForm.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  ADOTable1.Close;
end;

procedure TfrmGridForm.FormShow(Sender: TObject);
begin
  ADOTable1.Open;
end;

end.
```

Выполнив все перечисленные выше действия, вы получите форму, внешний вид которой показан на рис. 10.10.

8. Откомпилируйте и запустите программу. Внешний вид окна программы приведен на рис. 10.11.

После запуска программы, при открытии набора данных, в `TDBGrid` автоматически будут отображаться все поля таблицы `Сотрудники`. При этом таблица на форме будет содержать колонки, связанные с полями оригинальной таблицы

базы данных, имеющими тип данных, который в принципе невозможно отобразить в виде одной текстовой строки (например, поля MemoBLOB). В этом случае в колонке таблицы, соответствующей полю такого типа, отображается не информация, содержащаяся в поле, а его тип — как в последней колонке таблицы из нашего примера. Для отображения информации из таких полей необходимо использовать специальные компоненты: TDBMemo, TDBGraphic и т. п.



Рис. 10.10. Внешний вид табличной формы в редакторе форм

	код сотрудника	код должности	код физического лица	рейтинг	дата приема	дата увольнения	примечание
1	1	1	1	10	18.01.2000		(WideMemo)
2	2	2	2	6	28.02.2000	30.04.2002	(WideMemo)
3	5	2	2	8	01.05.2002		(WideMemo)
4	2	6	6	01.05.2002			(WideMemo)
5	3	3	3	0	14.02.2000	20.04.2005	(WideMemo)
6	3	4	3	30.04.2005			(WideMemo)
7	3	5	4	03.03.2000		01.06.2007	(WideMemo)
8	5	5	8	02.06.2007			(WideMemo)
9	3	7	1	10.06.2007			(WideMemo)
10	4	10	3	27.06.2005			(WideMemo)
11	6	8	4	22.08.2003			(WideMemo)
12	7	9	4	15.08.2003			(WideMemo)
13	8	11	3	11.10.2004			(WideMemo)
14	8	12	2	16.04.2005			(WideMemo)

Рис. 10.11. Табличная форма при выполнении программы

ПРИМЕЧАНИЕ

В принципе в ячейке таблицы TDBGrid можно отображать информацию любого типа, устанавливая свойство DefaultDrawing в значение false и программируя обработчик события OnDrawColumnCell. Однако, ввиду ограниченного объема книги, мы не имеем возможности подробно обсуждать данный вопрос.

Чтобы исключить отображение в таблице «лишних» полей, следует воспользоваться редактором столбцов, вызываемым командой Columns Editor контекстного меню компонента TDBGrid (см. главу 9). После открытия набора данных в таблице будут отображаться только те поля, которые указаны в редакторе

столбцов. Последовательность отображения столбцов также определяется в редакторе столбцов.

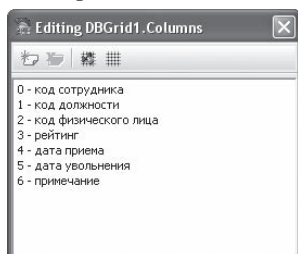


Рис. 10.12. Окно редактора столбцов

Воспользуемся данной возможностью для оптимизации созданной нами табличной формы. С этой целью исключим из таблицы поле Код сотрудника, которое не несет никакой информации и используется для связи с другими таблицами, а также поле Примечание, которое все равно не отображается в TDBGrid. Для реализации этого укажем в редакторе столбцов только используемые поля (рис. 10.12).

При добавлении полей в редакторе столбцов в компонент TDBGrid, помещенный на форму, автоматически добавляются соответствующие столбцы, видимые во время разработки формы (рис. 10.13). Ширина столбца (в пикселах) задается свойством Width класса TColumn.

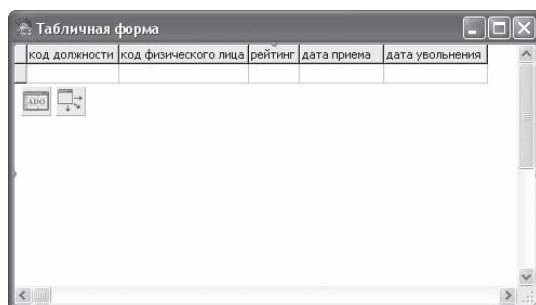


Рис. 10.13. Табличная форма с настроенными столбцами

После запуска программы в таблице будут отображаться только заданные поля (рис. 10.14).



Рис. 10.14. Табличная форма с настроенными столбцами при выполнении программы

Поскольку компонентом TDBGrid автоматически поддерживается навигация по набору данных, то при использовании табличных форм нет необходимости применять компонент TDBNavigator. Однако его использование в ряде случаев

может быть полезным, так как он обеспечивает ряд дополнительных возможностей навигации и редактирования набора данных:

- ☐ переход на первую и последнюю записи таблицы;
- ☐ удаление записи;
- ☐ вставка новой записи;
- ☐ отмена ошибочно введенных данных.

Кроме того, компонент `TDBNavigator` позволяет включить механизм контроля удаления записей. Для этого необходимо установить значение его свойства `ConfirmDelete` равным `true`.

Если во время выполнения программы в таблице изменить значение какого-либо поля, то внесенные изменения автоматически сохраняются при переходе на другую запись. С целью предотвращения случайного искажения данных перед сохранением изменений можно запрашивать подтверждение у пользователя. Реализацию этой процедуры можно осуществить разными методами. Наиболее просто использовать метод-обработчик события `BeforePost` компонента доступа к данным, который выполняется перед выполнением сохранения данных в таблице. Для возврата исходных значений можно использовать свойство `SelectedField` класса `TDBGrid`, указывающее на текущее поле сетки, и свойство `OldValue` класса `TField`, содержащее предыдущее значение поля. Для вывода запроса о подтверждении удаления следует создать специальное окно диалога, содержащее текст предупреждения об изменении данных, и две кнопки — подтверждение изменения и отмена изменения. В качестве такого диалога используется обычная форма, отображаемая в модальном режиме. Примерный внешний вид окна диалога подтверждения изменения показан на рис. 10.15, а текст процедуры-обработчика события `BeforePost` приведен в листинге 10.3.

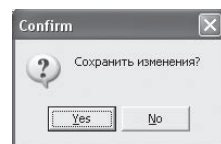


Рис. 10.15. Запрос на подтверждение модификации данных

Листинг 10.3. Обработчик события `BeforePost`

```
procedure TForm1.ADOTable1BeforePost(DataSet: TDataSet);
begin
    if MessageDlg('Сохранить изменения?', mtConfirmation, mbYesNo, 0) <> mrYes
    then
        DBGrid1.SelectedField.Value :=
            DBGrid1.SelectedField.OldValue;
end;
```

В том виде компонент `TDBGrid`, как он изображен на рис. 10.14, не очень нагляден, например должность отображается в виде кода. Мы можем сделать табличную форму более наглядной, воспользовавшись возможностями `lookup`-полей.

Чтобы создать `lookup`-поле, произведите следующую последовательность действий.

1. Выполните двойной щелчок мышью на компоненте `ADOTable1`, после чего появится окно, в котором можно работать с полями данного компонента.
2. Щелкните на открытом в предыдущем пункте окне правой кнопкой мыши и в появившемся контекстном меню (рис. 10.16) выберите команду `Add all fields`.

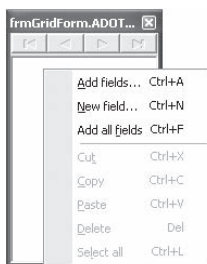


Рис. 10.16. Окно со списком полей компонента TADOTable

3. После выполнения команды **Add all fields** заполнится список полей компонента TADOTable.
4. Теперь нужно создать набор данных, из которого будет производиться подстановка в lookup-поле. Добавим в проект еще один компонент TADOTable, который будет связан с таблицей Должности базы данных employees.mdb. Для этого выполните шаги, аналогичные тем, что были произведены при добавлении компонента ADOTable1, только свяжите новый компонент с таблицей Должности.
5. Чтобы мы могли видеть не код должности, а ее наименование, в первую очередь необходимо создать собственно lookup-поле. Для этого в контекстном меню (рис. 10.14) выберем команду **New Field**.
6. В появившемся окне **New Field** зададим соответствующие значения, как показано на рис. 10.17.

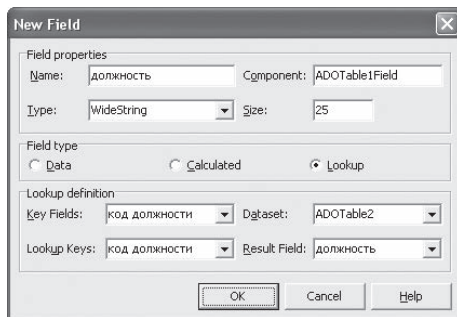
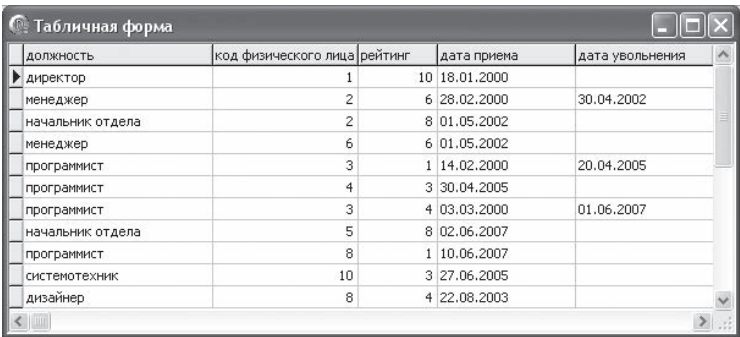


Рис. 10.17. Добавление нового поля в набор данных

7. В списке полей таблицы TADOTable появится новое поле с именем **Должность**.
8. Чтобы это поле отображалось в компоненте DBGrid1, нужно вызвать редактор столбцов и добавить это поле. При этом столбцу код должности свойство **Visible** можно установить в значение **False**, т. к. нам теперь отображать его не требуется. Табличная форма с добавленным полем **должность** показана на рис. 10.18.

Аналогичным образом можно вместо столбца код физического лица, который не очень нагляден, добавить столбец **физическое лицо**, который будет ото-



должность	код физического лица	рейтинг	дата приема	дата увольнения
директор	1	10	18.01.2000	
менеджер	2	6	28.02.2000	30.04.2002
начальник отдела	2	8	01.05.2002	
менеджер	6	6	01.05.2002	
программист	3	1	14.02.2000	20.04.2005
программист	4	3	30.04.2005	
программист	3	4	03.03.2000	01.06.2007
начальник отдела	5	8	02.06.2007	
программист	8	1	10.06.2007	
системотехник	10	3	27.06.2005	
дизайнер	8	4	22.08.2003	

Рис. 10.18. Табличная форма с добавленным столбцом должность

бражать содержимое соответствующего lookup, поля, содержащего, например, фамилию, имя и отчество.

Формы со вкладками

При разработке приложений баз данных часто требуется размещать на форме большой объем информации. Если размещаемые данные можно разделить на несколько групп, то в этом случае удобно использовать вкладки. Для создания форм со вкладками предназначен специальный элемент управления TPageControl.

Элемент управления TPageControl

Элемент TPageControl обеспечивает создание многостраничных окон. На каждой странице данного элемента можно размещать любые другие компоненты (включая TPageControl).

Добавлять и удалять вкладки можно как на стадии разработки программы, так и во время ее выполнения. Для добавления вкладки используется команда New Page контекстного меню компонента TPageControl. Удаление вкладок производится командой Delete Page контекстного меню.

Основные свойства элемента TPageControl приведены в табл. 10.6.

Таблица 10.6. Основные свойства компонента TPageControl

Свойство	Тип	Описание
ActivePage	TTabSheet	Активная вкладка
ActivePageIndex	Integer	Порядковый номер активной вкладки
PageCount	Integer	Количество страниц в компоненте (только для чтения)
Pages[Index: Integer]	TTabSheet	Массив вкладок, имеющих в TPageControl. Используется для прямого доступа ко вкладкам
HotTrack	Boolean	Определяет, будет (true) или нет (false) выделяться цветом заголовков, находящийся под указателем мыши
MultiLine	Boolean	Если значение данного свойства равно true, то заголовки вкладок могут размещаться в несколько строк. Используется при большом количестве вкладок

продолжение ↗

Таблица 10.6 (продолжение)

Свойство	Тип	Описание
Style	TTabStyle = (tsTabs, tsButtons, tsFlatButtons);	<p>Определяет внешний вид заголовков вкладок (рис. 10.19):</p> <ul style="list-style-type: none"> • tsTabs — обычные вкладки; • tsButtons — отображает заголовки в виде кнопок; • tsFlatButtons — отображает заголовки в виде плоских кнопок

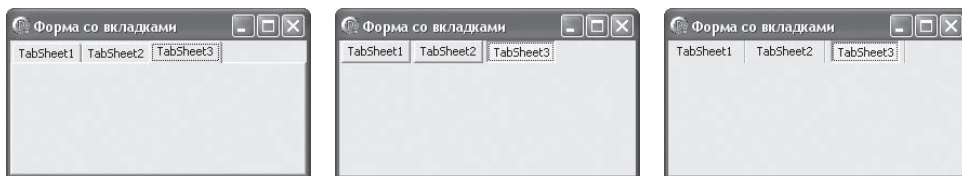


Рис. 10.19. Различные виды вкладок

Каждая отдельная вкладка является объектом класса TTabSheet. Поскольку при использовании вкладок обращаться к свойствам и методам отдельных страниц обычно не приходится, то здесь мы рассматривать их не будем. В случае необходимости обращайтесь к справочной системе Delphi.

Пример формы со вкладками

В базе данных employees.mdb, рассматриваемой нами в качестве примера, содержатся две логически связанные таблицы, в которых хранится информация о сотрудниках фирмы — это таблицы Сотрудники и Физические лица. В первой таблице содержатся служебные сведения о сотруднике: должность, разряд, зарплата и т. п. Во второй — персональные данные. При создании формы для просмотра и редактирования данных о сотрудниках целесообразно использовать одну форму, но разместить информацию из разных таблиц на разных вкладках. Содержание первой вкладки будет составлять персональная информация из таблицы Физические лица, на второй отобразим служебную информацию из таблицы Сотрудники.

Последовательность действий при создании простых форм будет примерно следующей.

1. Для создания нового приложения выполните команду **File ► New ► VCL Form Application — Delphi for Win32**.
2. Поместите на форму компонент TPageControl, расположенный на вкладке Win32 палитры компонентов.
3. С помощью команды **New Page** контекстного меню компонента TPageControl создайте две вкладки.
4. Отредактируйте с помощью инспектора объектов свойство Caption для каждой вкладки. Для первой задайте заголовок **Персональная информация**, для второй — **Служебная информация**. Обратите внимание на то, что при изменении этого свойства синхронно изменяются заголовки вкладок.
5. Разместите на форме по два компонента TADOTable (находится на вкладке dbGo палитры компонентов) и TDataSource.

6. Подключите к компонентам TADOTable таблицы **Физические лица** и **Сотрудники** базы данных employees.mdb (подключение таблицы к компоненту набора данных было подробно рассмотрено ранее, в примере создания простых форм).
7. Настройте источники данных TDataSource: один свяжите с набором данных таблицы **Физические лица**, второй — с набором данных таблицы **Сотрудники**.
8. Разместите необходимые элементы управления на вкладке **Персональная информация** и выполните их настройку. Размещение элементов на первой вкладке показано на рис. 10.20.

Рис. 10.20. Размещение элементов управления на вкладке **Персональная информация**

СОВЕТ

Размещение компонентов для данной вкладки можно сделать примерно таким же, как в случае простой формы из первого примера этой главы.

9. Разместите на вкладке **Служебная информация** элементы управления для отображения и редактирования информации из таблицы **Сотрудники**. Примерный вариант размещения показан на рис. 10.21.

Рис. 10.21. Размещение элементов управления на вкладке **Служебная информация**

10. Настройте элементы визуализации полей базы данных и элементы навигации по набору данных. Свяжите элементы, расположенные на вкладке **Персональная информация**, с набором данных таблицы **Физические лица**, а на вкладке **Служебная информация** — с набором данных таблицы **Сотрудники**.

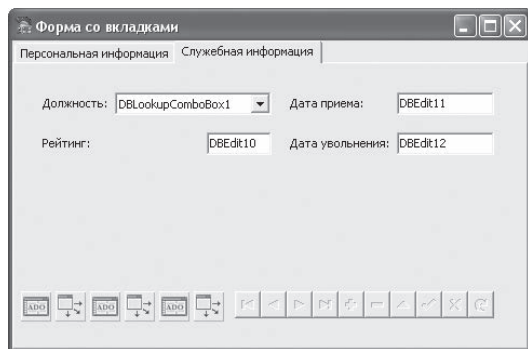


Рис. 10.21. Размещение элементов управления на вкладке **Служебная информация**

11. Добавьте в обработчик события **OnShow** главной формы вызов метода **Open** для каждого набора данных, а в обработчик события **OnClose** — вызов метода **Close**. Текст данных обработчиков приведен в листинге 10.4.

Листинг 10.4. Обработчики событий для формы со вкладками

```
procedure TfrmTabbed.FormShow(Sender: TObject);
begin
    ADOTable1.Open;
    ADOTable2.Open;
    ADOTable3.Open;
end;

procedure TfrmTabbed.FormClose(Sender: TObject;
    var Action: TCloseAction);
begin
    ADOTable1.Close;
    ADOTable2.Close;
    ADOTable3.Close;
end;
```

12. Откомпилируйте и запустите программу. Внешний вид ее окна приведен на рис. 10.22.

Работа с многотабличными базами данных

В большинстве случаев база данных состоит из нескольких взаимосвязанных таблиц. При редактировании информации в базе данных часто необходимо учитывать эту взаимосвязь. В рассмотренном выше примере создания формы для двух связанных таблиц связь между этими таблицами не принималась во внимание, поэтому изменение положения курсора данных в первом наборе никак не отражалось на информации, показываемой на второй вкладке.

Рис. 10.22. Форма со вкладками во время выполнения программы

Отсутствие связи между наборами данных приводит к тому, что персональная информация на первой вкладке может не соответствовать служебной информации на второй вкладке. Для синхронизации перемещения курсора данных в нескольких наборах необходимо во время разработки приложения установить между ними связь.

Связывание наборов данных

Для связывания наборов данных между собой используются свойства MasterSource и MasterFields компонентов набора данных.

При организации связи между таблицами одна из них является *главной* (master), а все остальные, участвующие во взаимосвязи, — *подчиненными* (detail). Перемещение курсора данных в главной таблице приводит к синхронному перемещению курсора и в подчиненных таблицах. Причем все это реализовано на уровне компонентов, поэтому не требуется написание дополнительного программного кода, достаточно только задать соответствующие свойства. В обратную сторону связь не действует — перемещение курсора данных в подчиненной таблице не приведет к изменению положения курсора данных в главной таблице.

Для установления связи набор данных главной таблицы не требует никаких дополнительных настроек.

В подчиненном наборе данных необходимо с помощью свойства MasterSource указать источник данных главной таблицы. Так устанавливается отношение между полями главной и подчиненной таблиц. В этом свойстве задаются имена полей, по которым устанавливается связь (если полей несколько, то их имена разделяются точкой с запятой).

ВНИМАНИЕ

Поля, между которыми устанавливается связь, обязательно должны быть индексированными.

Для установления связи между полями можно использовать редактор связей полей (рис. 10.23), который открывается при нажатии на кнопку с многообразием в поле значения свойства MasterFields в инспекторе объектов.

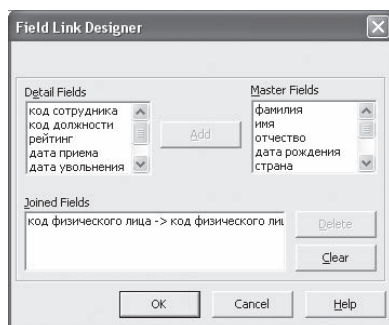


Рис. 10.23. Редактор связей полей

В окне редактора связей отображаются два списка — список полей подчиненной таблицы (**Detail Fields**) и список полей главной таблицы (**Master Fields**). Для создания связи необходимо выбрать необходимые поля в обоих списках и щелкнуть на кнопке **Add**. Созданные между полями связи отображаются в списке **Joined Fields**. Если таблицы связываются по нескольким полям, то следует задать несколько связей. Для удаления выбранной связи используется кнопка **Delete**. Щелчок на кнопке **Clear** удаляет все созданные связи.

Пример приложения со связанными таблицами

В качестве исходных данных будем использовать предыдущее приложение. На роль главной выберем таблицу **Физические лица**. Модифицируем программу следующим образом.

1. Оставьте только один компонент **TDBNavigator** и вынесите его за пределы вкладки.

Это делается потому, что для навигации по связанным таблицам необходимо перемещать курсор только в главной из них. Для этой цели достаточно одного элемента навигации. А поскольку этот элемент должен быть доступен с любой вкладки, то его желательно вынести за пределы компонента **TPageControl**. Единственный элемент навигации необходимо связать с главным набором данных.

2. Установите связь между наборами данных. Для этого в свойстве **MasterSource** набора данных, связанного с таблицей **Сотрудники**, задайте имя источника данных таблицы **Физические лица**.
3. Установите связь между полями. Для этого воспользуемся редактором связей полей. Общим для обеих таблиц является поле **Код физического лица**. Поэтому установим связь между этими полями, как показано на рис. 10.23.
4. Откомпилируйте и запустите программу. Теперь при нажатии на кнопки элемента навигации данные синхронно изменяются на обеих вкладках.

ПРИМЕЧАНИЕ

При установлении связи между полями их типы обязательно должны быть одинаковыми. Имена полей в разных таблицах не обязательно должны совпадать, однако все же желательно давать связанным полям одинаковые названия.

Часть III

Выборка данных

Глава 11

Выборка данных

Одной из задач, наиболее часто возникающих при работе с базами данных, является выборка данных, то есть извлечение из базы данных информации, отвечающей ряду требований, заданных пользователем.

Использование SQL для выборки данных из таблицы

Одним из наиболее эффективных и универсальных способов выборки данных из таблиц базы данных является использование запросов языка SQL. Команды SQL подразделяются на несколько категорий. Для выборки данных используются команды, относящиеся к так называемому языку запросов DQL (Data Query Language).

SQL-запросы можно использовать как при работе с локальными базами данных, так и с SQL-серверами баз данных (Oracle, Informix, Sybase, InterBase, Microsoft SQL Server). Причем при формировании SQL-запросов не имеет особого значения, какая система управления базами данных используется, так как команды языка SQL стандартизованы (стандарты ANSI SQL 92, 99 и 2003). Однако следует учитывать, что производители СУБД обычно предлагают свои реализации SQL, которые могут включать расширения команд стандарта и даже отклонения от него. Тем не менее большинство команд SQL имеют одинаковый или очень похожий синтаксис в различных реализациях. Поэтому, изучив одну из реализаций SQL, впоследствии можно легко перейти на другую.

В Delphi для работы с таблицами локальных баз данных с использованием BDE применяется собственная реализация языка SQL, называемая локальным SQL (Local SQL). Данная реализация является подмножеством языка SQL 92. Несмотря на то что она не содержит отклонений от стандарта, ее возможности несколько урезаны.

При работе с SQL-серверами обработка запроса выполняется на стороне сервера. Поэтому особенности реализации языка SQL в этом случае определяются используемым SQL-сервером.

Компоненты Delphi, работающие с базами данных через SQL-запросы

Как мы уже говорили выше, в Turbo Delphi Explorer для доступа к базам данных могут использоваться три механизма: dbG0, dbExpress и BDE. Наиболее современной из этих трех технологий является dbGo, ее мы и будем рассматривать.

Для работы с базами данных через SQL-запросы в dbGo могут использоваться компоненты TADOCommand, TADODataSet, TADOStoredProc и TADOQuery. Все эти компоненты могут быть использованы для выполнения SQL-запросов, но мы рассмотрим компонент TADOQuery, предназначенный специально для этих целей.

Компонент TADOQuery

При использовании компонента TADOQuery подготовка и диспетчеризация запросов выполняется dbG0, которая основана на ADO.Net.

По своим свойствам и назначению компонент TADOQuery подобен компоненту TADOTable. Отличие заключается только в способе получения данных: в TADOTable используются методы, инкапсулированные в классе TADOTable, а TADOQuery получает данные как результат выполнения SQL-запроса.

Применение языка SQL позволяет легко решать задачи, которые сложно или вообще невозможно решить в рамках класса TADOTable. Поэтому компонент TADOQuery является гораздо более мощным и гибким инструментом для работы с базами данных. Основные свойства класса TADOQuery приведены в табл. 11.1.

Таблица 11.1. Основные свойства класса TADOQuery

Свойство	Тип	Описание
DataSource	TDataSource	Задаёт источник данных, связанный с набором данных, поля которого используются в качестве параметров SQL-запроса
ParamCheck	Boolean	Определяет, следует ли обновлять параметры запроса при изменении свойства SQL во время выполнения программы
Parameters	TParameters	Содержит коллекцию параметров SQL-оператора
Prepared	Boolean	Определяет, будет ли запрос подготовлен заранее. Задав значение этого свойства равным true, вы избегаете подготовки запроса при каждом его открытии
RowsAffected	Integer	Содержит число, равное количеству записей, изменённых с момента последнего выполнения запроса
SQL	TStrings	Содержит текст SQL-запроса

В классе TADOQuery также имеется ряд методов, которые довольно часто используются при работе с базами данных через SQL-запросы:

- `procedure ExecSQL` — выполняет SQL-запрос, заданный в свойстве SQL. Обычно используется в тех случаях, когда в результате выполнения запроса не возвращаются данные (например, при выполнении команд INSERT, UPDATE, DELETE и CREATE TABLE).

ПРИМЕЧАНИЕ

Для выполнения команды `SELECT` необходимо использовать метод `Open` компонента `TADOQuery`.

Пример использования компонентов доступа к данным через SQL-запросы

Рассмотрим возможность практического использования перечисленных выше компонентов. В дальнейшем полученные результаты мы будем использовать в качестве примера для изучения команд `SQL`. В качестве исходных данных воспользуемся уже известной нам базой данных `InfSyst`.

Последовательность действий, выполняемых при создании форм ввода данных с использованием `SQL`-запросов, будет примерно следующей.

1. Для создания нового приложения выполните команду `File ► New ► VCL Form Application — Delphi for Win32`.
2. Разместите на форме компоненты `TADOQuery` и `TDataSource`. Последний необходим для связи набора данных с компонентами визуализации данных и расположен на вкладке `Data Access` палитры компонентов.
3. Подключите базу данных `Sales.mdb`.

Поскольку последовательность действий при подключении базы данных аналогична процедуре, рассмотренной в предыдущей главе для компонента `TADOTable`, здесь мы не будем подробно на этом останавливаться.

ПРИМЕЧАНИЕ

В отличие от `TADOTable`, класс `TADOQuery` не имеет свойств, в которых указывается связанная с ним таблица базы данных. При его использовании информация поступает в набор данных в результате выполнения `SQL`-запроса, заданного в свойстве `SQL`.

4. Для задания запроса щелкните на кнопке с многоточием в поле ввода свойства `SQL` в инспекторе объектов. При этом откроется окно простого текстового редактора, в котором формируется запрос. Сформируйте запрос, как показано на рис. 11.1. Его назначение состоит в возвращении выборки данных, содержащей все поля и все записи таблицы `Товары` нашей базы данных.

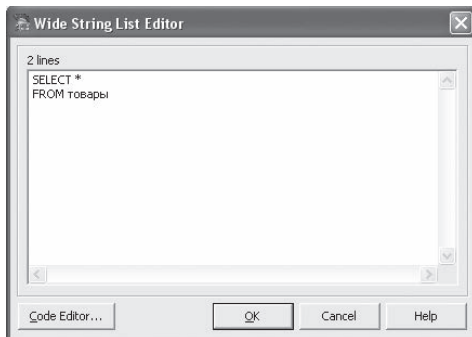


Рис. 11.1. Окно редактора `SQL`-запросов

5. Щелкните на кнопке **Code Editor**. Теперь текст запроса будет отображаться в окне редактора кода (рис. 11.2), причем ключевые слова языка SQL будут выделяться полужирным шрифтом, что снижает вероятность ошибок при написании запроса.

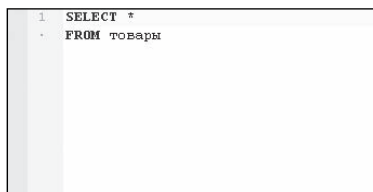


Рис. 11.2. Текст SQL-запроса в редакторе кода

6. Выполните настройку источника данных **TDataSource**. Она производится так же, как и в случае использования компонента **TADOTable** — в свойстве **DataSet** указывается имя объекта доступа к данным (по умолчанию — **ADOQuery1**). Далее следует разместить на форме необходимые элементы управления и выполнить их настройку.

7. Выберите следующие элементы:

- компонент **TMemo**, который будет использоваться для отображения и редактирования текста запроса;
- компонент отображения данных **TDBGrid** — для отображения результатов выполнения запроса;
- кнопка **TButton** — для подачи команды на выполнение запроса.

Примерный вариант размещения на форме необходимых компонентов показан на рис. 11.3.

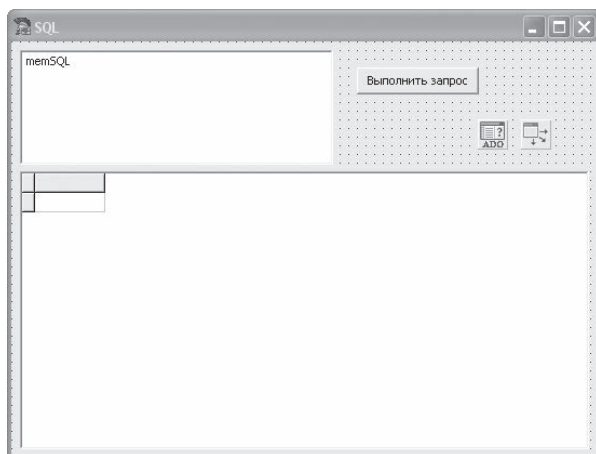


Рис. 11.3. Размещение элементов управления на форме

8. Для настройки компонента визуализации полей базы данных **TDBGrid** в его свойстве **DataSource** укажите имя источника данных (по умолчанию — **DataSource1**).

Следующий этап — реализация процедур открытия и закрытия набора данных.

9. Если в результате выполнения SQL-запроса возвращаются данные, для его выполнения необходимо воспользоваться методом `Open` класса `TADOQuery`. Данный метод следует выполнять при запуске приложения, например в обработчике события `OnShow` главной формы. При этом происходит выполнение запроса, заданного в свойстве `SQL`, а результаты его выполнения отображаются в таблице `DBGrid1`.
10. При закрытии приложения следует закрыть и набор данных. Вызовите метод `Open` в обработчике события `OnShow` главной формы, а метод `Close` — в обработчике ее же события `OnClose`.
11. Осталось написать обработчик события `OnClick` для кнопки **Выполнить запрос**. Данная кнопка понадобится в дальнейшем для возможности изменения текста запроса с последующим выполнением его без перекомпиляции программы. При нажатии на кнопку должен выполняться запрос. Будем полагать, что в результате выполнения запроса возвращаются данные. В этом случае при нажатии на кнопку следует выполнить следующие действия:
 - передать текст запроса из компонента `memSQL` в свойство `SQL` компонента `ADOADOQuery1`;
 - открыть набор данных, вызвав метод `Open` компонента `ADOQuery1`.

Текст модуля разработанной формы приведен в листинге 11.1.

Листинг 11.1. Главный модуль приложения

```
unit SQL_main;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, Grids, DBGrids, Db,
  StdCtrls, ExtCtrls, DBTables;

type
  TfrmMain = class(TForm)
    ADOADOQuery1: TADOQuery;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    memSQL: TMemo;
    btnOpenQuery: TButton;
    procedure FormShow(Sender: TObject);
    procedure FormClose(Sender: TObject;
      var Action: TCloseAction);
    procedure btnOpenQueryClick(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
```

```
frmMain: TfrmMain;

implementation

{$R *.DFM}

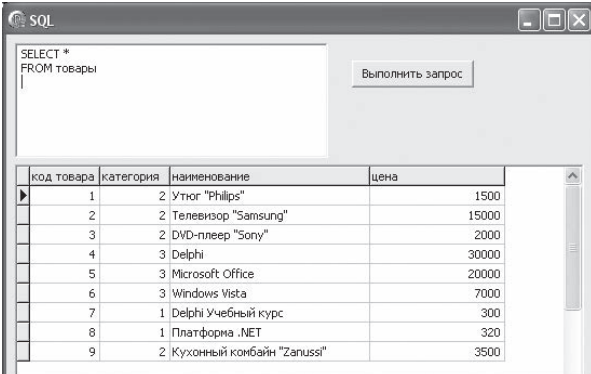
procedure TfrmMain.FormShow(Sender: TObject);
begin
    memSQL.Lines.Clear;
    memSQL.Lines.Assign(ADOQuery1.SQL);
    ADOQuery1.Open;
end;

procedure TfrmMain.FormClose(Sender: TObject;
    var Action: TCloseAction);
begin
    if ADOQuery1.Active then ADOQuery1.Close;
end;

procedure TfrmMain.btnOpenQueryClick(Sender: TObject);
begin
    if ADOQuery1.Active then ADOQuery1.Close;
    ADOQuery1.SQL.Clear;
    ADOQuery1.SQL.Assign(memSQL.Lines);
    ADOQuery1.Open;
end;

end.
```

12. Откомпилируйте и запустите приложение. После его запуска в компоненте DBGrid1 на форме отобразится информация, содержащаяся в таблице **Товары** базы данных **sales.mdb** (рис. 11.4).



код товара	категория	наименование	цена
1	2	Утюг "Philips"	1500
2	2	Телевизор "Samsung"	15000
3	2	DVD-плеер "Sony"	2000
4	3	Delphi	30000
5	3	Microsoft Office	20000
6	3	Windows Vista	7000
7	1	Delphi Учебный курс	300
8	1	Платформа .NET	320
9	2	Кухонный комбайн "Zanussi"	3500

Рис. 11.4. Результат выполнения SQL-запроса

Язык запросов DQL

Язык запросов, являющийся одной из категорий языка SQL, состоит всего из одной команды **SELECT**. Эта команда вместе с множеством опций и предложений используется для формирования запросов к базе данных. Запросы формируются

для извлечения из таблиц базы данных информации, соответствующей некоторым требованиям, задаваемым пользователем.

Оператор SELECT не используется автономно, вместе с ним обязательно должны задаваться уточняющие предложения. Предложения, используемые совместно с командой SELECT, могут быть *обязательными* и *дополнительными*. *Обязательным* является только одно предложение — FROM, без которого оператор SELECT не может использоваться.

Простейшая форма оператора SELECT

Оператор SELECT вместе с предложением FROM используется для получения информации из базы данных. Синтаксис простейшей формы оператора SELECT приведен ниже:

```
SELECT {* | ALL | DISTINCT field1, field2, ... , fieldN}  
FROM table1 {, table2, ... , tableN}
```

Здесь за ключевым словом SELECT следует список полей, которые возвращаются в результате выполнения запроса:

- ☐ имена полей в списке разделяются через запятую;
- ☐ для выборки всех полей таблицы (таблиц) используется символ подстановки «*»;
- ☐ опция ALL (задана по умолчанию) означает, что результат выборки будет содержать все записи, включая дублирующие друг друга;
- ☐ при использовании опции DISTINCT результат запроса не будет содержать дублирующихся строк.

Совместно с командой SELECT всегда используется предложение FROM, с помощью которого указывается имя таблицы (таблиц), из которой производится выборка. Если в предложении FROM указывается несколько таблиц, то их имена разделяются запятыми.

Выше мы уже рассмотрели пример использования оператора SELECT для выборки всей информации, содержащейся в таблице Товары. Чтобы выбрать не все поля, а лишь некоторые, необходимо после слова SELECT указать имена полей, которые будут включены в результат выборки. В качестве примера ниже приведен запрос, возвращающий значения только трех полей: Код товара, Наименование и Цена:

```
SELECT [код товара], наименование, цена  
FROM товары
```

СОВЕТ

Обратите внимание, что при указании в списке оператора SELECT имен полей, содержащих пробел, их необходимо заключать в квадратные скобки. Это правило необходимо выполнять и для имен таблиц с пробелами, указываемых, например, в предложении FROM.

В результате выполнения данного запроса возвращаются все записи, содержащиеся в трех полях таблицы Товары (рис. 11.5).

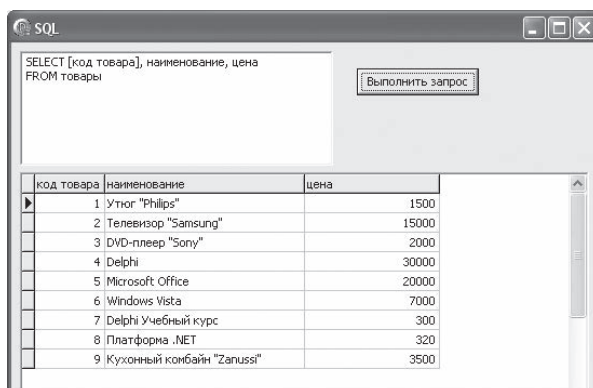


Рис. 11.5. Результат выбора трех полей

ПРИМЕЧАНИЕ

Для выполнения запроса нет необходимости перекомпилировать программу. Достаточно во время ее выполнения ввести текст запроса в поле ввода и нажать на кнопку Выполнить запрос.

Задание условий при выборке данных

Для ограничения отбираемой из базы данных информации оператор SELECT позволяет использовать условие, которое задается с помощью предложения WHERE. В случае реализации условной выборки оператор SELECT имеет следующий вид:

```
SELECT { * | ALL | DISTINCT field1, field2, ... , fieldN }  
FROM table1 { , table2, ... , tableN }  
WHERE условие
```

Специальные операторы языка SQL, применяемые для задания условия, можно разделить на следующие группы:

- ☐ операторы сравнения;
- ☐ логические операторы;
- ☐ операторы объединения;
- ☐ операторы отрицания.

Результатом выполнения каждого из этих операторов является логическое значение (true или false). Если для некоторой записи оператор возвращает значение true, то запись включается в результат выборки, если false — не включается.

Операторы сравнения

Операторы сравнения используются в запросах SQL для наложения ограничений на информацию, возвращаемую в результате выполнения запроса. Это типичные операторы, существующие во всех алгоритмических языках:

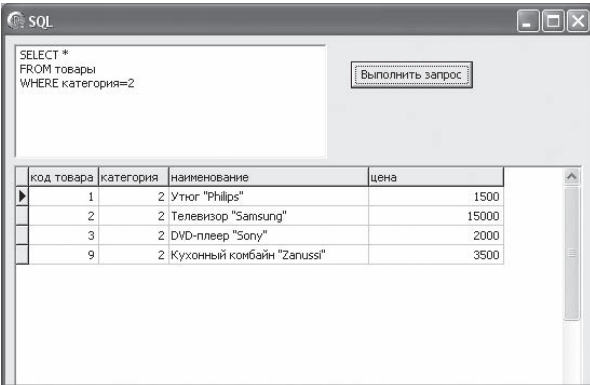
- ☐ оператор равенства «=» используется для отбора записей, в которых значение определенного поля точно соответствует заданному;

- ❑ оператор неравенства «<>» возвращает значение true, если значение поля не совпадает с заданным значением;
- ❑ операторы «меньше» и «больше» (соответственно, «<» и «>») позволяют отбирать записи, в которых значение определенного поля меньше или больше некоторой заданной величины;
- ❑ операторы «меньше или равно» и «больше или равно» (соответственно, «<=» и «>=») представляют собой объединение операторов «меньше» и «равно», «больше» и «равно». В отличие от операторов «<» и «>», операторы «<=» и «>=» возвращают значение true, если значение поля совпадает с заданным значением.

В качестве примера рассмотрим запрос, выбирающий из таблицы Товары только те записи, категория товаров в которых равна 2:

```
SELECT *  
FROM товары  
WHERE категория=2
```

Результат выполнения данного запроса показан на рис. 11.6.



The screenshot shows a window titled "SQL" with a text area containing the query: `SELECT *
FROM товары
WHERE категория=2`. To the right of the text area is a button labeled "Выполнить запрос". Below the text area is a table with the following data:

код товара	категория	наименование	цена
1	2	Утюг "Philips"	1500
2	2	Телевизор "Samsung"	15000
3	2	DVD-плеер "Sony"	2000
9	2	Кухонный комбайн "Zanussi"	3500

Рис. 11.6. Результат выполнения запроса с условием

Логические операторы

К *логическим* относятся операторы, в которых для задания ограничений на отбор данных используются специальные ключевые слова. В SQL определены следующие логические операторы: IS null, BETWEEN...AND, IN, LIKE, EXISTS, UNIQUE, ALL, ANY.

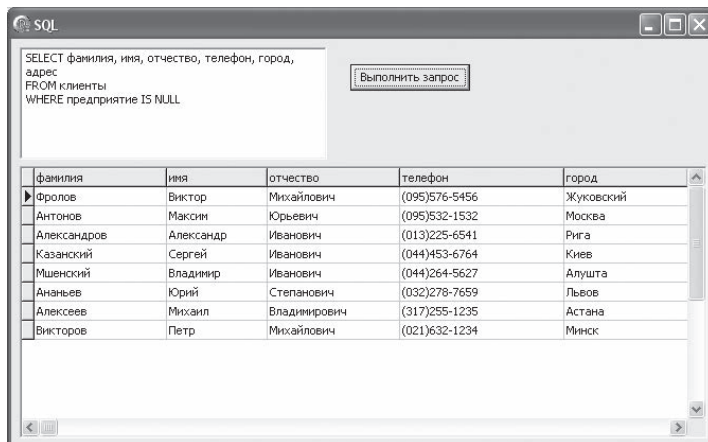
Оператор IS NULL

Оператор IS NULL предназначен для сравнения текущего значения поля со значением NULL. Он используется для отбора записей, в некоторое поле которых не занесено никакое значение.

Для иллюстрации использования этого оператора воспользуемся таблицей Клиенты. С помощью приведенного ниже запроса произведем выборку из нее записей клиентов, у которых не указано название предприятия, которое они представляют:


```
SELECT фамилия, имя, отчество, телефон, город, адрес  
FROM клиенты  
WHERE предприятие IS NULL
```

Результат выполнения запроса показан на рис. 11.7.



фамилия	имя	отчество	телефон	город
Фролов	Виктор	Михайлович	(095)576-5456	Жуковский
Антонов	Максим	Юрьевич	(095)532-1532	Москва
Александров	Александр	Иванович	(013)225-6541	Рига
Казанский	Сергей	Иванович	(044)453-6764	Киев
Мшенский	Владимир	Иванович	(044)264-5627	Алушта
Ананиев	Юрий	Степанович	(032)278-7659	Львов
Алексеев	Михаил	Владимирович	(317)255-1235	Астана
Викторов	Петр	Михайлович	(021)632-1234	Минск

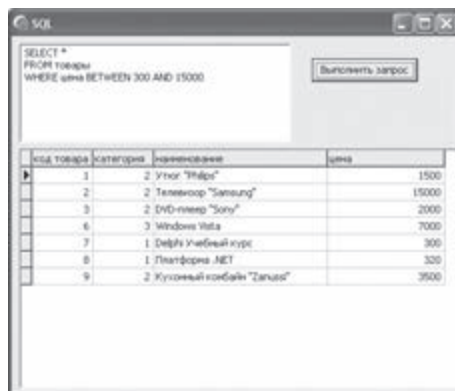
Рис. 11.7. Пример использования оператора IS NULL

Оператор BETWEEN...AND

Оператор BETWEEN...AND применяется для отбора записей, в которых значения поля находятся внутри заданного диапазона. Границы диапазона включаются в условие отбора. Чтобы продемонстрировать работу этого оператора, вернемся к таблице Товары и выберем в ней товары, цена которых находится в диапазоне от 300 до 15 000. Для этого сформируем следующий запрос:

```
SELECT *  
FROM товары  
WHERE цена BETWEEN 300 AND 15000
```

Результаты, возвращенные при выполнении данного запроса, приведены на рис. 11.8.



код товара	категория	наименование	цена
1	2	Утюг "Viktor"	1500
2	2	Телевизор "Samsung"	15000
3	2	DVD-плеер "Sony"	2000
6	3	Windows Vista	7000
7	1	Delphi учебный курс	300
8	1	Платформа .NET	320
9	2	Холодильный комбайн "Zanussi"	3500

Рис. 11.8. Пример использования оператора BETWEEN...AND

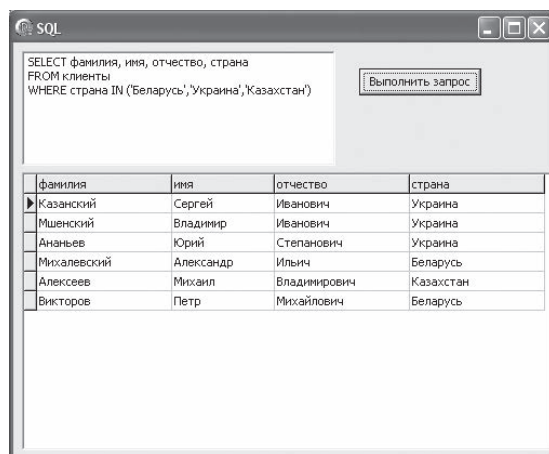
Оператор IN

Оператор IN используется для выборки записей, в которых значение некоторого поля соответствует хотя бы одному из значений заданного списка.

Выберем из таблицы Клиенты список клиентов, которые живут в Беларуси, Украине или Казахстане:

```
SELECT фамилия, имя, отчество, страна
FROM клиенты
WHERE страна IN ('Беларусь', 'Украина', 'Казахстан')
```

Результат выполнения данного запроса показан на рис. 11.9.



фамилия	имя	отчество	страна
Казанский	Сергей	Иванович	Украина
Мшенский	Владимир	Иванович	Украина
Ананьев	Юрий	Степанович	Украина
Михалевский	Александр	Ильич	Беларусь
Алексеев	Михаил	Владимирович	Казахстан
Викторов	Петр	Михайлович	Беларусь

Рис. 11.9. Пример использования оператора IN

Оператор LIKE

Оператор LIKE применяется для сравнения значения поля со значением, заданным при помощи шаблонов. Для задания шаблонов используются два символа:

- знак процента «%» — заменяет последовательность символов любой (в том числе и нулевой) длины;
- символ подчеркивания «_» — заменяет любой единичный символ.

Найдем в таблице Клиенты записи, в которых фамилия клиента начинается с буквы «М»:

```
SELECT фамилия, имя, отчество, телефон
FROM клиенты
WHERE фамилия LIKE 'М%'
```

В результате выполнения этого запроса выбрано три записи (рис. 11.10).

А теперь найдем в этой же таблице записи, для которых номер телефона начинается на цифры (095)5, а остальные цифры неизвестны:

```
SELECT фамилия, имя, отчество, телефон
FROM клиенты
WHERE телефон LIKE '(095)5_ _ _ _ _'
```

При выполнении данного запроса отобраны две записи (рис. 11.11).

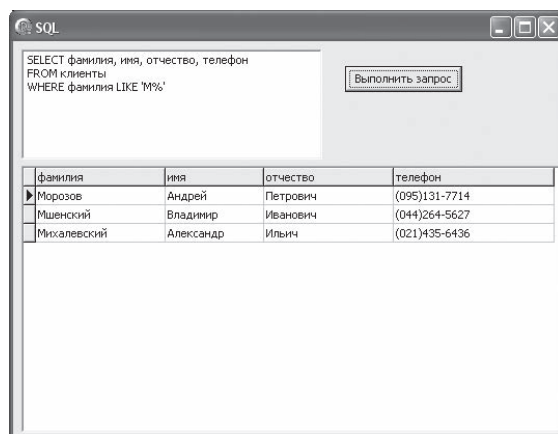


Рис. 11.10. Использование оператора LIKE с шаблоном «%»

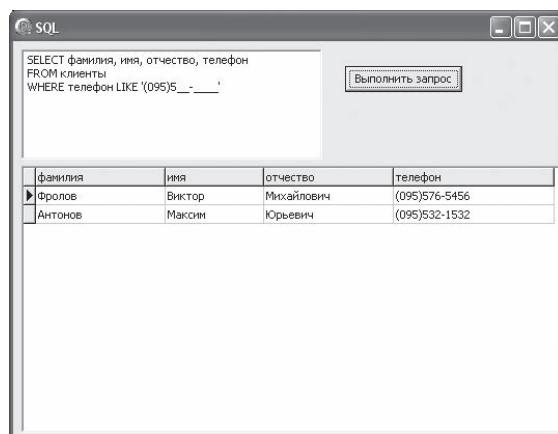


Рис. 11.11. Использование оператора LIKE с шаблоном «_»

Оператор EXISTS

Оператор EXISTS используется для отбора записей, соответствующих заданному критерию.

Для иллюстрации его работы рассмотрим следующий пример. Из таблицы Товары требуется отобрать список товаров, количество продаж которых превышает 10. Сведения о продажах содержатся в таблице Продажи в поле Продано. Для получения необходимой выборки воспользуемся оператором EXISTS:

```
SELECT наименование, цена
FROM товары
WHERE EXISTS (SELECT [код товара]
              FROM продажи
              WHERE (продажи.продано>30) AND
                    товары.[код товара]=[код товара])
```

В этом запросе после ключевого слова EXISTS следует оператор SELECT, отбирающий из таблицы Продажи записи, для которых количество продаж превышает 30. Оператор EXISTS отбирает из таблицы Товары записи, в которых значение поля Код товара соответствует отобранным из таблицы Продажи. Результат выполнения данного запроса приведен на рис. 11.12.

ПРИМЕЧАНИЕ

При использовании оператора EXISTS (а также еще трех логических операторов: UNIQUE, ALL и ANY) применяется *подзапрос* — оператор SELECT, следующий за ключевым словом EXISTS и заключенный в круглые скобки. Более подробно подзапросы будут рассмотрены ниже.

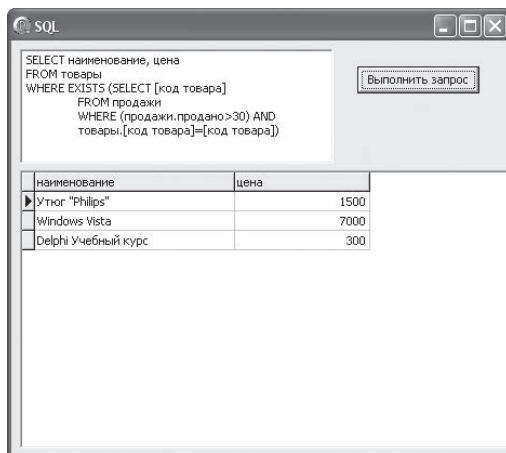


Рис. 11.12. Пример использования оператора EXISTS

Оператор UNIQUE

Оператор UNIQUE используется для проверки записи таблицы на уникальность. По своему действию он аналогичен оператору EXISTS. Единственное отличие заключается в том, что подзапрос, задаваемый после ключевого слова UNIQUE, не должен возвращать более одной записи.

Оператор ALL

Оператор ALL используется для сравнения исходного значения со всеми другими значениями, входящими в некоторый набор данных.

Например, для того чтобы выбрать из таблицы Товары те товары, которые имеют цену большую, чем цена всех товаров, проданных в количестве более 30, используется следующий запрос:

```
SELECT *
FROM товары
WHERE цена>ALL (SELECT продажи.цена
                FROM продажи
                WHERE продажи.продано>30)
```

Результат выполнения данного запроса приведен на рис. 11.13.

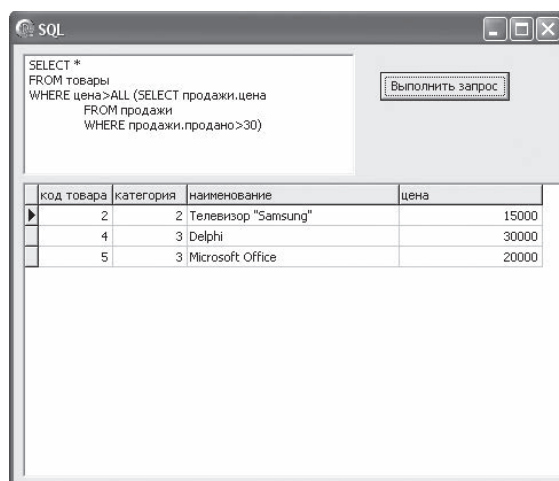


Рис. 11.13. Пример использования оператора ALL

Оператор ANY

Оператор ANY применяется для сравнения заданного значения с каждым из значений некоторого набора данных. Если в предыдущем примере заменить оператор ALL на ANY, то будет возвращен список товаров, цена которых больше, чем *хотя бы у одного* из товаров, проданных в количестве больше 30. Результат выполнения такого запроса показан на рис. 11.14.

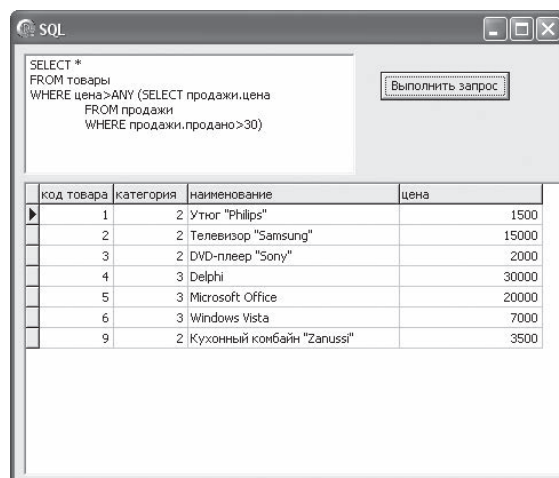


Рис. 11.14. Пример использования оператора ANY

Операторы объединения

Часто при написании запроса на выборку данных требуется задать сложное условие, для которого недостаточно использовать только один оператор. В этом

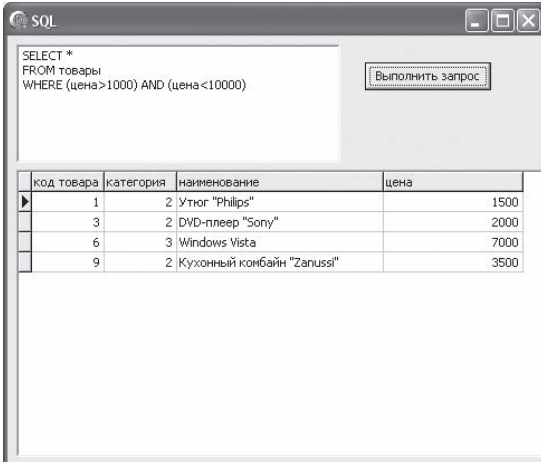
случае используется объединение нескольких условий с помощью специальных операторов. В SQL определены два таких оператора:

- ❑ оператор AND используется в тех случаях, когда необходимо отобрать записи, соответствующие нескольким условиям. Причем для каждой записи, включаемой в результат выборки, должны выполняться *все* заданные ограничения. Оператор AND объединяет несколько условий путем выполнения операции логического умножения результатов всех заданных ограничений. Результат true, соответственно, будет получен только в том случае, если все объединяемые условия принимают значение true;
- ❑ оператор OR выполняет операцию логического сложения результатов всех заданных условий. При использовании данного оператора запись включается в результирующую выборку в случае выполнения *хотя бы одного* из заданных ограничений.

При использовании операторов объединения каждое логическое выражение следует заключать в круглые скобки. Для примера произведем выборку данных о товарах, цена которых больше 1000, но меньше 10 000:

```
SELECT *  
FROM товары  
WHERE (цена>1000) AND (цена<10000)
```

Результат выполнения запроса приведен на рис. 11.15.



The screenshot shows a window titled "SQL" with a text area containing the query: `SELECT *
FROM товары
WHERE (цена>1000) AND (цена<10000)`. To the right of the text area is a button labeled "Выполнить запрос". Below the text area is a table with the following data:

код товара	категория	наименование	цена
1	2	Утюг "Philips"	1500
3	2	DVD-плеер "Sony"	2000
6	3	Windows Vista	7000
9	2	Кухонный комбайн "Zanussi"	3500

Рис. 11.15. Пример объединения логических выражений

Синтаксические правила использования оператора OR такие же, как и для оператора AND. Следующий запрос:

```
SELECT *  
FROM товары  
WHERE (цена<1000) OR (цена>10000)
```

возвратит список товаров, цена которых меньше 1000 или больше 10 000 (рис. 11.16).

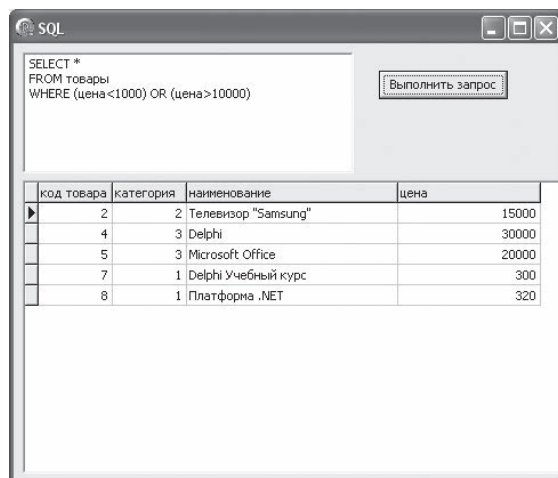


Рис. 11.16. Пример использования оператора объединения OR

Оператор отрицания

Для каждого из рассматриваемых операторов может быть выполнена операция отрицания, меняющая результат выполнения оператора на противоположный. Для реализации этой операции используется оператор NOT. Ниже приведены примеры использования этого оператора с логическими операторами:

```
IS NOT NULL  
NOT BETWEEN  
NOT IN  
NOT LIKE  
NOT EXISTS  
NOT UNIQUE
```

Упорядочение данных

Для упорядочения данных в выборке, полученной в результате выполнения запроса, используется предложение ORDER BY. Синтаксис оператора SELECT в этом случае будет следующим:

```
SELECT {* | ALL | DISTINCT field1, field2, ... , fieldN}  
FROM table1 {, table2, ... , tableN}  
WHERE условие  
ORDER BY field {ASC | DESC}
```

После ключевых слов ORDER BY указывается имя поля (полей), по которому производится сортировка, а затем указывается режим сортировки:

- ❑ ASC — режим, используемый по умолчанию. При этом информация располагается в порядке возрастания значения указанного поля (для текстовых полей — в алфавитном порядке);
- ❑ DESC — используется для вывода информации в порядке убывания значений указанного поля (для текстовых полей — в порядке, обратном алфавитному).

Например, чтобы отсортировать список товаров по алфавиту, следует использовать следующий запрос:

```
SELECT категория, наименование, цена  
FROM товары  
ORDER BY наименование
```

Результат выполнения данного запроса приведен на рис. 11.17.

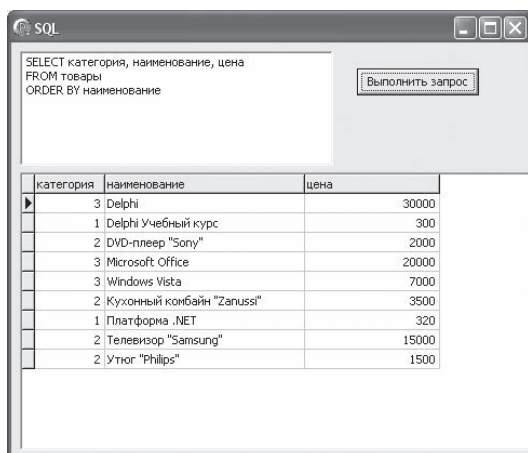


Рис. 11.17. Пример использования предложения ORDER BY

Вместо имени поля в предложении ORDER BY можно использовать целое число, определяющее порядковый номер поля в списке после ключевого слова SELECT (если производится выборка всех полей таблицы с помощью символа «*», то число указывает порядковый номер поля в таблице базы данных). Например, для вывода списка товаров в порядке убывания цены можно использовать следующий запрос:

```
SELECT категория, наименование, цена  
FROM товары  
ORDER BY 3 DESC
```

Результат выполнения запроса изображен на рис. 11.18.

Использование вычисляемых полей

Язык SQL позволяет создавать вычисляемые поля в тексте запроса. Для реализации этой функции в запросе просто приводится выражение, в котором используются арифметические и математические операторы, а также имена полей

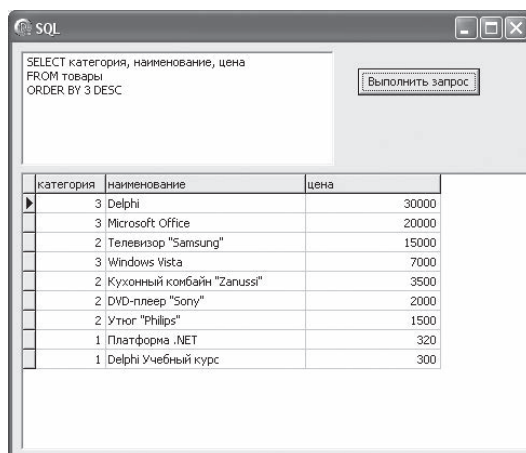


Рис. 11.18. Пример использования порядкового номера поля в предложении ORDER BY

в качестве переменных. В результате выполнения запроса с вычисляемыми полями выборка будет содержать не только ту информацию, которая содержится в таблицах базы данных, но и дополнительную информацию, полученную в результате вычисления заданного выражения.

ПРИМЕЧАНИЕ

Кроме математических операций в SQL поддерживается ряд строчковых функций, выполняющих такие операции, как конкатенация строк, выделение подстроки, поиск подстроки внутри строки и ряд других. В запросах SQL также могут применяться функции преобразования символьного типа в числовой, и наоборот, символьного типа в дату и т. п.

При создании вычисляемого поля можно использовать следующие арифметические операторы:

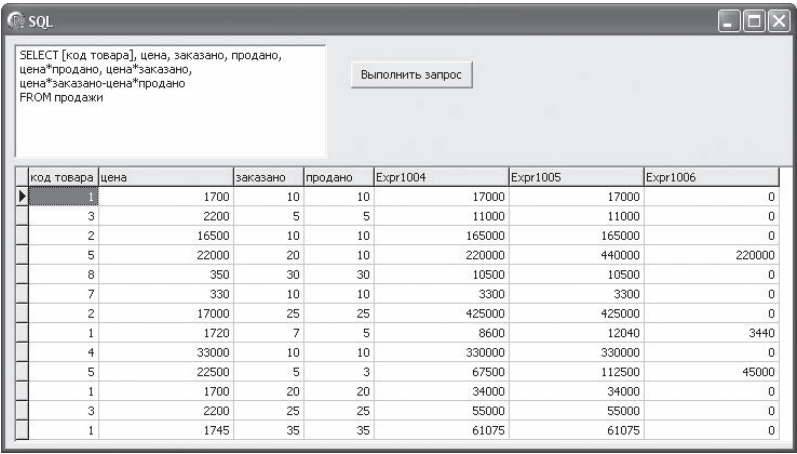
- ☐ оператор сложения «+»;
- ☐ оператор вычитания «-»;
- ☐ оператор умножения «*»;
- ☐ оператор деления «/».

Приоритет перечисленных операторов соответствует общепринятому: умножение и деление, затем сложение и вычитание. Порядком выполнения операторов можно управлять с помощью круглых скобок.

Рассмотрим пример использования вычисляемых полей. Для этого на основании данных таблицы Продажи вычислим для каждого товара сумму денег, полученных за проданный товар (произведение цены на количество проданного товара), и сумму, на которую заказано товаров (произведение цены на количество заказанного товара), а также разность между ними:

```
SELECT [код товара], цена, заказано, продано,  
цена*продано, цена*заказано,  
цена*заказано-цена*продано  
FROM продажи
```

Данный запрос содержит три вычисляемых поля. Результат его выполнения приведен на рис. 11.19.



код товара	цена	заказано	продано	Expr1004	Expr1005	Expr1006
1	1700	10	10	17000	17000	0
3	2200	5	5	11000	11000	0
2	16500	10	10	165000	165000	0
5	22000	20	10	220000	440000	220000
8	350	30	30	10500	10500	0
7	330	10	10	3300	3300	0
2	17000	25	25	425000	425000	0
1	1720	7	5	8600	12040	3440
4	33000	10	10	330000	330000	0
5	22500	5	3	67500	112500	45000
1	1700	20	20	34000	34000	0
3	2200	25	25	55000	55000	0
1	1745	35	35	61075	61075	0

Рис. 11.19. Результат выполнения запроса с вычисляемыми полями

Кроме арифметических операторов допускается использование ряда математических функций, например:

- ❑ ABS — вычисление абсолютного значения;
- ❑ ROUND — округление;
- ❑ SQR — извлечение квадратного корня;
- ❑ EXP — экспонента;
- ❑ LOG — натуральный логарифм;
- ❑ SIN, COS, TAN — тригонометрические функции.

Арифметические операторы и математические функции можно использовать как в списке полей после ключевого слова SELECT, так и в предложении, задающем условие выборки (WHERE).

ПРИМЕЧАНИЕ

Набор математических функций зависит от конкретной реализации языка SQL. Синтаксис одинаковых функций в разных реализациях также может различаться (например, функция вычисления квадратного корня может обозначаться либо SQR, либо SQRT).

Псевдонимы полей

В запросах SQL можно изменять имена полей. Задаваемые при этом новые имена называются *псевдонимами* (aliases). Их удобно применять при задании

в запросе вычисляемых полей. С помощью псевдонимов этим полям можно присваивать осмысленные имена. Псевдоним помещается после имени поля или после вычисляемого выражения через ключевое слово AS.

ВНИМАНИЕ

Переименование поля с помощью псевдонима действительно только в пределах конкретного запроса.

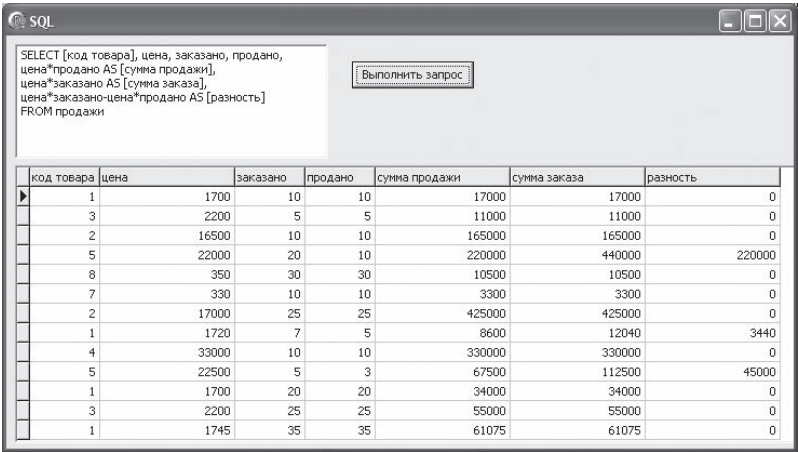


Рис. 11.20. Пример запроса с псевдонимами полей

В качестве примера воспользуемся предыдущим запросом, задав в нем псевдонимы для вычисляемых полей:

```
SELECT [код товара], цена, заказано, продано,  
цена*продано AS [сумма продажи],  
цена*заказано AS [сумма заказа],  
цена*заказано-цена*продано AS [разность]  
FROM продажи
```

Результаты выполнения данного запроса приведены на рис. 11.20.

ПРИМЕЧАНИЕ

Способы задания псевдонимов различаются в разных реализациях SQL. Часто псевдоним задается просто указанием нового имени через пробел после имени поля или вычисляемого выражения, без дополнительных ключевых слов.

Функции агрегирования

Функциями агрегирования называются функции, которые позволяют определить количество записей в таблице или количество значений в столбце таблицы, находят минимальное, максимальное и среднее значения для столбца таблицы, а также вычисляют сумму данных для столбца. Таким образом, агрегирующие функции обеспечивают получение некоторой обобщенной информации.

В SQL определены следующие стандартные функции агрегирования:

- ❑ COUNT — выполняет подсчет записей в таблице или подсчет ненулевых значений в столбце таблицы;
- ❑ SUM — возвращает сумму содержащихся в столбце значений;
- ❑ MIN — возвращает минимальное значение в столбце;
- ❑ MAX — возвращает максимальное значение в столбце;
- ❑ AVG — вычисляет среднее значение для содержащихся в столбце значений.

В качестве примера рассмотрим таблицу Продажи. Подсчитаем количество записей в поле Продано, минимальное и максимальное количество проданных товаров, общую сумму проданных товаров и среднее значение проданных товаров. Для этого зададим следующий запрос:

```
SELECT COUNT(продано) AS [всего записей],  
MIN(продано) AS minimum,  
MAX(продано) AS maximum,  
SUM(продано) AS [всего продано],  
AVG(продано) AS [среднее количество продаж]  
FROM продажи
```

Результат выполнения этого запроса показан на рис. 11.21.

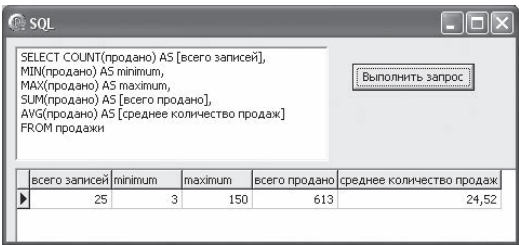


Рис. 11.21. Пример использования функций агрегирования

Со всеми функциями агрегирования можно использовать опцию DISTINCT. В этом случае выполняется обобщение информации только для различающихся строк.

ПРИМЕЧАНИЕ

Как правило, использование опции DISTINCT с агрегирующими функциями не имеет смысла, поскольку при подсчете обобщенных данных обычно следует учитывать все записи, а не только уникальные.

Группировка данных

Группировка данных — это объединение записей в соответствии со значениями некоторого заданного поля. Для группировки результатов выборки совместно с оператором SELECT используется предложение GROUP BY. Данное предложение должно следовать после предложения WHERE, но перед предложением ORDER BY. После ключевых слов GROUP BY указывается список полей, включенных в выборку с помощью оператора SELECT. Причем нужно обязательно указывать *все* отби-

раемые поля (за исключением полей, относящихся к агрегирующим функциям), хотя порядок их перечисления после предложения GROUP BY может не соответствовать порядку списка после слова SELECT.

Синтаксис оператора SELECT с предложением GROUP BY следующий:

```
SELECT field1, field2, ... , fieldN
FROM table1 {, table2, ... , tableN}
WHERE условие
GROUP BY field1, field2, ... , fieldN
ORDER BY field1 {ASC | DESC}
```

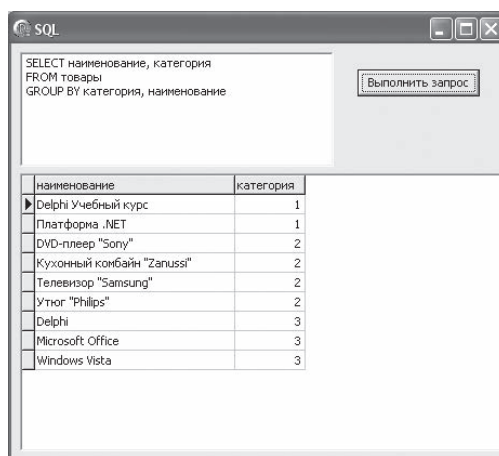
ПРИМЕЧАНИЕ

Применение предложения GROUP BY без дополнительных функций дает такой же результат, как и применение предложения упорядочения ORDER BY.

Например, если выбрать из таблицы Товары два поля — Наименование и Категория, а затем сгруппировать их с помощью запроса:

```
SELECT наименование, категория
FROM товары
GROUP BY категория, наименование
```

то результат выборки будет упорядочен по значению первого поля, указанного в предложении GROUP BY (рис. 11.22).



The screenshot shows a window titled "SQL" with a text area containing the following query:

```
SELECT наименование, категория
FROM товары
GROUP BY категория, наименование
```

To the right of the text area is a button labeled "Выполнить запрос". Below the text area is a table with two columns: "наименование" and "категория". The table contains the following data:

наименование	категория
Delphi Учебный курс	1
Платформа .NET	1
DVD-плеер "Sony"	2
Кухонный комбайн "Zanussi"	2
Телевизор "Samsung"	2
Утюг "Philips"	2
Delphi	3
Microsoft Office	3
Windows Vista	3

Рис. 11.22. Пример группировки данных

Если в запросе выбрать только одно поле и выполнить для него группировку, то результирующая выборка не будет содержать дублирующих друг друга записей. Например, если выполнить запрос, аналогичный предыдущему (рис. 11.23), но выбрать только поле Категория:

```
SELECT категория
FROM товары
GROUP BY категория
```

то выборка будет содержать только три записи (рис. 11.23).

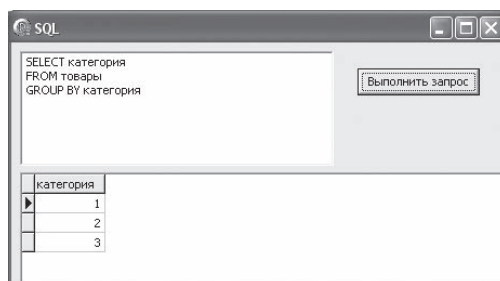


Рис. 11.23. Результат группировки одного поля

В этом случае группировка дает такой же результат, как применение оператора `SELECT` с опцией `DISTINCT` и предложением `ORDER BY`.

Поскольку применение одного предложения `GROUP BY` не дает никакого нового результата, то совместно с ним, как правило, используются функции агрегирования. В этом случае они применяются для вычисления итоговых значений по отдельным группам данных.

Например, чтобы подсчитать количество покупок товаров, сделанных каждым из клиентов, используется следующий запрос:

```
SELECT [код клиента],  
SUM(продано) AS [количество покупок]  
FROM продажи  
GROUP BY [код клиента]
```

Результат выполнения такого запроса приведен на рис. 11.24.

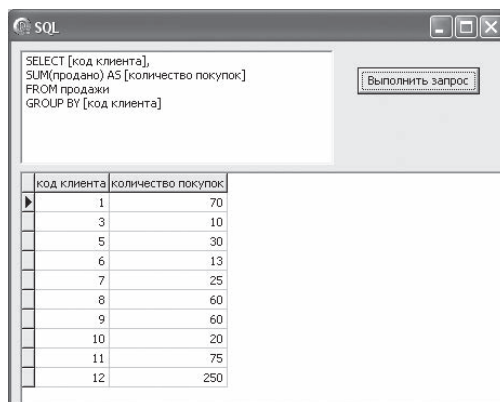
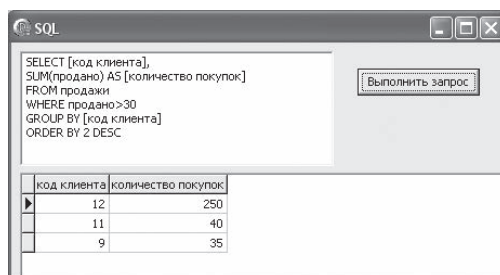


Рис. 11.24. Использование агрегирующих функций при группировке данных

Результаты группировки можно упорядочить с помощью ключевого слова `ORDER BY`, а в операторе `SELECT`, содержащем предложение группировки, можно использовать предложение `WHERE`. Для иллюстрации этой возможности модифицируем предыдущий запрос следующим образом: выберем только тех клиентов, которые сделали за один раз более 30 покупок, и упорядочим результаты выборки в порядке убывания:

```
SELECT [код клиента],  
SUM(продано) AS [количество покупок]  
FROM продажи  
WHERE продано>30  
GROUP BY [код клиента]  
ORDER BY 2 DESC
```

Результат выполнения данного запроса изображен на рис. 11.25.



The screenshot shows a window titled "SQL" with a text area containing the following query:

```
SELECT [код клиента],  
SUM(продано) AS [количество покупок]  
FROM продажи  
WHERE продано>30  
GROUP BY [код клиента]  
ORDER BY 2 DESC
```

To the right of the text area is a button labeled "Выполнить запрос". Below the text area is a table with the following data:

код клиента	количество покупок
12	250
11	40
9	35

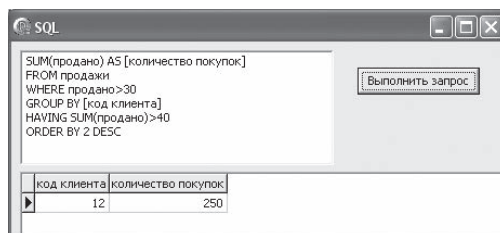
Рис. 11.25. Пример сортировки результатов выборки с помощью предложения GROUP BY

Для задания ограничений на создаваемые группы совместно с ключевым словом GROUP BY может использоваться предложение HAVING. Оно должно следовать после GROUP BY, но до предложения ORDER BY (если оно присутствует в запросе).

В предыдущем примере в качестве условия было задано количество покупок за один раз. Если мы хотим установить ограничение на общее количество покупок, то необходимо применить предложение HAVING:

```
SELECT [код клиента],  
SUM(продано) AS [количество покупок]  
FROM продажи  
WHERE продано>30  
GROUP BY [код клиента]  
HAVING SUM(продано)>40  
ORDER BY 2 DESC
```

Результат выполнения запроса приведен на рис. 11.26.



The screenshot shows a window titled "SQL" with a text area containing the following query:

```
SUM(продано) AS [количество покупок]  
FROM продажи  
WHERE продано>30  
GROUP BY [код клиента]  
HAVING SUM(продано)>40  
ORDER BY 2 DESC
```

To the right of the text area is a button labeled "Выполнить запрос". Below the text area is a table with the following data:

код клиента	количество покупок
12	250

Рис. 11.26. Пример использования предложения HAVING

ПРИМЕЧАНИЕ

В предложении HAVING необязательно использовать только те поля, которые заданы в списке оператора SELECT.

Модифицируем рассмотренный ранее пример таким образом, чтобы ограничение было наложено не на количество купленных товаров, а на их стоимость:

```
SELECT [код клиента],  
SUM(продано) AS [количество покупок]  
FROM продажи  
GROUP BY [код клиента]  
HAVING SUM(продано*цена)>250000  
ORDER BY 2 DESC
```

Данный запрос отбирает клиентов, купивших товаров более чем на 250 000, и отображает количество сделанных ими покупок (рис. 11.27).

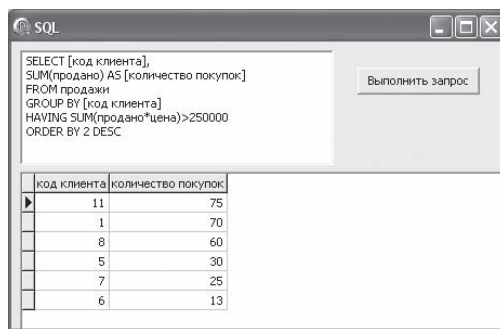


Рис. 11.27. Пример запроса с предложением HAVING

ПРИМЕЧАНИЕ

В предложении GROUP BY, в отличие от предложения ORDER BY, нельзя вместо имен выбранных полей использовать их порядковые номера в списке оператора SELECT.

Выборка данных из нескольких таблиц

Как правило, информация, хранящаяся в базе данных, содержится в нескольких связанных между собой таблицах. Язык SQL позволяет создавать запросы, извлекающие данные из нескольких таблиц. При этом выполняется операция *соединения*, состоящая в объединении нескольких таблиц с целью поиска в них запрошенных данных.

Существует несколько способов соединения таблиц. Наиболее часто встречаются следующие:

- ☐ соединение равенства;
- ☐ соединение неравенства;
- ☐ внешние соединения.

Для задания вида соединения используется предложение WHERE, в котором вид соединения указывается с помощью операторов сравнения или логических операторов.

Соединение равенства

Данное соединение является наиболее часто используемым. Соединение равенства обычно производится по общему для нескольких таблиц полю (которое, как правило, является первичным ключом).

Синтаксис оператора выборки для этого способа соединения таблиц будет следующим:

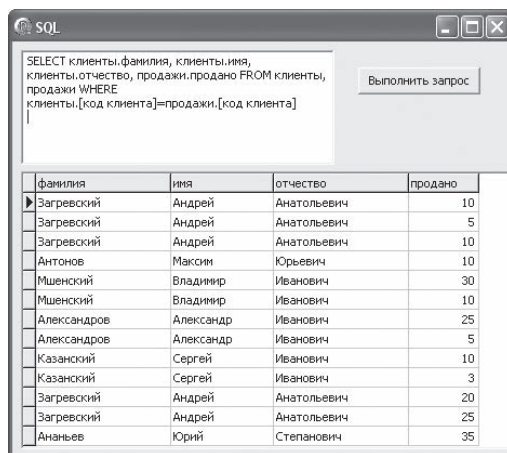
```
SELECT table1.field1, table2.field2 {, ... , tableN.fieldN}
FROM table1, table2 {, ... , tableN}
WHERE table1.common_field1 = table2.common_field1
{AND table1.common_field2 = table2.common_field2}
```

При формировании запроса на выборку из нескольких таблиц в списке полей после слова SELECT перед именем поля обычно указывается имя таблицы, к которой это поле относится. Такое действие называется *квалификацией* полей запроса. Квалификация обязательна только для полей, имеющих одинаковые имена в разных таблицах, из которых производится выборка.

Рассмотрим пример выборки из двух таблиц с использованием соединения равенства. Выберем из таблицы Клиенты поля, содержащие сведения об именах клиентов, а из таблицы Продажи — поля, в которых содержатся сведения о покупках, сделанных клиентами. Для связывания таблиц воспользуемся общим для обеих таблиц полем Код клиента:

```
SELECT клиенты.фамилия, клиенты.имя,
клиенты.отчество, продажи.продано
FROM клиенты, продажи
WHERE клиенты.[код клиента]=продажи.[код клиента]
```

Результат выполнения данного запроса приведен на рис. 11.28.



фамилия	имя	отчество	продано
Загrevский	Андрей	Анатолевич	10
Загrevский	Андрей	Анатолевич	5
Загrevский	Андрей	Анатолевич	10
Антонов	Максим	Юрьевич	10
Мшенский	Владимир	Иванович	30
Мшенский	Владимир	Иванович	10
Александров	Александр	Иванович	25
Александров	Александр	Иванович	5
Казанский	Сергей	Иванович	10
Казанский	Сергей	Иванович	3
Загrevский	Андрей	Анатолевич	20
Загrevский	Андрей	Анатолевич	25
Аняньев	Юрий	Степанович	35

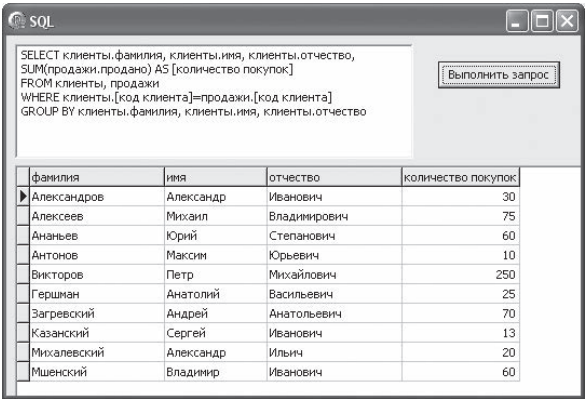
Рис. 11.28. Использование соединения равенства при выборке из двух таблиц

При связывании таблиц можно использовать предложение группировки. Изменим рассмотренный запрос таким образом, чтобы результаты были сгруппированы по полям Фамилия, Имя, Отчество и для каждого клиента выводилось суммарное количество покупок:

```
SELECT клиенты.фамилия, клиенты.имя, клиенты.отчество,
SUM(продажи.продано) AS [количество покупок]
FROM клиенты, продажи
```

```
WHERE клиенты.[код клиента]=продажи.[код клиента]
GROUP BY клиенты.фамилия, клиенты.имя, клиенты.отчество
```

Результаты, возвращаемые этим запросом, приведены на рис. 11.29.



The screenshot shows an SQL window with the following query:

```
SELECT клиенты.фамилия, клиенты.имя, клиенты.отчество,
SUM(продажи.продано) AS [количество покупок]
FROM клиенты, продажи
WHERE клиенты.[код клиента]=продажи.[код клиента]
GROUP BY клиенты.фамилия, клиенты.имя, клиенты.отчество
```

The results are displayed in a table with 4 columns: фамилия, имя, отчество, and количество покупок.

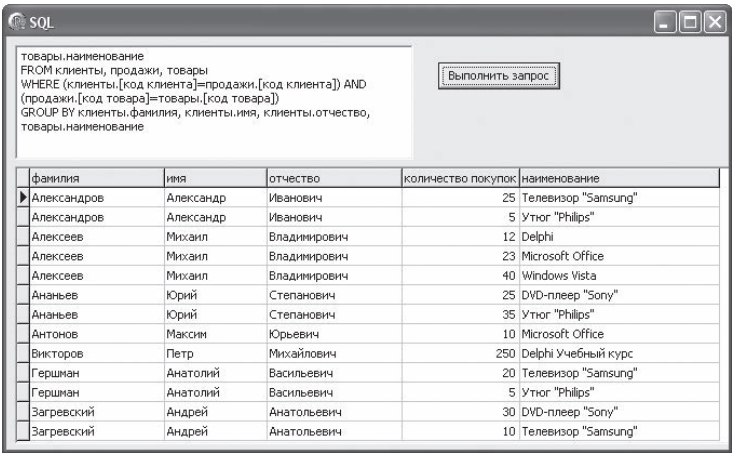
фамилия	имя	отчество	количество покупок
Александров	Александр	Иванович	30
Алексеев	Михаил	Владимирович	75
Ананиев	Юрий	Степанович	60
Антонов	Максим	Юревич	10
Викторов	Петр	Михайлович	250
Гершман	Анатолий	Васильевич	25
Загребский	Андрей	Анатолевич	70
Казанский	Сергей	Иванович	13
Михалевский	Александр	Ильич	20
Мшенский	Владимир	Иванович	60

Рис. 11.29. Пример группировки выборки из двух таблиц

Выборка из трех таблиц проводится аналогичным образом, только в предложении WHERE необходимо указать условие связи с третьей таблицей. Для примера дополним предыдущий запрос (рис. 11.29) таким образом, чтобы в выборку была включена информация о наименовании товара из таблицы Товары:

```
SELECT клиенты.фамилия, клиенты.имя, клиенты.отчество,
SUM(продажи.продано) AS [количество покупок],
товары.наименование
FROM клиенты, продажи, товары
WHERE (клиенты.[код клиента]=продажи.[код клиента]) AND
(продажи.[код товара]=товары.[код товара])
GROUP BY клиенты.фамилия, клиенты.имя, клиенты.отчество,
товары.наименование
```

Результаты выполнения данного запроса показаны на рис. 11.30.



The screenshot shows an SQL window with the following query:

```
товары.наименование
FROM клиенты, продажи, товары
WHERE (клиенты.[код клиента]=продажи.[код клиента]) AND
(продажи.[код товара]=товары.[код товара])
GROUP BY клиенты.фамилия, клиенты.имя, клиенты.отчество,
товары.наименование
```

The results are displayed in a table with 5 columns: фамилия, имя, отчество, количество покупок, and наименование.

фамилия	имя	отчество	количество покупок	наименование
Александров	Александр	Иванович	25	Телевизор "Samsung"
Александров	Александр	Иванович	5	Утюг "Philips"
Алексеев	Михаил	Владимирович	12	Delphi
Алексеев	Михаил	Владимирович	23	Microsoft Office
Алексеев	Михаил	Владимирович	40	Windows Vista
Ананиев	Юрий	Степанович	25	DVD-плеер "Sony"
Ананиев	Юрий	Степанович	35	Утюг "Philips"
Антонов	Максим	Юревич	10	Microsoft Office
Викторов	Петр	Михайлович	250	Delphi Учебный курс
Гершман	Анатолий	Васильевич	20	Телевизор "Samsung"
Гершман	Анатолий	Васильевич	5	Утюг "Philips"
Загребский	Андрей	Анатолевич	30	DVD-плеер "Sony"
Загребский	Андрей	Анатолевич	10	Телевизор "Samsung"

Рис. 11.30. Пример выборки из трех таблиц

Соединение неравенства

В случае применения соединения неравенства информация из двух таблиц объединяется таким образом, чтобы значения в заданном поле одной таблицы не совпадали со значениями соответствующего ему поля в другой таблице.

Синтаксис запроса при соединении неравенства аналогичен предыдущему случаю, только вместо оператора «=» в предложении WHERE используются операторы «<>», «<», «>» и т. п.

```
SELECT table1.field1, table2.field2 {, ... , tableN.fieldN}
FROM table1, table2 {, ... , tableN}
WHERE table1.common_field1 <> table2.common_field1
{AND table1.common_field2 > table2.common_field2}
```

Соединения неравенства используются довольно редко. В частности, для базы данных, используемой нами в качестве практической модели, довольно трудно привести пример такого соединения, имеющего практическое применение.

Внешние соединения

При использовании внешнего соединения результат запроса будет содержать все записи одной из таблиц, даже в том случае если в связанной с ней таблице отсутствуют совпадающие значения. Этот тип соединения реализуется с помощью оператора OUTER JOIN.

Внешние соединения подразделяются на три группы:

- ❑ левое внешнее соединение, LEFT OUTER JOIN, — выборка будет содержать все записи таблицы, имя которой указано слева от оператора OUTER JOIN;
- ❑ правое внешнее соединение, RIGHT OUTER JOIN, — выборка будет содержать все записи таблицы, имя которой указано справа от оператора OUTER JOIN;
- ❑ полное внешнее соединение, FULL OUTER JOIN, — в выборку включаются все записи из правой и левой таблицы.

Для внешнего соединения условие соединения указывается не с помощью предложения WHERE, а входит в оператор OUTER JOIN после ключевого слова ON:

```
SELECT table1.field1, table2.field2 {, ... , tableN.fieldN}
FROM table1
    LEFT | RIGHT | FULL {OUTER} JOIN table2
    ON условие
    {LEFT | RIGHT | FULL {OUTER} JOIN table3
    ON условие}
```

Рассмотрим следующий пример. Выберем из таблицы **Товары** список товаров, а из таблицы **Продажи** — суммарное количество проданных товаров:

```
SELECT товары.[наименование],
SUM(продажи.[продано]) AS [всего продано]
FROM товары LEFT OUTER JOIN продажи
ON товары.[код товара]=продажи.[код товара]
GROUP BY товары.[наименование]
```

Так как таблица **Товары** указана слева от оператора LEFT JOIN, то результирующая выборка будет содержать полный список товаров, включая даже те, которые ни разу не проданы (рис. 11.31).

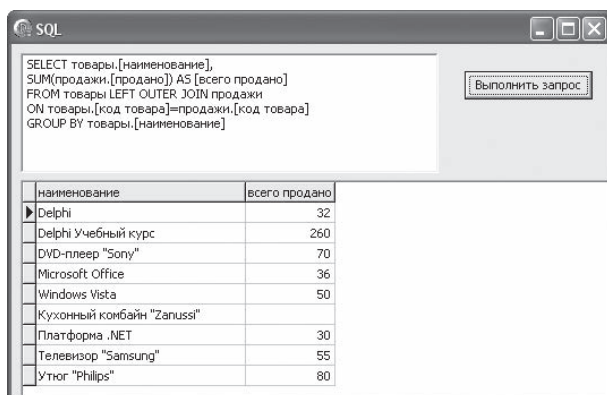


Рис. 11.31. Результат внешнего соединения двух таблиц

Подзапросы

Подзапрос представляет собой запрос, помещенный внутри другого запроса. Подзапросы применяются для получения данных, которые затем используются другим запросом.

Запрос, содержащий подзапрос, называется *сложным*. В процессе его выполнения сначала выполняется подзапрос, а затем — основной запрос. При создании сложного запроса необходимо следовать следующему набору правил:

- ☐ подзапросы должны заключаться в круглые скобки;
- ☐ предложение ORDER BY может быть использовано только в основном запросе;
- ☐ подзапросы, возвращающие более одной записи, могут использоваться только с многозначными операторами;
- ☐ в основном запросе нельзя использовать оператор BETWEEN.

Ниже приведен синтаксис оператора SELECT с подзапросом:

```
SELECT { * | ALL | DISTINCT field1, field2, ... , fieldN }
FROM table1 { , table2, ... , tableN }
WHERE условие (SELECT field1 { , field2, ... , fieldN }
FROM table1 { , table2, ... , tableN }
WHERE условие)
```

Для иллюстрации технологии использования подзапроса воспользуемся следующим примером. Выберем из таблицы Продажи информацию о продажах товара с наименованием «Delphi»:

```
SELECT [код клиента], заказано, продано, цена
FROM продажи
WHERE [код товара]=(SELECT [код товара]
FROM товары
WHERE наименование='Delphi')
```

Поскольку в таблице Продажи не содержится наименования товара, то с помощью подзапроса мы обращаемся к таблице Товары и определяем код товара заданного наименования. Затем в основном запросе выбираем интересующие

нас поля из таблицы Продажи, в которых код товара совпадает с тем, который получен в результате выполнения подзапроса. Результат, полученный при выполнении приведенного запроса, показан на рис. 11.32.

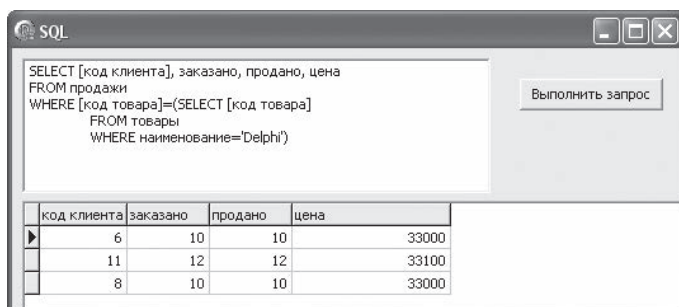


Рис. 11.32. Пример использования подзапроса

ПРИМЕЧАНИЕ

В подзапросе, так же как и в основном запросе, можно использовать подзапросы. Максимальный уровень вложенности подзапросов определяется конкретной реализацией SQL.

Объединение запросов

Язык SQL позволяет объединять несколько запросов с помощью специальных операторов. Запросы, включающие в себя несколько операторов SELECT, принято называть *составными*.

Составные запросы формируют один набор данных на основе результатов, полученных при выполнении каждого отдельного запроса, входящего в объединение. Во многих случаях составные запросы целесообразно использовать вместо простых запросов со сложным условием выборки. Это связано с тем, что разбиение сложного условия на несколько более простых запросов делает текст запроса более понятным. Как правило, проще написать составной запрос, чем аналогичный простой запрос со сложным условием отбора данных.

Для объединения запросов наиболее часто используются операторы UNION и UNION ALL (предусмотренные стандартом ANSI).

ПРИМЕЧАНИЕ

В стандарте ANSI определены также и другие операторы объединения: EXCEPT и INTERSECT, которые расширяют возможности составных запросов.

При объединении запросов, независимо от типа используемых операторов объединения, необходимо следовать следующим правилам:

- ❑ каждый из запросов, входящих в объединение, должен возвращать одинаковое количество полей (в том числе и вычисляемых);
- ❑ типы полей, возвращаемых в результате выполнения каждого запроса, должны совпадать.

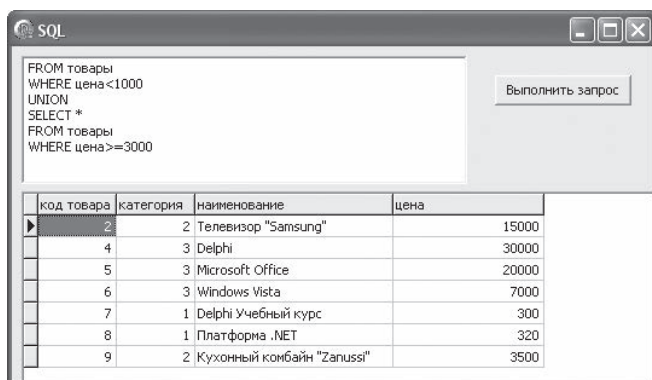
Оператор UNION

При использовании оператора UNION результаты выполнения отдельных запросов объединяются. При этом дублирующие друг друга записи исключаются из результирующего набора данных.

Для примера выберем из таблицы Товары список товаров, цена которых меньше 1000 или больше 3000. Такую выборку можно сделать, используя объединение логических операторов в предложении WHERE с помощью оператора OR либо путем объединения запросов:

```
SELECT *  
FROM товары  
WHERE цена<1000  
UNION  
SELECT *  
FROM товары  
WHERE цена>=3000
```

Здесь первый запрос отбирает товары, цена которых меньше 1000, а второй — товары, цена которых превышает 3000. С помощью оператора UNION результаты, возвращаемые отдельными запросами, объединяются в один набор данных (рис. 11.33).



The screenshot shows a window titled "SQL" with a text area containing the following query:

```
FROM товары  
WHERE цена<1000  
UNION  
SELECT *  
FROM товары  
WHERE цена>=3000
```

To the right of the text area is a button labeled "Выполнить запрос". Below the text area is a table with the following data:

	код товара	категория	наименование	цена
▶	2	2	Телевизор "Samsung"	15000
	4	3	Delphi	30000
	5	3	Microsoft Office	20000
	6	3	Windows Vista	7000
	7	1	Delphi Учебный курс	300
	8	1	Платформа .NET	320
	9	2	Кухонный комбайн "Zanussi"	3500

Рис. 11.33. Результат объединения запросов с помощью оператора UNION

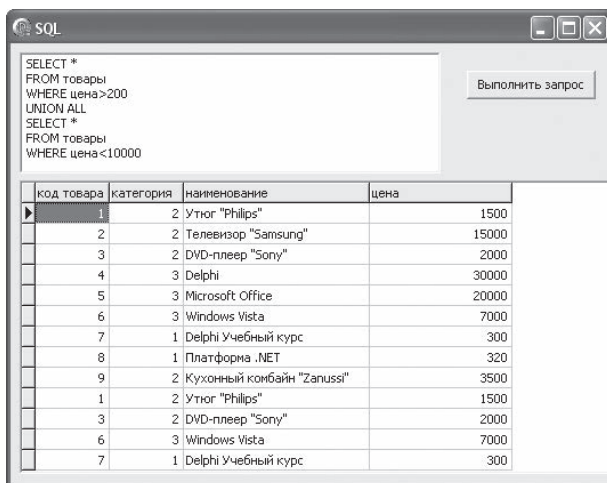
Оператор UNION ALL

Данный оператор аналогичен оператору UNION, за исключением того, что в результирующую выборку включаются дублирующие записи. Если в предыдущем примере (см. рис. 11.33) заменить UNION на UNION ALL, то результат не изменится, так как в нем не содержится дублирующих записей. Однако если задать запрос таким образом, что одни и те же записи попадут в результаты обоих запросов, входящих в объединение, то в результирующей выборке они также будут присутствовать два раза. Например, при выполнении запроса:

```
SELECT *  
FROM товары  
WHERE цена>200
```

```
UNION ALL  
SELECT *  
FROM товары  
WHERE цена<10000
```

результат будет содержать 16 записей (рис. 11.34), хотя в таблице Товары содержится всего лишь 9 записей. Это объясняется тем, что часть записей выбрана и в первом, и во втором запросе (это те товары, цена которых больше 200, но меньше 10 000), поэтому в результирующей выборке содержится несколько одинаковых записей.



код товара	категория	наименование	цена
1	2	Утюг "Philips"	1500
2	2	Телевизор "Samsung"	15000
3	2	DVD-плеер "Sony"	2000
4	3	Delphi	30000
5	3	Microsoft Office	20000
6	3	Windows Vista	7000
7	1	Delphi Учебный курс	300
8	1	Платформа .NET	320
9	2	Кухонный комбайн "Zanussi"	3500
1	2	Утюг "Philips"	1500
3	2	DVD-плеер "Sony"	2000
6	3	Windows Vista	7000
7	1	Delphi Учебный курс	300

Рис. 11.34. Объединение запросов с помощью оператора UNION ALL

Упорядочение и группировка данных в составных запросах

В составном запросе для упорядочения данных допускается использование предложения ORDER BY. Независимо от того, сколько запросов входит в объединение, можно использовать только одно предложение ORDER BY. Для указания полей, по которым производится сортировка, в этом предложении допускается использование как имен полей, так и их порядковых номеров в списке оператора SELECT.

В отличие от ORDER BY, предложение GROUP BY можно применять в каждом из запросов, входящих в объединение. Вместе с GROUP BY допускается применение оператора HAVING. Предложение GROUP BY можно применять и для группировки результатов выполнения составного запроса.

Работа с представлениями данных

Представление (view) — это предопределенный запрос, который хранится в базе данных. Представление можно рассматривать как виртуальную таблицу, которая формируется из одной или нескольких реальных таблиц базы данных (и/или ранее созданных представлений). Работа с представлением после его

создания полностью аналогична работе с таблицей. Представления обычно используются в двух случаях:

- ❑ для объединения данных, хранящихся в нескольких таблицах (разбиение на таблицы обычно производится при нормализации базы данных), с целью их представления в удобном для просмотра и редактирования виде;
- ❑ для разграничения доступа к информации — с помощью представлений можно разрешить пользователю доступ только к части информации, хранящейся в таблице базы данных.

Создание представлений

Для создания представления используется оператор `CREATE VIEW`. Поскольку представление всегда создается на основе таблиц и/или ранее созданных представлений, то оператор `CREATE VIEW` отличается от оператора создания таблицы — вместо указания имен и типов полей данных оператор должен содержать запрос:

```
CREATE VIEW имя_представления AS  
SELECT ...
```

Чтобы рассмотреть пример создания представления, следует несколько модифицировать программу, которую мы используем для изучения SQL. Дело в том, что оператор `CREATE VIEW` не возвращает никаких данных. Поэтому для его выполнения следует воспользоваться не методом `Open`, который мы использовали для выполнения оператора `SELECT`, а методом `ExecSQL`. Чтобы не усложнять задачу, просто добавим на форму еще одну кнопку (назовем ее `Exec SQL`), при щелчке на которой будет вызываться этот метод. Таким образом, для выполнения оператора `SELECT` следует щелкнуть на кнопке **Выполнить запрос**, а для выполнении операторов, не возвращающих данных, — на кнопке `Exec SQL`. Обработчик нажатия на кнопку `Exec SQL` приведен в листинге 11.2.

Листинг 11.2. Обработчик события `OnClick` кнопки `Exec SQL`

```
procedure TfrmMain.btnExecSQLClick(Sender: TObject);  
begin  
    if ADOQuery1.Active then ADOQuery1.Close;  
    ADOQuery1.SQL.Clear;  
    ADOQuery1.SQL.Assign(memSQL.Lines);  
    ADOQuery1.ExecSQL;  
end;
```

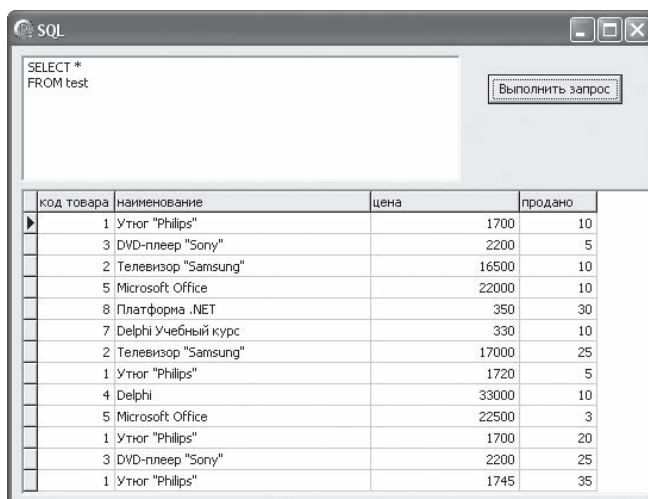
Теперь в качестве примера создадим представление на основе таблиц **Товары** и **Продажи**. Из первой таблицы выберем поля **Код товара** и **Наименование**, из второй — **Цена** и **Продано**. Для связи таблиц будем использовать соединение равенства. Запрос, создающий представление с именем `Test`, имеет следующий вид:

```
CREATE VIEW test AS  
SELECT товары.[код товара], товары.наименование,  
продажи.цена, продажи.продано  
FROM товары, продажи  
WHERE товары.[код товара]=продажи.[код товара]
```


После создания представления с ним можно работать как с обычной таблицей. Например, можно вызвать следующий запрос:

```
SELECT *
FROM test
```

результат выполнения которого (рис. 11.35) аналогичен результату, который возвратил бы запрос, следующий после ключевого слова AS в операторе CREATE VIEW.



код товара	наименование	цена	продано	
1	Утюг "Philips"	1700	10	10
3	DVD-плеер "Sony"	2200	5	5
2	Телевизор "Samsung"	16500	10	10
5	Microsoft Office	22000	10	10
8	Платформа .NET	350	30	30
7	Delphi Учебный курс	330	10	10
2	Телевизор "Samsung"	17000	25	25
1	Утюг "Philips"	1720	5	5
4	Delphi	33000	10	10
5	Microsoft Office	22500	3	3
1	Утюг "Philips"	1700	20	20
3	DVD-плеер "Sony"	2200	25	25
1	Утюг "Philips"	1745	35	35

Рис. 11.35. Результат выборки всех записей из представления Test

При создании представлений допускается использование вычисляемых полей. Например, можно создать представление, подобное рассмотренному выше, но с вычисляемым полем, в котором будет содержаться сумма закупленного товара:

```
CREATE VIEW test2 AS
SELECT товары.[код товара], товары.наименование,
       продажи.цена, продажи.продано,
       продажи.цена*продажи.продано AS [сумма продаж]
FROM товары, продажи
WHERE товары.[код товара]=продажи.[код товара]
```

Здесь для вычисляемого поля задан псевдоним Сумма продаж. Результат выборки всех записей из такого представления приведен на рис. 11.36.

Удаление представлений

Для удаления представлений используется оператор DROP VIEW, синтаксис которого представлен ниже:

```
DROP VIEW view_name
```

Команды, удаляющие созданные нами представления, имеют следующий вид:

```
DROP VIEW Test
DROP VIEW Test2
```

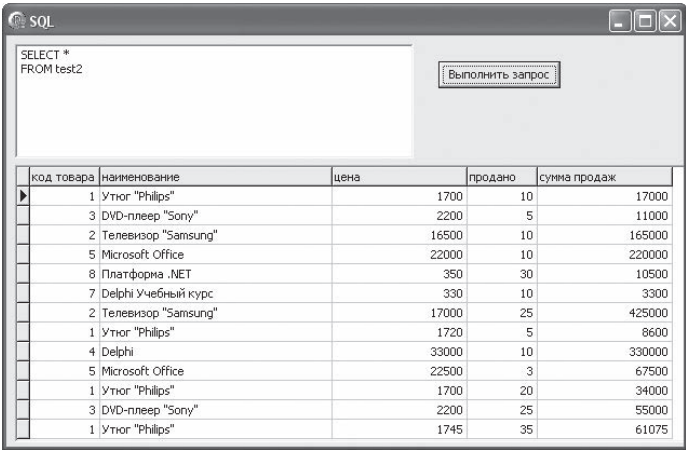


Рис. 11.36. Представление с вычисляемым полем

Использование параметров в SQL-запросах

При задании SQL-запроса можно использовать параметры — переменные, включаемые в оператор SQL, значения которых определяются во время выполнения программы. Использование параметров в значительной степени повышает гибкость SQL-запросов, обеспечивая возможность запрашивать у пользователя численные значения критериев выборки данных.

Параметры задаются в тексте SQL-запроса. Для определения параметра перед его именем указывается символ «:», например:

```
SELECT *
FROM table name
WHERE field1<:PARAM1
```

В данном запросе задан один параметр с именем PARAM1.

После ввода текста запроса в свойство SQL автоматически производится заполнение массива в свойстве Params. Одновременно значение свойства Params.Count устанавливается равным количеству заданных в запросе параметров. Последовательность заполнения массива Params соответствует порядку следования параметров в тексте запроса.

Свойства определенных в SQL-запросе параметров доступны для редактирования как во время разработки, так и во время выполнения программы:

- для редактирования свойств параметров во время разработки программы используется специальный редактор, который вызывается щелчком на кнопке с многоточием в поле ввода свойства Params в инспекторе объектов;

- ❑ для задания значения параметра во время разработки программы вначале необходимо определить его тип с помощью свойства `DataType` класса `TParam`;
- ❑ для доступа к свойствам параметров SQL-запроса во время выполнения программы можно либо воспользоваться свойством `Items` класса `TParams`, либо методом `ParamByName` этого же класса. Свойство `Items` предоставляет доступ к объектам параметров по их порядковым номерам, что не очень удобно. Обычно гораздо проще обращаться к параметрам по их именам с помощью метода `ParamByName`, возвращающего объект параметра, имя которого задается в качестве аргумента при вызове данного метода;
- ❑ в зависимости от значения свойства `ParamCheck` при изменении текста запроса во время выполнения программы список параметров в свойстве `Params` может либо автоматически обновляться (`ParamCheck = true`), либо оставаться прежним (`ParamCheck = false`).

Для иллюстрации использования концепции параметров на практике модифицируем программу, которую мы использовали ранее при изучении SQL. С этой целью добавим на форму два компонента `TEdit`, которым присвоим имена `edtParam1` и `edtParam2`. Эти компоненты обеспечивают возможность изменения значений параметров во время выполнения программы. Изменим код метода-обработчика события `OnClick` кнопки **Выполнить запрос**, как показано в листинге 11.3.

Листинг 11.3. Обработчик события `OnClick` кнопки «Выполнить запрос»

```
procedure TfrmMain.btnOpenQueryClick(Sender: TObject);
begin
    if ADOQuery1.Active then ADOQuery1.Close;
    ADOQuery1.SQL.Clear;
    ADOQuery1.SQL.Assign(memSQL.Lines);
    if ADOQuery1.Params.Count>0
    then begin
        ADOQuery1.Params.ParamByName('P_P1').Value:=
            StrToInt(edtParam1.Text);
        ADOQuery1.Params.ParamByName('P_P2').Value:=
            StrToInt(edtParam2.Text);
    end;
    ADOQuery1.Open;
end;
```

Теперь, если в запросе имеются параметры, то их значения будут считываться из полей ввода `edtParam1` и `edtParam2`.

После запуска программы зададим следующий запрос:

```
SELECT *
FROM товары
WHERE (цена>:P_P1) AND (цена<:P_P2),
```

в котором определены два параметра: `P_P1` и `P_P2`. Зададим их значения с помощью элементов `edtParam1` и `edtParam2` — в первом поле ввода укажем 200 (значение первого параметра), во втором — 2000 (значение второго параметра). Если теперь щелкнуть на кнопке **Выполнить запрос**, то результатом будет вывод списка товаров, цена которых больше 200 и меньше 2000 (рис. 11.37).

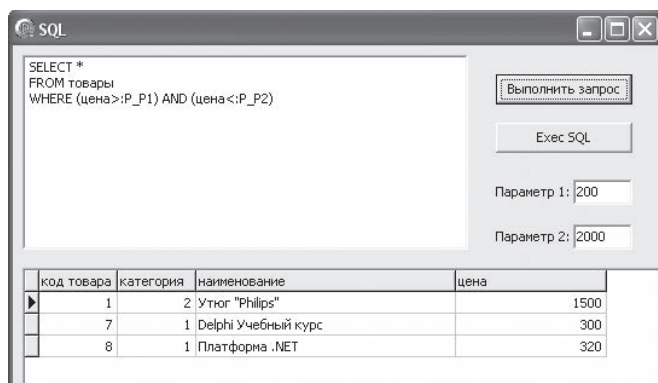


Рис. 11.37. Результат выполнения запроса с параметрами

Значения параметров могут передаваться из другого набора данных. Для этого в свойстве `DataSource` задается имя источника данных, связанного с набором данных, значения полей которого передаются в параметры. Имена параметров в этом случае должны совпадать с именами полей набора данных, заданного в свойстве `DataSource`. При перемещении по исходному набору данных текущие значения полей автоматически передаются в запрос. Этот механизм можно использовать для организации связи между таблицами базы данных, обращение к которым производится через SQL-запросы.

Рассмотрим простой пример организации связи между таблицами. Напишем программу, в которой осуществляется связь между таблицами `Товары` и `Продажи` по общему для обеих таблиц полю `Код товара`. Таблица `Товары` будет главной, `Продажи` — подчиненной.

1. Создайте новое приложение с помощью команды `File ► New ► New VCL Application — Delphi for Win32`.
2. Поместите на форму по два экземпляра следующих компонентов: `TADOQuery`, `TDataSource` и `TDBGrid`. Затем переименуйте шесть размещенных на форме компонентов следующим образом: `qryMaster`, `qryDetail`, `dsMaster`, `dsDetail`, `dbgMaster`, `dbgDetail`.
3. Для каждого компонента доступа к данным (`qryMaster` и `qryDetail`) установите связь с базой данных `sales.mdb`.
4. Свяжите источник данных `dsMaster` с компонентом доступа к данным `qryMaster`, а источник данных `dsDetail` — с компонентом `qryDetail`.
5. Свяжите элементы отображения данных `DBGMaster` и `DBGDetail` с источниками данных `dsMaster` и `dsDetail` соответственно.
6. Задайте в свойстве `SQL` компонента `qryMaster` следующий запрос:


```
SELECT *
FROM товары
```
7. Задайте в свойстве `SQL` компонента `qryDetail` запрос, приведенный ниже:

```
SELECT *
FROM продажи
WHERE [код товара]="код товара"
```

ВНИМАНИЕ

Если в качестве параметра используется имя поля таблицы, содержащее пробелы, то его необходимо заключать в кавычки.

- Выберите на форме компонент `qryDetail` и щелкните в инспекторе объектов на кнопке с многоточием в поле ввода свойства `Params`. В открывшемся окне редактора параметров выберите единственный параметр **Код товара** и задайте ему тип `Integer`. (Свойство `DataType` в инспекторе объектов.)

- Задайте обработчики событий формы `OnShow` и `OnClose`. В первом вызовите метод `Open` для обеих таблиц:

```
procedure TForm1.FormShow(Sender: TObject);
begin
    qryMaster.Open;
    qryDetail.Open;
end;
```

Во втором обработчике вызовите для двух таблиц метод `Close`:

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    qryMaster.Close;
    qryDetail.Close;
end;
```

- Откомпилируйте и запустите программу. В окне программы будут содержаться две таблицы, одна из которых показывает всю информацию из таблицы **Товары**, а во второй отображаются записи из таблицы **Продажи**, для которых значение поля **Код товара** равно значению поля **Код товара** выбранной записи в таблице **Товары** (рис. 11.38).

код товара	категория	наименование	цена
1	2	Утюг "Philips"	1500
2	2	Телевизор "Samsung"	15000
3	2	DVD-плеер "Sony"	2000
4	3	Delphi	30000
5	3	Microsoft Office	20000
6	3	Windows Vista	7000
7	1	Delphi Учебный курс	300
8	1	Платформа .NET	320
9	2	Кухонный комбайн "Zanussi"	3500

код продажи	код товара	код клиента	дата заказа	заказано	дата продажи	продано
1	1	1	15.01.2008	10	23.01.2008	
8	1	5	15.01.2008	7	25.01.2008	
11	1	1	12.02.2008	20	23.02.2008	
13	1	9	12.02.2008	35	22.02.2008	
18	1	10	10.01.2008	5	20.01.2008	
22	1	7	01.02.2008	5	12.02.2008	

Рис. 11.38. Пример связывания таблиц с использованием SQL-запросов

Созданная связь между таблицами соответствует отношению «один-ко-многим». Отношение «многие-ко-многим» создается точно так же. В этом случае подчиненная таблица аналогичным образом связывается с еще одной таблицей в качестве главной. Например, в нашем случае можно было бы связать таблицу Продажи с таблицей Клиенты по общему для них полю Код клиента.

Часть IV

Компоновка приложения и управление проектом

Глава 12

Система меню и панель инструментов приложения

Меню является одним из наиболее важных элементов программного интерфейса. С его помощью обеспечивается доступ пользователя практически ко всем функциям приложения. Именно поэтому такой элемент интерфейса нашел в программах широкое распространение.

Панели инструментов также впервые были использованы еще во времена DOS, но особого распространения не имели. Тогда их область применения в основном ограничивалась графическими редакторами. С развитием технологий программирования ситуация изменилась. В настоящее время панели инструментов присутствуют практически во всех программах.

Наличие в программах меню и панелей инструментов является своего рода стандартом для приложений.

Планирование приложения

Разработка любого приложения должна начинаться с его планирования — определения требований, предъявляемых к приложению, его функциональности.

Для приложения, предназначенного для работы с базами данных, можно условно выделить три основных этапа его создания:

- ☐ разработка структуры базы данных, то есть состава и структуры таблиц, из которых состоит база данных, а также отношений между ними;
- ☐ определение функций, выполняемых приложением;
- ☐ разработка интерфейса пользователя.

Перечисленные этапы необязательно должны выполняться в указанной последовательности. Однако обычно разработку приложения рекомендуется начинать именно с выбора структуры базы данных, так как значительная часть программного кода жестко привязана к ее структуре. Модификация структуры базы данных на завершающих стадиях разработки приложения может привести к возникновению трудно диагностируемых ошибок.

При разработке функций приложения определяются методы обработки данных, обеспечивается контроль вводимых пользователем значений, а также кон-

троль удаления и модификации содержащихся в таблицах данных. При разработке приложений, работающих с локальными базами данных, может также потребоваться реализация функций разграничения доступа к информации, содержащейся в базе данных.

Разработка интерфейса пользователя обычно сводится к разработке форм, диалоговых окон, системы меню и панелей инструментов. На этом этапе в первую очередь необходимо учесть потребности конечного пользователя и обеспечить простой и интуитивно понятный интерфейс.

При разработке интерфейса пользователя не стоит принимать слишком оригинальные решения. Лучше, если программа будет иметь «стандартный» вид, нежели обладать изысканным, но непривычным и малопонятным для «среднего» пользователя интерфейсом.

Меню окончательно формируется на последней стадии разработки приложения, когда все функции программы определены. Оно, как правило, содержит полный набор команд, обеспечивающих доступ пользователя к функциям приложения. Давать рекомендации по созданию меню для общего случая довольно сложно, можно лишь сформулировать ряд пожеланий:

- ☐ очень важно корректно сгруппировать команды меню, чтобы у пользователей не возникало лишних сложностей при поиске необходимой команды;
- ☐ не следует применять много уровней вложенности команд меню, так как это затрудняет их поиск;
- ☐ для наиболее часто используемых команд имеет смысл определить соответствующие им клавиатурные сокращения («горячие клавиши»).

Панели инструментов содержат кнопки, обеспечивающие быстрый доступ к командам меню (панели инструментов могут содержать и «инструменты» в прямом смысле этого слова — например инструменты для рисования в графических редакторах). Как правило, панели инструментов не дают доступа ко всем функциям приложения, то есть содержат сокращенный набор команд.

Современные средства проектирования позволяют без особых затруднений создавать настраиваемые панели инструментов. Поскольку заранее трудно определить оптимальный состав кнопок на панели инструментов, имеет смысл обеспечить возможность настройки панели инструментов пользователем. Стоит предусмотреть возможность сохранения избранной пользователем конфигурации при повторных запусках приложения.

Кроме того, полезно использовать в панелях инструментов всплывающие подсказки о назначении кнопок, так как размещаемые на них значки далеко не всегда вызывают правильные ассоциации с соответствующими командами.

Создание главного меню

Строка главного меню располагается в верхней части главной формы приложения. Доступ к ее командам может осуществляться одним из трех способов:

- ☐ с помощью мыши;
- ☐ с помощью клавиатуры (для входа в главное меню зарезервирована клавиша Alt, а для навигации по командам меню используются клавиши перемещения курсора);

- с помощью специальных комбинаций клавиш — клавиатурных сокращений, которые, однако, могут быть заданы не для всех команд меню.

Для создания главного меню в Delphi имеется специальный компонент TMainMenu, расположенный на странице Standard палитры компонентов.

Компонент TMainMenu обычно относят к невизуальным, хотя главное меню отображается во время разработки приложения. Основные свойства класса TMainMenu приведены в табл. 12.1.

Таблица 12.1. Основные свойства класса TMainMenu

Свойство	Тип	Описание
AutoMerge	Boolean	В зависимости от значения этого свойства меню дочерней формы SDI-приложения будет (true) или не будет (false) добавляться к меню главной формы. Положение добавляемых пунктов меню (TMenuItem) будет зависеть от значений их свойств GroupIndex. В MDI-приложениях меню дочерней формы всегда объединяется с меню главной формы, независимо от значения этого свойства
AutoHotkeys	TMenuAutoFlag	Используется для исключения конфликтов оперативных клавиш при изменении состава команд меню во время выполнения программы
Images	TCustomImageList	Подключает к меню коллекцию изображений (которая может быть задана, например, с помощью компонента TImageList). Изображения, содержащиеся в этом свойстве, используются для задания значков, отображаемых слева от соответствующих им команд меню
OwnerDraw	Boolean	Определяет, как будет происходить прорисовка меню. Если задано значение false, прорисовка выполняется автоматически. В случае значения true для отображения меню необходимо программировать обработчик события OnDrawItem. Обычно это используется для создания нестандартных меню

Среди методов, определенных в классе TMainMenu, наиболее часто используются два:

- procedure Merge(Menu: TMainMenu) — вызывается для объединения двух меню в SDI-приложениях. Например, вызывая этот метод, можно включить в состав меню главной формы пункты меню дочерней формы. Положение добавляемых пунктов меню определяется свойством GroupIndex элемента меню TMenuItem;
- procedure Unmerge(Menu: TMainMenu) — исключает из меню формы пункты, добавленные при слиянии двух меню.

ПРИМЕЧАНИЕ

Если свойство AutoMerge дочерней формы установлено равным true, то методы Merge и Unmerge вызываются автоматически при открытии и закрытии дочерней формы.

Класс TMenuItem

Каждый элемент меню является экземпляром класса TMenuItem. Причем объект TMenuItem может либо быть командой, либо содержать меню более низкого уровня. Количество уровней вложенности не ограничено. Основные свойства класса TMenuItem приведены в табл. 12.2.

Таблица 12.2. Основные свойства класса TMenuItem

Свойство	Тип	Описание
Action	TBasicAction	Определяет действие, связанное с данной командой меню
Bitmap	TBitmap	Определяет значок, показываемый слева от строки текста команды меню
Break	TMenuBarBreak = (mbNone, mbBreak, mbBarBreak)	Используется для отображения команд меню в несколько колонок: <ul style="list-style-type: none"> mbNone — элемент меню отображается в текущей колонке; mbBreak — начиная с текущего элемента пункты меню отображаются в следующей колонке; mbBarBreak — начиная с текущего, пункты меню отображаются в следующей колонке, причем между колонками располагается разделитель
Caption	String	Текст команды
Checked	Boolean	Используется для команд меню, функционирующих подобно элементу CheckBox (флажок). Если данное свойство имеет значение true, то команда меню помечается (флажок устанавливается)
Default	Boolean	Используется для команд подменю. Если значение свойства Default задано равным true, то данная команда будет выполняться при двойном щелчке на заголовке подменю. Из всех команд, входящих в подменю, только одна может иметь значение этого свойства равным true
Enabled	Boolean	Определяет, доступна команда меню (true) или нет (false). Недоступные команды отображаются серым цветом
GroupIndex	Byte	Используется при объединении двух меню
ImageIndex	TImageIndex	Указывает на изображение в коллекции, заданной в свойстве Images объекта TMainMenu, которое будет отображаться слева от текста команды. Данное свойство имеет более высокий приоритет, чем свойство Bitmap
MenuItemIndex	Integer	Указывает на порядковый номер пункта меню. Изменение этого свойства приводит к изменению расположения пунктов меню
RadioItem	Boolean	Используется для создания команд меню, функционирующих подобно элементу TRadioButton. Для группировки пунктов меню используется свойство GroupIndex
ShortCut	TShortCut	Определяет клавиатурные сокращения для команды меню

В классе TMenuItem содержится также ряд методов, которые могут быть полезны при работе с меню. Рассмотрим основные из них:

- ❑ procedure Add(Item: TMenuItem) — добавляет в конец меню новую команду;
- ❑ procedure Clear — удаляет все элементы меню и освобождает занимаемую ими память;

- ❑ procedure Click — вызывает событие OnClick для данного элемента меню;
- ❑ function Find(ACaption: string): TMenuItem — возвращает указатель на элемент меню, соответствующий указанному в ACaption тексту команды. Если такой элемент не найден, возвращает nil;
- ❑ function IndexOf(Item: TMenuItem): Integer — возвращает порядковый номер элемента меню;
- ❑ procedure Insert(Index: Integer; Item: TMenuItem) — добавляет в меню элемент Item, располагая его в позиции Index;
- ❑ function InsertNewLineAfter(AItem: TMenuItem): Integer — добавляет разделитель после указанного элемента меню;
- ❑ function InsertNewLineBefore(AItem: TMenuItem): Integer — добавляет разделитель перед указанным элементом меню;
- ❑ function IsLine: Boolean — определяет, является элемент меню разделителем или нет;
- ❑ procedure Remove(Item: TMenuItem) — удаляет указанный элемент меню.

В классе TMenuItem определены четыре обработчика событий. Один из них — OnClick — используется для задания реакции на выбор команды меню. Остальные три — OnAdvancedDrawItem, OnDrawItem и OnMeasureItem — предназначены для создания меню, обладающих возможностями, не поддерживаемыми стандартными средствами. Их применение возможно только в том случае, если свойство OwnerDraw компонента TMainMenu задано равным true.

Работа с редактором меню

Для создания меню в Delphi используется специальный редактор (рис. 12.1), который запускается двойным щелчком на компоненте TMainMenu, помещенном на форму, или щелчком на кнопке с многоточием в поле ввода свойства Items этого компонента в инспекторе объектов.

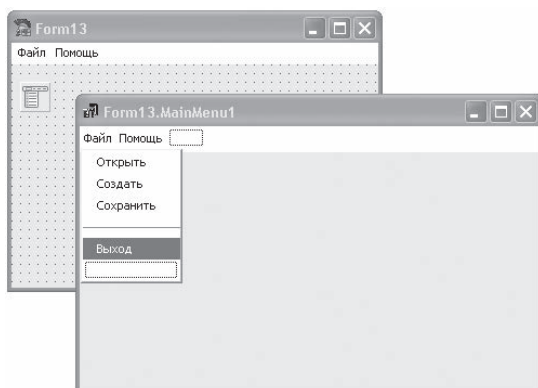


Рис. 12.1. Окно редактора меню

Создание и удаление команд меню

Для добавления новых элементов меню (команд или новых пунктов меню) сделайте следующее:

- ❑ выберите с помощью мыши (или клавишами перемещения курсора) свободный элемент (такие элементы всегда есть внизу каждого выпадающего меню и в правой части строки меню);
- ❑ введите текст команды в поле ввода свойства `Caption` в инспекторе объектов.

Для вставки новых элементов меню между уже существующими используйте команду `Insert` контекстного меню редактора или клавишу `Ins`. Созданный при этом новый элемент будет располагаться выше (или левее) того элемента, который был выделен перед выполнением команды `Insert`.

Чтобы удалить элемент меню, можно воспользоваться либо клавишей `Del`, либо командой `Delete` контекстного меню.

При написании текста команды можно использовать служебный символ «&» для назначения командам меню соответствующих им клавиш ускоренного доступа. Клавиша, соответствующая букве, перед которой помещен этот символ, становится клавишей ускоренного доступа. При этом в тексте команды меню эта буква выделяется подчеркиванием.

ПРИМЕЧАНИЕ

Не следует путать клавиши ускоренного доступа к командам с клавиатурными сокращениями. Нажатие на клавишу ускоренного доступа вызывает соответствующую команду только при условии, что меню в момент нажатия имеет фокус ввода. В отличие от них клавиатурные сокращения никак не привязаны к состоянию меню и всегда функциональны.

Использование значков в командах меню

При создании меню имеется возможность добавления значка к команде меню. Он показывается слева от текста команды меню (рис. 12.2).

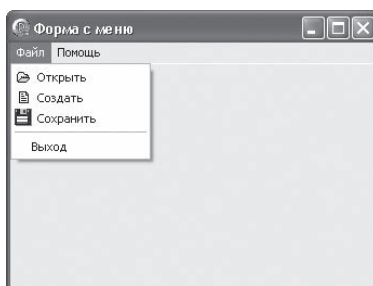


Рис. 12.2. Пример меню со значками

Для добавления значка можно использовать либо свойство `ImageIndex`, либо свойство `Bitmap` класса `TMenuItem`. В первом случае значок выбирается из коллекции изображений, задаваемой свойством `Images` класса `TMainMenu`. Во втором случае устанавливается BMP-файл с подходящей картинкой. Если изображения заданы в обоих свойствах, то в команде меню будет отображаться значок, указанный в свойстве `ImageIndex`.

ПРИМЕЧАНИЕ

В редакторе меню значки, добавленные к команде меню, не отображаются. Стандартная коллекция пиктограмм по умолчанию устанавливается в каталог /Images/Default.

Назначение командам меню оперативных клавиш

Для задания оперативных клавиш используется свойство ShortCut класса TMenuItem. В инспекторе объектов поле ввода данного свойства содержит выпадающий список, в котором предлагается ряд вариантов клавиатурных сокращений. Если ни один из вариантов вас не устраивает, можно ввести свою комбинацию с клавиатуры в виде текстовой строки. Для служебных клавиш в этом случае приняты следующие обозначения: Alt, Ctrl, Shift, BkSp, Ins, Del.

Комбинация оперативных клавиш, назначенных для команды меню, отображается справа от текста команды (рис. 12.3).

Создание разделителей

Часто команды меню, входящие в одно выпадающее меню, дополнительно группируются с помощью разделителей в виде горизонтальных линий (например, в меню, показанном ранее на рис. 12.2, разделитель отделяет команду Выход).

Разделители можно создавать в редакторе меню точно так же, как и обычные команды. Для этого в свойстве Caption элемента меню следует просто ввести символ «-».

Создание подменю

Чтобы создать подменю, выполните следующие действия:

- ☐ установите указатель мыши на команду меню;
- ☐ нажмите правую кнопку мыши;
- ☐ выберите команду Create SubMenu контекстного меню редактора меню.

После этих действий справа от текущей команды меню появится символ >, обозначающий подменю.

Задание реакции на выбор команды меню

Выбор команд меню во время выполнения программы приводит к совершению определенных действий. Для их задания можно использовать два способа:

- ☐ задать обработчик события OnClick элемента меню;
- ☐ задать элементу меню соответствующее значение свойства Action.

Программирование обработчика события OnClick

Задать обработчик события OnClick необходимой команды меню можно следующими способами:

- ☐ выполнить двойной щелчок на команде в редакторе меню;
- ☐ выбрать эту команду в меню формы во время разработки приложения.

Delphi автоматически активизирует окно редактора кода и сгенерирует заголовков процедуры-обработчика события `OnClick`:

```
procedure TForm1.N4Click(Sender: TObject);  
begin
```

```
end;
```

Для программирования реакции на выбор команды меню достаточно написать соответствующий код и вставить его между словами `begin` и `end` процедуры-обработчика. Например, для отображения окна диалога открытия файла при выборе команды меню **Файл ► Открыть** необходимо ввести следующий код:

```
procedure TForm1.N4Click(Sender: TObject);  
begin  
    if OpenDialog1.Execute()  
    then begin  
        ...  
    end;  
end;
```

Использование свойства Action

При использовании этого способа необходимо применять компонент `TActionList`, расположенный на вкладке **Standard** палитры компонентов. Данный компонент представляет собой нечто вроде хранилища функций, являющихся реакциями на определенные события. Компонент `TActionList` имеет всего три опубликованных свойства: `Name`, `Images` и `Tag`. В свойстве `Images` указывается ссылка на коллекцию изображений, которые можно связывать с задаваемыми действиями.

Для задания действия следует воспользоваться редактором действий (рис. 12.3), который открывается двойным щелчком мыши на компоненте `TActionList`, помещенном на форму.

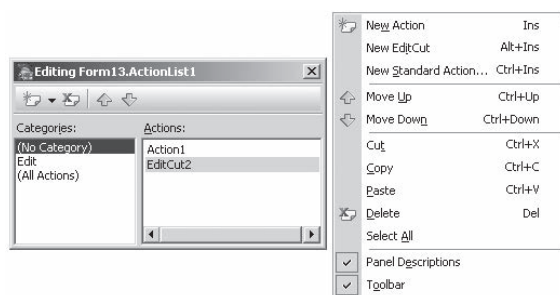


Рис. 12.3. Окно редактора действий

Для добавления нового действия используется команда **New Action** контекстного меню редактора действий. Каждое заданное действие является экземпляром класса `TAction`, причем все основные опубликованные свойства этого класса дублируют ряд свойств класса `TMenuItem`: `Caption`, `Checked`, `Enabled`, `ImageIndex` и `Shortcut`. Свойство `Category` ни на что не влияет и используется для разделения заданных действий на группы в редакторе действий.

Функция, связанная с действием, задается с помощью метода-обработчика события `OnExecute` класса `TAction`.

Если указать некоторое действие в свойстве `Action` элемента меню, то данная команда меню будет отображаться согласно параметрам, указанным для действия (используются заданные для действия текст команды, значок, оперативные клавиши). При выборе этой команды будет выполняться функция, заданная в обработчике события `OnExecute` указанного действия.

ПРИМЕЧАНИЕ

На первый взгляд может показаться, что использование действий неоправданно усложняет программирование откликов на выбор команд меню. Действительно, во многих случаях для команд меню проще использовать обработчик события `OnClick`. Однако для достаточно больших программ, в которых одно и то же действие может выполняться для разных событий (например, при выборе команды меню и при щелчке на кнопке панели инструментов), удобнее использовать действия. В этом случае код программы будет лучше читаться и снижается вероятность ошибки.

Создание контекстного меню

В Delphi имеется возможность связывания контекстного меню практически с любым визуальным компонентом. Для этой цели предназначено свойство `PopupMenu`.

Процедура создания самого контекстного меню предполагает использование специального компонента — `TPopupMenu`. Технология реализации такого меню практически не отличается от создания обычного меню — используется тот же редактор меню, а свойства и методы класса `TPopupMenu` аналогичны свойствам и методам класса `TMenuItem`, которые уже были рассмотрены ранее.

Реакция на выбор команды контекстного меню задается точно так же, как и для команд главного меню — либо используется обработчик события `OnClick`, либо задается действие, которое указывается в свойстве `Action` компонента `TPopupMenu`.

Панель инструментов

Панели инструментов в последнее время стали таким же привычным элементом интерфейса, как и меню. Во многих случаях панель инструментов является альтернативой меню, обеспечивая более оперативный доступ к командам.

Хотя панели инструментов могут создаваться несколькими способами, наиболее простой и естественный из них — использование специального компонента `TToolBar`, который по умолчанию находится в палитре компонентов Win32.

Класс `TToolBar`

Класс `TToolBar` объединяет в одном объекте сами кнопки и контейнер для них. В контейнере компонента `TToolBar` допускается размещать и другие элементы управления, такие как `TEdit`, `TComboBox` и т. п. Основные свойства класса `TToolBar` приведены в табл. 12.3.

Таблица 12.3. Основные свойства класса TToolBar

Свойство	Тип	Описание
ButtonWidth	Integer	Ширина кнопок, расположенных на панели инструментов
ButtonHeight	Integer	Высота кнопок, расположенных на панели инструментов
Images	TCustomImageList	Ссылка на список картинок, которые будут отображаться на кнопках, находящихся в обычном состоянии
DisabledImages	TCustomImageList	Ссылка на список картинок, отображаемых на неактивных кнопках
Flat	Boolean	Внешний вид кнопок: обычные (false) или плоские (true)
HotImages	TCustomImageList	Коллекция картинок, отображаемых на кнопках при нахождении над ними указателя мыши
Indent	Integer	Интервал между левой границей панели и левой границей элемента управления, расположенного на ней
List	Boolean	Способ расположения изображения и надписи на кнопке друг относительно друга: если true, то текст располагается справа от картинки, если false — то над картинкой
ShowCaptions	Boolean	Определяет, отображать (true) или нет (false) надписи на кнопках
Transparent	Boolean	Способ отображения панели: прозрачная (true) или нет (false)
Wrapable	Boolean	Если значение данного свойства равно true, то элементы управления на панели инструментов могут автоматически распределяться по нескольким строкам

ПРИМЕЧАНИЕ

Методы, инкапсулированные в классе TToolBar, используются довольно редко, поэтому мы их рассматривать не будем.

В классе TToolBar определено довольно большое количество событий, среди которых наиболее важными являются события, генерируемые при перемещении панели с помощью мыши: OnStartDrag, OnStartDock, OnEndDrag, OnEndDock, OnDragDrop, OnDragOver и OnDockOver.

Класс TToolButton

Каждая кнопка, расположенная на панели инструментов, является экземпляром класса TToolButton, основные свойства которого приведены в табл. 12.4.

Таблица 12.4. Основные свойства класса TToolButton

Свойство	Тип	Описание
Action	TBasicAction	Действие, связанное с кнопкой
Caption	TCaption	Надпись на кнопке (отображается в том случае, если значение свойства ShowCaptions компонента TToolBar задано равным true)

продолжение ➤

Таблица 12.4 (продолжение)

Свойство	Тип	Описание
AllowAllUp	Boolean	Определяет, могут (true) или нет (false) все кнопки, входящие в группу с зависимым нажатием, быть «отжаты» одновременно
AutoSize	Boolean	Если значение данного свойства равно true, то размеры кнопки автоматически изменяются таким образом, чтобы полностью отобразить изображение и надпись на кнопке
Down	Boolean	Определяет, нажата кнопка (true) или нет (false)
DropDownMenu	TPopupMenu	Указывает на меню, связанное с кнопкой
Grouped	Boolean	Используется для организации группы кнопок с зависимым нажатием
Hint	String	Текст всплывающей подсказки
ImageIndex	TImageIndex	Задаёт картинку из списка Images панели инструментов, отображаемую на кнопке
Indeterminate	Boolean	Если значение данного свойства равно true, то кнопка находится в неопределённом состоянии
Marked	Boolean	Если значение данного свойства равно true, то кнопка выделяется цветом
MenuItem	TMenuItem	Пункт меню, соответствующий кнопке
ShowHint	Boolean	Если значение данного свойства равно true, всплывающие подсказки показываются, если false — нет
Style	TToolButtonStyle = (tbsButton, tbsCheck, tbsDropDown, tbsSeparator, tbsDivider);	Стиль кнопки: <ul style="list-style-type: none">• tbsButton — обычная кнопка;• tbsCheck — кнопка с двумя состояниями (аналог CheckBox);• tbsDropDown — кнопка с выпадающим меню;• tbsSeparator, tbsDivider — разделители

Чтобы поместить кнопку на панели инструментов во время разработки программы, следует использовать команду **New Button** контекстного меню компонента **TToolBar**. Размеры всех кнопок, расположенных на панели инструментов, одинаковы и определяются свойствами **ButtonWidth** и **ButtonHeight** компонента **TToolBar**. Кнопки можно группировать, используя разделители, которые помещаются на панель инструментов с помощью команды **New Separator** контекстного меню. После размещения кнопок и разделителей на панели инструментов их можно перемещать с помощью мыши.

Из методов, инкапсулированных в классе **TToolButton**, рассмотрим два, которые используются наиболее часто:

- ❑ **function CheckMenuDropDown: Boolean** — отображает выпадающее меню, связанное с кнопкой, и возвращает значение true, если для кнопки задано свойство **DropDownMenu**. Возвращает false, если кнопка не имеет меню;
- ❑ **procedure Click** — генерирует событие **OnClick** кнопки. Используется для программного «нажатия» на кнопку.

События, определённые для класса **TToolButton**, генерируются или при нажатии на кнопку (**OnClick**), либо при перетаскивании кнопки (аналогичны событиям, рассмотренным для **TToolBar**).

Так же как и команды меню, кнопки на панели инструментов могут использоваться в качестве аналогов элементов управления `TCheckBox` и `TRadioButton`. В первом случае достаточно присвоить свойству `Style` кнопки значение `tbsCheck`. После этого кнопка может находиться в двух состояниях: нажатом и отжатом. О состоянии кнопки сигнализирует свойство `Down`.

Чтобы использовать кнопки в качестве переключателей, необходимо сгруппировать их, присвоив свойству `Grouped` каждой кнопки, включаемой в группу, значение `true`. Причем значение свойства `Style` для всех кнопок группы следует установить в значение `tbsCheck`. После этого находиться в нажатом состоянии сможет только одна кнопка из группы. Если же свойству `AllowAllUp` кнопки, входящей в группу, присвоить значение `true`, то ее можно будет «отжать», даже если все остальные кнопки группы «отжаты».

Для визуального объединения родственных кнопок в группы используются разделители. Поскольку разделители также являются экземплярами класса `TToolButton`, то они обладают всеми свойствами кнопок. Фактически разделитель — это обычная кнопка, для которой задан стиль `tbsSeparator` или `tbsDivider` (однако ширина разделителя может отличаться от ширины обычной кнопки).

Различие между стилями `tbsSeparator` и `tbsDivider` заключается в следующем: в первом случае кнопки разделяются интервалом, на котором никакие дополнительные элементы не отображаются (первый разделитель на рис. 12.4). Во втором случае кнопки разделяются вертикальной чертой (второй разделитель на рис. 12.4).



Рис. 12.4. Форма с панелью инструментов

Задание реакции на нажатие кнопки

Реакция на нажатие кнопки панели инструментов может задаваться тремя способами:

- ☐ с помощью обработчика события `OnClick`;
- ☐ путем указания действия в свойстве `Action`;
- ☐ путем указания пункта меню, соответствующего кнопке, с помощью свойства `MenuItem`.

Первые два способа ничем не отличаются от рассмотренных ранее для меню. При использовании третьего способа в свойстве `MenuItem` указывается имя объекта `MenuItem`. В этом случае нажатие на кнопку инициирует выполнение действия, связанного с этим объектом. Причем неважно, каким образом задана реак-

ция на выбор этого пункта меню — через действие или с помощью обработчика события `OnClick`.

Контейнеры для панелей инструментов

Часто приложение содержит несколько панелей инструментов, которые размещаются в специальных контейнерах. Примером этого может служить среда разработки Delphi, главное окно которой фактически представляет собой контейнер, в котором содержатся различные панели инструментов: меню и т. п.

В Delphi функции контейнера для панелей инструментов выполняет `TCoolBar`.

Компонент `TCoolBar`

Компонент `TCoolBar` фактически является коллекцией объектов `TCoolBand`, каждый из которых может содержать панель инструментов `TToolBar`. Таким образом, для размещения панели инструментов в контейнере `TCoolBar` необходимо сначала создать в нем элемент `TCoolBand`. Для этого следует использовать специальный редактор, который открывается двойным щелчком мыши на компоненте `TCoolBar`, помещенном на форму, или при щелчке на кнопке с многоточием в инспекторе объектов, в поле ввода свойства `Bands`. Панель инструментов связывается с элементом `TCoolBands` с помощью свойства `Control` класса `TCoolBand`.

Панели инструментов, помещенные в контейнер `TCoolBar`, можно сделать «плавающими». Для этого достаточно присвоить свойству `DockSite` контейнера `TCoolBar` и главной формы значение `true`, а свойству `DragMode` панели инструментов — значение `dmAutomatic`. После этого во время работы программы можно будет с помощью мыши перемещать панели инструментов из контейнера на форму, и наоборот.

Путем настройки свойств `DockSite` и `DragMode` можно реализовать механизм «drag-and-dock», не написав ни одной строки программного кода. Однако следует иметь в виду, что на практике часто возникают ситуации, где требуется организовать механизм перетаскивания «вручную», то есть задавая соответствующий программный код. К сожалению, ограниченный объем книги не позволяет рассмотреть этот вопрос более подробно.

Глава 13

Управление проектом и создание приложения

Основой любого приложения, создаваемого в среде Delphi, является проект (или группа проектов). Проект объединяет в себе все отдельные структурные блоки приложения, определяющие интерфейс пользователя программы и выполняемые ей функции. Проект также обеспечивает взаимодействие структурных блоков — как между собой, так и со средой разработки Delphi.

С помощью Delphi можно решать различные задачи:

- ☐ разрабатывать исполняемые приложения (EXE-модули);
- ☐ разрабатывать динамически связываемые библиотеки (библиотеки DLL);
- ☐ создавать компоненты VCL для Delphi;
- ☐ создавать приложения для работы в Интернете.

В случаях создания приложений VCL разработка начинается с проекта, причем структура проекта и его свойства зависят от типа решаемой задачи. В данной главе мы рассмотрим вопросы управления проектом при создании приложения.

Структура проекта

Разработка нового приложения всегда начинается с создания нового проекта. Создать новый проект приложения можно с помощью команды **File ► New ► VCL Forms Application — Delphi For Win 32** главного меню.

ПРИМЕЧАНИЕ

Всякий раз после запуска Delphi по умолчанию создается новый проект приложения.

После создания нового приложения автоматически генерируются три объекта: модуль проекта, форма и модуль формы, которые при сохранении записываются в файлы с расширениями **DPR**, **DFM** и **PAS** соответственно. Кроме этих трех

основных файлов проект содержит еще три файла, в которых находится служебная информация:

- ❑ файлы с расширениями DOF и CFG содержат сведения о настройках проекта, задаваемых в окне диалога **Project Options**;
- ❑ файл с расширением RES содержит ресурсы проекта.

Модуль формы проекта

При создании нового приложения в редактор кода автоматически загружается файл с расширением PAS, содержащий код модуля формы (листинг 13.1). По структуре этот файл является обычным модулем языка Object Pascal и содержит следующую информацию:

- ❑ объявления используемых модулей — указываются основные стандартные модули Delphi, необходимые для обеспечения функционирования компонентов, входящих в состав приложения. Если затем на форму помещается компонент, для которого требуется модуль, не указанный ранее в разделе **uses** модуля формы, то объявление этого модуля добавляется автоматически;
- ❑ объявление нового класса формы TForm1, являющегося потомком стандартного класса TForm. В данном классе будут инкапсулированы все компоненты, размещаемые на форме;
- ❑ объявление переменной Form1 — экземпляра класса TForm1. Если затем с помощью инспектора объектов изменить имя формы, то имя этой переменной также будет автоматически изменено;
- ❑ подключение файла ресурсов формы — директива `{ $R *.DFM }`.

Листинг 13.1. Код модуля формы

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs;

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

var
  Form1: TForm1;
```

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

end.

ПРИМЕЧАНИЕ

Файл модуля проекта фактически является шаблоном, в который затем добавляют различные фрагменты кода, обеспечивающие функционирование приложения — описания типов и переменных, обработчики событий, процедуры и функции, определяемые пользователем и т. п.

Главный файл проекта

Главный файл проекта содержит всего несколько строк кода (листинг 13.2). На практике обычно не требуется изменять этот файл, поэтому при создании проекта он даже не загружается в редактор кода.

Чтобы загрузить файл проекта в редактор кода, следует воспользоваться командой **Project ► View Source** главного меню Delphi IDE.

Листинг 13.2. Главный файл проекта

```
program Project1;  
uses  
    Forms,  
    Unit1 in 'Unit1.pas' {Form1};  
{ $R *.RES }  
begin  
    Application.Initialize;  
    Application.CreateForm(TForm1, Form1);  
    Application.Run;  
end.
```

Главный файл проекта имеет структуру файла программы языка Object Pascal и содержит следующую информацию:

- ❑ объявления используемых модулей: **Forms** — стандартный модуль Delphi, **Unit1** — модуль, содержащий описание формы проекта;
- ❑ подключение файла ресурсов — директива **{ \$R *.RES }**;
- ❑ вызов методов класса **TApplication** (более подробно этот класс будет рассмотрен ниже):
 - **Initialize** — инициализация приложения;
 - **CreateForm(TForm1, Form1)** — создание формы **Form1**;
 - **Run** — запуск приложения.

Файл описания формы проекта

Файл формы (DFM) содержит описание свойств формы и компонентов, размещенных на форме. Этот файл может быть либо текстовым, либо двоичным — в зависимости от состояния флажка **New form as text** на вкладке **Preferences** окна диалога **Environment Options**. Но в любом случае текст описания формы можно загрузить в редактор кода с помощью команды **View as Text** контекстного меню редактора форм.

ПРИМЕЧАНИЕ

Чтобы вернуться к визуальному редактору форм, используйте команду View as Form контекстного меню редактора кода.

Пример содержимого файла описания формы (рис. 13.1), на которой размещены кнопка (компонент TButton) и поле ввода (компонент TEdit), приведен в листинге 13.3.

Листинг 13.3. Файл описания формы

```
object Form1: TForm1
  Left = 0
  Top = 0
  Caption = 'Form1'
  ClientHeight = 206
  ClientWidth = 339
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'Tahoma'
  Font.Style = []
  OldCreateOrder = False
  PixelsPerInch = 96
  TextHeight = 13
  object Button1: TButton
    Left = 48
    Top = 96
    Width = 75
    Height = 25
    Caption = 'Button1'
    TabOrder = 0
  end
  object Edit1: TEdit
    Left = 48
    Top = 48
    Width = 121
    Height = 21
    TabOrder = 1
    Text = 'Edit1'
  end
end
```

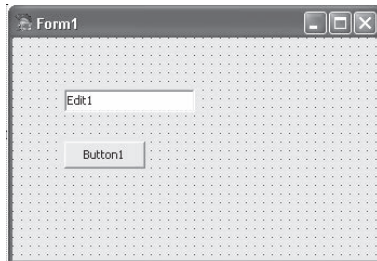


Рис. 13.1. Пример формы проекта

Данный файл содержит только те свойства формы и размещенных на ней элементов, которые определяют их внешний вид и расположение на форме.

ПРИМЕЧАНИЕ

Добавить новый компонент на форму можно с помощью редактирования текста описания формы — просто дополнить его описанием требуемого объекта, а также включить этот объект в описание класса формы (в файле модуля формы). Однако гораздо удобнее использовать визуальный редактор, поэтому обычно не требуется изменять файл описания формы вручную.

Добавление к проекту форм и модулей

Если проект содержит несколько форм, то одна из них является главной — то есть той формой, которая будет отображаться при запуске приложения. При каждом создании нового проекта приложения (команда меню **File** ► **New** ► **VCL Forms Application — Delphi for Win32**) автоматически создается одна форма, которая и является главной.

Если проект должен включать несколько форм, то их необходимо добавлять к проекту, используя команду **File** ► **New** ► **Form — Delphi for Win32** главного меню. Главной формой при этом остается та форма, которая была создана первой. Каждой форме приложения обязательно соответствует модуль, который создается автоматически в процессе генерации новой формы.

Проект может также включать в себя программные модули, не связанные с формами. Такие модули обычно являются библиотеками процедур и функций, общих для всего приложения. Для создания нового модуля используйте команду **File** ► **New** ► **Unit — Delphi for Win32** главного меню.

Для добавления в проект уже существующих (созданных ранее) модулей и форм используется команда **Project** ► **Add to Project**. При выполнении этой команды открывается стандартное окно диалога выбора файла, в котором указывается файл модуля, добавляемого к проекту. Если выбранный модуль связан с формой, то она также добавляется к проекту и включается в список автоматически создаваемых форм.

Для удаления из проекта модулей и/или форм применяется команда **Project** ► **Remove from Project**. При ее выполнении отображается окно диалога **Remove From Project** (рис. 13.2), содержащее список всех форм и модулей, принадлежавших проекту. Чтобы удалить модуль, выделите его в списке и щелкните на кнопке **OK**. В данном окне допускается выделение сразу нескольких объектов — для этого при выполнении операции выделения следует удерживать нажатой клавишу **Ctrl**.

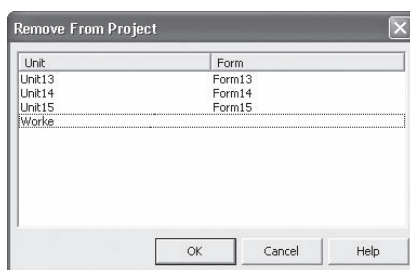


Рис. 13.2. Окно диалога **Remove From Project** предназначено для удаления модуля из проекта

Для подключения вновь созданных модулей (не важно, связанных с формами или нет) используйте команду **File** ► **Use Unit** главного меню. Ее вызов открывает окно диалога **Use Unit** (рис. 13.3), содержащее список всех модулей проекта, которые не объявлены в разделе **uses** текущего файла в редакторе кода.

Выберите из списка те модули, которые вы хотите подключить, и щелкните на кнопке OK.

Добавлять формы и модули в проект можно через контекстное меню менеджера проектов. Команды контекстного меню **Add New ► Form** и **Add New ► Unit** инициируют появление тех же окон диалога. Напомним, менеджер проектов вызывается командой **View ► Project Manager** либо комбинацией клавиш **Ctrl+Alt+F11** и обычно располагается в правом верхнем углу среды разработки.

СОВЕТ

Подключение модулей можно также выполнить вручную — просто указав имена нужных модулей в разделе `uses`. При этом для избежания конфликтов лучше объявлять модули проекта в разделе `implementation`, а не `interface`.

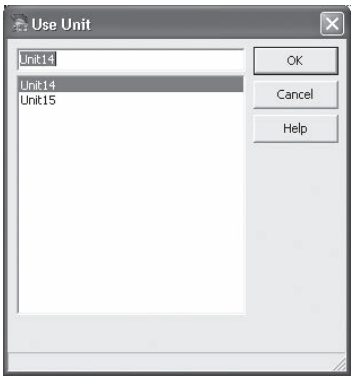


Рис. 13.3. Окно диалога Use Unit используется для подключения модулей

Класс TApplication

Любое приложение, создаваемое в среде Delphi, является экземпляром класса `TApplication`. Данный класс реализует взаимодействие приложения с операционной системой Windows. Переменная, обеспечивающая доступ к свойствам и методам класса `TApplication`, для любого приложения имеет одно и то же имя — `Application`. Данная переменная объявляется в модуле `Forms` и всегда доступна во всех модулях приложения. Основные свойства класса `TApplication` приведены в табл. 13.1.

Таблица 13.1. Основные свойства класса `TApplication`

Свойство	Тип	Описание
<code>Active</code>	<code>Boolean</code>	Значение свойства равно <code>true</code> , если приложение активно
<code>CurrentHelpFile</code>	<code>string</code>	Имя файла контекстной справки
<code>HelpFile</code>	<code>string</code>	Имя файла справки приложения
<code>ExeName</code>	<code>string</code>	Имя исполняемого файла приложения
<code>Handle</code>	<code>HWND</code>	Дескриптор главного окна приложения

Свойство	Тип	Описание
Icon	TIcon	Значок приложения
MainForm	TForm	Главная форма приложения
Hint	string	Текст подсказки

В классе `TApplication` определен также ряд методов, с помощью которых можно управлять как внешним видом, так и процессом выполнения приложения. Основные из них приведены в табл. 13.2.

Таблица 13.2. Основные методы класса `TApplication`

Метод	Описание
procedure BringToFront	Помещает окно приложения поверх всех других окон
procedure Minimize	Сворачивает главное окно приложения
procedure ProcessMessages	Прерывает выполнение приложения для обработки сообщений Windows
procedure Terminate	Закрывает приложение
procedure Restore	Восстанавливает свернутое окно приложения

В классе `TApplication` определен также ряд событий. Для программирования обработчиков этих событий следует использовать специальный компонент `TApplicationEvents`, расположенный на странице **Additional** палитры компонентов. Отметим основные события, обрабатываемые классом `TApplication`:

- ❑ `OnActionExecute` — вызывается при выполнении действия, определенного с помощью компонента `TActionList`;
- ❑ `OnActivate` — вызывается при активизации приложения;
- ❑ `OnDeactivate` — вызывается при деактивизации приложения;
- ❑ `OnMessage` — вызывается при получении сообщения от операционной системы;
- ❑ `OnShortCut` — вызывается при нажатии на любую клавишу.

Управление формами проекта

При создании каждой новой формы в проекте она автоматически заносится в список форм, создаваемых по умолчанию. Управление формами, содержащимися в этом списке, происходит автоматически. Это значит, что при запуске приложения для них выделяются необходимые ресурсы, а при завершении приложения выделенные под эти формы ресурсы освобождаются.

В некоторых случаях автоматическое управление формами является нежелательным, так как приложение, содержащее много форм, будет нерационально использовать память — хранить информацию, ненужную в текущий момент. Кроме того, из-за этого замедляется загрузка программы. Поэтому в ряде случаев программисту целесообразно взять на себя управление созданием и удалением некоторых форм.

Управление списком автоматически создаваемых форм осуществляется с помощью вкладки Forms окна диалога Project Options (рис. 13.4).

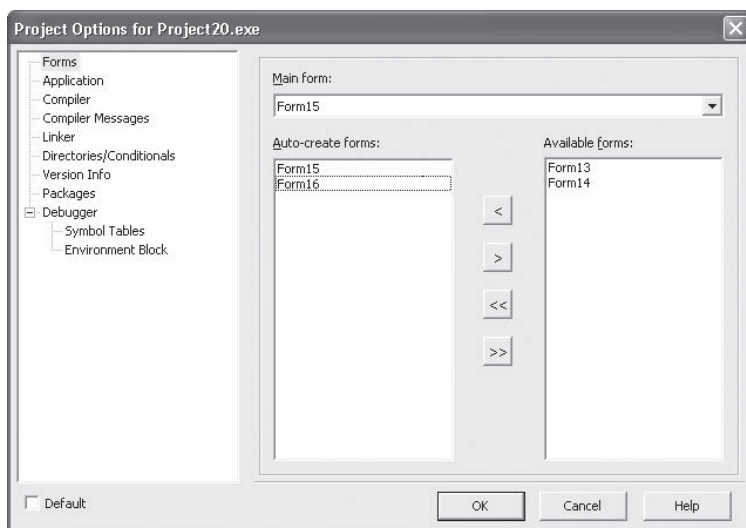


Рис. 13.4. Вкладка Forms окна диалога Project Options предназначена для управления формами

На данной вкладке содержится ряд элементов управления:

- ❑ два списка: Auto-create forms и Available forms. Первый включает имена форм, которые создаются автоматически. Во втором отображаются имена форм, содержащихся в проекте, но не создаваемых при запуске приложения;
- ❑ кнопки:
 - < и > — для перемещения выделенной формы из одного списка в другой;
 - << и >> — для перемещения всех форм, находящихся в одном списке, в другой список;
- ❑ выпадающий список Main form, с помощью которого можно выбрать из всех форм проекта главную форму.

ПРИМЕЧАНИЕ

Главная форма обязательно должна быть автоматически создаваемой. Поэтому если выбрать в списке Main form форму, находящуюся в списке Available forms, она будет автоматически перемещена в список Auto-create forms.

Управлять процессом создания форм можно также путем редактирования кода главного файла проекта. Для всех автоматически создаваемых форм перед выполнением метода Run класса Application сначала выполняются методы CreateForm этого же класса. Если удалить строку, создающую форму, то эта форма перемещается в список Available forms.

СОВЕТ

Главной является форма, создаваемая первой, поэтому, изменяя порядок вызова методов `CreateForm`, можно управлять выбором главной формы приложения.

Если форма не содержится в списке **Auto-create forms**, то перед ее отображением (с помощью методов `Show` или `ShowModal`) необходимо создать экземпляр класса формы, вызвав конструктор `Create` этого класса. По завершении работы с формой следует освободить занимаемые ею ресурсы, вызвав деструктор `Release`. Фрагмент кода программы, открывающий форму, не содержащуюся в списке **Auto-create forms**, должен выглядеть примерно так:

```
Form2:=TForm2.Create(Application);  
Form2.ShowModal;  
Form2.Release;
```

Работа с группой проектов

В Delphi существует возможность объединения нескольких проектов в одну группу. Создание группы проектов полезно, например, в случае если приложение использует динамически загружаемые библиотеки, разрабатываемые совместно с ним. И приложение и библиотека DLL являются отдельными проектами, поэтому при совместной разработке логично их объединить в одну группу проектов.

Создание группы проектов

Для создания новой группы проектов выберите команду **File ► New ► Other** и затем в открывшемся окне хранилища объектов выберите элемент **Project Group** (рис. 13.5).

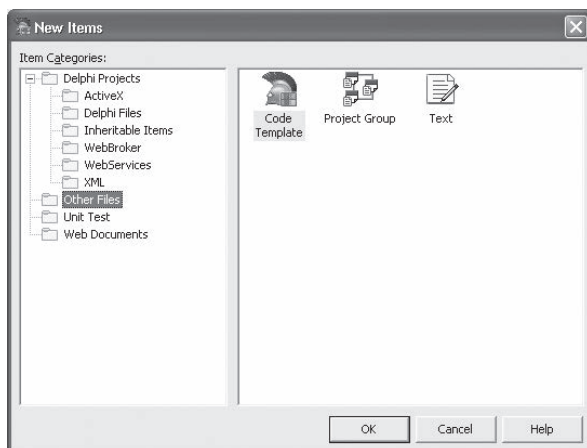


Рис. 13.5. Создание группы проектов

В окне менеджера проектов появится пустая группа проектов, в которую с помощью команд контекстного меню можно добавить существующие и новые проекты.

Чтобы добавить в группу новый проект, выберите команду **Add New Project** контекстного меню менеджера проектов. Созданные ранее проекты добавляются в группу с помощью команды **Add Existing Project** контекстного меню (рис. 13.6).

Группу проектов можно также создать на основе уже существующего проекта. Для реализации этого достаточно открыть окно менеджера проектов с помощью команды **View ► Project Manager** и добавить к существующему проекту другие проекты (новые или созданные ранее).

При создании нового проекта для него автоматически создается своя группа, что автоматически отражается в окне менеджера проектов.

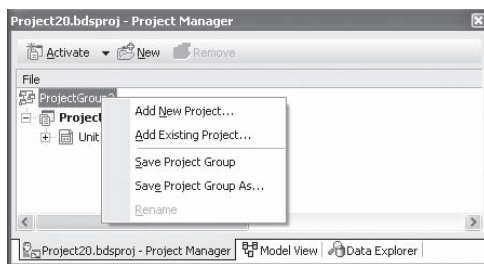


Рис. 13.6. Контекстное меню менеджера проектов

При создании группы проектов генерируется файл ее описания, имеющий расширение **BPG** и содержащий информацию о проектах, входящих в группу.

ПРИМЕЧАНИЕ

Можно считать, что единичный проект также является группой проектов, включающей в себя лишь один проект. Однако файл описания группы проектов создается лишь в том случае, если в группу входит несколько проектов.

Управление группой проектов

Управление группой проектов осуществляется с помощью менеджера проектов. В его окне отображается структура как группы в целом, так и каждого отдельного проекта. В качестве иллюстрации сказанного на рис. 13.7 приводится окно менеджера проектов для группы, включающей в себя приложение и библиотеку **DLL**. В нем проект приложения содержит одну форму, один модуль, связанный с формой, и один модуль, не относящийся к формам. Проект библиотеки **DLL** содержит один модуль, не связанный с формой.

Из всех проектов, содержащихся в группе, только один может быть активным. Именно активный проект запускается на выполнение командой **Run ► Run**. Выбор активного проекта производится в окне менеджера проектов одним из следующих способов:

- ☐ щелчком на кнопке **Activate**;
- ☐ выбором проекта из выпадающего списка;
- ☐ двойным щелчком на имени проекта.

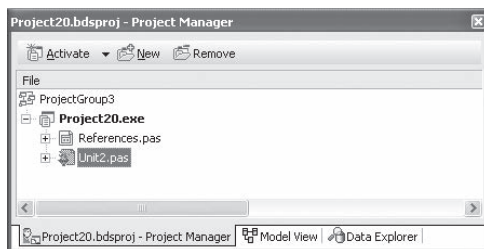


Рис. 13.7. Окно менеджера проектов

Настройка параметров проекта

Для настройки параметров проекта используется команда **Project ► Options**, открывающая окно диалога **Project Options**. Оно содержит семь вкладок: **Forms**, **Application**, **Compiler**, **Compiler Messages**, **Linker**, **Directories/Conditionals**, **Version Info**, и **Packages**. Каждая из перечисленных вкладок включает ряд настроек, влияющих на проект:

- ☐ вкладка **Forms** уже была рассмотрена нами выше, в разделе «Управление формами проекта»;
- ☐ на вкладке **Directories/Conditionals** определяются пути размещения скомпилированных файлов;
- ☐ вкладка **Version Info** дает возможность добавлять к проекту информацию о версии приложения;
- ☐ вкладка **Packages** используется для настроек пакетов (см. раздел «Пакеты» в главе 8);
- ☐ вкладка **Compiler Messages** содержит список сообщений компилятора, которые будут выводиться при отладке.

ПРИМЕЧАНИЕ

В процессе отладки Delphi выдает достаточно много сообщений. В этой массе часто трудно найти нужную информацию. Тем более что компилировать приложение в процессе отладки приходится не один раз. Для упрощения работы мы рекомендуем отключать предупреждения и замечания, не существенные на определенном этапе отладки, не забывая затем включать их обратно.

Назначение элементов управления, размещенных на остальных вкладках, рассмотрено ниже.

Вкладка Application

Вкладка **Application** (рис. 13.8) содержит ряд настроек, управляющих внешним видом программы во время выполнения:

- ☐ **Title** — название приложения, которое отображается в виде подсказки при наведении указателя мыши на свернутое окно программы;
- ☐ **Help file** — имя справочного (help) файла приложения;

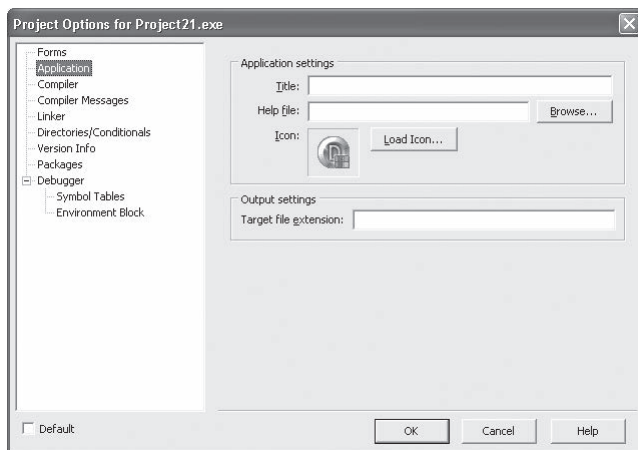


Рис. 13.8. Вкладка Application окна диалога Project Options

- ❑ Icon — это поле отображает текущий значок приложения;
- ❑ Load Icon — щелчок на этой кнопке открывает окно диалога для загрузки файла значка приложения (это должен быть файл с расширением ICO);
- ❑ Target file extension — расширение файла приложения. Обычно компилятор автоматически определяет тип файла (EXE или DLL), однако в некоторых случаях его следует задавать. Например, при разработке элементов ActiveX компилятор должен создать файл с расширением OCX.

Вкладка Compiler

Вкладка Compiler (рис. 13.9) содержит настройки, предназначенные для оптимизации работы компилятора. Данные настройки могут быть также изменены с помощью директив компилятора:

- ❑ Optimization — включает режим оптимизации генерируемого компилятором кода, {\$O};
- ❑ Aligned record fields — включает выравнивание данных по двойному слову, {\$A}. Установка данного флажка приводит к повышению скорости выполнения программы при одновременном увеличении объема программы;
- ❑ Pentium-save FDIV — генерирует код, программно исправляющий ошибку вычисления, допускаемую первыми процессорами Pentium, {\$U};
- ❑ Range checking — проверяет нахождение значения переменной в допустимом диапазоне, {\$R};
- ❑ I/O checking — проверяет наличие ошибок при выполнении операций ввода/вывода, {\$I};
- ❑ Overflow checking — диагностирует ошибки переполнения при выполнении целочисленных операций, {\$Q};

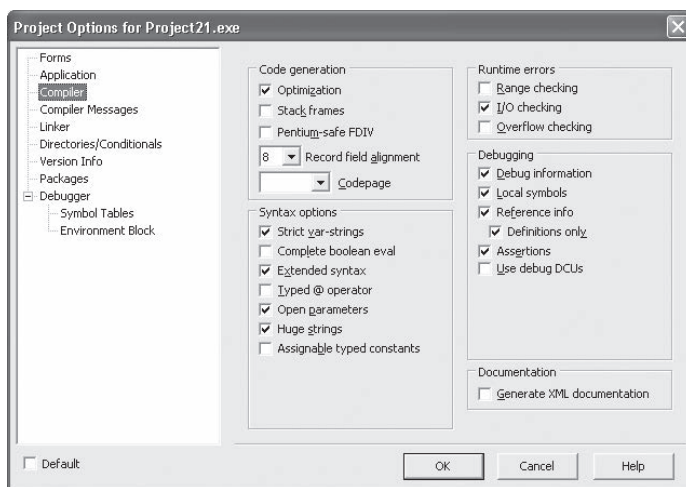


Рис. 13.9. Вкладка Compiler окна диалога Project Options

- ☐ **Strict var-strings** — проверяет соответствие типов формальных и фактических параметров строкового типа, `{ $V }`. Данная настройка выполняется только в случае установки флажка **Open parameters**;
- ☐ **Complete boolean eval** — полное вычисление логического выражения, даже если результат известен заранее, `{ $B }`;
- ☐ **Extended syntax** — допускает вызов функции как процедуры (без выполнения операции присваивания). Результат, возвращаемый функцией, игнорируется, `{ $X }`;
- ☐ **Typed @ operator** — определяет, типизированный или нет указатель возвращается оператором `@`, `{ $T }`;
- ☐ **Open parameters** — допускает использование параметров-«открытых строк» (то есть длина строки, передаваемой в качестве строкового параметра, может отличаться от длины, заданной при объявлении формального параметра), `{ $P }`.

ПРИМЕЧАНИЕ

Директивы компилятора помещаются в тексте программы. Они заключаются в фигурные скобки и состоят из символа `$` и следующей за ним буквы латинского алфавита (идентифицируемой с назначаемым режимом), за которой следует либо знак «+» (включает режим), либо знак «-» (выключает режим). Директивы компилятора имеют более высокий приоритет, чем установки IDE.

Вкладка Linker

Вкладка **Linker** (рис. 13.10) содержит настройки, влияющие на компоновку проекта:

- ☐ флажок **Generate console application**. В случае его установки создается консольное приложение;

- ❑ флажок **Include remote debug symbols** следует установить, если необходимо использовать удаленную отладку приложения;
- ❑ **Memory sizes** — на этой вкладке указывается минимальный и максимальный размеры стека;

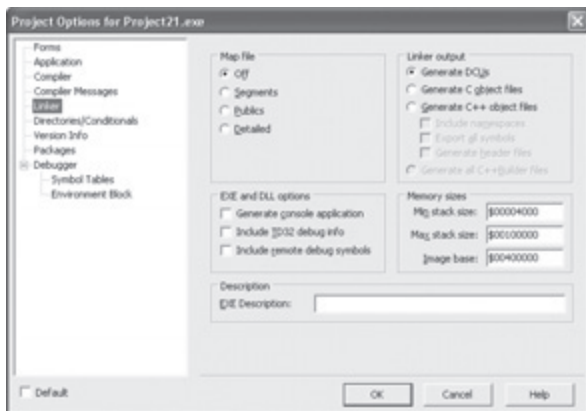


Рис. 13.10. Вкладка Linker окна диалога Project Options

- ❑ в поле **EXE Description** можно указать комментарий к исполняемому файлу — до 255 символов.

Компиляция и запуск приложения

Компиляция и запуск приложения осуществляются с помощью команд меню **Project** и **Run**. Следует отметить, что в Delphi перед запуском приложение обязательно полностью компилируется, даже если оно запускается из-под отладчика Delphi IDE. При попытке запустить не откомпилированный проект его компиляция будет выполнена автоматически.

Команды компиляции проекта

Все рассматриваемые ниже команды расположены в меню **Project**. Каждая из них имеет альтернативную ей комбинацию оперативных клавиш, которые указаны в скобках после названия команды:

- ❑ команда **Compile (Ctrl+F9)** компилирует файлы, в которые были внесены изменения после выполнения последней компиляции. Затем выполняется компоновка приложения, в результате чего формируется либо исполняемая программа, либо динамически связываемая библиотека. Для группы проектов выполняется компиляция активного проекта;
- ❑ команда **Build** отличается от команды **Compile** только тем, что при ее выполнении компилируются все файлы проекта, независимо от того, вносились в них изменения после последней компиляции или нет. Для группы проектов выполняется компиляция активного проекта;

- ❑ команда **Add to Respository** добавляет проект в хранилище ваших удачных решений. Код, помещенный в хранилище, возможно использовать при построении других приложений;
- ❑ команда **Compile All Projects** выполняет компиляцию всех проектов, входящих в текущую группу, в том порядке, в каком они перечисляются в окне менеджера проектов. При выполнении данной команды компилируются только те файлы, которые изменялись со времени последней компиляции;
- ❑ команда **Build All Projects** аналогична предыдущей команде, только выполняет компиляцию всех файлов, независимо от того, изменялись они или нет.

Команды запуска приложения

Все команды запуска приложения расположены в меню **Run**. Большинство из них используется для отладки приложения:

- ❑ команда **Run (F9)** запускает текущий проект на выполнение. Если проект не откомпилирован (или в проект вносились изменения после последней компиляции), то перед запуском автоматически проводится его компиляция;
- ❑ команды **Step Over (F8)** и **Trace Into (F7)** используются в режиме отладки для пошагового выполнения программы. При выполнении команды **Step Over** строки, содержащие вызов процедуры или функции, выполняются за один шаг, без прохождения отдельных строк вызываемых подпрограмм. Пошаговая отладка с «заходом» в вызываемые процедуры и функции выполняется с помощью команды **Trace Into**;
- ❑ команда **Trace to Next Source Line (Shift+F7)** обеспечивает пошаговую отладку подпрограмм косвенного вызова;
- ❑ команда **Run to Cursor (F4)** запускает программу и выполняет ее до строки, на которой расположен текстовый курсор в редакторе кода.

Глава 14

Коллективная разработка приложений

Создание крупных информационных систем требует согласованной работы целой группы программистов.

Несколько лет назад проблемы организации взаимодействия отдельных разработчиков при создании крупных проектов были актуальны в основном для крупных фирм — производителей программного обеспечения. Однако с появлением и развитием систем быстрой разработки приложений (RAD, Rapid Application Development) ситуация изменилась. Внедрение средств RAD позволяет повысить производительность труда как отдельных программистов, так и рабочих групп. Благодаря этому полный цикл разработки крупных проектов может выполняться существенно меньшими коллективами. Таким образом, проблемы обеспечения согласованной работы отдельных программистов, выполняющих разработку крупного проекта, стали актуальны и для небольших рабочих групп. Это нашло свое отражение на рынке программного обеспечения.

Наличие в системе быстрой разработки приложений эффективных средств, обеспечивающих поддержку коллективной разработки, становится одним из факторов, повышающих конкурентоспособность данного программного продукта.

Структура средств коллективного проектирования и решаемые ими задачи

Рассмотрим спектр задач, решаемых системами обеспечения коллективной разработки приложений. Основными из них являются: обеспечение управляемости и контроля процессов разработки, сопровождение приложения. Для этого необходимо обеспечить выполнение как минимум двух функций:

- ☐ регистрации всех изменений, вносимых в проект;
- ☐ централизованного хранения файлов проекта.

Под *проектом* мы будем понимать множество файлов с исходными текстами программ, а также файлов ресурсов и всех прочих файлов (исполняемые файлы, библиотеки DLL, ActiveX, объектные модули, документы), необходимых для выполнения компиляции и запуска приложения.

Обе указанные выше функции реализуются с помощью так называемых *систем контроля версий проектов* (PVCS, Project Version Control Systems). Системой контроля версий проектов называется комплекс программного обеспечения, назначением которого является централизованное хранение и обработка всех или большей части объектов (файлов), из которых состоит проект. Для решения задач управления разработкой проекта применяются методы и средства, обеспечивающие:

- ❑ идентификацию состояния как отдельных компонентов, так и проекта в целом;
- ❑ контроль над вносимыми в компоненты и структуру проекта изменениями;
- ❑ координированное управление всеми составляющими проекта.

Идентификация

Чтобы осуществлять управление объектами, необходимо их идентифицировать. При идентификации объектов в системах PVCS используется понятие *версии*. Версией проекта называется некий уникальный идентификатор, обозначающий текущий номер разработки. Так как в отдельные составляющие проекта во время разработки могут вноситься изменения, каждому из помещенных в хранилище PVCS объектов присваиваются идентификаторы версии самого объекта и версии проекта в целом. Это позволяет определить, какие именно файлы должны быть использованы для сборки заданной версии приложения.

Хранилище файлов и контроль за изменением файлов

Хранилища объектов, используемые PVCS, могут организовываться с использованием самых разных технологических решений, вплоть до применения специальных баз данных. Возможно также использование одной PVCS нескольких способов хранения одновременно.

В процессе работы над проектом промежуточное состояние файлов периодически сохраняется в хранилище проекта. Одновременно с этим ведутся записи о времени сохранения и соответствии друг другу нескольких вариантов разных файлов проекта. Кроме этого, фиксируются имена разработчиков, ответственных за тот или иной файл, состав файлов промежуточных версий проекта и пр. Это позволяет при необходимости вернуться к какому-либо из предыдущих состояний файла (например, при обнаружении ошибки, которую в данный момент трудно исправить).

В хранилище обычно содержатся все версии файлов проекта, любая из которых может быть оттуда извлечена. Во избежание бесполезного расходования дискового пространства обычно сохраняются только изменения базовой версии файла.

Блокировки

Система управления разработкой обязательно должна обеспечивать функции *блокировки*. Блокировка преследует две основные цели.

1. Обеспечение централизованного управления файлами проекта. В этом случае задачей блокировки является устранение возможности случайной или намеренной модификации исходных текстов файлов проекта после его отладки и принятия версии (всего проекта или одной из его частей) как окончательной. Для защиты финальной версии проекта (или отдельных его составляющих) от модификации обычно используются различные схемы с применением паролей для снятия блокировки, шифрование и некоторые другие.
2. Исключение конфликтов при одновременной модификации одной и той же составляющей несколькими участниками проекта. Возможность таких конфликтов обусловлена тем, что практически никогда нельзя разделить проект на несколько полностью изолированных друг от друга частей. Поэтому ряд файлов проекта может одновременно относиться к нескольким частям проекта и, следовательно, их могут модифицировать разные программисты.

Последовательность работы с PVCS

Мы рассмотрели основные функции, выполняемые PVCS. Теперь приведем последовательность операций, выполняемых при работе с PVCS.

1. Ввод исходной информации о структуре проекте и его составляющих. Создание первой версии проекта в хранилище PVCS.
2. Определение авторов проекта, назначение ответственных за отдельные составляющие проекта, задание связей между отдельными объектами, настройка прав доступа (возможность чтения, внесения изменений, удаления и т. п.) разработчиков как к отдельным объектам, так и ко всему проекту в целом.
3. Выдача отдельных составляющих проекта для изменения с учетом прав доступа и возможностью блокировки получения копии этой версии объекта до момента помещения модифицированного объекта в хранилище.
4. Занесение в хранилище PVCS измененных (или вновь созданных) составляющих проекта с присвоением им номера версии самой составляющей, а также проекта в целом.
5. Выдача всех составляющих проекта заданной версии для компиляции либо всего проекта, либо отдельного его компонента.

Существует множество инструментов, предназначенных для управления версиями проектов. Наиболее популярны StarTeam, TeamSource, Microsoft Visual Source Safe QSC Team Conerence.

В качестве примера рассмотрим систему управления версиями TeamSource фирмы Borland, получившую широкое распространение благодаря тому, что входила в поставку 5–7 версий Delphi.

Программа TeamSource

Хотя система TeamSource является средством групповой разработки, она может использоваться и в однопользовательском режиме.

TeamSource позволяет решать большинство задач, о которых мы говорили выше. Хранилище составляющих проекта в TeamSource реализовано по файловому принципу. Имеется возможность создания собственного расширения для управления хранилищем версий, например для использования базы данных в качестве такого хранилища. Поскольку система TeamSource поставляется вместе с исходными текстами, написание расширений не представляет собой сверхсложной задачи.

Структура системы TeamSource

Функционирование системы TeamSource основано на использовании подключаемых модулей (plug-ins), разрабатываемых на основе TeamSource Extension API. Все операции над отдельными составляющими проекта осуществляются при помощи так называемых *контроллеров*, посредством которых реализуется доступ к хранилищу версий файлов проекта, генерация и обработка номеров версий файлов, заполнение комментариев к файлам и проектам, а также ряд других операций. Контроллеры располагаются в подключаемых модулях расширения, представляющих собой файлы с расширением tsx. В базовую поставку входят два подключаемых модуля:

- ❑ `izlib.tsx` — основной контроллер версий, осуществляющий хранение файлов проекта в библиотеках формата ZLib (совместимого с форматом zip, но в отличие от последнего не требующего лицензирования);
- ❑ `tscomments.tsx` — контроллер ввода комментариев к файлам и проектам.

Идентификация проекта и его составляющих в TeamSource

Версии проекта и его составляющих назначаются контроллером версий TeamSource. Номер версии составляющих проекта состоит из двух двузначных чисел. Основной контроллер формирует версию каждой из составляющих проекта в момент помещения ее в хранилище, увеличивая на единицу правую часть номера версии, исходное значение которой (для первой версии файла, помещенной в хранилище) равно 1.0. Когда правая часть достигает значения 99, левая увеличивается на единицу, а правая обнуляется.

ПРИМЕЧАНИЕ

Можно также реализовать свой собственный генератор версий, создав специальное расширение TeamSource.

Версия проекта задается при его описании и не генерируется автоматически.

Отдельные версии проекта можно отмечать путем установки закладок (Bookmark). Установка закладки отмечает текущую версию всех составляющих проекта.

Использование закладок в значительной степени упрощает управление файлами при проведении сборки проекта, а также при указании текущей версии проекта. При необходимости закладку можно снабдить комментариями.

Хранилище TeamSource

Как уже отмечалось выше, хранилище TeamSource организовано по файловому принципу. Для каждого проекта выделяется каталог, называемый корневым (root), в котором создается структура подкаталогов и файлов, соответствующая файлам и каталогам, включенным в описание проекта. Изначально для каждого корневого каталога создается следующая структура файлов и подкаталогов:

- ❑ Archives — каталог, в котором содержатся версии файлов проекта. Файлы хранятся в архивированном виде, в формате ZLib. Каталог содержит все версии каждого из файлов проекта. Имена присваиваются файлам по следующему принципу: к имени исходного файла (включая и расширение) добавляется расширение .z (например, файл project.dpr будет иметь имя project.dpr.z). Кроме файлов проекта данный каталог содержит еще два файла:
 - файл с информацией о проекте (название проекта, версия TeamSource и уникальное имя контроллера версий, получаемое от соответствующего модуля расширения);
 - файл, содержащий версию проекта;
- ❑ History — каталог, в котором сохраняется информация об изменениях файлов в хранилище. Имена файлов в этом каталоге имеют вид <код даты и времени>.<имя рабочей станции>. Файл истории содержит имя пользователя, работавшего с проектом, дату и время сеанса, а также список измененных файлов;
- ❑ Locks — каталог, предназначенный для хранения информации о блокировках. Обычно содержит один файл lockinfo.dat;
- ❑ logs.txt — журнал работы с проектом;
- ❑ summary.txt — результирующие данные о каждом сеансе работы с проектом.

Работа с программой TeamSource

Хотя установка TeamSource производится отдельно от установки Delphi, возможна интеграция систем контроля версий и Delphi. В этом случае команды системы контроля версий станут доступны из подменю Team главного меню Delphi.

Первый запуск TeamSource

При первом запуске программы TeamSource открывается окно диалога Welcome to TeamSource, в котором запрашивается ряд параметров, необходимых для идентификации пользователя:

- ❑ имя пользователя (user name) — данный параметр не может быть изменен в окне Welcome to TeamSource и является сетевым именем компьютера, на который установлена TeamSource;
- ❑ полное имя (full name) — полное имя пользователя, которое используется внутри проектов TeamSource;
- ❑ адрес электронной почты (e-mail) пользователя.

Хотя окно диалога Welcome to TeamSource открывается только один раз (при первом запуске программы), параметры, которые в нем задаются, могут быть изменены впоследствии с помощью команд меню программы TeamSource.

Настройка параметров программы TeamSource

После заполнения полей окна диалога Welcome to TeamSource открывается главное окно программы TeamSource (рис. 14.1). Внешне оно очень похоже на главное окно программы Microsoft Outlook.

Меню TeamSource содержит всего пять пунктов. Рассмотрим их более подробно:

- ❑ меню File включает стандартный набор команд: создание нового проекта — New Project; открытие ранее созданного проекта — Open Project; закрытие проекта — Close Project и выход из программы — Exit;
- ❑ команды меню Project становятся доступными только в том случае, если открыт проект TeamSource. Поскольку сразу после запуска ни один проект не открыт, то все команды данного меню пользователю недоступны. Мы их рассмотрим несколько ниже;
- ❑ команды меню View определяют режим работы с проектом. Пока проект не открыт, выбор той или иной команды этого меню ни на что не влияет. Обратите внимание, что команды этого меню дублируются кнопками панели инструментов Views, расположенной в левой части окна программы TeamSource;
- ❑ меню Options содержит единственную команду Preferences, с помощью которой задаются настройки программы TeamSource;
- ❑ меню Help обеспечивает доступ к справочной системе.

Перед тем как создавать проект, познакомимся с основными типами настроек, выполняемых с помощью команды Options ► Preferences. При выборе данной команды открывается окно диалога Preferences, содержащее две вкладки (рис. 14.2). На первой вкладке General (см. рис. 14.2, а) задаются параметры пользователя программы TeamSource: имя пользователя и адрес электронной почты (которые задавались при первом запуске программы), а также параметры, характеризующие режим работы программы (группа флажков File Handling):

- ❑ Update local file on each checkin (get after put) — выполнять обновление локальных копий файлов проекта при каждой записи в хранилище. Данная настройка используется в том случае, когда контроллер версий осуществляет автоматическую модификацию исходного текста при записи в хранилище (операция Check-In), например проставляет дату и время выполнения Check-In, версию и т. п.;

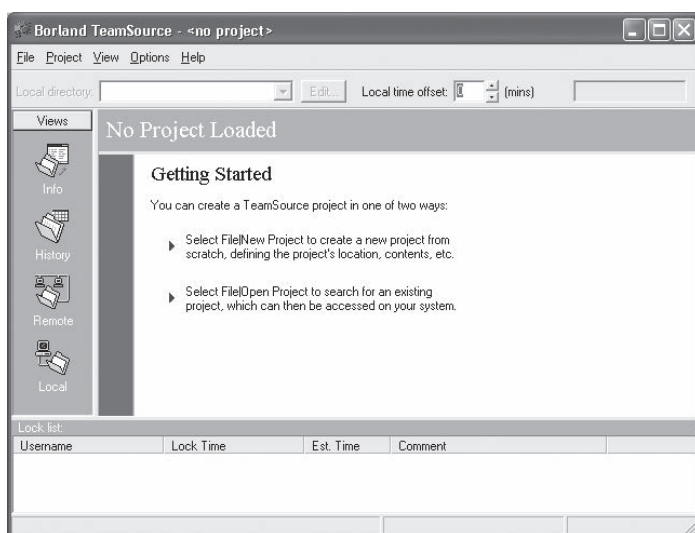


Рис. 14.1. Главное окно программы TeamSource

- ❑ Ignore spaces in files during compares — игнорировать пробелы в начале и в конце строк при сравнении текстовых файлов. Данная установка позволяет избежать выделения подобных строк как измененных в окне сравнения версий;
- ❑ Automatically import comments during reconcile — автоматически импортировать комментарии из окна Recommended changes при выполнении операции Check-In.

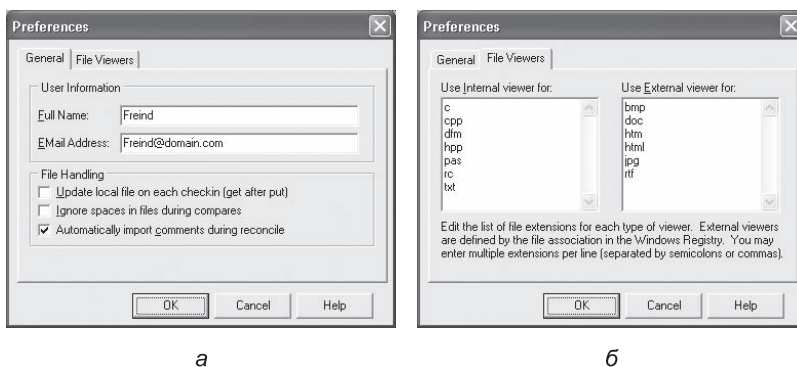


Рис. 14.2. Вкладки окна диалога Preferences

На вкладке File Viewers окна диалога Preferences указываются средства просмотра файлов различных типов (см. рис. 14.2, б). По умолчанию файлы с расширениями c, cpp, dfm, hpp, pas, rc и txt располагаются в списке Use Internal viewers for. Это говорит о том, что они просматриваются с использованием внут-

ренных средств программы TeamSource. В список Use External viewers for помещаются файлы, содержимое которых отображается с использованием внешних программ, ассоциированных с расширением файла.

ПРИМЕЧАНИЕ

При редактировании параметров этой страницы следует учитывать, что внутренние средства просмотра TeamSource предназначены для отображения только текстовых файлов в формате ASCII.

Создание проекта

Для создания нового проекта выберите команду File ► New Project. При этом откроется окно диалога New Project, в котором предлагается выбор из двух вариантов: создать новый проект (предлагается по умолчанию) или импортировать данные из уже существующего проекта. Поскольку мы создаем новый проект, то следует оставить предлагаемый по умолчанию вариант Create new project from scratch.

После щелчка на кнопке OK диалога New Project запускается мастер создания проекта. Создание проекта выполняется за семь шагов.

1. На первом шаге (рис. 14.3) требуется задать:

- имя проекта (это имя будет затем использоваться во всех операциях TeamSource);
- имя файла проекта (без указания пути);
- контроллер версий (выбирается из списка, в котором отображаются все доступные контроллеры, подключенные к TeamSource).

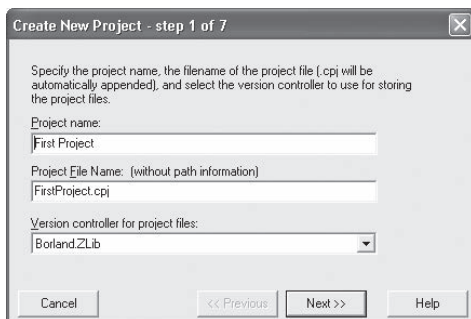


Рис. 14.3. Первый шаг создания проекта TeamSource

ПРИМЕЧАНИЕ

Указанное имя проекта в дальнейшем можно изменить только редактированием файла проекта.

2. На втором шаге (рис. 14.4) указывается каталог, в котором будут храниться файлы проекта.

ПРИМЕЧАНИЕ

Каталог проекта может располагаться не только на локальном, но и на удаленном компьютере.

3. На третьем шаге (рис. 14.5) задаются каталоги для хранилища версий (Archives), файлов истории (History) и блокировок (Locks).

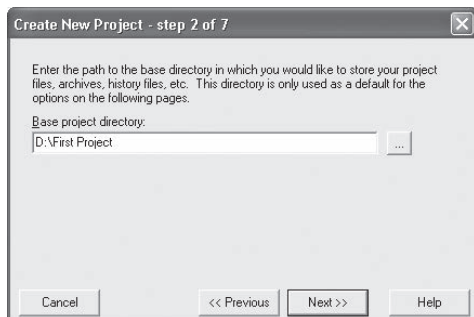


Рис. 14.4. Второй шаг создания проекта TeamSource

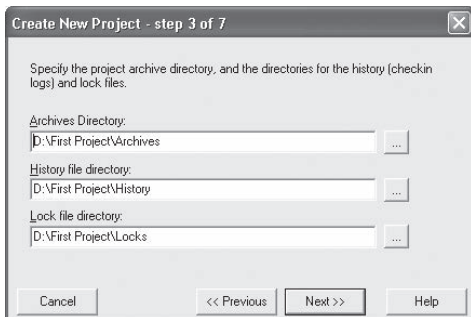


Рис. 14.5. Третий шаг создания проекта TeamSource

Любой из этих трех каталогов может быть размещен в любом месте (даже на разных рабочих станциях в сети). Однако обычно удобнее всего располагать их в каталоге проекта, как и предлагается по умолчанию.

4. На следующем (четвертом) шаге (рис. 14.6) задаются параметры режима создания резервной копии хранилища версий.

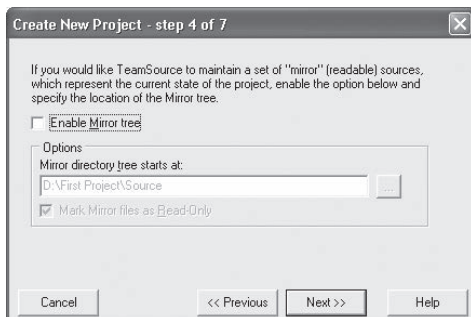


Рис. 14.6. Четвертый шаг создания проекта TeamSource

Если установить флажок **Enable Mirror tree**, то в указанном ниже каталоге будет создаваться «зеркало» (mirror) хранилища версий. Это делается для повышения надежности хранения файлов. При установке флажка **Mark Mirror files as Read-Only** файлы, сохраняемые в каталоге «зеркала», будут помечаться атрибутом read-only (только для чтения).

5. На пятом шаге (рис. 14.7) задаются имена и местоположение на диске (или в сети) файлов истории и журнала. Кроме того, указывается SMTP-сервер,

который будет использоваться для рассылки сообщений подсистемой оповещения.

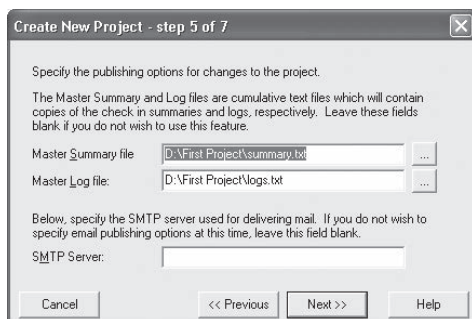


Рис. 14.7. Пятый шаг создания проекта TeamSource

6. Если на пятом шаге указан SMTP-сервер, то при щелчке на кнопке **Next** мастер создания проекта переходит к шестому шагу: заданию адресов электронной почты для рассылки информации (рис. 14.8). В противном случае (если SMTP-сервер не задан) будет произведен переход сразу к седьмому шагу.

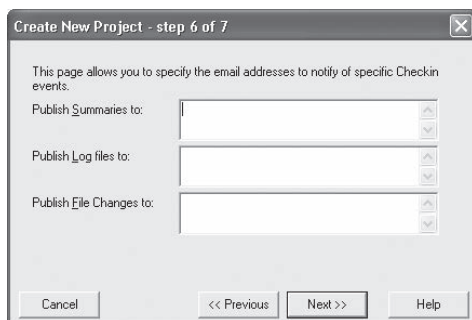


Рис. 14.8. Шестой шаг создания проекта TeamSource

Поле, в котором задан адрес электронной почты, определяет, какую информацию будет получать каждый из участников проекта при изменениях в хранилище версий.

7. На седьмом шаге выводится окно (рис. 14.9), содержащее всю информацию о проекте, введенную на предыдущих шагах. При обнаружении ошибки можно вернуться назад (кнопка **Previous**) и внести необходимые коррективы.

После щелчка на кнопке **Finish** TeamSource запрашивает данные о локальных каталогах проекта и о типах файлов, обрабатываемых контроллером версий. Данная информация вводится с помощью специального мастера **Content Wizard** (рис. 14.10). Окно диалога **Local Directory Wizard** позволяет указывать список типов файлов, обрабатываемых контроллером версий для каждого из локальных каталогов. Можно также удалить любой из каталогов, кроме самого верхнего.

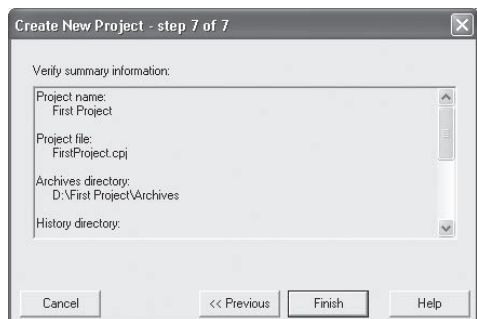


Рис. 14.9. Заключительный шаг создания проекта TeamSource

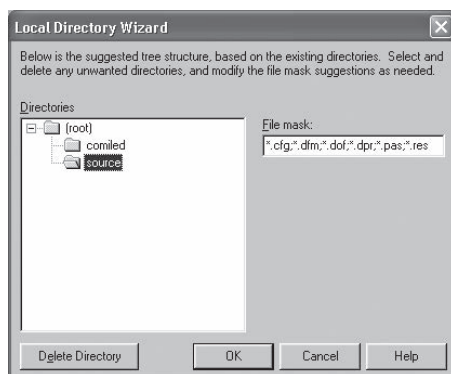


Рис. 14.10. Окно мастера настройки структуры локальных каталогов Local Directory Wizard

ПРИМЕЧАНИЕ

Вызов мастера Content Wizard может быть осуществлен после создания проекта с помощью команды меню Project ► Content Wizard. Доступ к данной команде возможен только при включенной блокировке файлов проекта. Как производится включение и снятие блокировки, будет рассмотрено несколько ниже.

Настройка параметров проекта

Параметры проекта настраиваются с помощью окна диалога Project Options, открываемого командой меню Project ► Options. Данная команда доступна при любом режиме работы TeamSource. Однако при снятой блокировке файлов проекта никаких изменений в окне Project Options производить нельзя, о чем свидетельствует надпись Read Only в его левом нижнем углу. Поэтому для выполнения настройки проекта в первую очередь необходимо включить блокировку. О том, как это сделать, будет рассказано ниже, в разделе «Работа с проектом TeamSource». После задания блокировки настройки в окне диалога Project Options становятся доступными для редактирования.

Окно диалога Project Options содержит четыре вкладки. Рассмотрим, какие установки задаются с помощью каждой из них.

На вкладке General (рис. 14.11) расположены следующие элементы управления:

- ❑ текстовое поле Project name отображает имя текущего проекта. Данное поле недоступно для редактирования;
- ❑ текстовое поле Version info file name отображает имя файла, содержащего информацию о версии проекта. Этот файл создается в корневом каталоге локального проекта автоматически при выполнении операции выдачи всех составляющих проекта заданной версии для сборки приложения (операция Pull);
- ❑ при установке флажка Detect new local directories TeamSource будет проверять, не содержит ли корневой каталог локального проекта новых подкаталогов, и в случае их обнаружения автоматически запускать мастер Content Wizard;

- ❑ при установке флажка **Require summary comments** TeamSource будет требовать ввода общих комментариев при выполнении операции записи файлов проекта в хранилище (операция **Check-In**);
- ❑ установка флажка **Require file comments** приведет к тому, что TeamSource при выполнении операции **Check-In** будет требовать ввода комментариев к каждому файлу;
- ❑ текстовое поле **Build numbers** предназначено для задания исходного номера версии проекта.

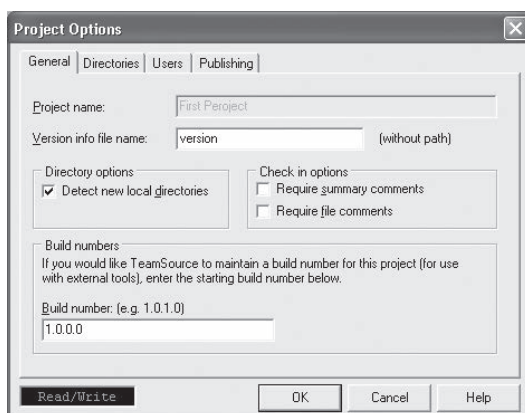


Рис. 14.11. Вкладка General окна диалога настройки параметров проекта

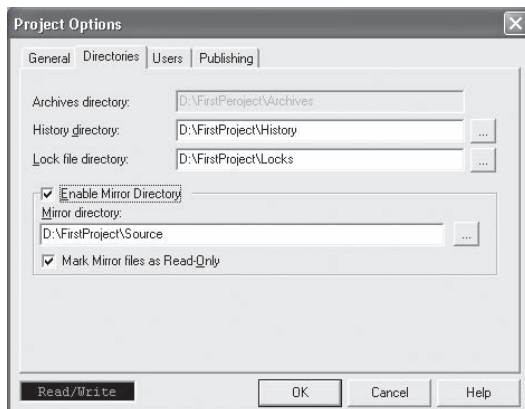


Рис. 14.12. Вкладка Directories окна диалога настройки параметров проекта

Вкладка **Directories** (рис. 14.12) используется для задания путей к файлам проекта:

- ❑ в текстовом поле **Archives directory** выводится информация о местоположении файлов хранилища версий. Данное поле недоступно для редактирования;

- ☐ поле ввода History directory позволяет изменить каталог, содержащий файлы истории;
- ☐ в текстовом поле Lock file directory можно изменить путь к файлам блокировки проекта;
- ☐ флажок Enable Mirror Directory позволяет включить (или выключить) режим создания резервной копии («зеркала») проекта;
- ☐ текстовое поле Mirror directory доступно для редактирования при включенном флажке Enable Mirror Directory и задает путь к «зеркалу» проекта;
- ☐ флажок Mark Mirror files as Read-Only используется для задания файлам «зеркала» атрибута «только для чтения».

На вкладке Users (рис. 14.13) указываются имена участников проекта и права их доступа к нему. Список Authorized users содержит имя пользователя (столбец Username) и информацию о его правах доступа (столбец Access).

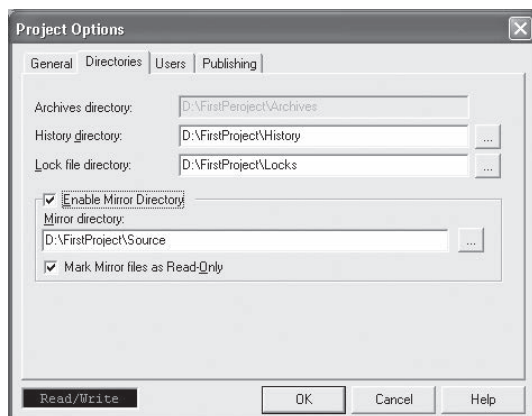


Рис. 14.13. Вкладка Users окна диалога настройки параметров проекта

Для добавления нового участника проекта следует щелкнуть на кнопке Add. При этом открывается окно диалога User Information (рис. 14.14), в котором задаются:

- ☐ имя нового участника проекта — текстовое поле Username;
- ☐ права доступа к проекту — группа переключателей Access Rights:
 - Read-Only — только чтение файлов проекта;
 - Read-Write — возможность модификации файлов проекта;
 - User is an Administrator — пользователь с правами администратора. Этот режим обеспечивает ряд дополнительных возможностей, в частности позволяет создавать не ограниченные по времени блокировки проекта и изменять состав участников проекта.

Установка флажка Allow Guest access to this project на вкладке Users открывает всеобщий доступ к проекту в режиме Read-Only.

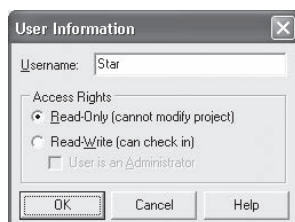


Рис. 14.14. Окно диалога User Information

На вкладке Publishing (рис. 14.15) задаются адреса электронной почты, по которым производится рассылка различного рода информации:

- ☐ сводных отчетов об операциях с хранилищем версий;
- ☐ журналов операций с хранилищем версий;
- ☐ подробных отчетов об изменениях в отдельных файлах при выполнении операций Check-In.

На этой вкладке также указываются имена файлов, содержащих рассылаемую информацию.

ПРИМЕЧАНИЕ

Ряд параметров, общих для всего проекта, можно также задавать в окне диалога Controller Options, которое открывается командой меню Project ► Controllers. Однако обращаться к данным настройкам имеет смысл только в том случае, если используются расширения TeamSource, так как контроллер версий, входящий в поставку TeamSource, не имеет параметров, которые можно настраивать в данном окне.

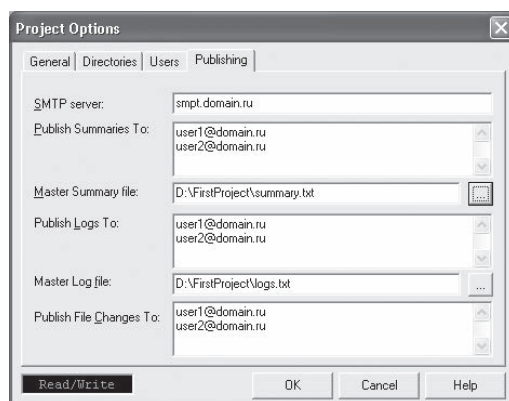


Рис. 14.15. Вкладка Publishing окна диалога настройки параметров проекта

Работа с проектом TeamSource

Работа с проектом TeamSource может выполняться в четырех режимах, которые задаются либо с помощью команд меню View, либо с помощью кнопок панели инструментов Views:

- ❑ **Info** — режим информации. При работе в данном режиме в окне TeamSource отображается информация о текущем проекте. Режим Info нами фактически уже рассмотрен, так как все возможные в нем действия ограничены настройкой параметров программы TeamSource и параметров проекта;
- ❑ **Local Project** — работа с локальной копией проекта;
- ❑ **Remote Project** — работа с хранилищем версий;
- ❑ **History** — работа с историей версий проекта.

Работа с локальной копией проекта

В режиме Local контроллер версий программы TeamSource проверяет совпадение версий и времени создания файлов, находящихся в локальном каталоге, с файлами, расположенными в хранилище версий. Результаты сравнения отображаются в главном окне TeamSource на двух панелях: Recommended changes to your Local project и Recommended changes to the Remote project (рис. 14.16).

В панели Recommended changes to the Remote project приводится список файлов локального проекта, имеющих отличия от копий этих файлов, расположенных в хранилище версий, а также указываются рекомендуемые действия. Причем действия связаны с изменением состава файлов в хранилище. Например, согласно рис. 14.16, в локальном проекте обнаружены следующие изменения:

- ❑ создан новый файл Calculation.pas, отсутствующий в хранилище версий. Рекомендуется занести его в хранилище (Check-In);
- ❑ обнаружены изменения в файлах SQL.exe, SQL_main.dcu, SQL.dpr, SQL.cfg, SQL.dof, SQL_main.dfm, SQL_main.pas. Перечисленные файлы имеют более новую версию или более позднее время создания, чем их аналоги в хранилище. Рекомендуемое действие — занести их в хранилище (Check-In);
- ❑ файл SQLBDE.res удален из локальной копии проекта. Рекомендуемое действие — удалить копии этого файла из хранилища версий.

В панели Recommended changes to your Local project также приводится список файлов локального проекта, отличающихся от соответствующих файлов в хранилище. Однако в данном случае рекомендуются действия по изменению файлов в локальном проекте, а не в хранилище. Например, в случае, соответствующем рис. 14.16, рекомендуется внести изменения в два файла:

- ❑ время создания файла SQL.res отличается от времени создания последней версии этого файла в хранилище. Однако размер и контрольная сумма не изменились. Рекомендуемое действие — привести время создания файла в соответствие с версией в хранилище (Touch);
- ❑ файл SQLBDE.dpr изменен, однако время его создания более раннее, чем у версии, находящейся в хранилище (конфликт версий). Такого рода конфликты не могут быть разрешены автоматически. Поэтому рекомендуется исправить файл вручную (Correct by hand).

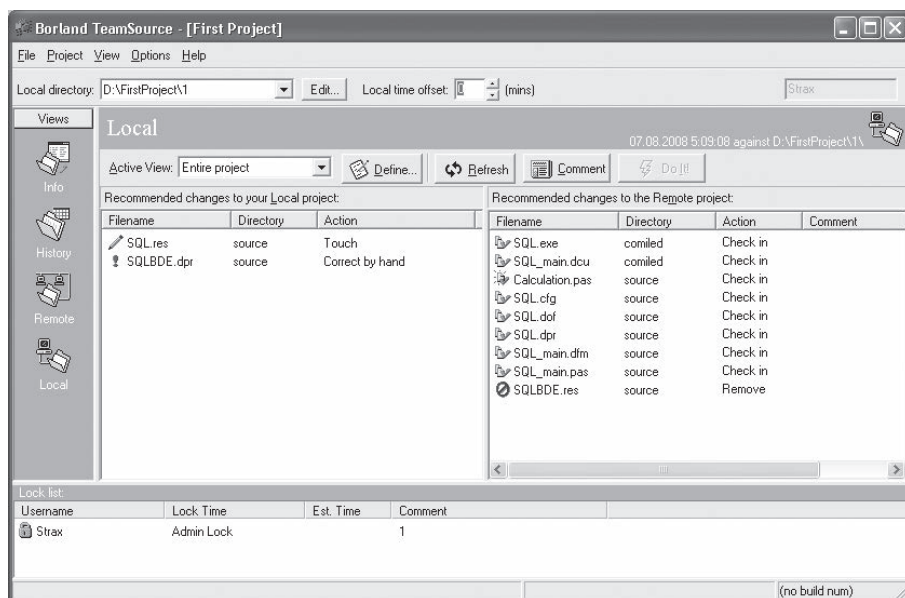


Рис. 14.16. Главное окно TeamSource при работе в режиме Local

Для выполнения рекомендованных действий над файлом выделите его в списке и нажмите на кнопку **Do It!**.

Если после выделения одного или нескольких файлов щелкнуть правой кнопкой мыши на списке, содержащем выбранные файлы, то раскроется контекстное меню, содержащее ряд команд, позволяющих выполнить некоторые дополнительные действия. Состав команд меню несколько различается для разных панелей. Меню, раскрывающееся в панели **Recommended changes to the Remote project**, содержит следующие команды:

- ☐ первый пункт меню в разных случаях может быть различным, но это всегда команда, выполняющая рекомендуемое действие;
- ☐ **View Local Changes** — просмотр изменений в файле локальной копии проекта в сравнении с версией, находящейся в хранилище;
- ☐ **View Remote Changes** — просмотр изменений одной версии файла в хранилище относительно другой версии, также взятой из хранилища;
- ☐ **View All Changes** — просмотр всех изменений (сравниваются локальный файл и разные версии из хранилища);
- ☐ **View File Info** — вывод информации о файле: даты и времени модификации локальной копии, даты и времени модификации копии в хранилище, номера последней версии файла в хранилище;
- ☐ **Ignore (move to other pane)** — игнорировать изменения и переместить файл в другой список;
- ☐ **Revert** — записать файл в хранилище поверх предыдущих версий;

- ❑ Edit File Comment — редактировать комментарии к файлу;
- ❑ Import Comments — импортировать комментарии из предыдущей версии файла;
- ❑ Select All — выделить все файлы, относящиеся к данной панели.

Контекстное меню панели Recommended changes to your Local project содержит только одну команду, отличающуюся от рассмотренных выше: Change File Status — изменить рекомендуемое действие для выделенного файла (файлов).

При выборе в контекстном меню команд просмотра изменений, произошедших от одной версии файла к другой, открывается окно Comparing, в котором отображается текст выбранного файла и указываются произошедшие изменения.

На рис. 14.17 приведен пример окна Comparing для файла, в котором произошло изменение одной строки. В тексте отображаются оба варианта строки. Старый вариант выделяется красным цветом и помечается символом «-». Новый вариант выделяется желтым цветом и помечается символом «+». В нижней части окна Comparing выводится комментарий к измененному файлу.

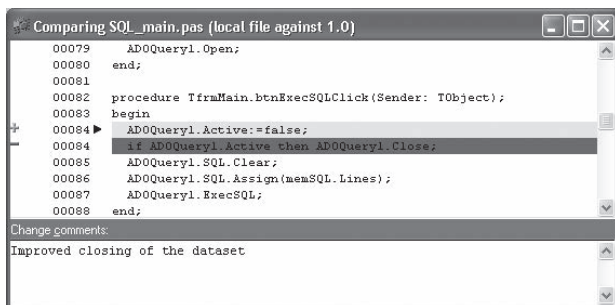


Рис. 14.17. Окно диалога Comparing

Работа с хранилищем версий

В режиме Remote в главном окне TeamSource отображается состав хранилища версий (рис. 14.18). Так же как и в режиме Local, главное окно делится на две панели. В левой панели в данном случае отображается иерархическая структура каталогов хранилища версий. Для обозначения корневого каталога, независимо от его имени, используется имя root. В правой панели отображается список файлов, содержащихся в каталоге, выделенном в левой панели.

При снятой блокировке проекта можно лишь просматривать содержимое хранилища версий в режиме Read-Only. Поэтому в том случае, когда требуется внести изменения в хранилище (например, удалить лишние файлы), следует произвести включение блокировки.

При щелчке правой кнопкой мыши на списке файлов открывается контекстное меню, содержащее восемь команд:

- ❑ View Tip Revision — просмотр текущей версии выделенного файла (файлов). Файл отображается в окне, обладающем возможностями простейшего текстового редактора (рис. 14.19);

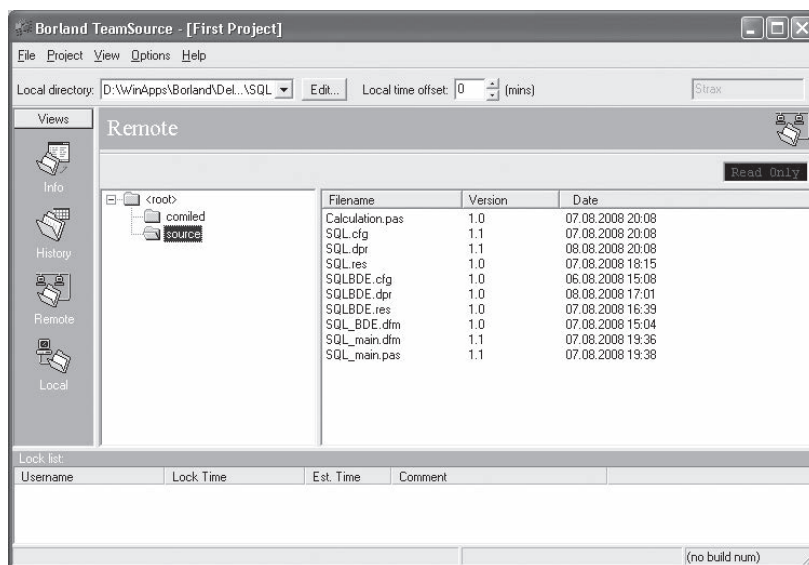


Рис. 14.18. Окно программы TeamSource в режиме Remote

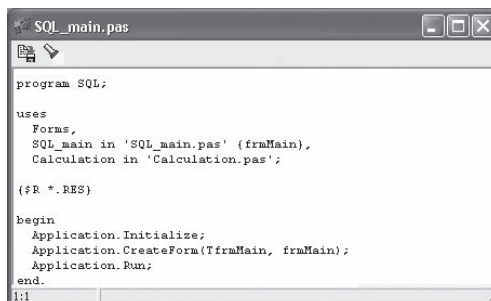


Рис. 14.19. Окно просмотра текстового файла проекта

ПРИМЕЧАНИЕ

Если перед выбором этой команды было выделено несколько файлов, то каждый из них открывается в своем окне.

- ❑ View Any Revision — просмотр любой версии выделенного файла. При выборе этой команды открывается окно диалога Select A Revision (рис. 14.20), в котором предлагается выбрать версию файла среди всех, имеющихся в хранилище. Выбранная версия файла отображается в окне просмотра, полностью аналогичному тому, которое изображено ниже на рис. 14.21;
- ❑ Save Revision As — сохранение выделенной версии файла поверх любой из версий, имеющихся в хранилище. При выборе этой команды открывается окно диалога Select A Revision (см. рис. 14.20), в котором указывается версия, поверх которой нужно произвести запись;

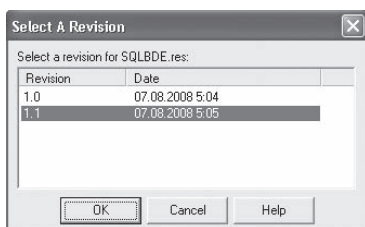


Рис. 14.20. Окно диалога Select A Revision

- ❑ Remove from project — полностью удаляет выделенный файл (файлы) из хранилища версий;
- ❑ View Archive Report — вывод отчета о текущем состоянии хранилища версий;
- ❑ Compare Revisions — сравнение двух версий выделенного файла. При выборе данной команды открывается окно диалога Select Revision Pair (рис. 14.21), в котором следует выбрать две сравниваемые версии. Результаты сравнения выводятся в окне, подобном тому, что изображено ранее на рис. 14.17;

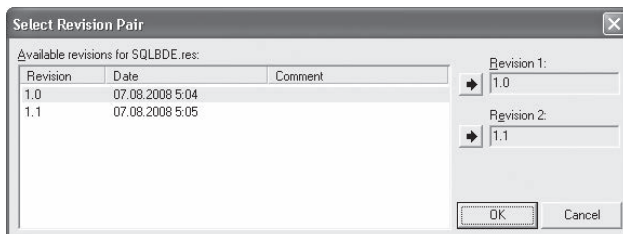


Рис. 14.21. Окно диалога Select Revision Pair

- ❑ Set Revision Number — присвоить выделенному файлу любой номер версии из тех, что имеются в хранилище. Для выбора версии открывается окно диалога Select A Revision;
- ❑ Fix Tip Revisions — проверка соответствия указателей на текущую версию файла. При выборе данной команды проверяется соответствие указателя текущей версии списку версий. В том случае если обнаруживается несоответствие, открывается окно диалога (см. рис. 14.20), в котором следует указать корректный номер версии.

В режиме Remote имеется возможность изменять структуру каталогов проекта. Для этого используются команды контекстного меню, открывающегося при щелчке правой кнопкой мыши на имени любого из каталогов, отображаемых в левой панели. В данном меню содержатся всего три команды:

- ❑ Add — добавить к проекту новый каталог;
- ❑ Delete — удалить выделенный каталог. Данная команда неприменима к корневому каталогу (root);
- ❑ Properties — изменить свойства выделенного каталога.

При выборе последней команды открывается окно диалога **Directory Properties** (рис. 14.22), в котором можно изменить параметры выделенного каталога хранилища версий и режим обработки соответствующего ему локального каталога.

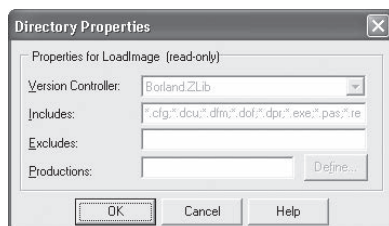


Рис. 14.22. Окно диалога **Directory Properties**

В окне **Directory Properties** отображаются и задаются следующие параметры:

- ❑ **Version Controller** — используемый контроллер версий. Отображается для информации, редактировать данную строку нельзя;
- ❑ **Includes** — список шаблонов файлов, заносимых в хранилище версий из локального каталога. Например, для Delphi типичными шаблонами для файлов с исходными текстами являются *.dpr; *.pas; *.dfm;
- ❑ **Excludes** — список шаблонов файлов, которые не следует заносить в хранилище из локального каталога и изменения в которых не надо отслеживать. Например, если в список **Includes** поместить шаблон *.dfm, а в список **Excludes** — шаблон tmp*.dfm, то dfm-файлы, имена которых начинаются с букв tmp, не будут отслеживаться контроллером версий;
- ❑ **Productions** — данная настройка позволяет отменить наблюдение за изменениями в автоматически генерируемых файлах, даже если их тип указан в списке **Includes**. Соответствие между исходным и генерируемым файлом задается в окне диалога, открываемом при щелчке на кнопке **Define**.

Работа в режиме History

В режиме **History** показываются все изменения, внесенные в проект, с момента его создания. Главное окно программы TeamSource в данном режиме делится на две панели (рис. 14.23). В левой панели отображаются дата и время внесения изменения в хранилище версий, а также имя пользователя, внесшего данные изменения. В правой панели указывается, какие файлы проекта были изменены.

Блокировка проекта

При работе с проектом многие операции выполняются только в режиме блокировки. Уже отмечалось, что настройку параметров проекта и модификацию хранилища версий можно выполнять только при включенной блокировке.

Блокировка проекта может быть выполнена любым пользователем, обладающим соответствующими правами доступа к проекту. Для включения блокировки используется команда **Project ► Request Lock** главного меню TeamSource.

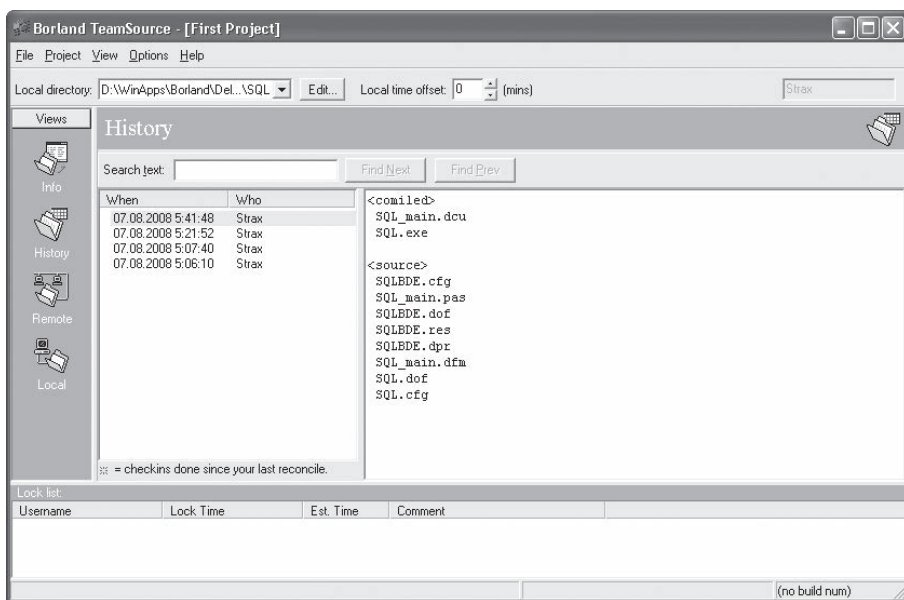


Рис. 14.23. Использование программы TeamSource в режиме History

При этом открывается окно диалога Lock Information (рис. 14.24), в котором задаются следующие параметры блокировки:

- ☐ комментарии к блокировке — текстовое поле Lock Comment;
- ☐ время блокировки — счетчик Estimated time you will need the lock;
- ☐ задание признака блокировки администратором — флажок Lock as Administrator Lock. При наличии этого признака блокировка не имеет ограничения по времени.

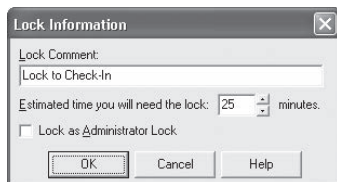


Рис. 14.24. Окно диалога Lock Information

После создания блокировки ее параметры можно изменить с помощью команд контекстного меню, открывающегося при щелчке правой кнопкой мыши в поле Lock list главного окна TeamSource. Команды данного меню позволяют:

- ☐ снять блокировку (Clear Lock);
- ☐ изменить комментарий блокировки (Edit Lock Comment);
- ☐ продлить время действия блокировки (Extend lock);

- ❑ передать блокировку другому пользователю (Yield to);
- ❑ проверить состояние блокировки проекта другим пользователем (Verify Current Lock).

Использование закладок

Закладки (Bookmarks) применяются для пометки версии проекта. Обычно эта пометка используется при выполнении сборки проекта (операция Pull).

Управление закладками осуществляется с помощью окна диалога Bookmarks (рис. 14.25), открывающегося после выбора команды Project ► Bookmarks главного меню TeamSource.

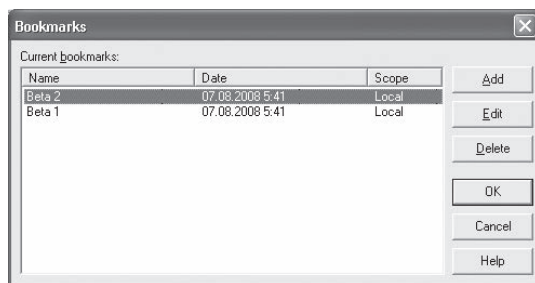


Рис. 14.25. Окно управления закладками проекта

Для добавления новой закладки следует щелкнуть на кнопке Add. Редактирование ранее созданной закладки выполняется после щелчка на кнопке Edit. После выполнения любого из этих действий открывается окно диалога Bookmark Properties (рис. 14.26), в котором задаются (или изменяются) свойства закладки:

- ❑ название закладки — текстовое поле Name;
- ❑ дата и время версии проекта, для которой создается закладка, — список Date;
- ❑ тип закладки (глобальная или локальная) — переключатели группы Scope (Local или Global).

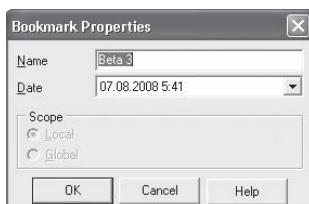


Рис. 14.26. Окно диалога Bookmark Properties

Глобальные закладки видны и доступны для использования всем участникам проекта. Создавать глобальные закладки имеет право только администратор. Локальные закладки могут просматриваться и использоваться только пользователем, который их создал.

Сборка проекта

Для сборки проекта необходимо получить из хранилища версий все файлы, требуемые для проведения компиляции. Это могут быть как файлы, соответствующие текущей версии проекта, так и файлы любой более ранней версии, отмеченной закладкой.

Для получения всех файлов проекта из хранилища следует выбрать в главном меню TeamSource команду **Project ► Pull to**. При этом открывается окно диалога Pull (рис. 14.27), в котором содержатся следующие элементы управления:

- ❑ список **Bookmark**, с помощью которого указывается закладка, соответствующая версии запрашиваемых файлов. Если производится сборка текущей версии, то в данном списке следует оставить предлагаемый по умолчанию вариант **None**;
- ❑ флажок **Fast Pull** — при установке этого флажка будут копироваться только те файлы, которые отсутствуют в локальных копиях (или локальные версии которых старше запрашиваемой).

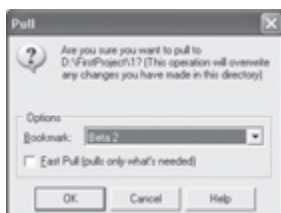


Рис. 14.27. Окно диалога Pull

Глава 15

Справочная система приложения

Многие пользователи, работающие с информационными системами, часто не являются специалистами в области компьютерных технологий. Кроме того, большая часть пользователей осваивает приложение в процессе работы с ним. Поэтому при разработке любых, даже не очень сложных приложений следует ориентироваться на не очень «продвинутого» пользователя и принимать все возможные меры для облегчения работы с программой:

- ❑ во-первых, как уже отмечалось выше, приложение должно иметь простой и интуитивно понятный интерфейс, соответствующий сложившимся на сегодняшний день стандартам;
- ❑ во-вторых, приложение должно иметь справочную систему.

В данной главе рассматриваются вопросы создания и использования справочных систем различных типов, принятых на сегодняшний день для Windows-приложений.

Основные компоненты справочной системы

Прежде всего определимся, что мы будем понимать под *справочной системой* приложения. Традиционно под справочной системой подразумевают *файл справки*, содержащий подробное (или не очень) описание функций программы, который открывается при выборе соответствующего пункта меню или при нажатии на клавишу F1. Однако в большинстве случаев наличия этого недостаточно, так как пользователю часто требуется получить оперативную подсказку по программе, а поиск необходимой информации в текстовом файле может занять много времени. Поэтому наряду с описанием функций программы необходимо наличие дополнительных модулей справки. Это, в первую очередь, *контекстно-зависимая справка*, позволяющая быстро получать информацию о любом окне программы, диалоге, команде меню или элементе управления во время их использования.

Кроме контекстной справки приложение следует также снабдить системой *всплывающих подсказок*, которые отображаются, когда пользователь задерживает указатель мыши над каким-либо элементом управления. Всплывающие подсказки особенно актуальны для кнопок панелей инструментов, так как значки, отображаемые на этих кнопках, часто не дают ясного представления об их назначении.

Все вышеперечисленные элементы справочной системы позволяют максимально быстро находить необходимую справочную информацию и облегчают работу с приложением. Однако кроме этого полезно также предусмотреть наличие средств, информирующих о текущем состоянии приложения, о выполняемых им действиях. Это позволит, например, уведомить пользователя о том, что приложение не «зависло», а выполняет какую-либо длительную операцию. Обычно для вывода информации о приложении используется так называемая *строка состояния*, которая располагается в нижней части окна приложения. Таким образом, в наиболее завершенном виде справочная система должна включать в себя следующие элементы:

- ☐ файл справки, содержащий подробную информацию о работе с приложением, изложенную в доступной для «среднего» пользователя форме;
- ☐ контекстно-зависимую справку, вызываемую нажатием на клавишу F1. В большинстве случаев содержимое контекстно-зависимой справки включается в файл справки;
- ☐ систему всплывающих подсказок;
- ☐ строку состояния, в которой отображается информация о текущем состоянии приложения.

Реализация первых двух элементов справочной системы осуществляется с помощью специальных программ, предназначенных для разработки файла справки и его просмотра. Delphi в этом случае используется лишь для интеграции приложения с файлами справочной системы.

Последние два элемента справочной системы (всплывающие подсказки и строка состояния) обеспечиваются средствами Delphi.

Вопросы разработки файлов справки и интеграции их в приложения Delphi будут рассмотрены ниже, а в первую очередь мы коснемся технологии создания всплывающих подсказок и строки состояния.

Создание всплывающих подсказок

Всплывающие подсказки могут создаваться для любых визуальных компонентов Delphi. Это обусловлено тем, что они обладают рядом свойств, предназначенных для создания и управления параметрами всплывающих подсказок. Поэтому создание подсказки для какого-либо компонента сводится просто к изменению значений некоторых его свойств. Информация о свойствах визуальных компонентов Delphi, отвечающих за отображение «всплывающих» подсказок, приведена в табл. 15.1.

Таблица 15.1. Свойства визуальных компонентов, используемые для создания всплывающих подсказок

Свойство	Тип	Описание
Hint	string	Текст всплывающей подсказки
ShowHint	Boolean	Определяет, отображать подсказку (true) или нет (false)
ParentShowHint	Boolean	Определяет, наследовать (true) или нет (false) параметры подсказки родительского компонента

Текст подсказки, задаваемый в свойстве Hint, может состоять из двух частей, которые разделяются между собой символом «|». Например, текст подсказки может иметь следующий вид:

Печать | Печать текущего документа

В этом случае во всплывающей подсказке отображается только первая часть текста. Вторая часть подсказки (обычно более развернутая) используется для отображения информации, например в строке состояния приложения. Чтобы получить доступ к каждой части подсказки, используются две стандартные функции Delphi:

- ❑ `function GetShortHint(const Hint: string): string` — возвращает первую часть подсказки;
- ❑ `function GetLongHint(const Hint: string): string` — возвращает вторую часть подсказки, если она есть. Если подсказка не разделена на две части, то возвращается строка Hint.

Все визуальные компоненты Delphi обладают свойствами, приведенными в табл. 15.1. Кроме того, рядом важных свойств, определяющих параметры отображения подсказки, обладает также невидимый объект Application. Рассмотрим эти свойства:

- ❑ `ShowHint: Boolean` — определяет, будут ли отображаться подсказки при работе приложения. Данное свойство имеет более высокий приоритет, чем аналогичное свойство визуальных компонентов, то есть если значение данного свойства установлено равным false, то, независимо от значений свойства Hint визуальных компонентов, подсказки отображаться не будут;
- ❑ `HintColor: TColor` — цвет фона, на котором отображается подсказка;
- ❑ `HintHidePause: Integer` — интервал времени (в миллисекундах), в течение которого будет отображаться текст подсказки;
- ❑ `HintPause: Integer` — задержка (в миллисекундах) перед отображением подсказки (интервал времени между моментом помещения указателя мыши на визуальный компонент и моментом вывода подсказки на экран).

Перед отображением подсказки возникает событие OnShowHint объекта Application. Заголовок процедуры-обработчика этого события имеет следующий вид:

Параметры этой процедуры имеют следующий смысл:

- ❑ `HintStr` — текст подсказки, который может быть изменен в теле процедуры-обработчика данного события;

- ❑ CanShow — параметр, показывающий, отображать подсказку или нет. Если в обработке события OnShowHint данному параметру присвоить значение false, то подсказка отображаться не будет;
- ❑ HintInfo — запись, определяющая параметры отображения подсказки, такие как цвет фона, координаты подсказки, длительность отображения и задержка появления подсказки и т. п.

Создание строки состояния приложения

Строка состояния приложения предназначена для того, чтобы информировать пользователя о текущем состоянии базы данных, о процессе выполнения различных операций, а также для вывода различного рода подсказок.

Для создания строки состояния приложения в VCL Delphi существует специальный компонент, имеющий название TStatusBar и расположенный на странице Win32 палитры компонентов. Компонент TStatusBar, в зависимости от значения свойства SimplePanel, может состоять из одной (SimplePanel = true) или нескольких панелей (SimplePanel = false), на которые выводится какая-либо информация.

В том случае когда значение свойства SimplePanel задано равным true, в строку состояния можно выводить только текстовые сообщения. Для вывода текста в строку состояния используется свойство SimpleText класса TStatusBar. Например, для вывода в строке состояния количества записей, выбранных в результате выполнения SQL-запроса, можно использовать следующий код:

```
StatusBar1.SimpleText := 'Выбрано' +  
    IntToStr(Query1.RecordCount) + ' записей';
```

При создании более сложных строк состояния можно использовать несколько панелей компонента TStatusBar. Для создания панелей строки состояния во время разработки приложения используется специальный редактор панелей (рис. 15.1). Для его открытия можно воспользоваться одним из следующих способов:

- ❑ выполнить двойной щелчок на компоненте TStatusBar, размещенном на форме;
- ❑ выбрать пункт Panels Editor контекстного меню данного компонента;
- ❑ щелкнуть на кнопке с многоточием в поле ввода свойства Panels компонента TStatusBar в инспекторе объектов.

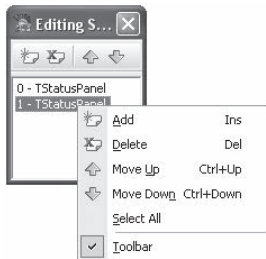


Рис. 15.1. Окно редактора панелей компонента TStatusBar

Для добавления, удаления и изменения местоположения панелей используются либо кнопки панели инструментов редактора панелей, либо команды контекстного меню, которые не требуют пояснений. Каждая панель представляет собой экземпляр класса `TStatusPanel`. Количество свойств у этого класса невелико и из них отметим только четыре:

- ☐ `Bevel` — определяет вид панели:
 - ☐ `pbLowered` — вдавленная;
 - ☐ `pbRaised` — выпуклая;
 - ☐ `pbNone` — панель без выделения;
- ☐ `Text` — текстовая строка, отображаемая на панели;
- ☐ `Width` — ширина панели в пикселах;
- ☐ `Style` — свойство, управляющее рисованием на панели:
 - ☐ `psText` — на панель выводится только текст;
 - ☐ `psOwnerDraw` — обеспечивает вывод информации в любой форме.

При использовании стиля панели `psOwnerDraw` разработчик программы должен сам позаботиться о прорисовке прямоугольной области, занимаемой панелью строки состояния. Для этого следует использовать событие `OnDrawPanel` компонента `TStatusBar`. В этом случае на панель можно выводить не только текст, но и любые изображения.

Создание файла справки в формате WinHelp 4

Справочная система WinHelp представляет собой специальную утилиту, предназначенную для просмотра справочных файлов (имеющих расширение `hlp`). Данная система использовалась еще в Windows 3.0. В состав Windows 95 включена обновленная версия данной утилиты, называемая WinHelp 4, которая состоит из одного файла с именем `winhelp.exe`. В Windows XP данная утилита также присутствует.

В настоящее время фирма Microsoft объявила справочную систему WinHelp устаревшей и предложила новый формат справочных файлов (так называемый `HtmlHelp`, основанный на файлах формата `chm`, которые представляют собой сжатые файлы формата `html`). Однако несмотря на это, справочные файлы в формате WinHelp до сих пор достаточно широко используются, и Borland Delphi обеспечивает интеграцию программ с файлами справки WinHelp.

Основные элементы справочной системы WinHelp 4

Файл справки в формате WinHelp состоит из нескольких элементов, которые создаются во время разработки справочной системы. Рассмотрим их более подробно.

Темы

Основным элементом справочной системы является *тема* (`topic`). Тема представляет собой фрагмент справочной системы, отображаемый в окне приложения

winhelp.exe. Он может содержать текст, таблицы, графические изображения и кнопки.

Обычно справочная система представляет собой гипертекстовый документ, включающий множество тем, связанных между собой перекрестными ссылками. Каждый раздел обычно имеет заголовок, отображаемый в верхней части окна просмотра, строковый идентификатор, числовой идентификатор, набор ключевых слов, по которым можно найти раздел, а также ссылки на другие разделы.

Перекрестные ссылки

Отдельные темы справочной системы связываются между собой при помощи ссылок. Для пользователя ссылки представляются в виде выделенного цветом и подчеркиванием текста или в виде кнопок. При разработке справочной системы можно создать ссылку на другую тему либо на временное окно. Временное окно обычно используется для пояснения термина. Можно также создавать ссылки, отображающие тему во вторичном окне.

Содержание и указатели

Содержание справочной системы фактически является отдельной темой, ничем не отличающейся от других тем, за исключением того, что при создании справочной системы она описывается специальным образом и содержит прямые или косвенные ссылки на все остальные темы. При использовании справочной системы WinHelp 4 содержание обычно помещается в отдельный файл с расширением `snt`, имеющий такое же имя, как и файл справки.

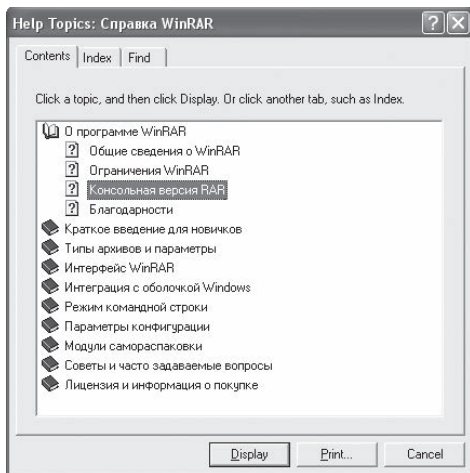


Рис. 15.2. Окно Help Topics справочной системы

В отличие от остальных тем, содержание отображается в специальном окне, называемом Help Topics. Данное окно открывается при открытии файла справки с помощью WinHelp 4 и содержит три вкладки (рис. 15.2):

- ❑ **Contents (Содержание)**, на которой отображается древовидная структура справочной системы, состоящая из *разделов* и *тем*. Каждый *раздел* (heading) содержит одну или несколько *тем* (topic). Раздел также может содержать другие разделы. При двойном щелчке на заголовке раздела отображается его содержание. При двойном щелчке на теме открывается окно, содержащее текст, соответствующий выбранной теме. В содержании могут быть объединены несколько справочных файлов, каждому из которых соответствует раздел;
- ❑ **Index (Указатель)** — предметный указатель справочной системы, обеспечивающий поиск по ключевым словам. Предметный указатель составляется разработчиком справочной системы путем связывания ключевых слов с идентификаторами соответствующих тем;
- ❑ **Find (Поиск)**. С помощью этой вкладки выполняется поиск по всему тексту справочной системы.

Окна

Поясняющий текст в системе WinHelp 4 может отображаться в окнах нескольких видов. Наиболее часто применяются три типа окон:

- ❑ основное окно, в котором отображается текст выбранной темы;
- ❑ вторичное окно, используемое, чтобы вывести дополнительную информацию, не покидая текущий раздел помощи, отображаемый в основном окне. В этом окне могут располагаться перекрестные ссылки, причем при щелчке на них тема, соответствующая переходу, будет отображена в основном окне, а текст дополнительного окна останется прежним. Такие окна используются, например, в справочной системе Delphi для вывода списков свойств, методов или событий классов;
- ❑ всплывающее (pop-up) окно, которое используется для вывода пояснений к терминам. Такие окна также часто используются при организации контекстно-зависимой справки приложения для краткого описания размещенного на экране элемента.

WinHelp 4 позволяет разработчикам справочных систем создавать до 255 различных классов вторичных окон, а отдельные темы можно связывать с определенным стилем окна. Вторичные окна могут содержать инструментальные кнопки. Имеется возможность указать относительный размер и положение окна на экране. Можно также создавать окна, высота которых устанавливается автоматически, так чтобы весь текст темы уместился на экране.

Кнопки и макрокоманды

В тексте справки разработчик справочной системы может определить кнопки нескольких типов, предназначенные для выполнения перехода, вызова всплывающего меню или макрокоманды.

В WinHelp 4 определено более 30 макрокоманд. Основные из них приведены в табл. 15.2.

Таблица 15.2. Основные макрокоманды WinHelp 4

Макрокоманда	Описание
KLink	Переход к теме, соответствующей обычным ключевым словам. Если таких тем несколько, то открывается окно Найденные разделы и пользователь может выбрать тему для просмотра
ALink	Переход к теме, соответствующей скрытым ключевым словам. Если таких тем несколько, то открывается окно Найденные разделы и пользователь может выбрать тему для просмотра
Find	Открывает окно диалога Help Topics с вкладкой Поиск
ExecFile	Запускает указанную программу
ShellExecute	Запускает указанную программу, печатает файл или открывает файл с помощью связанной с ним программы
ShortCut	Запускает указанную программу, если она не запущена, или делает ее активной и передает ей сообщение <code>WM_COMMAND</code> , если программа запущена
ControlPanel	Запускает приложение панели управления
MPrintID	Печатает тему
BrowseButtons	Добавляет к окну справки кнопки просмотра вперед и назад (>> и <<)
CreateButton	Создает в окне справки кнопку с указанным именем, связанную с указанным макросом
InsertMenu	Добавляет новый пункт к главному меню окна справки
AppendMenu	Добавляет новый пункт к указанному подменю

Создание файла справки

Существует довольно большое количество как коммерческих, так и бесплатных средств разработки справочных файлов. Мы рассмотрим процесс создания файла справки с помощью программы Microsoft Help Workshop.

Для создания тем справок кроме программы Microsoft Help Workshop необходим текстовый редактор, способный сохранять тексты в формате RTF. Для этого можно использовать редактор MS Word (мы используем MS Word 2003), так как он в полной мере обеспечивает все необходимые возможности по созданию тем справочной системы.

Разработка текстов тем справочной системы

Исходный текст справочной системы представляет собой файл в формате RTF, содержащий текст тем справочной системы. Каждая тема снабжается необходимыми атрибутами, которые будут рассмотрены ниже. Темы отделяются друг от друга символом разрыва страницы, который внедряется в документ командой **Вставка ► Разрыв**. Порядок следования тем в RTF-файле не имеет значения, за исключением темы «Содержание», которая должна располагаться первой.

Тема может снабжаться рядом атрибутов, которые определяют ее свойства. Каждая тема обязательно содержит по крайней мере один атрибут. Задание атрибутов тем в редакторе MS Word производится с помощью сносок. Символ сноски определяет вид атрибута темы. Чтобы задать сноску, выберите команду **Вставка ► Ссылка ► Сноска** главного меню. При этом откроется окно диалога (рис. 15.3), в кото-

ром задаются параметры сноски. Изменять в этом окне следует только символ сноски.

Все атрибуты следует помещать в самом начале темы, перед ее текстом.

После того как вы вставите сноску, в нижней части страницы появится область для редактирования сносок. Текст сноски, являющийся атрибутом темы, вводится после соответствующего символа в области редактирования сносок. Между символом сноски и текстом сноски должен быть только один символ пробела.

Атрибуты тем

Рассмотрим, какими атрибутами может обладать тема и какие сноски соответствуют этим атрибутам:

- ☐ идентификатор темы (сноска #) — уникальная текстовая строка, используемая в дальнейшем для ссылки на раздел. Это единственный обязательный атрибут. Строка может задаваться как латинскими, так и русскими буквами и может состоять из нескольких слов;
- ☐ заголовок темы (сноска \$) — наименование темы;
- ☐ обычные ключевые слова (сноска K — заглавная латинская буква) — задает ключевые слова, по которым производится поиск в предметном указателе справки (окно Help Topics, вкладка Index (Указатель)). Текст сноски может состоять из нескольких слов, разделенных пробелами. С помощью одной сноски можно задавать несколько ключевых слов, разделяя их точкой с запятой. Также можно задавать ключевые слова двух уровней — в этом случае вначале указывается ключевое слово первого уровня, затем через запятую ключевое слово второго уровня;
- ☐ скрытые ключевые слова (сноска A — заглавная латинская буква) — задает ключевые слова, которые не включаются в предметный указатель, а используются только для переходов с помощью макроса ALink;
- ☐ номер в последовательности просмотра (сноска +) — позволяет задать положение темы в последовательности просмотра, производимого с помощью кнопок >> и <<. Значения этого атрибута можно не указывать, в этом случае они будут устанавливаться автоматически в соответствии с последовательностью тем в RTF-файле;
- ☐ макрос, связанный с темой (сноска !) — один или несколько макросов, запускаемых перед отображением темы;
- ☐ директива компилятора (сноска *) — позволяет включать или не включать тему в справочную систему в зависимости от параметров компиляции;
- ☐ идентификатор окна (сноска >) — указывает идентификатор окна, в котором будет отображаться тема.

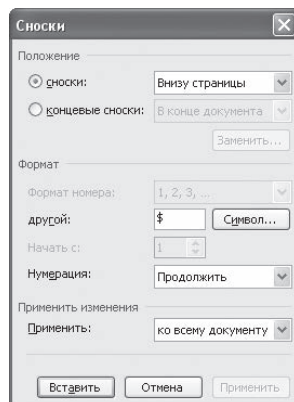


Рис. 15.3. Окно диалога задания параметров сноски

Текст темы

Текст темы должен следовать после атрибутов. При написании текста можно использовать все возможности редактора MS Word — внедрять в текст формулы, рисунки из графических файлов или буфера обмена, использовать различные виды форматирования текста и разные шрифты.

ПРИМЕЧАНИЕ

Векторные картинки, нарисованные непосредственно в редакторе Word, включены в текст справки не будут.

Если объем текста темы велик и не уместается полностью в окне просмотра, то автоматически появляется полоса прокрутки. Однако иногда требуется, чтобы часть текста (например, заголовок темы) постоянно отображалась в окне просмотра и не прокручивалась. Для создания области без прокрутки выполните следующее.

1. Выделите абзац, текст которого не должен прокручиваться.
2. Выберите команду главного меню MS Word **Формат** ► **Абзац**. Откроется окно диалога **Абзац**.
3. Перейдите на вкладку **Положение на странице** и установите флажок **Не отрывать от следующего** (рис. 15.4).

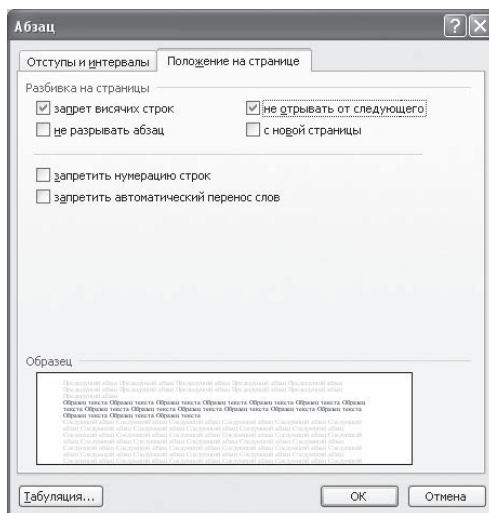


Рис. 15.4. Задание области текста без прокрутки

Перекрестные ссылки

В текстах тем могут быть перекрестные ссылки, позволяющие переходить на другие темы. Ссылки создаются непосредственно в тексте темы, для чего используется соответствующее шрифтовое оформление — перечеркнутый или двукратно подчеркнутый текст, однократно подчеркнутый текст и скрытый текст.

Различаются ссылки следующих типов:

- ❑ *непосредственные переходы* — при щелчке на ссылке производится переход на другую тему;
- ❑ *переходы по ключевым словам* — переход осуществляется не на заранее заданную тему, а на темы, соответствующие заданным ключевым словам;
- ❑ переходы к темам, отображаемым во *всплывающем* окне;
- ❑ переходы к темам, отображаемым во *вторичном* окне.

При использовании первых двух типов ссылок текст темы, на которую они указывают, отобразится в главном окне. Третий и четвертый тип ссылок выводят текст указываемых тем в дополнительно открываемых окнах. Содержимое главного окна справки при этом остается неизменным. Рассмотрим процесс создания ссылок различных типов более подробно.

Чтобы задать *непосредственный переход*, выполните следующие действия.

1. Выделите слово или сочетание слов, которые соответствуют ссылке, и выберите команду **Формат** ► **Шрифт**.
2. В открывшемся окне диалога **Шрифт** установите в разделе **Подчеркивание** вариант двойного подчеркивания (рис. 15.5).

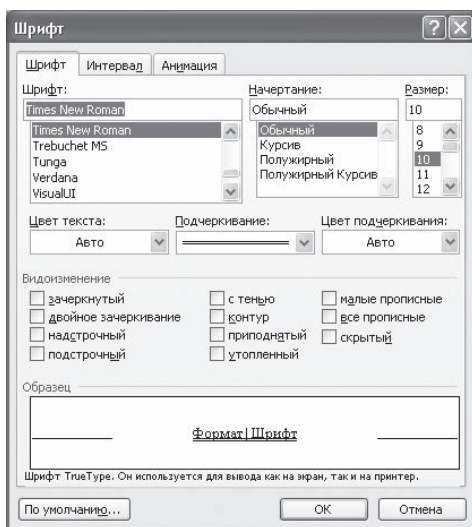


Рис. 15.5. Установка двойного подчеркивания

3. Сразу (без пробела) после слов, являющихся ссылкой, укажите идентификатор темы (заданный с помощью сноски #), на которую будет производиться переход.
4. Выделите идентификатор темы и выберите команду **Формат** ► **Шрифт**. Откроется окно диалога **Шрифт**.
5. Установите в группе **Видоизменение** флажок **Скрытый**.

ПРИМЕЧАНИЕ

Скрытый текст по умолчанию не отображается в редакторе Word. Если вы хотите видеть его, следует установить флажок **Скрытый текст** на вкладке **Вид** окна диалога **Параметры**, которое открывается при помощи команды **Сервис** ► **Параметры** главного меню MS Word.

Для создания переходов по ключевым словам следует использовать макросы KLink или ALink. Оба этих макроса действуют одинаково и имеют одинаковый синтаксис. Различие заключается только в том, что они производят поиск по разным ключевым словам: Klink — по ключевым словам, заданным с помощью сносков K, а ALink — по ключевым словам, заданным с помощью сносков A. Далее мы рассмотрим только макрос Klink, поскольку синтаксис макроса ALink будет аналогичным:

!KLink(<ключевые слова>,<тип>,<идентификатор темы>,<идентификатор окна>)

Прокомментируем параметры вызова этого макроса:

- ❑ <ключевые слова> разделяются точкой с запятой. Если одно из ключевых слов содержит запятую, то весь список заключается в кавычки. В том случае если при поиске будет найдено несколько тем, откроется окно **Найденные разделы**, в котором пользователю будет предложено выбрать тему, на которую следует перейти;
- ❑ <тип> определяет вид действия макроса при обнаружении одного или нескольких ключевых слов. Этот параметр может принимать одно или несколько (разделенных пробелами) следующих значений (можно использовать как символьные, так и числовые значения):
 - JUMP (1) — если найдена только одна тема, то на нее сразу производится переход;
 - TITLE (2) — если ключевые слова найдены более чем в одном файле, то отображается окно **Найденные разделы**, в котором приводится список тем, соответствующих заданным ключевым словам, и после названия темы указывается имя файла справки, в котором эта тема расположена;
 - TEST (4) — возвращает значение, показывающее, нашлось ли хотя бы одно соответствие заданным ключевым словам;
- ❑ <идентификатор темы> определяет тему, на которую будет осуществлен переход, если ни одно из ключевых слов не найдено;
- ❑ <идентификатор окна> определяет окно для отображения темы, соответствующей найденным ключевым словам. Если он не указан, то используется окно, заданное при описании темы; если и оно не задано, то используется тип окна, заданный по умолчанию.

Из всех параметров только первый является обязательным.

При использовании макросов ссылка создается точно так же, как и при прямом переходе, только вместо идентификатора темы задается вызов макроса.

В тексте темы можно также создавать ссылки, при нажатии на которые открывается окно, отображающее небольшой комментарий. Такое всплывающее окно не имеет области заголовка и полос прокрутки, однако текст, располагае-

мый в нем, может содержать ссылки любого вида. Для создания ссылки, открывающей всплывающее окно, выполните следующие действия.

1. Создайте тему, содержащую тот текст, который необходимо вывести во всплывающем окне. Создание такой темы не имеет никаких особенностей.
2. В тексте темы, где вы организуете ссылку, открывающую всплывающее окно, выделите текст, который будет выполнять роль ссылки, и задайте для него одинарное подчеркивание.
3. Сразу за подчеркнутым текстом (без пробелов) укажите идентификатор темы, которая должна быть показана во всплывающем окне.
4. Выделите идентификатор темы и укажите в окне диалога настройки шрифта, что он является скрытым текстом.

Тему справки, на которую указывает ссылка, можно также вывести во *вторичном* окне — окне, которое открывается поверх основного. Ссылка, выводящая тему в такое окно, создается следующим образом.

1. Выделите текст ссылки и установите в окне диалога **Шрифт** вариант двойного подчеркивания.
2. Сразу за подчеркнутым текстом (без пробелов) укажите идентификатор темы, которая будет выводиться во вторичное окно.
3. После идентификатора темы введите символ «>», а сразу за ним (без пробела) — идентификатор окна, в которое должна выводиться тема.
4. Выделите (одним блоком) идентификатор темы, символ «>» и идентификатор окна и укажите в окне диалога настройки шрифта, что они являются скрытым текстом.

ПРИМЕЧАНИЕ

Вторичное окно, идентификатор которого указывается в рассмотренной выше ссылке, должно быть создано в файле проекта справки. Более подробно вопрос создания окон будет рассмотрен ниже.

Кнопки

В текст тем справочного файла можно включать кнопки, при нажатии на которые будут выполняться связанные с ними макросы. Для включения в текст кнопки используется следующая запись:

```
{button <надпись>, <макросы>}
```

Здесь:

- <надпись> — надпись на кнопке;
- <макросы> — список макросов, которые выполняются при нажатии кнопки. Макросы отделяются друг от друга двоеточием.

Изображения

В текст темы можно внедрять изображения, хранящиеся в файлах форматов bmp, dib, wmf, shg, mrf. Для включения в текст изображения используются следующие три команды:

- ❑ {bmc <имя файла>} — включает изображение в строку как символ. При этом оформление абзаца (например, межстрочный интервал) будет применяться и к изображению;
- ❑ {bml <имя файла>} — размещает изображение с левой стороны страницы, текст располагается справа от картинки;
- ❑ {bmr <имя файла>} — помещает изображение в правой части страницы, текст располагается слева от изображения.

Рисунок можно так же просто поместить в текст, используя возможности редактора MS Word. Изображения, размещенные в тексте RTF-файла, можно использовать в качестве ссылок точно так же, как и текст.

Компиляция файла справки

Чтобы создать файл справки в формате hlp, необходимо выполнить следующие действия с помощью программы Microsoft Help Workshop.

1. Создайте новый файл проекта справки.
2. Выполните необходимые настройки проекта (укажите RTF-файл с текстом тем, создайте необходимые окна и т. п.).
3. Выполните компиляцию.

Рассмотрим каждый из перечисленных этапов создания файла справки более подробно.

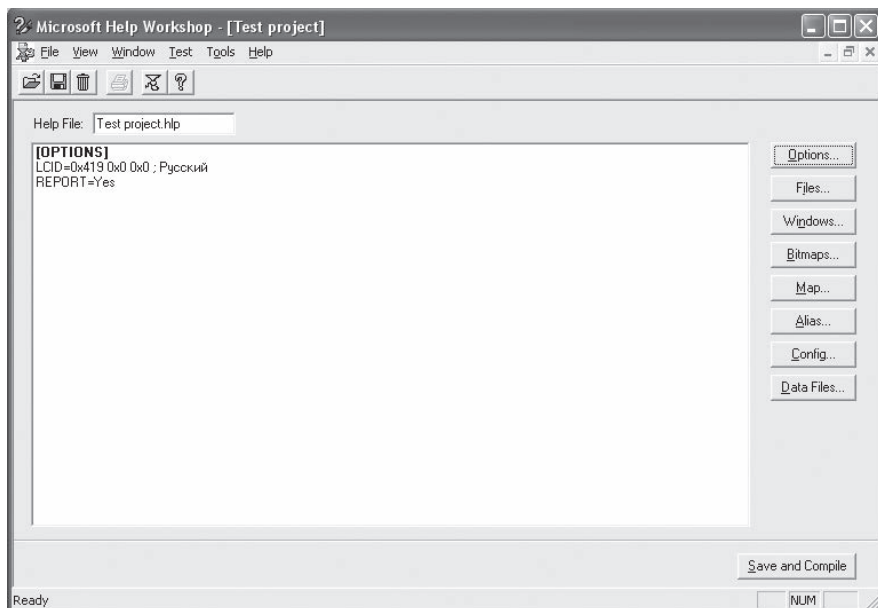


Рис. 15.6. Окно Microsoft Help Workshop со вновь созданным файлом проекта справки

Создание файла проекта справки

Файл проекта справки является обычным текстовым файлом в кодировке cp 1251 и может быть создан с помощью любого простейшего текстового редактора. Однако такой подход требует детального изучения синтаксиса этого файла. Программа Help Workshop в значительной степени облегчает разработку файла проекта справки, позволяя выполнить все необходимые настройки в интерактивном режиме.

Для создания файла проекта справки выполните следующее.

1. Откройте программу Microsoft Help Workshop — запустите на выполнение один из файлов: hcw.exe или hcrtf.exe (любой из них, не важно какой).
2. Выберите команду File ► New в главном меню открывшегося окна Help Workshop, затем в открывшемся окне диалога New выберите в списке строку Help Project и щелкните на кнопке OK.
3. Задайте имя файла проекта и укажите его местоположение на диске в открывшемся окне диалога Project File Name.

После выполнения указанных действий будет создан новый файл — проект справки, текст которого отображается в окне Microsoft Help Workshop (рис. 15.6).

Настройка файла проекта справки

Следующим шагом по созданию файла справки является выполнение всех необходимых настроек проекта. Главное здесь — указать имя файла (файлов), содержащего текст тем, и создать окна, используемые при просмотре справки. Кроме того, следует задать ряд дополнительных параметров, необходимых для организации взаимодействия файла справки с приложением. Все настройки выполняются в окнах диалога, открывающихся при нажатии кнопок, расположенных в окне Microsoft Help Workshop справа от текста файла проекта. Рассмотрим их более подробно.

Окно диалога Topic Files (рис. 15.7) открывается при щелчке на кнопке Files и служит для указания файлов, содержащих тексты тем. Для добавления файла к проекту щелкните на кнопке Add и выберите в открывшемся окне диалога открытия файла RTF-файл с текстами тем. Если таких файлов несколько, то повторите эту операцию необходимое количество раз. Ошибочно указанный файл

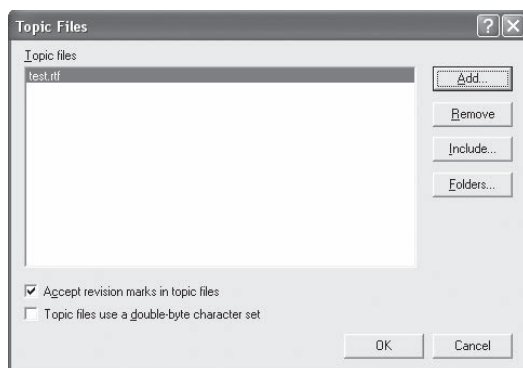


Рис. 15.7. Окно диалога Topic Files

можно удалить с помощью кнопки Remove. Кнопка Include используется для включения в проект текстового файла, содержащего список файлов тем. Кнопка Folders предназначена для указания каталога, в котором будет производиться поиск файлов тем при компиляции проекта.

Окно диалога Windows Properties (рис. 15.8) открывается при щелчке на кнопке Windows и используется для описания окон, которые применяются для отображения тем справки.

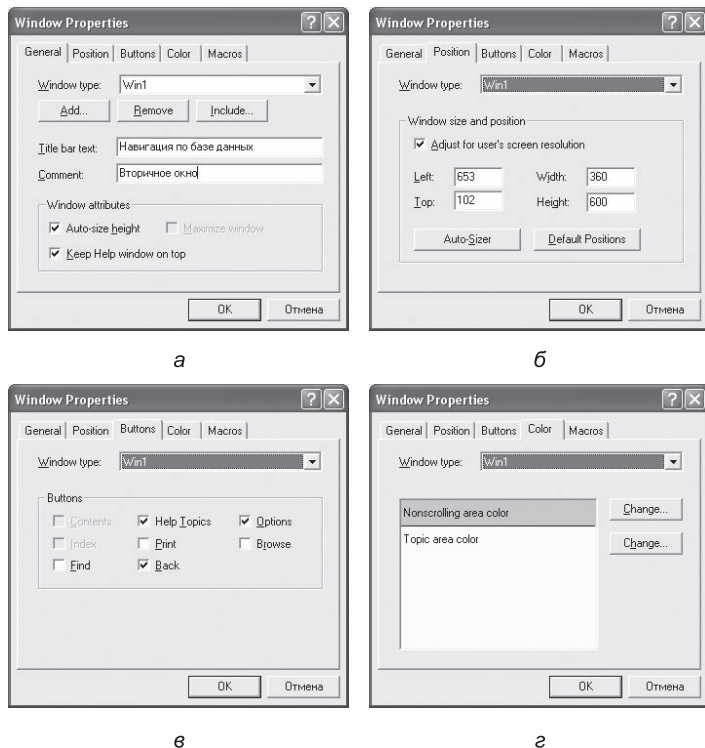


Рис. 15.8. Вкладки окна диалога Windows Properties

Окно диалога Windows Properties содержит пять вкладок, на которых указываются различные свойства создаваемого окна:

- ❑ на вкладке General (см. рис. 15.8, а) содержатся следующие настройки:
 - поле ввода Title bar text, в котором указывается текст заголовка окна;
 - флажок Auto-size height, позволяющий задать автоматический выбор высоты окна;
 - флажок Keep Help window on top, который обеспечивает расположение данного типа окна всегда поверх всех остальных окон;
- ❑ вкладка Position (см. рис. 15.8, б) позволяет задавать размер и расположение окна на экране;

- ❑ вкладка **Buttons** (см. рис. 15.8, *в*) используется для задания кнопок панели инструментов окна;
- ❑ вкладка **Color** (см. рис. 15.8, *з*) используется для задания цвета фона окна как для основной области (**Topic area**), так и для области текста без прокрутки (**Nonscrolling area**);
- ❑ на вкладке **Macros** можно ввести текст макроса, который будет выполняться при открытии окна.

Окно диалога **Bitmap Folders** открывается при щелчке на кнопке **Bitmaps** и позволяет указать каталоги, в которых следует искать графические файлы, включаемые в тексты тем с помощью команд `bmc`, `bm1` или `bmr`.

Окно диалога **Map** (рис. 15.9) открывается при щелчке на кнопке **Map**. Оно предназначено для создания карты соответствий символьных идентификаторов тем справки целочисленным номерам, которые необходимы для создания контекстно-зависимой справки приложения.

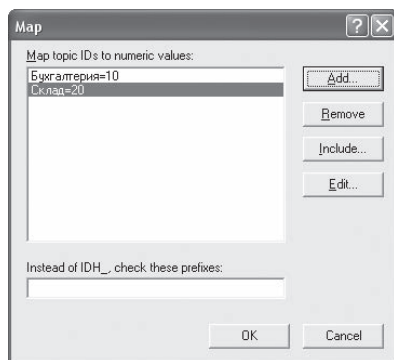


Рис. 15.9. Окно диалога Map

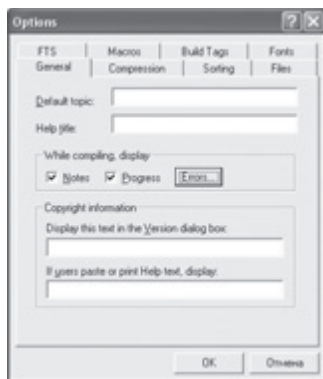
Окно **Topic ID Alias** открывается при щелчке на кнопке **Alias** и используется для задания псевдонимов идентификаторов тем. Псевдонимы используются для указания идентичности двух идентификаторов. Это может понадобиться, например, при объединении двух (или нескольких) тем в одну.

Окно **Configuration Macros**, открывающееся при щелчке на кнопке **Config**, предназначено для задания макросов, которые будут выполняться при каждом обращении к справке.

Окно диалога **Options**, с помощью которого задается ряд параметров файла справки, открывается при щелчке на кнопке **Options**. Это окно содержит восемь вкладок:

- ❑ на вкладке **General** (рис. 15.10, *а*) задаются:
 - тема, к которой справочная система будет обращаться по умолчанию (поле ввода **Default topic**);
 - заголовок окна справочной системы (поле ввода **Help title**);
 - текст, отображаемый в окне диалога, открываемом при выборе команды **Справка?Версия** главного меню программы WinHelp при просмотре данного справочного файла (поле ввода **Display this text in the Version dialog box**);

- текст, присоединяемый к тексту темы при его печати или копировании в буфер обмена (поле ввода If user paste or print Help text, display);
- информация, выводимая в процессе компиляции (элементы управления, расположенные в группе While compiling, display);



а



б



в



г

Рис. 15.10. Вкладки окна диалога Options

- на вкладке Compression (см. рис. 15.10, б) устанавливаются параметры сжатия файла справки. Рекомендуется устанавливать максимальное сжатие (для чего следует установить флажок Maximum), так как при этом размер файла справки будет минимальным. Правда, из-за этого увеличивается время компиляции;
- вкладка Sorting позволяет задать язык файла справки и порядок сортировки в предметном указателе;
- вкладка Files (см. рис. 15.10, в) позволяет задать:
 - имя результирующего hlp-файла (поле ввода Help File);
 - имя log-файла, в который будут заноситься все сообщения компилятора;

- имя RTF-файла с текстами тем (список Rich Text Format Files);
 - имя файла содержания справки (поле ввода Contents file);
 - каталог для хранения временных файлов (поле ввода TMP folder);
 - путь для поиска RTF- и графических файлов, включаемых в справочный файл (поле ввода Substitute path prefix);
- ❑ на вкладке FTS (см. рис. 15.10, з) можно выполнить установки для создания полнотекстового индексного файла, используемого при поиске по всему тексту. Обычно такой файл создавать не следует;
 - ❑ вкладка Font позволяет изменить шрифты, используемые для отображения текстов тем, и задать шрифт для окон диалога программы WinHelp;
 - ❑ на вкладке Macros можно связать ключевые слова в тексте справки с соответствующими им макросами;
 - ❑ вкладка Build Tags позволяет задать директивы компилятору, в зависимости от которых файлы тем с заданными сносками * будут включаться или не включаться в файл справки.

Последнее, что необходимо выполнить перед компиляцией файла проекта справки — создать файл содержания и связать его с файлом проекта. Файл содержания представляет собой текстовый файл (с расширением cnt), который можно создавать с помощью Microsoft Help Workshop.

Для создания нового файла содержания выполните следующее.

1. Выберите команду File ► New в главном меню программы Help Workshop и в открывшемся окне диалога New выберите Help Contents. После этого будет создан и загружен в редактор Help Workshop пустой файл содержания.
2. Для добавления к содержанию новых пунктов используйте кнопки Add Above и Add Below. Первая из них добавляет новый пункт к содержанию выше текущего (на котором находится курсор), а вторая — ниже. При щелчке на любой из этих кнопок открывается окно диалога Edit Contents Tab Entry (рис. 15.11).

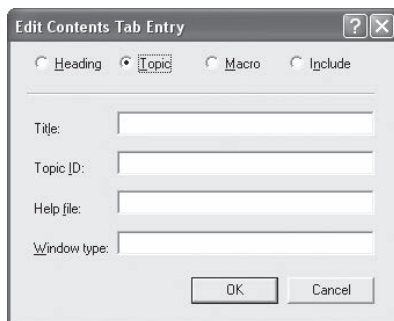


Рис. 15.11. Окно диалога Edit Contents Tab Entry

В верхней части окна Edit Contents Tab Entry расположены четыре переключателя, предназначенные для выбора типа нового пункта содержания:

- ❑ **Heading** — создается заголовок с названием, заданным в поле ввода **Title**. Все остальные поля ввода при выборе этого переключателя становятся недоступными;
- ❑ **Topic** — создается ссылка на тему файла справки:
 - наименование ссылки задается в поле ввода **Title**;
 - идентификатор темы, которая будет открыта при нажатии на ссылку, — в поле ввода **Topic ID**;
 - имя файла справки, в котором расположена данная тема, — в поле ввода **Help file** (если файл справки один, то заполнять данное поле не обязательно);
 - идентификатор окна, в котором будет отображаться тема, — в поле ввода **Window type** (если тема должна отображаться в окне, заданном по умолчанию, то в этом поле ввода ничего указывать не надо);
- ❑ **Macro** — создается ссылка на макрос. Наименование ссылки задается в поле ввода **Title**, макрос — в поле ввода **Macro**;
- ❑ **Include** — включает внешний файл содержания.

Каждый вновь введенный пункт содержания сразу отображается в окне Microsoft Help Workshop (рис. 15.12).

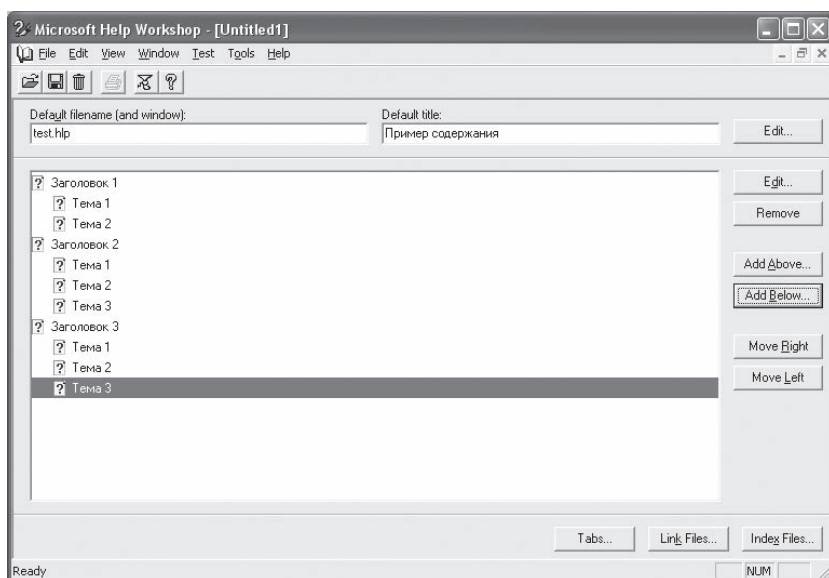


Рис. 15.12. Окно редактора содержания Microsoft Help Workshop

Кнопки **Move Right** и **Move Left** позволяют сдвигать выделенные пункты содержания вправо или влево, изменяя их уровень вложенности.

Чтобы связать созданный файл содержания с файлом проекта справки, используется окно диалога **Options**, рассмотренное выше.

Компиляция файла проекта справки

После завершения всех настроек файла проекта следует произвести компиляцию. В результате компиляции будет создан файл справки в формате hlp. Для выполнения компиляции щелкните на кнопке **Save and Compile**, размещенной в правом нижнем углу окна **Help Workshop**, либо сохраните файл проекта и выберите команду **File ► Compile** главного меню. После проведения компиляции в окне **Help Workshop** будет выведена информация о результатах компиляции.

В том случае если компилятор обнаружит ошибки в файле проекта или в файле тем, сообщения о них также будут отображены в окне **Help Workshop**.

Тестирование файла справки

На последнем этапе создания файла справки следует провести тестирование полученных результатов. Для проверки работоспособности разработанной справочной системы удобно воспользоваться возможностями программы **Microsoft Help Workshop**. После завершения компиляции выберите команду **File ► Run WinHelp** главного меню. При этом откроется окно диалога **View Help File** (рис. 15.13), с помощью которого можно проверить работу файла справки в разных режимах.

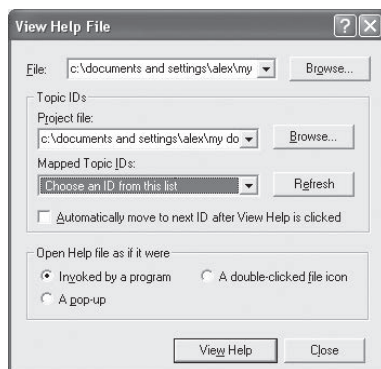


Рис. 15.13. Окно тестирования файла справки

С помощью группы переключателей раздела **Open Help file as if it were** задается способ открытия файла справки:

- ☐ **Invoked by a program** — открывает файл справки таким же способом, как при вызове из приложения;
- ☐ **A pop-up** — отображает тему во всплывающем окне;
- ☐ **A double-clicked file icon** — открывает файл справки таким же образом, как при двойном щелчке на его значке.

В списке **Mapped Topic IDs** можно выбрать идентификатор темы, которая будет выводиться первой при обращении к справке. Использование этого списка позволяет моделировать работу контекстно-зависимой справки.

ПРИМЕЧАНИЕ

В список Mapped Topic IDs заносятся только те темы, идентификаторам которых поставлены в соответствии числовые значения в окне диалога Map.

После задания всех настроек в окне диалога View Help File щелкните на кнопке View Help для запуска программы WinHelp.

Создание файла справки в формате HTML Help

Рост интереса к Всемирной сети Интернет вызвал бурное развитие средств разработки для Веб. Практически все современные средства разработки включают в себя инструментарий для работы с Веб. Поэтому нет ничего удивительного в том, что появился новый вид справочных систем, основанный на использовании языка HTML.

Такие справочные системы базируются на использовании браузера Internet Explorer. В настоящее время наиболее используются для их создания программы HTML Help Workshop и Microsoft Document Explorer. Они, как правило, включены во все основные продукты Microsoft.

Система HTML Help несомненно имеет ряд преимуществ перед традиционной системой WinHelp, среди которых главными являются следующие:

- ❑ язык разметки гипертекста (HTML) обеспечивает широкие возможности навигации по большим массивам информации;
- ❑ система HTML Help более эффективна при использовании для публикации крупных документов.

Развитием методов разработки справочных систем приложений является размещение справки на сервере разработчиков систем проектирования. Такой подход позволит производить постоянную корректировку справочной информации. Доступ же к справке через Интернет для современных пользователей проблем не представляет.

Основные элементы HTML Help

Несмотря на все достоинства, язык HTML не очень пригоден для создания контекстно-зависимых справочных систем прикладных программ. В первую очередь это связано со следующими факторами:

- ❑ HTML-документы обычно состоят из большого количества файлов, информация в которых хранится в несжатом виде. Вследствие этого дисковое пространство используется неэффективно. Кроме того, это может вызвать определенные проблемы при распространении приложений;
- ❑ в HTML-документах нет средств создания содержания, поиска по ключевым словам, а также полнотекстового поиска;
- ❑ обычные браузеры малоприспособлены для просмотра файлов справки;
- ❑ нет возможности создавать различные типы окон для вывода текста справки.

Для устранения перечисленных недостатков Microsoft дополнила стандартный язык HTML рядом средств:

- ❑ компилируемым файловым форматом (.chm), который предусматривает сжатие и объединение всех файлов HTML-документа в единый файл;
- ❑ стандартными средствами навигации: оглавлением, предметным указателем и средствами полнотекстового поиска;
- ❑ настраиваемым интерфейсом, напоминающим интерфейс WinHelp и снабженным настраиваемыми окнами и панелями инструментов;
- ❑ контекстно-зависимым API для организации взаимодействия с прикладными программами.

Все эти дополнения делают HTML Help очень похожим на WinHelp.

Создание файла справки в формате HTML

Для разработки файлов справки существует несколько различных инструментов. Мы будем рассматривать создание справочного файла с помощью бесплатного инструментального пакета HTML Help Workshop корпорации Microsoft. Его можно обнаружить на сервере Microsoft.

Процесс разработки файла справки с помощью Microsoft HTML Help Workshop в целом напоминает разработку hlp-файла, о которой мы говорили выше, и включает в себя следующие основные этапы.

1. Создание исходных файлов справочной системы.
2. Компиляцию файла проекта, после выполнения которой все файлы справочной системы объединяются в один файл справки;
3. Тестирование и отладку справочной системы.

Рассмотрим каждый из перечисленных этапов более подробно.

Создание исходных файлов справочной системы

Исходные файлы справочной системы состоят из:

- ❑ html-файлов с описанием отдельных тем справки;
- ❑ файла проекта;
- ❑ файла содержания, задающего иерархическую структуру разделов, подразделов и страниц справки, которая отображается на вкладке Содержание (Contents);
- ❑ индексного файла, используемого для быстрого поиска информации по ключевым словам.

Создание файлов тем

Файлы с текстами тем создаются с использованием любого из редакторов, поддерживающих формат HTML (например, Microsoft Word). При подготовке текстов тем справочной системы можно использовать все возможности языка HTML. Кроме обычного текста и изображений на страницах можно размещать мультимедийную информацию, фреймы, формы и пользовательские сценарии.

Это одно из преимуществ, обусловленных зависимостью HTML Help от Internet Explorer: при разработке страниц не нужно заботиться о вопросах совместимости с браузером.

Создание файла проекта

После подготовки файлов тем необходимо сформировать файл проекта, который в дальнейшем понадобится компилятору. Файл проекта является обычным текстовым (ASCII) файлом, в котором содержатся имена и адреса файлов, используемых в проекте. Кроме того, он содержит разнообразные варианты настройки интерфейса и поведения справочной системы.

Для создания файла проекта выполните следующие действия.

1. Запустите программу HTML Help Workshop и создайте новый проект справочной системы, выбрав команду **File ► New**.
2. Выберите в открывшемся окне диалога **New** вариант **Project** (рис. 15.14) и щелкните на кнопке **OK**. Появляется окно диалога **New Project**. С этого момента к созданию справочной системы подключается мастер проекта.
3. Если у вас есть проект в формате WinHelp, то вы можете, установив флажок **Convert WinHelp Project**, конвертировать его в проект в формате HTML. Мы же создадим проект с нуля, поэтому флажок **Convert WinHelp Project** устанавливать не будем.
4. Щелкните на кнопке **Далее** для перехода к окну диалога **New Project — Destination** и укажите в нем название файла проекта и каталог, в котором он будет храниться. Затем щелкните на кнопке **Далее**.

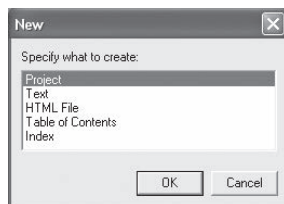


Рис. 15.14. Окно диалога New

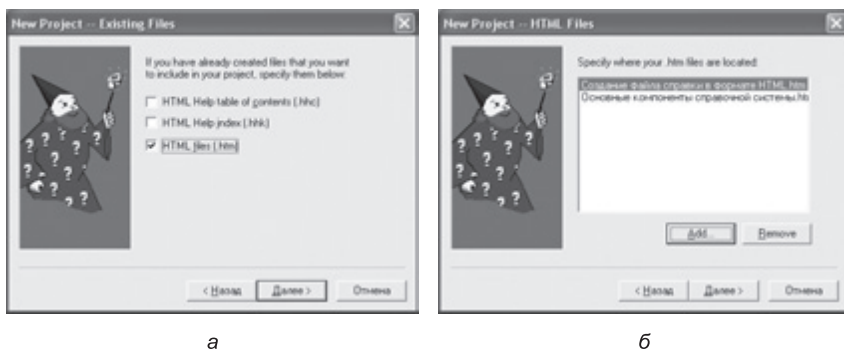


Рис. 15.15. Окна диалога подключения файлов тем к проекту справки

5. Выберите в открывшемся окне **New Project — Existing Files** (рис. 15.15, а) форматы файлов тем, которые должны быть включены в состав справочной системы. Если, как было предложено в пункте 1 данного алгоритма, вы создали

темы справочной системы в виде HTML-файлов, то вам потребуется установить флажок HTML files (.htm) и щелкнуть на кнопке Далее.

6. В открывшемся окне New Project — HTML Files (рис. 15.15, б), используя кнопки Add и Remove, включите в проект ранее созданные HTML-файлы с темами. Затем, щелкнув на кнопке Далее, перейдите в окно диалога New Project — Finish и щелкните на кнопке Готово.

После выполнения перечисленных действий отображается окно HTML Help Workshop (рис. 15.16). Рассмотрим основные элементы управления этого окна.

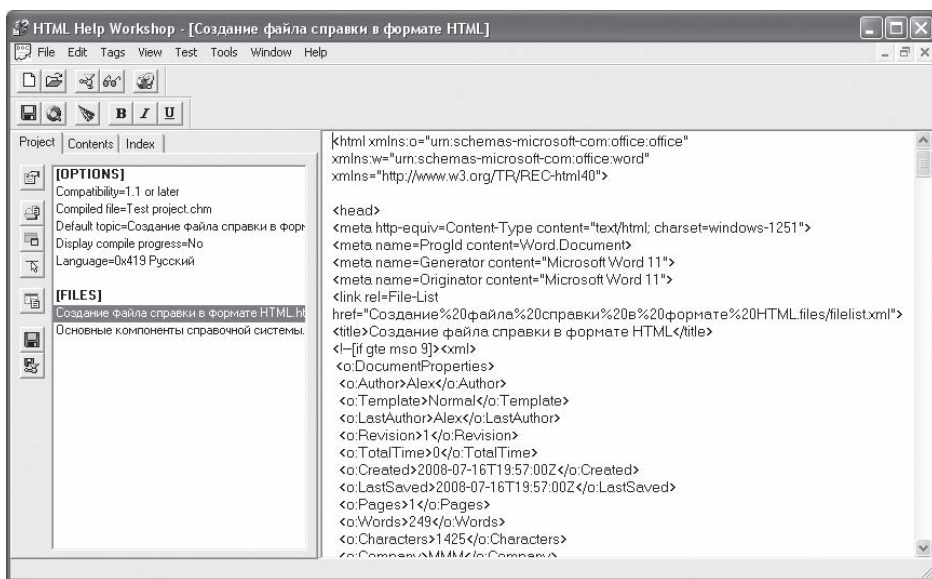


Рис. 15.16. Главное окно программы HTML Help Workshop

Окно программы HTML Help Workshop (см. рис. 15.16) состоит из двух частей:

- ❑ в левой части находятся вкладки Project, Contents и Index. Вдоль левой границы окна размещена панель инструментов. Состав кнопок на панели зависит от того, какая вкладка является активной;
- ❑ в правой части окна отображается содержимое исходного текста HTML-файла выбранной темы справочной системы. Этот файл можно не только просматривать, но и вносить в него изменения. Правда, для этого необходимо обладать знаниями языка HTML.

HTML Help Workshop предоставляет возможность просмотра содержимого файлов с темами в веб-браузере (рис. 15.17). Для реализации этой возможности выделите требуемый файл в разделе [FILES] и щелкните на кнопке Display in Browser.

Ряд важных параметров проекта задается с помощью окна диалога Options (рис. 15.18), которое открывается при двойном щелчке на разделе [Options] или при щелчке на кнопке Change Project Options на панели инструментов, расположенной в левой части окна HTML Help Workshop.

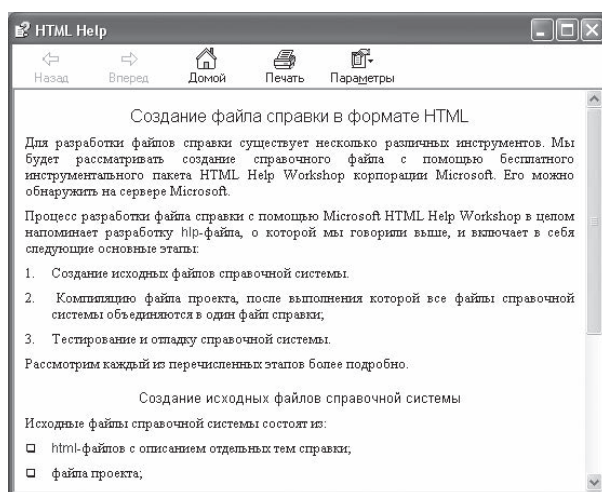


Рис. 15.17. Окно просмотра текста темы

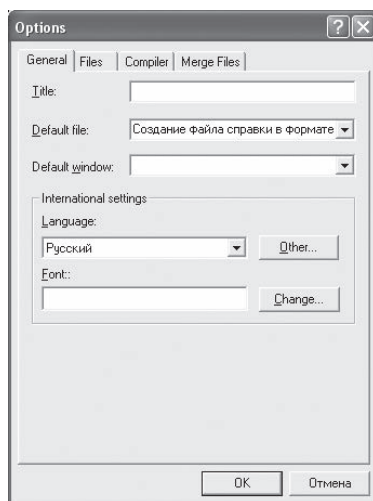


Рис. 15.18. Окно диалога Options

Окно диалога Options содержит четыре вкладки: General, Files, Compiler и Merge Files:

- ☐ вкладка General предназначена для определения следующих параметров:
 - Title — заголовок окна справочной системы;
 - Default file и Default window — соответственно файл темы и окно, выбираемые при открытии справочной системы;
 - Language и Font — язык и шрифт, используемые для отображения справочной системы;

- ❑ вкладка **Files** используется для указания расположения файлов справочной системы (**Compiled file**), файлов с указателями (**Index file**) и содержанием (**Contents file**);
- ❑ на вкладке **Compiler** задаются параметры компиляции справочной системы;
- ❑ на вкладке **Merge Files** можно задать откомпилированные файлы справки, которые должны сливаться в процессе работы приложения.

Так же как и WinHelp, система HTML Help позволяет разработчику справки определять свои типы окон, используемых для вывода справочной информации. Создание и настройка пользовательских типов окон производится в окне диалога **Window Types**, которое открывается при щелчке на кнопке **Add/Modify window definitions** панели инструментов вкладки **Project**. Это окно диалога содержит семь вкладок, на которых выполняются различные виды настроек, определяющих внешний вид окна, в котором отображается тема справки:

- ❑ вкладка **General** используется для задания заголовка окна справки, а также для добавления и удаления пользовательских типов окон;
- ❑ на вкладке **Buttons** определяются кнопки, которые будут включаться в состав панели инструментов программы просмотра файла справки;
- ❑ на вкладке **Positions** задаются размеры и исходное положение на экране окна программы просмотра справки;
- ❑ вкладка **Files** используется для указания файлов, связанных с окном, таких как файл содержания, индексный файл, файл темы, открываемой по умолчанию в окне данного типа и т. п.;
- ❑ вкладка **Navigation Panel** служит для настройки параметров панели навигации. Здесь указывается, отображать ли панель навигации при открытии окна, какую вкладку панели делать активной при открытии справки и т. п.;
- ❑ вкладки **Styles** и **Extended Styles** используются для настройки внешнего вида окна справки.

Чтобы созданный файл справки можно было впоследствии использовать в приложениях, необходимо связать файлы тем с численными идентификаторами. HTML Help Workshop использует двухступенчатое связывание: сначала каждой теме присваивается псевдоним (**Alias**), а затем псевдонимы связываются с численными значениями. (Псевдонимы в системе HTML Help фактически являются полным аналогом символьного идентификатора темы в WinHelp.)

Псевдонимы присваиваются темам с помощью окна диалога **HtmlHelp API information**, которое открывается при щелчке на одноименной кнопке панели инструментов. Для задания псевдонимов выполните следующее.

1. Откройте окно диалога **HtmlHelp API information**.
2. Перейдите на вкладку **Alias** (рис. 15.19) и щелкните на кнопке **Add**.
3. Выберите в раскрывающемся списке **Use it to refer to this HTML file** открывшегося окна диалога **Alias** (рис. 15.20) имя файла темы и введите в расположенное над ним поле ввода псевдоним, который будет связан с этим файлом.
4. Повторите пункт 3 необходимое количество раз в соответствии с имеющимся количеством файлов справочной системы.
5. Сохраните файл проекта справки.

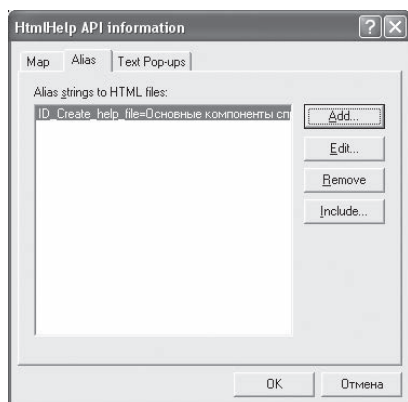


Рис. 15.19. Вкладка Alias окна диалога HtmlHelp API information

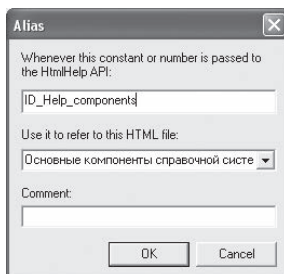


Рис. 15.20. Задание псевдонима для файла темы

Обратите внимание, что после задания псевдонимов в левой части окна HTML Help Workshop появился новый раздел с названием [Alias].

Создание и подключение файла связи

Связи между псевдонимами тем и их численными значениями должны быть записаны в отдельном файле связи в синтаксисе define-определений языка C. Данный файл состоит из строк, содержащих ключевое слово `#define`, за которым следуют разделенные пробелом псевдоним и индекс темы:

```
#define Alias 5
```

При вызове контекстной справки из приложения будут использоваться именно численные идентификаторы тем. Поэтому в файле связи должны быть описаны все идентификаторы, по которым будет осуществляться контекстный вызов. Файл должен иметь расширение `h`.

После создания файла связи его необходимо связать с файлом проекта. Для этого откройте окно диалога `HtmlHelp API information` на вкладке `Map` и добавьте файл связи в список подключаемых файлов. После этого в левой части окна HTML Help Workshop появился новый раздел [Map].

Создание файла тем, отображаемых во всплывающих окнах

Справочная система HTML Help, так же как и WinHelp, позволяет отображать темы справки во всплывающих окнах. Однако в отличие от WinHelp темы при этом должны быть подготовлены специальным образом. Для обеспечения возможности отображения тем во всплывающих окнах необходимо создать два дополнительных файла: файл с текстами тем и файл связи.

Файл с текстами тем создается в любом простейшем текстовом редакторе. Структура файла следующая: тема начинается с управляющей строки, содержащей директиву `.topic`, за которой через пробел указывается символьный идентификатор темы. Строки, следующие за управляющей — текст темы.

```
.topic символьный идентификатор темы № 1
Текст темы № 1
```

.topic символьный идентификатор темы № 2

Текст темы № 2

...

Файл с текстами тем, отображаемых во всплывающих окнах, должен иметь расширение txt.

Для тем, отображаемых во всплывающих окнах, необходимо создать отдельный файл связи. Синтаксис данного файла описан выше.

Подключение созданных файлов к файлу проекта справки выполняется с помощью окна диалога **HtmlHelp API Information**, на вкладке **Text Pop-ups**: для подключения файла с текстами тем следует щелкнуть на кнопке **Text file**, для подключения файла связи — на кнопке **Header file**.

Создание файла содержания

Содержание является полезным средством, позволяющим получить представление об общей схеме содержимого справочного файла. Для включения содержания в справочную систему необходимо создать отдельный файл содержания. В **HTML Help Workshop** для создания темы «Содержание» следует выполнить следующие действия.

1. Перейдите на вкладку **Contents**. В том случае если файл содержания ранее не был подключен к файлу проекта справки, появится окно диалога **Table of Contents Not Specified** (рис. 15.21).

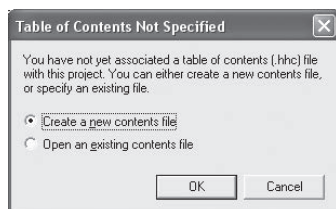


Рис. 15.21. Окно диалога **Table of Contents Not Specified**

2. Если файл содержания справочной системы не был создан вами ранее, то оставьте предлагаемый по умолчанию вариант **Create a new contents file** и щелкните на кнопке **OK**. Появится окно диалога **Сохранение**.
3. Укажите каталог и имя создаваемого файла содержания и щелкните на кнопке **Сохранить**. В результате будет создан пустой файл содержания, не имеющий никакой информации.
4. Для добавления заголовка или строки ссылки на тему справочной системы щелкните соответственно на одной из кнопок: **Insert a heading** или **Insert a page**. В первом случае в содержание добавляется заголовок, во втором — ссылка на тему. Но в любом случае откроется окно диалога **Table of Contents Entry** (рис. 15.22).
5. Задайте в окне **Table of Contents Entry** наименование строки содержания в поле **Entry title** и имя связанного с ней html-файла в расположенном ниже текстовом поле ввода. Щелкните на кнопке **OK**. В результате в разделе **Contents** появится строка содержания с заданным наименованием.

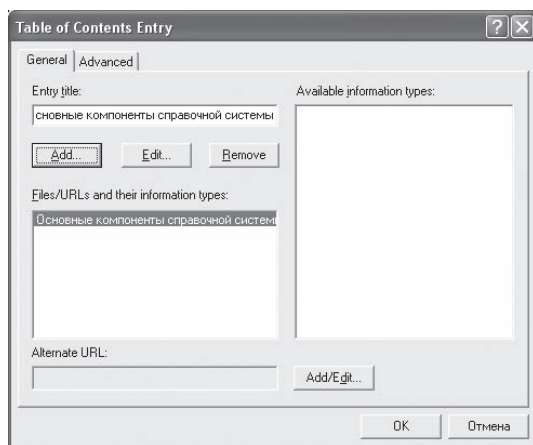


Рис. 15.22. Окно диалога Table of Contents Entry

6. Повторите эту процедуру для всех страниц данного раздела и всех разделов проекта. При необходимости с помощью кнопок со стрелками можно изменить положение элементов содержания в иерархической структуре.

Создание индексных файлов

Кроме содержания система HTML Help предусматривает еще два основных интерфейса для навигации:

- ☐ предметный указатель;
- ☐ средства полнотекстового поиска.

Чтобы их можно было использовать, необходимо сформировать индексные файлы.

Ключи к темам предназначены для организации предметного указателя. Как и строки, определяющие содержание тем и страниц справочной системы, ключи хранятся в специальном файле, который также сначала надо создать, а затем наполнить конкретным материалом. Это можно сделать следующим образом.

1. Для создания файла с ключами поиска перейдите на вкладку Index. При этом откроется окно диалога Index Not Specified, подобное окну Table of Contents Not Specified (см. рис. 15.21).
2. Примите предлагаемый по умолчанию вариант Create a new index file и щелкните на кнопке OK. Появится окно диалога Сохранение, в котором следует указать каталог и имя индексного файла. После этого будет создан специальный файл, предназначенный для хранения информации о ключевых словах справочной системы, не содержащий никакой информации.

Теперь пора приступить к наполнению индексного файла конкретным содержанием. Используя кнопки панели инструментов вкладки Index, можно создать новый ключ, редактировать ранее созданный ключ или удалить ключ.

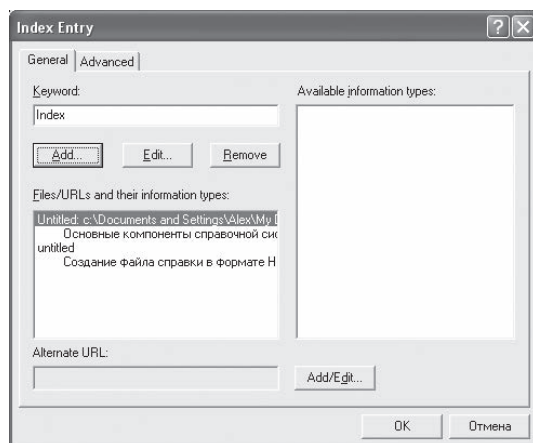


Рис. 15.23. Окно диалога Index Entry

3. Для добавления нового ключа щелкните на кнопке Insert а keyword, в результате чего откроется окно диалога Index Entry (рис. 15.23).
4. Введите в поле Keyword ключевую фразу, а затем, используя кнопку Add, добавьте в список Files/URLs and their information types темы справочной системы, на которые данная ключевая фраза должна ссылаться.
5. Для изменения уже заданных ключевых фраз используйте кнопку Edit, а для их удаления — кнопку Remove.

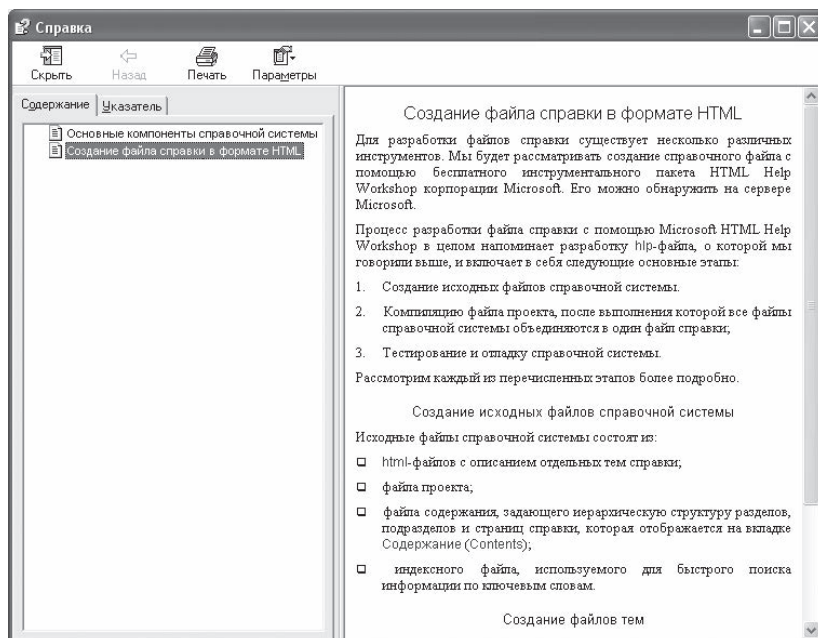


Рис. 15.24. Окно справочной системы HTML Help

ПРИМЕЧАНИЕ

Для обеспечения возможности полнотекстового поиска достаточно перейти в окно диалога Options, выбрать вкладку Compiler и установить флажок Compile full-text search information. При такой установке компилятор сформирует поисковую базу данных и сохранит ее в СНМ-файле.

Компиляция и тестирование файла справки

Окончательным шагом после подготовки проекта справочной системы является выполнение его компиляции. Перед этим необходимо сохранить все файлы проекта, для чего следует перейти на вкладку Project и щелкнуть на кнопке Save project, contents and index files на панели инструментов этой вкладки.

Для компиляции созданного проекта щелкните на кнопке Compile HTML file на панели инструментов HTML Help Workshop. Сообщения компилятора будут отображены в правой части главного окна HTML Help Workshop.

Чтобы просмотреть полученный файл справки, щелкните на кнопке View compiled file. При этом откроется окно, примерный вид которого представлен на рис. 15.24.

Использование справочной системы в приложениях

Итак, мы рассмотрели процедуру создания справочных файлов как в формате WinHelp, так и в формате HTML. Однако для создания полноценной справочной системы этого недостаточно — необходимо также объединить файлы справки с приложением.

Метод интеграции справочных файлов и приложения зависит от формата используемых файлов. Рассмотрим процедуру организации взаимодействия приложения со справочными файлами более подробно.

Подключение к приложению справочных файлов формата WinHelp

Все визуальные компоненты Delphi и класс TApplication обладают рядом свойств, обеспечивающих их взаимодействие со справочной системой WinHelp. Поэтому при использовании справки в формате WinHelp подключение справочных файлов обычно сводится к установке значений свойств объекта Application и визуальных компонентов. Для интеграции справочного файла с приложением следует выполнить два основных блока шагов.

1. Укажите имя файла справки, с которым будет взаимодействовать приложение. Имя hlp-файла задается с помощью свойства HelpFile объекта Application. Для этого используется окно диалога Options, открываемое после выбора команды Project ► Options главного меню Delphi. В этом окне выберите вкладку Application (рис. 15.25) и укажите имя файла справки в поле ввода Help file. После этого в файле проекта (текст которого можно открыть в редакторе кода с помощью команды Project ► View Source) появится следующая строка:

```
Application.HelpFile := 'TEST.HLP';
```

ПРИМЕЧАНИЕ

Справочный файл также можно присоединить к приложению, просто вручную написав эту строку.

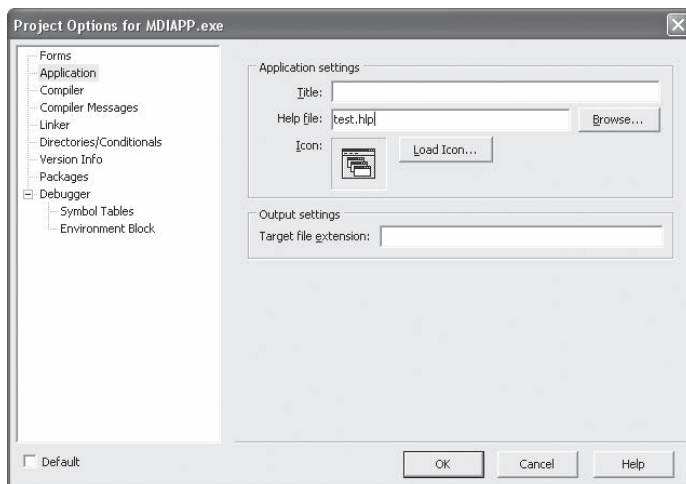


Рис. 15.25. Подключение файла справки к приложению

Рассмотренным способом задается файл справки сразу для всего приложения. Можно также связать файл справки только с одной формой приложения, для чего используется свойство `HelpFile` формы (класса `TForm`). Значение этого свойства изменяется в инспекторе объектов. Таким образом, можно связывать разные формы с разными файлами справки. Если значение свойства `HelpFile` формы не задано, то форма наследует его от объекта `Application`.

2. На втором шаге следует связать визуальные компоненты с соответствующими им темами в файле справки. Для этого используется свойство `HelpContext`, которым обладают все визуальные компоненты Delphi, способные иметь фокус ввода. В данном свойстве с помощью инспектора объектов следует указать числовой идентификатор темы, с которой должен быть связан данный компонент.

После выполнения указанных выше действий мы получим работающую контекстно-зависимую справку. При нажатии пользователем клавиши F1 будет открываться окно справочной системы WinHelp, в котором отображается тема, соответствующая свойству `HelpContext` компонента, имеющего фокус ввода.

Однако наряду с этим обычно также требуется обеспечить возможность обращения пользователя сразу к содержанию или к указателю справочного файла с использованием команд меню. Кроме того, в ряде случаев справочную информацию требуется выводить не в окне программы WinHelp, а во всплывающем окне. Для обеспечения всех этих возможностей следует использовать методы и события объекта `Application`, предназначенные для взаимодействия со справочной системой. Класс `TApplication` включает три метода, используемых для вызова справки:

- ❑ `function HelpCommand (Command: Word; Data: Longint): Boolean` — вызывает команду WinHelp API (application programming interface), указанную в параметре `Command`, передавая ей параметр `Data`;
- ❑ `function HelpContext (Context: THelpContext): Boolean` — открывает окно программы WinHelp, в котором выводит тему с номером, соответствующим параметру `Context`;
- ❑ `function HelpJump (const JumpID: string): Boolean` — аналогичен предыдущему методу, только тема задается не числовым идентификатором, а строковым.

Наиболее универсальным является метод `HelpCommand`, обеспечивающий доступ ко всем командам API WinHelp. Таких команд около 20, поэтому мы рассмотрим только основные.

- ❑ `HELP_COMMAND` — запускает макрос WinHelp. Параметр `Data` задает адрес строки, содержащей текст макроса. В строке можно задавать несколько макросов, разделенных точкой с запятой.
- ❑ `HELP_CONTENTS` — открывает окно с содержанием справочной системы. Параметр `Data` должен быть равен 0.
- ❑ `HELP_CONTEXT` — открывает окно WinHelp с темой, соответствующей параметру `Data`.
- ❑ `HELP_CONTEXTPOPUP` — отображает тему, номер которой задан параметром `Data`, во всплывающем окне.
- ❑ `HELP_INDEX` — отображает предметный указатель справочной системы. Параметр `Data` в этом случае задается равным 0.

При любой попытке обращения к справочной системе из приложения генерируется событие `OnHelp` объекта `Application`. Метод-обработчик этого события имеет следующий формат:

```
function (Command: Word; Data: Longint; var CallHelp: Boolean): Boolean
```

Здесь `Command` — команда API WinHelp; `Data` — параметр команды; `CallHelp` — параметр, указывающий, вызывать справочную систему или нет.

Сообщение `OnHelp` можно использовать для изменения способа отображения темы сообщения. Например, иногда бывает необходимо выводить контекстную справку во всплывающем окне. Однако по умолчанию при нажатии клавиши `F1` вызывается команда WinHelp API `HELP_CONTEXT`, которая отображает тему в обычном окне. В этом случае следует воспользоваться обработчиком события `OnHelp`.

Если задать его следующим образом, то темы с числовыми идентификаторами, лежащими в диапазоне от 1 до 10, будут отображаться во всплывающем окне:

```
function TForm1.ApplicationEvents1Help(Command: Word; Data: Integer; var
CallHelp: Boolean): Boolean;
const
    F : boolean = true;
begin
    case Data of
        1..10 :
```

```
        if F then begin
            F:=false;
            CallHelp:=false;
            Application.HelpCommand(HELP_CONTEXTPOPUP,Data);
        end
        else F:=true;
    end;
    result:=true;
end;
```

Обратите внимание на использование в приведенном примере дополнительной переменной F. Если бы мы задали в обработчике OnHelp следующий код, то при выполнении программы это привело бы к ошибке переполнения стека (Stack Overflow):

```
case Data of
    1..10 :
        begin
            CallHelp:=false;
            Application.HelpCommand(HELP_CONTEXTPOPUP,Data);
        end
end;
```

Дело в том, что вызов метода HelpCommand генерирует событие OnHelp, поэтому приведенный выше фрагмент кода приведет к бесконечной рекурсии.

Использование дополнительной переменной-флага приводит к тому, что при одном обращении к контекстной справке (нажатии пользователем клавиши F1) событие OnHelp будет генерироваться только дважды:

- ❑ первый раз событие OnHelp генерируется как реакция на нажатие клавиши F1. При этом выполняются операторы, расположенные в секции then оператора if; вызывается метод HelpCommand, а значение переменной F устанавливается равным false;
- ❑ второе обращение к обработчику OnHelp происходит вследствие генерации события OnHelp методом HelpCommand. Однако поскольку значение переменной-флага в этот момент равно false, то выполняться будут операторы секции else оператора if; переменной F будет присвоено значение true, что обеспечит вывод темы во всплывающее окно при повторном обращении к справке.

Использование в приложениях Delphi справочной системы HTML Help

Компоненты Delphi не поддерживают взаимодействие со справочными файлами в формате HTML Help. Поэтому для использования таких файлов всегда требуется вызывать функции API системы HTML Help. Заголовки этих функций содержатся в файлах `htmlhelp.h` и `htmlhelp.lib`, которые входят в поставку HTML Help Workshop. Однако эти файлы могут быть использованы только при разработке приложений на языке C/C++. Чтобы иметь возможность вызова функций API HTML Help, необходимо переписать файлы заголовков в синтаксисе языка Object Pascal. Это необязательно делать самим, в Интернет можно найти уже готовые решения. Например, библиотеку модулей Delphi с заголовка-

ми функций API HTML Help, а также с примерами и подробным описанием можно бесплатно загрузить с сервера <ftp://delphi-jedi.org/api/HtmlHelp.zip>. Основу данной библиотеки составляет модуль `HtmlHlp.pas`, содержащий описание констант, типов данных и функций, необходимых для работы с системой HTML Help.

Функция `HtmlHelp`

Взаимодействие с программой просмотра `chm`-файлов обеспечивается единственной функцией:

```
function HtmlHelp(hwndCaller: HWND; pszFile: PAnsiChar;
uCommand: UINT; dwData: DWORD): HWND; stdcall;
```

Ее параметры имеют следующий смысл:

- ❑ `hwndCaller` — дескриптор окна, которое является владельцем окна программы просмотра `chm`-файла;
- ❑ `pszFile` — путь либо к `chm`-файлу, либо к теме внутри `chm`-файла, в зависимости от параметра `uCommand`;
- ❑ `uCommand` — команда HTML Help;
- ❑ `dwData` — параметр команды.

В качестве параметра `hwndCaller` в большинстве случаев можно задавать значение, равное 0. В этом случае владельцем окна справки будет являться рабочий стол (desktop) Windows.

Формат строки `pszFile`, задающейся в качестве параметра, в общем случае имеет следующий вид:

`chm-файл [::\путь\тема.htm] [идентификатор окна]`

Путь к файлу темы должен полностью соответствовать пути, показываемому в разделе [FILES] файла проекта справки.

Команды `HtmlHelp`

Для обеспечения контекстно-зависимой справочной системы приложения необходимо вызывать функцию `HtmlHelp` с различными командами. Количество команд HTML Help довольно велико, поэтому мы рассмотрим только основные:

- ❑ `HH_DISPLAY_TOC` — открывает окно справочной системы, в левой части которого активна вкладка **Содержание**, а в правой части отображается заданная тема. Параметр `dwData` в этом случае может использоваться для задания темы. Данная команда может использоваться тремя способами:

```
HtmlHelp(0, PChar('HelpFile.chm'), HH_DISPLAY_TOC, 0);
HtmlHelp(0, PChar('HelpFile.chm::<путь>\TOPIC.htm '),
HH_DISPLAY_TOC, 0);
HtmlHelp(0, PChar('HelpFile.chm'), HH_DISPLAY_TOC,
DWORD(PChar('<путь>\TOPIC.htm')));
```

В первом случае в окне справки будет показана тема, заданная в качестве темы по умолчанию при создании файла справки. Во втором и третьем случаях отображается тема, содержащаяся в файле `topic.htm`. Файл темы указывается двумя путями: либо задается его полный путь в параметре `pszFile`,

либо в качестве параметра `dwData` передается адрес строки, содержащий путь к теме;

- ❑ `HH_DISPLAY_INDEX` — открывает окно справки с активной вкладкой *Указатель*. Параметр `dwData` в этом случае должен содержать адрес строки, в которой задано ключевое слово, поиск которого будет производиться. В левой части окна справки отображается тема, заданная по умолчанию. Вызов функции `HtmlHelp` с командой `HH_DISPLAY_INDEX` выглядит следующим образом:

```
HtmlHelp(0, PChar('HelpFile.chm'), HH_DISPLAY_INDEX,
         DWORD(PChar('<ключевое слово>')));
```

- ❑ `HH_HELP_CONTEXT` — открывает окно справки с темой, соответствующей числовому идентификатору, переданному в параметре `dwData`. Вызов функции `HtmlHelp` в этом случае имеет следующий вид:

```
HtmlHelp(0, PChar('HelpFile.chm'), HH_HELP_CONTEXT,
         <Topic ID>);
```

- ❑ `HH_DISPLAY_TOPIC` — открывает окно справочной системы с темой, заданной одним из двух способов: либо указанием полного пути к теме в параметре `pszFile`, либо передачей в качестве параметра `dwData` адреса строки, содержащей путь к теме. Варианты вызова функции `HtmlHelp` с данной командой имеют следующий вид:

```
HtmlHelp(0, PChar('HelpFile.chm>Main'), HH_DISPLAY_TOPIC,
         DWORD(PChar('<путь>\Topic.htm')));
HtmlHelp(0, PChar('HelpFile.chm:./<путь>\Topic.htm'),
         HH_DISPLAY_TOPIC, 0);
```

- ❑ `HH_TP_HELP_WM_HELP` — отображает текст темы во всплывающем окне. Данная команда может быть использована только для тем, созданных в специальном текстовом файле *Pop-Up* (см. выше). Формат вызова функции `HtmlHelp` для данной команды будет следующим:

```
HtmlHelp(<дескриптор элемента управления>,
         PChar('HelpFile.chm:./PopUp.txt'),
         HH_TP_HELP_WM_HELP, DWORD(@Ids));
```

При использовании команды в качестве первого параметра следует передавать дескриптор элемента управления, связанного с вызываемой темой. В качестве параметра `dwData` используется адрес массива, имеющего следующие особенности:

- тип элементов массива — `DWORD`;
- количество элементов должно быть четным;
- последние два элемента должны быть равны 0;
- все остальные элементы массива делятся на пары «дескриптор компонента — числовой идентификатор темы из файла *Pop-Up-тем*».

Например, следующий фрагмент кода отображает во всплывающем окне тему, связанную с компонентом `Edit1` и имеющую числовой идентификатор 100:

```
...
Ids[0] := Edit1.Handle;
Ids[1] := 100;
```

```

Ids[2] := 0;
Ids[3] := 0;
HtmlHelp(Edit1.Handle, PChar('test.chm:/PopUp.txt'),
  HH_TP_HELP_WM_HELP, DWORD(@Ids));
...

```

Внедрение справочного файла в формате HTML Help в приложение, разрабатываемое в Delphi

Итак, мы рассмотрели основные возможности функции, позволяющей производить вызов программы просмотра chm-файлов в различных режимах. Теперь покажем, как ее можно использовать в Delphi.

Вызов справки с помощью команд меню в данном случае проблем не вызывает и ничем не отличается от вызова справки WinHelp — достаточно в соответствующих методах производить вызов функции HtmlHelp с нужными параметрами.

Для обеспечения вызова контекстной справки следует использовать событие OnHelp объекта Application.

ПРИМЕЧАНИЕ

Для задания обработчика этого события следует использовать компонент ApplicationEvents, размещенный на странице Additional палитры компонентов.

Процесс подключения chm-файла к приложению состоит из следующих шагов.

1. Задайте в свойстве Application.OnHelp имя файла справки.

Хотя файл у нас не hlp-формата, это вполне допустимо. Вызывать программу WinHelp мы не будем, поэтому можем занести в свойство OnHelp любое строковое значение.

2. Задайте в свойстве HelpContext компонентов соответствующие им числовые идентификаторы тем файла справки.
3. Задайте функцию обработчик события OnHelp следующим образом:

```

function TForm1.ApplicationEvents1Help(Command: Word;
Data: Integer; var CallHelp: Boolean): Boolean;
begin
  // запретим вызов WinHelp
  CallHelp := False;
  // вызовем тему, соответствующую элементу,
  // который обладает фокусом ввода
  HtmlHelp(0, PChar(Application.HelpFile),
    HH_HELP_CONTEXT, Screen.ActiveControl.HelpContext);
  Result := True;
end;

```

Мы рассмотрели простейший случай, когда все темы будут открываться в одинаковых окнах. Если для разных элементов управления нужно отображать справку в окнах различного типа, то текст обработчика OnHelp будет несколько сложнее. В этом случае потребуется проанализировать, какой именно элемент управления сгенерировал данное событие, и в зависимости от этого изменять вызов функции HtmlHelp.

Часть V

Программирование для Интернета

Глава 16

Особенности интернет-приложений

В данной главе приводятся основные сведения о способах взаимодействия компьютеров в сети Интернет, рассматриваются основные типы веб-приложений и способы публикации данных в Интернете.

Основные сведения о сети Интернет

Рассмотрим некоторые основные сведения об Интернете и дадим определения ряда терминов, используемых при обсуждении вопросов программирования для Интернета.

Протокол IP как основа сети Интернет

Прообраз существующей сейчас Глобальной сети Интернет был создан более тридцати лет назад. Это была так называемая сеть ARPANET, разработанная Министерством обороны США. Одной из целей ее создания было исследование методов построения сетей, способных продолжать нормальное функционирование при частичном повреждении. В модели ARPANET всегда поддерживалась связь между компьютером-источником и компьютером-приемником. Основной принцип построения сети состоял в том, что любой компьютер мог связаться с любым другим компьютером. Передача данных между компьютерами была организована на основе протокола IP (Internet Protocol).

ПРИМЕЧАНИЕ

Под *протоколом* понимаются правила и описание работы сети, включающие правила установления и поддержания связи в сети, правила обращения с IP-пакетами и их обработки, описания сетевых пакетов IP.

Сеть проектировалась таким образом, чтобы для работы в ней не требовалось никакой информации о конкретной структуре сети. Для процесса передачи по ней компьютер-источник помещал данные в некий «конверт», указывал

на этом «конверте» конкретный адрес компьютера-приемника и передавал получившийся в результате этих действий пакет в сеть.

Именно протокол IP, являющийся очень удачным способом организации связи между совершенно разными компьютерами (которые к тому же могут работать под управлением разных операционных систем), в дальнейшем и стал основой для создания и развития компьютерных сетей.

ПРИМЕЧАНИЕ

Важность протокола IP как основы сети Интернет подчеркивается определением, данным некоторое время назад: «Интернет — это все сети, использующие протокол IP, которые кооперируются для формирования единой сети своих пользователей».

Однако следует заметить, что в настоящее время существуют способы подключения к Интернету и не IP-сетей. Поэтому приведенное определение сейчас уже не совсем корректно.

Многоуровневая сетевая модель

Для управления сетевым обменом данными используется несколько протоколов. Это обусловлено наличием большого количества правил, регламентирующих сетевое взаимодействие. Даже в простейшем случае, при передаче последовательности битов, необходимо разделить исходные данные на пакеты, к каждому пакету добавить служебную информацию (заголовок) и обеспечить его доставку. При приеме пакета необходимо проверить корректность данных и при необходимости организовать повторную его передачу. В более сложных случаях, при передаче каких-либо вполне конкретных данных (файлов, документов и т. п.), необходимо также включать в пакеты информацию о том, что представляет собой передаваемая последовательность битов и как ее интерпретировать на месте назначения.

Таким образом, при обмене информацией по сети требуется оговаривать множество деталей, поэтому протокол, реализующий все правила обмена данными, был бы чрезмерно сложным и неудобным в использовании. Поэтому применяют несколько протоколов, решающих задачу передачи данных на разных уровнях.

Взаимодействие протоколов разных уровней определяется многоуровневой сетевой моделью. Для сетевого обмена в Интернете используется модель DOD (*Department of Defense*), определяющая четыре уровня обмена данными:

- ☐ уровень сетевого доступа;
- ☐ межсетевой уровень;
- ☐ транспортный уровень;
- ☐ уровень приложений.

Низшие уровни трактуют пакеты высших уровней как данные, к которым добавляют служебную информацию для пакета соответствующего уровня на приемной стороне. При передаче же информации на более высокий уровень служебная информация более низкого уровня удаляется.

Уровень сетевого доступа

Уровень сетевого доступа является самым низким уровнем взаимодействия в сети. Он обеспечивает безошибочную передачу блоков данных. Только этот уровень оперирует такими элементами, как битовые последовательности, методы кодирования, маркеры. Уровень сетевого доступа несет ответственность за правильную передачу пакетов на участках между непосредственно связанными элементами сети и обеспечивает управление доступом к среде передачи. Вследствие своей сложности уровень сетевого доступа разделяется на два подуровня:

- MAC (Medium Access Control) — управление доступом к среде;
- LLC (Logical Link Control) — управление логической связью (каналом).

Уровень MAC управляет доступом к сети (с передачей маркера в сетях Token Ring или распознаванием конфликтов в сетях Ethernet) и обеспечивает управление сетью. Уровень LLC, действующий над уровнем MAC, и есть собственно тот уровень, который посылает и получает сообщения с данными.

Межсетевой уровень

Межсетевой уровень обеспечивает передачу данных в различные точки, разбросанные по всему миру. Различные части Интернета (отдельные локальные сети) соединяются между собой посредством компьютеров, которые называются узлами. Соединяемые сети могут быть сетями Ethernet, Token Ring, сетями на телефонных линиях и т. п. На узлах принимается решение о том, как перемещать данные (пакеты) по сети. Отдельные узлы сети не имеют прямых связей со всеми остальными узлами. Поэтому для работы такой системы необходимо, чтобы каждый узел имел информацию о существующих связях и о том, на какой из узлов следует передать пакет для его оптимальной доставки в точку назначения.

ПРИМЕЧАНИЕ

Процесс определения пути пакета называется *маршрутизацией*.

Для осуществления маршрутизации каждый узел имеет таблицу (называемую таблицей маршрутизации), где адресу точки назначения поставлен в соответствие адрес узла, на который следует послать данные. В Интернете составление и модификация таблиц маршрутизации (этот процесс тоже является частью маршрутизации и называется также маршрутизацией) определяется протоколами ICMP (Internet Control Message Protocol), RIP (Routing Internet Protocol) и OSPF (Open Shortest Path First). Узлы, выполняющие функции маршрутизации, называются *маршрутизаторами*.

Адресация пакетов на межсетевом уровне обеспечивается протоколом IP. В заголовок пакета помещается информация, называемая адресом IP, которой достаточно, чтобы определить, куда доставить пакет данных. IP-адрес состоит из четырех байт. При текстовой записи байты отделяются друг от друга точками, например 127.0.0.1.

Каждый компьютер, подключенный к Интернету, имеет уникальный адрес. Однако на межсетевом уровне определяется лишь сеть, в которой находится

конкретный компьютер. Для определения места расположения в локальной сети компьютера с данным числовым IP-адресом локальные сети используют свои собственные протоколы сетевого уровня (например, локальные сети Ethernet для отыскания Ethernet-адреса по IP-адресу компьютера, находящегося в данной сети, используют протокол ARP).

Информация, пересылаемая по сетям IP, делится по границам байтов на пакеты. Размер пакета обычно лежит в диапазоне от 1 до 1500 байт.

Транспортный уровень

Транспортный уровень определяет правила поддержки сетевых соединений. Типичным протоколом транспортного уровня является протокол ТСП (Transmission Control Protocol). Протокол ТСП занимается проблемой пересылки больших объемов информации, основываясь на возможностях протокола IP. ТСП делит информацию, которую надо переслать, на несколько частей и нумерует их, чтобы обеспечить возможность последующего восстановления. Каждая порция информации вместе с номером образует ТСП-пакет, который затем помещается в отдельный IP-пакет, с которым сеть уже «умеет» обращаться.

Получатель (ТСП-процесс) распаковывает IP-пакеты и получает ТСП-пакеты, далее распаковывает их и объединяет данные. Если какой-то информации недостает, ТСП требует переслать эту часть информации снова. Благодаря такой технологии информация собирается в нужном порядке и полностью восстанавливается.

При пересылке из-за наличия помех на линиях связи пакеты могут не только теряться, но и искажаться. Протокол ТСП решает и эту проблему. Для этого используется специальная система кодов, исправляющих ошибки. Наиболее простым примером таких кодов является код, использующий добавление к каждому пакету контрольной суммы (а к каждому байту — бита контроля четности). При создании ТСП-пакета вычисляется контрольная сумма, которая записывается в ТСП-заголовок. Если при приеме информации вычисленная сумма не совпадает с той, что указана в заголовке, это свидетельствует о том, что при передаче произошла ошибка и следует переслать этот пакет заново.

Таким образом, протокол ТСП обеспечивает гарантированную доставку пакетов, освобождая прикладные процессы от необходимости использовать режим ожидания и повторные передачи для обеспечения надежности.

ПРИМЕЧАНИЕ

Как уже отмечалось, протокол ТСП тесно связан с протоколом IP. Поэтому эти протоколы вместе часто именуют как ТСП/IP. Термин ТСП/IP обычно означает все, что связано с протоколами ТСП и IP. Он охватывает целое семейство протоколов, прикладные программы и даже саму сеть.

Уровень приложений

Уровень приложений определяет интерфейс между двумя системами на уровне приложений. На этом уровне определяется, как компьютер обрабатывает полученные данные. Для его поддержки разработано несколько протоколов, используемых для передачи вполне определенной информации:

- ❑ *передача гипертекстовых документов* — протокол HTTP (Hypertext Transfer Protocol);
- ❑ *передача файлов* — протокол FTP (File Transfer Protocol);
- ❑ *передача сообщений электронной почты* — протоколы SMTP (Simple Mail Transfer Protocol) и POP (Post Office Protocol).

Особенностью протоколов уровня приложений является то, что обмен служебной информацией между ними производится в символьном виде.

Адресация в Интернет

Как уже отмечалось выше, компьютеры в Интернете идентифицируются по IP-адресу, уникальному в пределах всего Интернета. Однако пользователям крайне неудобно производить обращение к требуемому серверу с использованием IP-адресов, так как они не несут никакого осмысленного значения и трудны для запоминания. Поэтому серверам Интернета присваивают *символьные адреса*. Все приложения Интернета позволяют пользоваться символьными именами вместо числовых IP-адресов.

Доменная система имен

При присваивании серверу символьного имени используется так называемая *доменная система имен* (Domain Name System), основанная на иерархии доменов. Доменом называется область иерархического пространства имен Интернета, которая обозначается уникальным доменным именем. Доменным именем называется символьное имя домена, и это имя должно быть уникальным в рамках данного домена. Полное имя домена имеет вид нескольких идентификаторов, разделенных точками:

domain_n. ... domain_2.domain_1

Чем дальше (правее) расположен в адресе домен, тем шире охватываемая им область. Домен высшего уровня (самый правый) представляет собой либо двухбуквенный шифр страны, либо трехбуквенный код, описывающий род деятельности владельца. Основные двухбуквенные домены: Россия — ru (или su), США — us, Германия — de, Англия — uk и т. д.

Трехбуквенная система ранее первоначально применялась исключительно в США, но в настоящее время множество подобных имен принадлежит компаниям и организациям, расположенным за пределами США. Трехбуквенные домены имеют следующий смысл:

- ❑ com — коммерческие организации. Международная доменная зона .com является наиболее престижной, популярной и «дорогой». Самые известные интернет-адреса зарегистрированы именно в ней;
- ❑ edu — учебные организации;
- ❑ gov — правительственные организации;
- ❑ int — международные организации;
- ❑ mil — военные организации;

- ❑ net — сетевые организации;
- ❑ org — некоммерческие организации.

За доменами верхнего уровня следуют домены, определяющие либо регионы, либо организации. Далее следуют уровни иерархии, которые могут быть закреплены либо за небольшими организациями, либо за подразделениями крупных организаций.

Регистрацией и распределением доменных имен ведает международная организация InterNIC. В ней существует специальная служба WhoIs для поиска владельца домена по имени домена или IP-адресу.

При обращении к серверу по символьному имени компьютер должен преобразовать имя в IP-адрес. Для этого производится запрос у так называемого DNS-сервера — узла, обладающего соответствующей базой данных, в число обязанностей которого входит обслуживание такого рода запросов. DNS-сервер начинает обработку имени с правого его конца и, перемещаясь по нему влево, постепенно сужает поиск. DNS-сервер необязательно должен «знать» IP-адрес, соответствующий запрошенному символьному имени. Для выяснения этого адреса он может связываться с другими DNS-серверами.

Порты и службы

IP-адрес позволяет точно идентифицировать компьютер, но в ряде случаев этого недостаточно. Дело в том, что на каждом узле могут быть запущены самые разные службы Интернет, обеспечивающие передачу электронной почты, файлов, гипертекстовой информации и т. п. Каждая служба использует свой протокол прикладного уровня. Например, для передачи гипертекстовых документов используется протокол HTTP, передача файлов производится по протоколу FTP, для работы с электронной почтой используются протоколы SMTP, POP3, IMAP и т. д.

Для упорядочения работы каждой службе отведен отдельный *порт*, представляющий собой число от 0 до 65534. Для наиболее популярных служб зарезервированы стандартные номера портов. Так, для FTP это 21, для HTTP — 80, SMTP — 25, POP3 — 110. Однако это всего лишь общепринятые значения по умолчанию, поэтому владелец узла может настроить эти службы на работу с совершенно другими портами. Часто это позволяет легко решать некоторые проблемы, например обеспечить поддержку различных кодировок кириллицы в Веб. Для реализации этого достаточно предусмотреть автоматическую перекодировку документа на сервере в зависимости от того, с каким портом общается клиентское приложение.

Унифицированный указатель ресурсов

Унифицированные указатели ресурсов (Uniform Resource Locator, URL) предназначены для адресации сетевых ресурсов документов, файлов и т. п. В самом общем виде URL записывается следующим образом:

```
[протокол]://[имя][:пароль]@[адрес][:порт][/путь/]  
[документ][?дополнительная информация]
```

Здесь:

- ❑ протокол — символьное обозначение протокола, используемого для доступа к ресурсу (например, ftp, http и т. д.);
- ❑ имя — имя пользователя;
- ❑ пароль — используется в сочетании с именем пользователя при работе с ресурсами, доступ к которым ограничен;
- ❑ адрес — адрес узла в доменной или цифровой форме;
- ❑ порт — номер порта. Если он отсутствует, то используется порт по умолчанию для данного протокола;
- ❑ путь — путь на сервере от его корневого каталога либо относительно текущего каталога;
- ❑ документ — имя документа;
- ❑ дополнительная информация — используется при работе с серверными приложениями.

ПРИМЕЧАНИЕ

Далеко не все составляющие URL являются обязательными. Во многих случаях достаточно указать только адрес узла.

Основы веб-программирования

В данном разделе приводятся некоторые базовые сведения, необходимые для понимания основ создания приложений для веб-серверов. Дается также обзор наиболее часто используемых типов веб-приложений и особенностей их использования.

Основные понятия и термины

В настоящее время наиболее развитой частью Интернета является WWW (World Wide Web) — *система публикации ресурсов*, представленных в виде гипертекстовых документов. Под *публикацией* обычно понимается возможность размещения на сервере некоторого гипертекстового документа, содержащего как статические, так и динамические данные. Для взаимодействия с сервером, предназначенным для веб-публикаций (веб-сервером), используется протокол HTTP.

ПРИМЕЧАНИЕ

Термин «веб-сервер» имеет несколько различных трактовок, наиболее распространенными из которых являются:

1. *компьютер*, предназначенный для публикации гипертекстовых документов;
2. *программный продукт*, предназначенный для обеспечения доступа к гипертекстовым документам, расположенным на компьютере. В этом случае веб-сервер реализует обработку запросов, поступающих от клиентов по протоколу HTTP.

Чтобы избежать путаницы, в дальнейшем изложении под веб-сервером мы будем понимать только программный продукт. Компьютер же, на котором функционирует веб-сервер, будем называть WWW-сервером.

Для просмотра гипертекстовых документов (которые часто называются также веб-страницами) используются специальные программы, называемые *браузерами*. На сегодняшний день наиболее известными являются два браузера: Microsoft Internet Explorer и Mozilla Firefox. Программа-браузер выполняет интерпретацию команд языка разметки гипертекста (HTML) и отображает содержимое HTML-документа.

По структуре организации веб-страницы можно подразделить на статические и динамические:

- ❑ *статические* страницы содержат некоторую жестко заданную информацию, для изменения которой необходимо вносить изменения в гипертекстовый документ;
- ❑ *динамические* страницы позволяют отображать данные, которые могут модифицироваться без изменения самого HTML-документа (например, информацию, извлекаемую из базы данных). Для создания динамических HTML-страниц обычно используют специальные серверные расширения, называемые *скриптами*, или *сценариями* (также иногда используется термин «*веб-приложения*»). Типичная задача, выполняемая сценарием, — получение информации из некоторого внешнего источника (например, из базы данных), которая затем представляется в виде HTML-документа и передается серверу, а он, в свою очередь, отправляет ее клиенту. Кроме того, сценарии позволяют обеспечить интерактивное взаимодействие с клиентом, обрабатывая данные, передаваемые от клиента к серверу. Подобным образом реализуется, например, возможность поиска либо выборка из базы данных именно той информации, которую запрашивает пользователь.

Веб-дизайн и веб-программирование

Реализация и поддержка функционирования веб-серверов предполагает решение следующих трех основных задач:

- ❑ подготовка материалов к веб-публикации, редактирование, дизайн, соблюдение единого стиля и единообразного оформления веб-страниц, поддержка связности веб-документов и т. п.;
- ❑ обеспечение динамического представления информации на веб-странице: создание интерактивных веб-страниц, организация разных видов поиска информации, статистика посещений страницы, регистрация пользователей, предоставление регламентированного доступа к информации, организация доступа к базам данных;
- ❑ администрирование веб-сервера как компонента системы World Wide Web. Таким образом, специалисты, разрабатывающие и поддерживающие веб-сервер, должны обладать знаниями в весьма различных областях;
- ❑ первая задача обычно решается специалистом в конкретной предметной области — автором будущих веб-страниц, а также специалистом по оформлению веб-страниц. Здесь выполняются разработка и создание статической веб-страницы, или, точнее, статической части веб-документа. Для решения этой задачи часто используются специальные визуальные средства разработки HTML-документов;

ПРИМЕЧАНИЕ

В настоящее время для оформления веб-страниц кроме языка HTML широко применяются фрагменты на языках Java, JavaScript, VBScript, предоставляющие гораздо большие возможности для представления информации, чем язык HTML.

- ❑ вторая задача подразумевает разработку средств, расширяющих возможности веб-сервера, и поэтому решается специалистами в области программирования. Ее цель состоит в реализации динамического изменения содержимого страницы в сочетании с возможностью интерактивного режима работы. Данная задача решается путем разработки сценариев;
- ❑ третья задача решается специалистами по системному администрированию. При использовании серверов, работающих под управлением операционной системы Windows, серверные расширения могут создаваться с помощью самых разных средств разработки (практически все современные средства разработки приложений для Windows обеспечивают возможность разработки веб-приложений).

В процессе разработки веб-страницы принято выделять две составляющие — *веб-дизайн* и *веб-программирование*. Между ними нет четкой границы. Чаще всего под веб-дизайном понимают разработку статической части веб-страницы на языке HTML. Включение же в HTML-документ фрагментов на языке Java обычно относят к области веб-программирования. Исключительно к области веб-программирования относят разработку расширений для веб-сервера.

ПРИМЕЧАНИЕ

В дальнейшем под веб-программированием мы будем понимать задачу разработки сценариев, необходимых для организации динамических документов.

Для более подробного изучения процесса разработки дизайна веб-приложений рекомендуем книгу Томаса А. Пауэлла «Web-дизайн».

Протокол HTTP

Программы, обеспечивающие работу WWW, используют для обмена данными протокол HTTP (напомним, что это протокол уровня приложений). Поэтому если вы собираетесь заниматься разработкой веб-приложений, необходимо получить хотя бы основные представления об этом протоколе.

Как уже отмечалось выше, протоколы приложений могут обмениваться только текстовой информацией. Для обеспечения возможности передачи двоичных файлов по протоколу HTTP используется спецификация MIME (Multipurpose Internet Mail Extension). Согласно спецификации MIME, формат данных описывается следующим образом:

<тип>/<подтип>

Тип определяет, какого рода информация содержится в двоичном файле (текст, приложение, изображение, видеозапись и т. п.), а подтип — формат файла.

Сеанс взаимодействия с сервером HTTP в наиболее общем виде состоит из следующих шагов:

- ☐ установление TCP-соединения;
- ☐ запрос клиента;
- ☐ ответ сервера;
- ☐ разрыв TCP-соединения.

Запрос клиента представляет собой просто требование на передачу HTML-документа или какого-либо другого ресурса. Ответ сервера — код запрашиваемого ресурса.

Запрос клиента

Запрос клиента состоит из четырех компонентов:

- ☐ строки состояния;
- ☐ поля заголовка;
- ☐ пустой строки;
- ☐ тела запроса.

Строка состояния имеет следующий формат:

<метод запроса> <URL ресурса> <версия протокола HTTP>

Прокомментируем отдельные структурные блоки этого формата:

- ☐ Метод запроса определяет вид воздействия на ресурс, указанный с помощью URL. Наиболее важны два метода: GET и POST:
 - метод GET предназначается для получения ресурса с указанным URL-адресом. При получении запроса GET сервер должен включить код ресурса в ответ клиенту. При этом ресурс необязательно является гипертекстовым документом;
 - основное назначение метода POST — передача данных на сервер. Однако на практике метод POST может применяться по-разному, в том числе и для получения информации с сервера;
- ☐ Версия протокола обычно задается в следующем формате:

HTTP/<версия>

Например, при использовании версии HTTP 1.0 данная строка выглядит так:

HTTP/1.0

Поля заголовка используются для передачи серверу дополнительной информации. Каждое поле заголовка имеет следующий формат:

<имя поля>: <значение>

Рассмотрим назначение некоторых наиболее часто используемых полей заголовка:

- ☐ Host — доменное имя или IP-адрес сервера, к которому обращается клиент;
- ☐ From — адрес электронной почты пользователя;

- ❑ Accept — MIME-типы данных, обрабатываемые клиентом. Может содержать несколько значений, разделяемых запятыми. Обычно используется для того, чтобы сообщить серверу о типах графических файлов, поддерживаемых клиентом;
- ❑ Accept-Language — идентификаторы, с помощью которых сообщаются языки, поддерживаемые клиентом. Разделяются запятыми;
- ❑ Accept-Charset — идентификаторы, сообщающие серверу о поддерживаемых клиентом кодировках. Разделяются запятыми;
- ❑ Content-Type — MIME-тип данных, содержащихся в теле запроса;
- ❑ Content-Length — число символов, содержащихся в теле запроса;
- ❑ Connection — управляет TCP-соединением. Если в этом поле задано значение Close, то после обработки запроса соединение разрывается. Если задано значение Keep-Alive, то соединение сохраняется и может быть использовано для последующих запросов;
- ❑ User-Agent — информация о клиенте.

Тело запроса в большинстве случаев отсутствует. Наиболее часто тело запроса используется в тех случаях, когда требуется передать серверу информацию, введенную пользователем.

Ниже приведен пример запроса:

```
GET http://www.altavista.com HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/4.04 [en] (Win95; I)
Host: www.altavista.com
Accept: image/gif, image/jpeg, image/png, */*
Accept-Language: en, ru
Accept-Charset: ISO8859-1, Win1251, *
```

Ответ сервера

С точки зрения веб-программирования структура ответа сервера гораздо более важна, чем структура запроса клиента. Выполняющиеся на сервере программы (разработкой которых, собственно, и занимается веб-программист), должны быть способны сами сформировать ответ клиенту.

Основные компоненты ответа полностью аналогичны компонентам запроса клиента и включают в себя следующие элементы:

- ❑ строку состояния;
- ❑ поля заголовка;
- ❑ пустую строку;
- ❑ тело запроса.

Строка состояния имеет следующий формат:

<версия протокола> <код ответа> <пояснения>

Здесь:

- ❑ Версия протокола задается в том же формате, что и в запросе клиента;
- ❑ Код ответа представляет собой трехзначное десятичное число, обозначающее результат обработки запроса сервером;

- ❑ Пояснения представляют собой расшифровку кода ответа в символьном виде. Это просто строка символов, не обрабатываемая клиентом и предназначенная для системного администратора.

Коды ответов подразделяются на пять групп. Группа, к которой относится код ответа, определяется старшим разрядом кода:

- ❑ 1 — информационное сообщение. Означает, что сервер продолжает обработку запроса клиента. Используется довольно редко;
- ❑ 2 — сообщение об успешной обработке запроса клиента;
- ❑ 3 — сообщение о перенаправлении запроса;
- ❑ 4 — сообщение об ошибке в запросе клиента;
- ❑ 5 — сообщение об ошибке сервера.

Наиболее часто встречающиеся коды ответов приведены в табл. 16.1.

Таблица 16.1. Коды ответов сервера

Код ответа	Строка пояснения	Описание
100	Continue	Часть запроса принята, сервер ожидает от клиента продолжения запроса
190	OK	Запрос успешно обработан, в ответе передается затребованный ресурс
191	Created	Запрос успешно обработан, на сервере создан новый ресурс
192	Accepted	Запрос принят сервером, но его обработка еще не закончена
301	Multiple Choice	Запрос указывает более чем на один ресурс
302	Moved Permanently	Затребованный ресурс больше не располагается на сервере
400	Bad Request	Запрос клиента содержит синтаксическую ошибку
403	Forbidden	Затребованный ресурс недоступен для данного пользователя
404	Not Found	Затребованный ресурс отсутствует на сервере
405	Method Not Allowed	Сервер не поддерживает указанный в запросе метод
500	Internal Server Error	Внутренняя ошибка сервера
501	Not Implemented	Сервер не обладает необходимыми функциональными возможностями для выполнения запроса
503	Service Unavailable	Служба недоступна
505	HTTP Version not Supported	Указанная в запросе версия HTTP не поддерживается сервером

Поля заголовка в ответе сервера имеют такую же структуру, что и в запросе клиента. Наиболее важны следующие поля:

- ❑ Server — наименование и номер версии веб-сервера;
- ❑ Allow — список методов, доступных для данного сервера;
- ❑ Content-Language — перечень языков, которые должен поддерживать клиент для корректного отображения передаваемого ресурса;
- ❑ Content-Type — MIME-тип данных, содержащихся в теле ответа сервера;

- ❑ Content-Length — размер данных, содержащихся в теле ответа сервера;
- ❑ Last-Modified — дата и время последнего изменения затребованного ресурса;
- ❑ Date — дата и время создания ответа сервера;
- ❑ Expires — дата и время, определяющие момент, когда информация, переданная клиенту, считается устаревшей;
- ❑ Location — адрес реального расположения ресурса. Используется для переадресации запроса;
- ❑ Cache-Control — директивы управления кэшированием.

В теле ответа содержится код передаваемого клиенту ресурса. Это может быть HTML-документ или любой другой ресурс. Способ обработки ресурса указывается в поле заголовка Content-type.

Ниже приведен пример ответа сервера, полученный в ответ на запрос HTML-документа:

```
HTTP/1.1 190 OK
Date: Sat, 11 Nov 1900 14:23:07 GMT
Server: Apache/1.3.6 (Unix) PHP/3.0.7 rus/PL28.12
Connection: close
Content-Type: text/html; charset=windows-1251
Expires: Thu, 01 Jan 1970 00:00:01 GMT
Last-Modified: Sat, 11 Nov 1900 14:24:44 GMT
Vary: accept-charset, user-agent
```

```
<html>
<head>
<meta name="author" content="WEBLab">
<title>Novgorod On-Line. Добро пожаловать в Великий
Новгород.</title>
</head>
<STYLE TYPE="text/css"><!--
A {text-decoration: none}
--></STYLE>
...
```

ПРИМЕЧАНИЕ

Чтобы получить приведенный выше ответ сервера, использовалась простейшая клиентская программа telnet.exe, входящая в поставку Widows. С помощью нее можно установить связь с любым сервером, имя которого указывается пользователем (для установления связи с веб-сервером следует использовать порт 80). Запрос к серверу формируется вручную. Ответ сервера никак не интерпретируется, а просто отображается в виде текста в окне терминала.

Язык HTML

Хотя разработка HTML-документов относится к области веб-дизайна, разработчик веб-приложений также должен знать этот язык. Результатом выполнения сценария, как правило, является создание HTML-документа. Поэтому нельзя разрабатывать веб-приложения без знания хотя бы основ языка разметки гипертекста (HTML).

В рамках одного раздела невозможно дать полное описание всех возможностей и особенностей языка HTML (хотя он и не относится к числу очень сложных). Поэтому здесь приводятся лишь основные сведения, достаточные для создания простых HTML-документов.

Структура HTML-документа

HTML-документ представляет собой обычный текстовый файл, содержащий текст документа и специальные языковые конструкции — *теги*, используемые для разметки документа и управления его отображением. Для создания HTML-документа можно использовать любой простейший текстовый редактор.

ПРИМЕЧАНИЕ

Если в состав HTML-документа входят графические изображения, то они хранятся в отдельных файлах. При этом в тексте HTML-документа указывается ссылка на соответствующий файл. Для хранения изображений в основном используются файлы форматов JPEG, GIF и PNG.

Теги обычно используются парами, состоящими из открывающего и закрывающего тега. Все теги начинаются с символа < и оканчиваются символом >. Открывающий тег имеет следующий формат:

<имя тега [атрибуты]>

Закрывающий тег имеет следующий вид:

</имя тега>

Любой документ в формате HTML начинается с открывающего тега <HTML> и заканчивается тегом </HTML>. Он состоит из двух частей:

- ❑ раздела заголовка (определяемого тегом HEAD);
- ❑ тела, которое включает собственно содержимое документа и определяется тегом BODY.

В общем виде документ HTML имеет следующую структуру:

```
<HTML>
  <HEAD>
    Раздел заголовка
  </HEAD>
  <BODY>
    Тело документа
  </BODY>
</HTML>
```

Раздел заголовка

Раздел заголовка содержит различного рода служебную информацию (например, ключевые слова, используемые поисковыми машинами), не считающуюся содержимым документа. Наиболее часто в заголовке применяются следующие теги:

- ❑ <TITLE> — заголовок HTML-документа, который отображается в строке заголовка окна браузера;

- ❑ `<BASE>` — базовый адрес, используемый при обработке *относительных URL-адресов* (это понятие будет рассмотрено ниже);
- ❑ `<LINK>` — тег, используемый для связи с другими HTML-документами;
- ❑ `<META>` — дополнительная информация об HTML-документе.

ПРИМЕЧАНИЕ

Из всех приведенных выше тегов только один используется в паре с закрывающим тегом — `TITLE`.

Тело документа

Тело документа содержится между тегами `<BODY>` и `</BODY>` и включает всю информацию, которая отображается в окне браузера.

ПРИМЕЧАНИЕ

В некоторых случаях вместо тега `BODY` используется тег `FRAMESET`, который определяет специальный тип документа — HTML-документ с фреймами (или кадрами). В этой книге мы не будем рассматривать такие HTML-документы.

В теле документа используются специальные теги, обеспечивающие форматирование текста документа и включающие в него изображения, таблицы и формы.

Теги форматирования текста

Приведем описание основных тегов, используемых для задания формата отображения текста в окне браузера.

Заголовки

HTML-документ может содержать шесть уровней заголовков. Для задания заголовка используются пары тегов:

`<H1>` заголовок первого уровня `</H1>`

`<H2>` заголовок второго уровня `</H2>`

...

`<H6>` заголовок шестого уровня `</H6>`

В открывающем теге можно указать дополнительный атрибут `ALIGN`, определяющий способ выравнивания текста заголовка. Данному атрибуту можно задавать одно из трех значений:

- ❑ `LEFT` — выравнивание по левому краю;
- ❑ `RIGHT` — выравнивание по правому краю;
- ❑ `CENTER` — выравнивание по центру.

Например, для создания заголовка первого уровня, выровненного по центру, используется следующая строка:

`<H1 ALIGN=CENTER> Текст заголовка </H1>`

Абзацы

Текст, относящийся к одному абзацу, заключается между тегами `<P>` и `</P>`. Каждый абзац отделяется от предыдущего увеличенным межстрочным интервалом. Так же как и для заголовков, для абзаца можно задавать способ выравнивания текста с помощью атрибута `ALIGN`.

Если требуется начать текст с новой строки в пределах одного абзаца, используется тег `
`. При использовании этого тега тип выравнивания не изменяется.

Списки

HTML-документ может содержать как маркированные, так и нумерованные списки. Для создания маркированных списков используются теги `` и ``, нумерованных — `` и ``. В обоих случаях каждый элемент списка помещается между тегами `` и ``. Допускается создание вложенных списков.

Выделение фрагментов текста

Язык HTML позволяет выделять отдельные слова и даже символы документа. Приведем основные теги, используемые при выделении:

- ☐ ` ... ` — выделение полужирным шрифтом;
- ☐ `<I> ... </I>` — выделение курсивом;
- ☐ `<U> ... </U>` — выделение подчеркиванием.

Указанные теги задают способ выделения фрагмента текста явным образом. Наряду с ними можно использовать теги, которые лишь указывают, что текст должен быть выделен, не обозначая способа выделения. В этом случае выбор способа выделения определяется браузером:

- ☐ ` ... ` — выделенному тексту должно уделяться внимание;
- ☐ ` ... ` — выделенному тексту должно уделяться особое внимание;
- ☐ `<KBD> ... </KBD>` — обозначение клавиши клавиатуры;
- ☐ `<VAR> ... </VAR>` — обозначение переменной;
- ☐ `<CITE> ... </CITE>` — цитата.

Рассмотрим пример HTML-документа, в котором используется большинство из рассмотренных тегов:

```
<HTML>
  <HEAD>
    <TITLE> Пример HTML-документа </TITLE>
  </HEAD>
  <BODY>
    <H2 ALIGN=CENTER> Пример использования списков </H2>
    <P>
      Маркированный список:
      <UL>
        <LI>Элемент <B>1</B></LI>
        <LI>Элемент <B>2</B></LI>
        <LI>Элемент <B>3</B></LI>
      </UL>
      Нумерованный список:
      <OL>
```

```

<LI>Элемент 1</LI>
<LI>Элемент 2</LI>
<LI>Элемент 3</LI>
</OL>
Вложенный список:
<OL>
<LI><U>Элемент 1</U></LI>
<UL>
<LI><I>Элемент</I> 1.1</LI>
<LI><I>Элемент</I> 1.2</LI>
</UL>
<LI><U>Элемент 2</U></LI>
<UL>
<LI><I>Элемент</I> 2.1</LI>
<LI><I>Элемент</I> 2.2</LI>
</UL>
<LI><U>Элемент 3</U></LI>
</OL>
</P>
</BODY>
</HTML>

```

В приведенном выше HTML-коде указан один параметр заголовка — заголовок окна браузера, задан один заголовок текста с выравниванием по центру и создано три вида списков — маркированный, нумерованный и вложенный. Вид данного документа в окне браузера Mozilla Firefox показан на рис. 16.1.

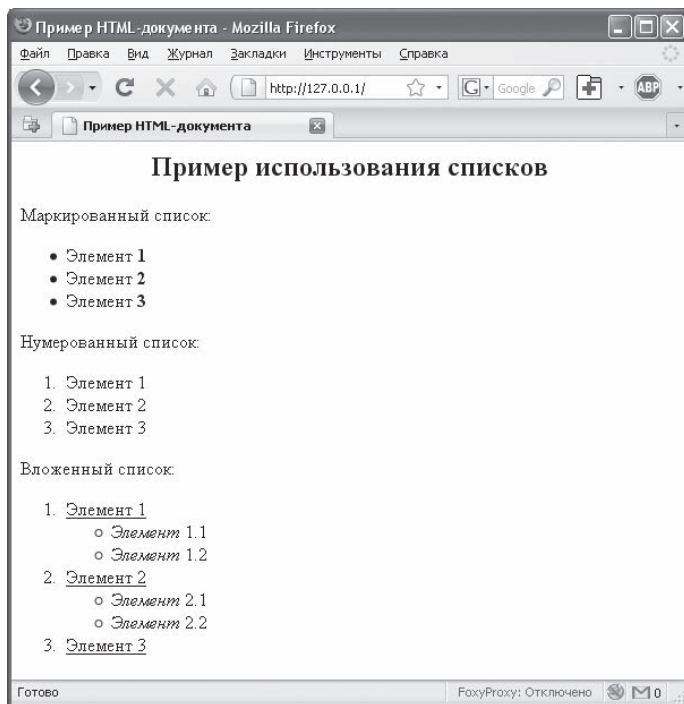


Рис. 16.1. Пример HTML-документа

Гиперссылки

Гиперссылки обеспечивают связь между различными HTML-документами. Гиперссылка представляет собой фрагмент HTML-документа (текст или изображение), щелчок на котором приводит к загрузке другого документа.

Для создания гиперссылки используется пара тегов `<A>` и ``. Заключенный между ними фрагмент HTML-документа при просмотре будет отображаться как гиперссылка. Тег `<A>` обязательно должен использоваться совместно с атрибутом `HREF`. С помощью него задается ссылка на документ, к которому должен быть произведен переход при щелчке на гиперссылке.

Таким образом, фрагмент HTML-документа, задающий гиперссылку, в общем виде выглядит так:

```
<A HREF=URL_ресурса> фрагмент документа </A>
```

Здесь URL-адрес, задаваемый атрибутом `HREF`, может быть двух видов: *абсолютным* и *относительным*:

- ❑ *абсолютный URL-адрес* уже был рассмотрен нами выше. В нем содержится полная информация о местоположении ресурса и протоколе обращения к ресурсу;
- ❑ *относительный URL-адрес* указывает расположение ресурса относительно местоположения текущего HTML-документа. Например, строка

```
<A HREF=doc1.html> Переход к документу 1 </A>
```

создает гиперссылку, указывающую на гипертекстовый документ, содержащийся в файле `doc1.html`, который размещен в том же каталоге, что и текущий документ.

Если абсолютный адрес документа, содержащего приведенную гиперссылку, выглядит, например, как `http://www.domen.ru/information/main.html`, то абсолютным адресом, на который эта гиперссылка ссылается, будет `http://www.domen.ru/information/doc1.html`.

Тег `<A>` имеет еще одно назначение — кроме создания гиперссылок он позволяет устанавливать маркеры для организации переходов по гиперссылкам в пределах одного документа HTML. Для задания маркера тег `<A>` используется совместно с атрибутом `NAME`:

```
<A NAME="имя_маркера"> текст </A>
```

В этом случае текст, заключенный между тегами `<A>` и ``, при отображении никак не выделяется, но к помеченному таким образом фрагменту HTML-документа можно перейти с помощью гиперссылки следующего вида:

```
<A HREF="#имя_маркера"> текст </A>
```

Гиперссылки такого вида удобно использовать в документах большого объема.

СОВЕТ

Имя маркера должно задаваться латинскими буквами и может содержать цифры (кроме первого символа).

Формы

Формы предназначены для организации интерактивного режима работы пользователя, обеспечивая взаимодействие между пользователем, работающим на клиентской машине, и веб-приложениями, выполняющимися на стороне сервера.

Для создания формы используется пара тегов `<FORM>` и `</FORM>`. Между ними располагаются строки, описывающие различные элементы управления: кнопки, поля ввода, флажки и т. п.:

```
<FORM>
описание элементов управления
</FORM>
```

Совместно с тегом `<FORM>` практически всегда используются атрибуты `ACTION` и `METHOD`:

- ❑ атрибут `ACTION` предназначен для указания URL-адреса программы (сценария), которая будет выполнять обработку данных, введенных пользователем;
- ❑ атрибут `METHOD` определяет метод, с помощью которого данные, введенные пользователем, будут передаваться на сервер. Он может принимать одно из двух значений: `GET` или `POST`.

Основные элементы управления создаются с помощью тега `<INPUT>`, который используется без закрывающего тега. Тип управляющего элемента задается с помощью атрибута `TYPE` данного тега. Кроме атрибута `TYPE`, тег `<INPUT>` содержит ряд других атрибутов, определяющих параметры элемента управления.

Поля ввода

Для создания полей ввода атрибуту `TYPE` следует присвоить значение «`TEXT`» (кавычки обязательны). Параметры поля ввода задаются следующими атрибутами тега `<INPUT>`:

- ❑ `NAME` — идентификатор элемента управления;
- ❑ `VALUE` — начальное значение, отображаемое в поле ввода сразу после загрузки документа;
- ❑ `SIZE` — максимальное количество отображаемых символов;
- ❑ `MAXLENGTH` — максимальное количество символов, которые могут быть введены с помощью данного поля ввода.

Например, приведенный ниже фрагмент кода создает форму, содержащую текстовое поле `txt1` длиной 19 символов:

```
<FORM METHOD="POST"
ACTION="http://domen.ru/scripts/test.cgi">
Имя: <INPUT TYPE="TEXT" SIZE=19 NAME="txt1">
</FORM>
```

Имеется еще одна разновидность полей ввода — поля, предназначенные для ввода пароля. Для создания такого поля ввода атрибуту `TYPE` следует задать значение «`PASSWORD`». Все символы, вводимые в этом поле ввода, будут отображаться на экране в виде звездочек (*). В остальном этот элемент управления ничем не отличается от обычного поля ввода.

Флажки

Для создания флажка атрибуту TYPE тега <INPUT> задается значение «CHECKBOX». Параметры флажка определяются следующими атрибутами:

- ☐ NAME — идентификатор элемента управления;
- ☐ VALUE — атрибут, определяющий значение, которое передается на сервер в случае, если флажок установлен;
- ☐ CHECKED — атрибут, указывающий, что после загрузки документа флажок должен быть установлен. Данному атрибуту не задается никакого значения.

Например, следующий код создает флажок, передающий на сервер значение «YES» и по умолчанию являющийся установленным:

```
<FORM METHOD="POST"
ACTION="http://domen.ru/scripts/test.cgi">
<INPUT TYPE="CHECKBOX" NAME="chk1" VALUE="YES" CHECKED>
Запомнить
</FORM>
```

Переключатели

Переключатель представляет собой группу элементов управления, подобных флажкам. Однако в отличие от последних в установленном состоянии может находиться только один из элементов управления, входящих в группу. Для создания переключателей атрибуту TYPE задается значение «RADIO». Параметры переключателей определяются следующими атрибутами:

- ☐ NAME — идентификатор переключателя. Он должен быть одинаковым для всех элементов управления, входящих в одну группу;
- ☐ VALUE — значение, передаваемое серверу при установленном значении элемента управления;
- ☐ CHECKED — атрибут, указывающий, какой из элементов управления должен быть установлен в группе при загрузке документа.

Следующий фрагмент HTML-кода содержит описание переключателя с тремя положениями:

```
<FORM METHOD="POST"
ACTION="http://domen.ru/scripts/test.cgi">
<INPUT TYPE="RADIO" NAME="rb1" VALUE="1" CHECKED>
сегодня <BR>
<INPUT TYPE="RADIO" NAME="rb1" VALUE="2">
за последнюю неделю <BR>
<INPUT TYPE="RADIO" NAME="rb1" VALUE="3">
за последний месяц <BR>
</FORM>
```

Кнопки

Различают два вида кнопок:

- ☐ кнопка SUBMIT производит передачу введенных пользователем данных на сервер;
- ☐ кнопка RESET сбрасывает все элементы управления в исходные состояния.

Для создания кнопок атрибуту TYPE задается значение либо «SUBMIT», либо «RESET» — в зависимости от того, какую кнопку требуется создать.

Надпись на кнопке задается с помощью атрибута VALUE.

Приведенный ниже фрагмент кода создает пару кнопок, одна из которых имеет тип SUBMIT, а вторая — RESET:

```
<FORM METHOD="POST" ACTION="http://domen.ru/scripts/test.cgi">
<INPUT TYPE="SUBMIT" VALUE="OK" CHECKED>
<INPUT TYPE="RESET" VALUE="CANCEL">
</FORM>
```

В заключение приведем пример формы, содержащей все основные элементы управления:

```
<HTML>
<HEAD>
<TITLE> Пример HTML-документа </TITLE>
</HEAD>
<BODY>
<H2 ALIGN="CENTER"> Пример создания форм </H2>
<FORM METHOD="POST"
ACTION="http://domen.ru/scripts/test.cgi">
Имя: <INPUT TYPE="TEXT" SIZE=19 NAME="txt1"> <BR><BR>
<INPUT TYPE="CHECKBOX" NAME="chk1" VALUE="YES" CHECKED>
Запомнить <BR><BR>
<INPUT TYPE="RADIO" NAME="rb1" VALUE="1" CHECKED>
сегодня <BR>
<INPUT TYPE="RADIO" NAME="rb1" VALUE="2">
за последнюю неделю <BR>
<INPUT TYPE="RADIO" NAME="rb1" VALUE="3">
за последний месяц <BR><BR>
<INPUT TYPE="SUBMIT" VALUE="OK" CHECKED>
<INPUT TYPE="RESET" VALUE="CANCEL">
</FORM>
</BODY>
</HTML>
```

Вид приведенного HTML-документа в окне браузера Mozilla Firefox показан на рис. 16.2.

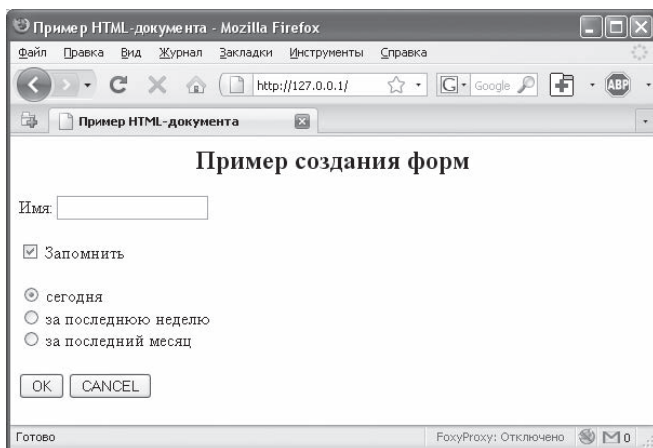


Рис. 16.2. Пример создания HTML-формы

ПРИМЕЧАНИЕ

Среда проектирования Turbo Delphi Explorer предоставляет возможности визуального проектирования HTML-страниц. Для этого необходимо выполнить команду File ► New ► Other, а затем в списке категорий выбрать Web Documents, после чего выбрать в списке справа значок HTML-page. Там же можно выбрать создание XML-схемы, JavaScript и т. д. Содержание доступных компонентов Tool Palette изменится автоматически.

Язык XML

Расширяемый язык разметки Extensible Markup Language (XML) является дальнейшим развитием языков разметки веб-документов. В настоящее время это XML — общепринятый открытый стандарт.

С помощью языка XML разработчики получили возможность создавать свои собственные языки разметки. Для этого достаточно в терминах XML описать предполагаемые нововведения, и если описание произведено корректно, браузер должен понять такой документ. Так, с помощью XML был расширен язык HTML. Новая версия языка — XHTML.

Язык XML также основан на применении тегов разметки. Для примера определим правила своей разметки путем описания типа документа Document Type Definition (DTD), который определяет то, каким образом следует пользоваться новой языковой конструкцией, а также указывает, какие элементы могут содержать вложенные элементы, значения атрибутов и т. д.

Простой шаблон DTD, определяющий классификационный язык для учащихся институтов, таков:

```
<!-- DTD Классы -->
<!ELEMENT grades (student+)>
<!ELEMENT student (course+)>
<!ATTLIST student name CDATA #REQUIRED
               sex (M|F) # REQUIRED
               level (1|2|3|4|5|6) #REQUIRED>
<!ELEMENT course EMPTY>
<!ATTLIST course title CDATA #REQUIRED
               grade (PASS|FAIL) #REQUIRED>
```

На такой файл **grades.dtd** возможна ссылка в XML-файлах, например,

```
<?xml version="1.0"?>
<!DOCTYPE GRADES SYSTEM "grades.dtd">
<!--образец -->
<grades>
<student name = "Виктор" sex= "M" level= "3">
    <course title= "Высшая математика" grade= "PASS"/>
    <course title= "Английский язык" grade= "PASS"/>
</student>
</grades>
```

В приведенном примере мы ввели конструкции для оперирования записями об оценках учащихся институтов по системе зачет-незачет.

Использование языка XML постоянно расширяется. Такая популярность объясняется тем, что XML позволяет представлять данные не в двоичном, а в текстовом

формате, что очень удобно для их передачи и обработки. Язык XML позволяет описывать сколь угодно сложно организованные данные и их структуру.

Очень полезна возможность преобразовывать наборы данных из различных внутренних форматов (например, баз данных) в стандартный формат XML (текстовый), и наоборот. Эта возможность позволяет стыковать работу приложений, создаваемых разными производителями, и расширять функциональные возможности хотя и устаревших, но находящихся в эксплуатации систем, не меняя их исходных текстов.

Типы веб-приложений

Задачи, решаемые веб-сервером, в основном сводятся к поддержке HTTP-протокола и передаче клиенту запрашиваемых ресурсов. Однако часто возникает необходимость выполнения каких-либо нестандартных действий. В этом случае используются специальные программы, выполняемые на сервере и взаимодействующие как с веб-сервером, так и с клиентом.

На сегодняшний день существует довольно большое количество различных типов приложений, использующихся в качестве расширений веб-серверов. Мы рассмотрим только основные из них, которые могут создаваться с помощью средств разработки приложений, функционирующих под управлением операционной системы Windows:

- ❑ CGI-сценарии;
- ❑ ISAPI/NSAPI-расширения.

CGI-сценарии

Аббревиатура CGI расшифровывается как Common Gateway Interface — интерфейс общего шлюза. *CGI-сценарии* можно отнести к классике интернет-приложений — это первый и общепринятый интерфейс для создания расширений веб-серверов. Данный факт в значительной степени определяет как достоинства, так и недостатки CGI-сценариев.

CGI-сценарий представляет собой обычное консольное приложение, обменивающееся данными с сервером через переменные окружения. Все недостатки CGI-сценариев обусловлены именно этим:

- ❑ CGI-сценарий выполняется в своем адресном пространстве (а не в адресном пространстве веб-сервера), поэтому обеспечивает довольно низкую скорость взаимодействия с сервером;
- ❑ производить обмен данными через переменные окружения в достаточной степени неудобно.

Указанные недостатки преодолены в других типах интернет-приложений, в частности таких как ISAPI и ASP. Но тем не менее сценарии CGI до сих пор имеют широчайшее распространение в WWW. Это объясняется их универсальностью — расширения CGI поддерживаются практически всеми существующими веб-серверами, работающими на любых платформах. Кроме того, CGI является основным видом серверных расширений для веб-серверов, работающих под

управлением различных разновидностей операционной системы UNIX (Linux, FreeBSD, Solaris и т. п.). А поскольку UNIX является наиболее распространенной операционной системой в Интернете, то и CGI имеет широкое распространение.

Расширения ISAPI

Спецификация ISAPI (Internet Server Application Programming), так же как и CGI, определяет правила взаимодействия между веб-сервером и другими приложениями. Главное отличие ISAPI-расширения от CGI-сценария заключается в том, что приложение ISAPI представляет собой динамически связываемую библиотеку (DLL), которая при вызове загружается не как отдельный процесс, а как поток, принадлежащий веб-серверу. Благодаря этому ISAPI-расширения обладают тремя существенными преимуществами перед CGI:

- ❑ поток требует существенно меньших ресурсов, чем отдельный процесс, что приводит к меньшей загрузке сервера;
- ❑ расширение ISAPI выполняется в адресном пространстве веб-сервера, поэтому работает быстрее, чем отдельный процесс;
- ❑ в отличие от CGI, приложение ISAPI может оставаться постоянно загруженным в память, а не загружаться каждый раз при поступлении нового запроса, как CGI. Благодаря этому снижается нагрузка на сервер и уменьшается время обработки запроса (так как не тратится время на загрузку приложения в память).

Однако, несмотря на все свои плюсы, ISAPI-расширения не лишены и недостатков, главным из которых является то, что ошибки, возникающие при выполнении ISAPI-расширения, могут повлечь за собой нарушение работы веб-сервера. Этот недостаток неразрывно связан с преимуществами ISAPI-программ, поскольку обусловлен тем, что расширения ISAPI выполняются в едином адресном пространстве с веб-сервером.

Доступ к базам данным с использованием Интернета

При публикации информации в Интернете широко используются базы данных, что значительно расширяет возможности веб-сервера и позволяет решить ряд проблем. Прежде всего это относится к проблемам ограничения доступа к информации и использованию широких возможностей СУБД для поиска необходимых данных.

С другой стороны, использование веб-браузера в качестве клиентской программы для базы данных также имеет свои преимущества. Главное из них — возможность работы с базой данных, размещенной на веб-сервере, абсолютно любой клиентской машины, независимо от того, какая операционная система используется на стороне клиента (достаточно, чтобы для этой операционной системы существовал веб-браузер). При этом не требуется разрабатывать специальные приложения для каждой платформы. Это связано с тем, что язык

HTML, являясь стандартным, одинаково интерпретируется браузерами, независимо от того, в какой операционной системе они работают — Windows, Linux или MacOS. Кроме того, при внесении каких-либо изменений в базу данных нет необходимости проводить обновление программного обеспечения пользователей этой базы данных, так как все, что необходимо, хранится на веб-сервере и доступно всем, кто имеет право доступа к этому серверу.

Благодаря всем этим достоинствам использование доступа к базам данных на основе веб-технологии нашло применение и в локальных сетях. Такие сети, использующие технологию WWW для доступа к данным, носят название *интрасетей* (интранет).

Модель взаимодействия с базой данных в рамках веб-технологий в случае реализации доступа к базе данных, размещенной на стороне сервера, можно изобразить в виде схемы, приведенной на рис. 16.3.

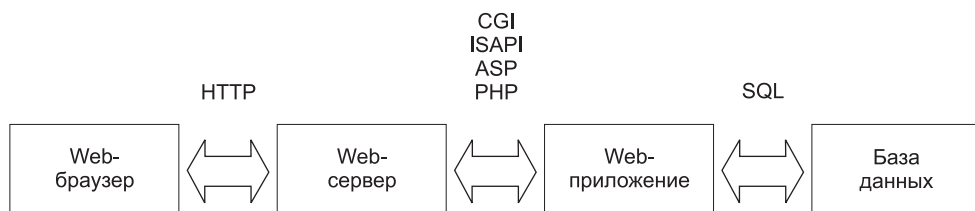
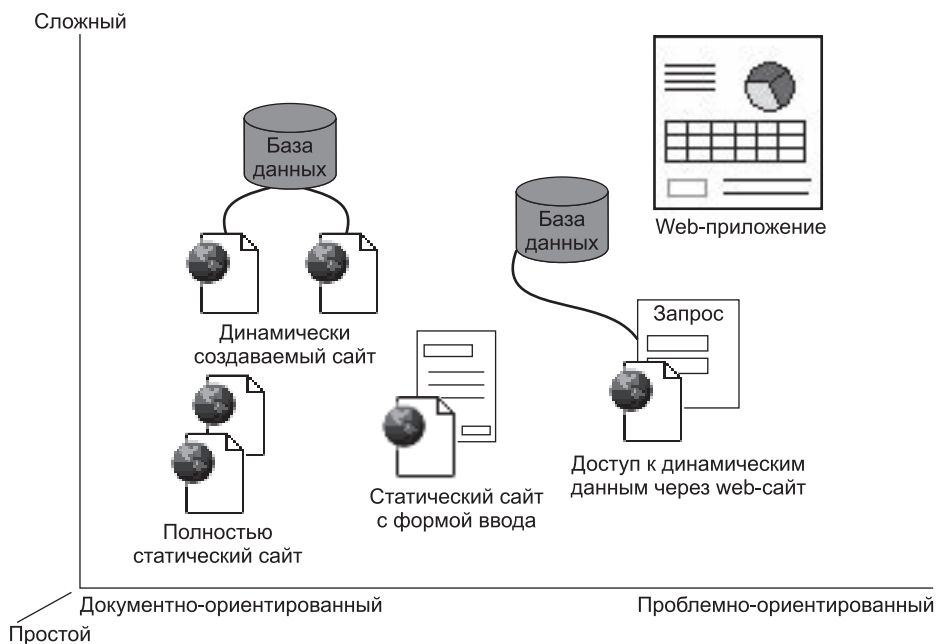


Рис. 16.3. Модель доступа к базе данных через веб

При обеспечении доступа к базам данных через WWW возможен ряд путей — комплексов технологических и организационных решений. Выбор конкретных решений доступа зависит от специфики конкретной СУБД и ряда других факторов, таких как наличие специалистов, способных с минимальными издержками освоить определенную ветвь технологических решений, или существование других БД, WWW-доступ к которым осуществляется с минимальными дополнительными затратами.

Для доступа к базам данных через веб наиболее часто используется один из двух подходов:

- ❑ однократное или периодическое преобразование содержимого базы данных в статические гипертекстовые документы. В этом случае база данных периодически просматривается специальной программой, создающей множество связанных HTML-документов, содержащих информацию из базы данных. Полученные HTML-файлы размещаются на одном или нескольких WWW-серверах. Доступ к ним осуществляется как к статическим гипертекстовым документам (полностью статический сайт, рис. 16.4). Этот вариант характеризуется минимальными начальными расходами. Он достаточно эффективен при работе с небольшими, редко обновляемыми базами данных, имеющими простую структуру, а также при пониженных требованиях к актуальности данных, предоставляемых через WWW;
- ❑ динамическое создание гипертекстовых документов на основе информации, содержащейся в базе данных, и информации, переданной клиентом веб-серверу. В этом варианте доступ к базе данных обеспечивается специальным

**Рис. 16.4.** Типы веб-сайтов

приложением (CGI, ISAPI, ASP (для .NET — ASP), PHP и т. п.), вызываемым WWW-сервером в ответ на запрос, полученный от клиента. Приложение обрабатывает запрос, производит необходимую выборку из базы данных и на ее основе формирует выходной HTML-документ, возвращаемый клиенту. Такое решение эффективно для больших баз данных со сложной структурой. Данный вариант позволяет также обеспечить возможность изменения информации, хранящейся в базе данных, с помощью веб-интерфейсов.

Глава 17

Разработка интернет-приложений

Современный подход к проектированию приложений для Интернет предполагает, что проектирование веб-приложений должно быть по возможности таким же простым, как и проектирование обычных приложений. Такой подход основан на использовании готовых компонентов, максимально освобождающих разработчика от написания кода вручную.

В библиотеке VCL Turbo Delphi Explorer присутствует большое количество удобных компонентов для создания приложений для Интернет.

Начиная с версии 7 в Delphi появилась группа компонентов, значительно упрощающих разработку приложений для веб и позволяющих разработчику не вникать в особенности интернет-протоколов.

В данной главе рассматривается разработка приложений для Интернет с использованием компонентов Turbo Delphi Explorer.

Разработка сценариев CGI

Для начала рассмотрим небольшой пример CGI-приложения, иллюстрирующий процесс передачи данных посредством методов GET и POST.

При создании CGI-приложения решаются две основные задачи: разработка веб-интерфейса и разработка собственно приложения. Для разработки веб-интерфейсов необходимо знать хотя бы основы языка HTML, по крайней мере набор основных тегов HTML, которые были рассмотрены в предыдущей главе.

Запуск CGI-приложения

Приложение CGI может запускаться двумя способами:

- ☐ щелчком на кнопке SUBMIT на форме (эта кнопка создается с помощью тега `<INPUT TYPE="SUBMIT">`);
- ☐ щелчком на ссылке.

В первом случае имя и местоположение CGI-сценария указываются в теге `<FORM>` с помощью атрибута `ACTION`, например:

```
<FORM ACTION="/scripts/test.cgi" METHOD="GET">
```

Во втором случае ссылка на приложение CGI указывается в теге `<A>` с помощью атрибута `HREF`:

```
<A HREF="/scripts/test.cgi"> Run CGI </A>
```

Наиболее часто используется первый способ, так как именно он позволяет организовать интерактивную работу и обеспечить возможность передачи на сервер данных, введенных пользователем.

Для тестирования веб-приложений необходим веб-сервер. Веб-сервер представляет собой обычное приложение, взаимодействующее с другими приложениями по протоколу HTTP. Такое приложение может быть установлено на любой локальный компьютер, даже не подключенный к сети. Воспользуемся входящими в стандартную поставку Windows XP Professional информационными службами Интернета (IIS).

Обычно после установки Windows XP эти службы не установлены. Чтобы их установить, воспользуйтесь диалоговым окном «Установка и удаление программ». После установки IIS на диске C: (по умолчанию) создается каталог `Inetpub`, предназначенный для хранения публикуемых в Веб документов и расширений сервера. Данный каталог содержит несколько подкаталогов, из которых наибольший для нас интерес представляют два:

- `Wwwroot` — корневой каталог для веб-страницы. При обращении к компьютеру по умолчанию будет производиться загрузка документа, хранящегося в этом каталоге и имеющего имя `default.htm` или `default.asp`;
- `Scripts` — каталог, предназначенный для хранения расширений сервера.

Простейшее CGI-приложение

Как уже отмечалось выше, приложения CGI представляют собой обычные консольные приложения. Поэтому для их разработки не требуется никаких специальных средств. Для вывода результатов выполнения сценария CGI используются обычные процедуры вывода на консоль. В языке Object Pascal это процедуры `write` и `writeln`. Однако выводимая таким образом информация должна соответствовать протоколу HTTP. Первая строка заголовка (`HTTP/1.0 200 OK`) формируется веб-сервером. Информационные же поля заголовка и тело ответа должны формироваться приложением CGI. В большинстве случаев достаточно одного поля — `Content-type`. Не следует также забывать, что заголовок и тело ответа должны разделяться пустой строкой.

В качестве примера создадим в Delphi простейшее CGI-приложение, результатом действия которого будет просто вывод строки текста (например, классической «Hello, world!»).

1. Выберите в главном меню Delphi IDE команду `File ► New ► Other`, затем выберите в открывшемся окне хранилища объектов в списке `Item Categories` категорию `Delphi Projects` и в данной категории выберите в списке справа зна-

чок Console Application и щелкните на кнопке OK. После этого будет создан шаблон консольного приложения, имеющий следующий вид:

```
program Project1;

{$APPTYPE CONSOLE}

uses SysUtils;

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

2. Введите следующий текст программы:

```
program console;

{$APPTYPE CONSOLE}

uses
  SysUtils;

begin
  // Выводим поле заголовка Content-type
  writeln('Content-Type: text/html');
  // Выводим пустую строку, отделяющую
  // заголовок от тела ответа
  writeln;
  // Построчно выводим HTML-документ
  writeln('<HTML>');
  writeln('<HEAD>');
  writeln('<TITLE>Пример CGI-приложения</TITLE>');
  writeln('</HEAD>');
  writeln('<BODY>');
  writeln('<H2 ALIGN=CENTER>Hello, World!</H2>');
  writeln('</BODY>');
  writeln('</HTML>');
end.
```

3. Откомпилируйте полученное приложение и запишите полученный исполняемый файл в каталог, предназначенный для размещения расширений веб-сервера (по умолчанию это каталог scripts).

ПРИМЕЧАНИЕ

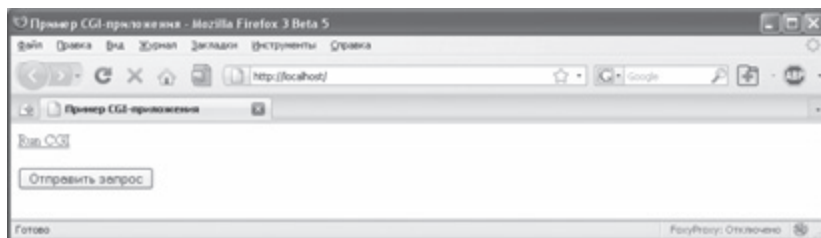
В результате компиляции будет создан исполняемый файл, по умолчанию имеющий расширение exe. Personal Web Server различает формат запускаемого сценария по расширению, поэтому исполняемый файл следует переименовать, присвоив ему расширение cgi.

4. Для тестирования полученного приложения необходимо создать HTML-документ, из которого будет производиться вызов CGI-приложения. Поскольку в нашем примере не требуется получать какие-либо данные от пользователя, то не важно, какой способ вызова использовать — форму или обычную ссылку. Создадим HTML-документ, в котором используются оба способа вызова сценария:

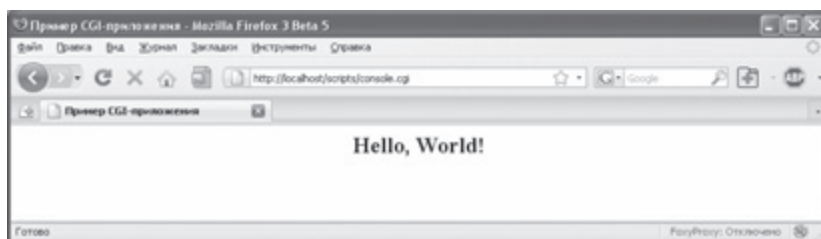
```
<HTML>
<HEAD>
```

```
<TITLE> Пример CGI-приложения </TITLE>
</HEAD>
<BODY>
<A HREF="/scripts/console.cgi"> Run CGI </A><BR><BR>
<FORM METHOD="GET" ACTION="/scripts/console.cgi">
<INPUT TYPE="SUBMIT">
</FORM>
</BODY>
</HTML>
```

5. Назовите созданный HTML-документ `default.htm` и поместите его в каталог `Wwwroot`. После этого откройте окно веб-браузера и наберите в строке адреса имя `localhost`, или IP-адрес `127.0.0.1`, или имя своего компьютера. В окне браузера отобразится документ, содержащий одну ссылку и одну кнопку (рис. 17.1, а). При щелчке на любом из этих элементов в окне браузера отобразится документ, соответствующий ответу запущенного сценария CGI. В нашем случае это просто строка текста «Hello, World!», выведенная по центру стилем заголовка второго уровня (рис. 17.1, б).



а



б

Рис. 17.1. Тестовый HTML-документ (а) и результат выполнения простого CGI-сценария (б)

Рассмотренный выше пример не имеет большого практического значения. Главной функцией, выполняемой веб-приложениями, является организация интерактивной работы пользователя. Поэтому результат выполнения сценария должен зависеть от данных, введенных пользователем.

Строка передаваемых параметров

Ввод данных пользователем производится средствами интерфейса, реализованного с помощью веб-формы. Щелчок на кнопке `SUBMIT`, расположенной на форме, вызывает CGI-сценарий, указанный в теге `<FORM>` с помощью атрибута

ACTION. Перед запуском сценария сервер формирует строку параметров. Содержимое этой строки будет определяться интерактивными элементами, расположенными на форме. Каждый из этих элементов имеет идентификатор, задаваемый атрибутом NAME, и значение, определяемое атрибутом VALUE или последовательностью символов, введенных пользователем. Из идентификаторов элементов управления и их значений формируется строка параметров следующего формата:

идентификатор1=значение1&идентификатор2=значение2...

Каждый параметр этой строки соответствует одному элементу управления и представляет собой имя управляющего элемента и его значение, разделенные знаком равенства. Различные (относящиеся к разным элементам управления) параметры разделяются в строке символами &.

Если символы = или & входят в состав имени или значения элемента управления, то они кодируются последовательностью из трех знаков: первый знак — %, за ним следуют две шестнадцатеричные цифры, являющиеся кодом символа (например, символ = кодируется как %3D, а символ & — как %26). Кроме этих двух знаков, трехсимвольными последовательностями обычно кодируются все знаки, за исключением латинских букв, цифр и символа пробела. Символ пробела заменяется символом +.

Полученную строку параметров прежде всего следует декодировать. Этот процесс можно представить в виде последовательности следующих действий:

- ☐ разделить строку на пары «идентификатор_N=значение_N»;
- ☐ выделить в каждой паре идентификатор и значение;
- ☐ заменить в каждом идентификаторе и каждом значении символы + пробелами;
- ☐ преобразовать каждую трехсимвольную последовательность, начинающуюся со знака %, в символ ASCII.

Таким образом, алгоритм декодирования довольно прост и сводится к нескольким операциям работы со строками.

Методы передачи и получения строки параметров

Строка параметров может передаваться веб-серверу двумя способами: либо с использованием метода GET, либо с помощью метода POST. Метод передачи данных определяется значением атрибута METHOD в теге <FORM>:

- ☐ при использовании метода GET строка параметров передается вместе с URL-адресом вызываемого CGI-приложения. Для разделения URL и строки параметров используется символ ?;
- ☐ в случае использования метода POST строка параметров передается в теле HTTP-запроса.

При разработке CGI-сценария важно знать не только способ передачи строки параметров, но и технологию получения ее в CGI-приложении. В зависимости от метода передачи строки параметров различаются и методы ее получения:

- ❑ при использовании метода GET строка параметров передается CGI-приложению через переменную окружения QUERY_STRING;
- ❑ при использовании метода POST данные передаются приложению CGI через стандартный поток ввода консольной программы. Длина строки в этом случае может быть определена через переменную окружения CONTENT_LENGTH.

Считывание строки параметров при использовании метода GET

При использовании метода GET для получения строки параметров в CGI-приложении следует использовать функцию, возвращающую значение переменной окружения с заданным именем. Для этого можно использовать следующую функцию Win32 API:

```
function GetEnvironmentVariable(lpName: PChar; lpBuffer: PChar; nSize: DWORD):  
DWORD; stdcall
```

Здесь:

- ❑ lpName — имя переменной окружения;
- ❑ lpBuffer — строка PChar, в которую будет занесено значение указанной переменной окружения;
- ❑ nSize — длина строки lpBuffer.

Значение, возвращаемое функцией GetEnvironmentVariable, равно нулю в том случае, если переменная окружения (ее имя указано параметром lpName) не определена.

Один из возможных вариантов фрагмента программы, выполняющего считывание данных из переменной окружения QUERY_STRING, имеет следующий вид:

```
var  
  buff: PChar;  
  St1: String;  
...  
begin  
...  
  // Выделяем память под строку параметров  
  GetMem(buff,200);  
  // Получаем строку параметров  
  GetEnvironmentVariable('QUERY_STRING',buff,200);  
  // Преобразуем строку PChar в паскалевскую строку  
  St1:=StrPas(buff);  
  // Освобождаем память  
  FreeMem(buff);  
...  
end;
```

Считывание строки параметров при использовании метода POST

При использовании метода POST строка параметров считывается из стандартного потока ввода. При этом следует выполнять считывание именно такого количества символов, какое содержится в передаваемой строке, — попытка считать больше символов, чем есть, приведет к «зависанию» программы. Если же считать не все символы, то часть информации будет потеряна.

Для реализации процедуры считывания данных из стандартного потока ввода проще всего использовать стандартную процедуру `Read` языка Pascal. Количество символов, которые требуется считать, передается через переменную окружения `CONTENT_LENGTH`. Таким образом, при использовании метода `POST` фрагмент программы, выполняющий считывание строки параметров, имеет более сложный вид, так как требует и реализации двух процедур: обращения к переменной окружения и считывания данных из стандартного потока ввода:

```
var
    buff: PChar;
    ContentLength,i: Integer;
    St1: String;
    C: Char;
...
begin
...
// Выделяем память под строку PChar для считывания
// данных из переменной окружения CONTENT_LENGTH
    GetMem(buff,50);
// Считываем данные из переменной
// окружения CONTENT_LENGTH
    GetEnvironmentVariable('CONTENT_LENGTH',buff,50);
// Преобразуем строку в число
    ContentLength:=StrToInt(StrPas(buff));
// Освобождаем выделенную память
    FreeMem(buff);
// Считываем ContentLength из стандартного потока ввода
    St1:='':
    for i:=1 to ContentLength do begin
        Read(C);
        St1:=St1+C;
    end;
...
end;
```

Получение дополнительной информации

При разработке CGI-сценариев можно использовать и ряд других переменных окружения, через которые веб-сервер передает дополнительную информацию о пользователе, вызвавшем сценарий на выполнение. Ряд переменных окружения, содержащих наиболее важную информацию, приведен в табл. 17.1.

Таблица 17.1. Основные переменные окружения

Название	Описание
<code>REQUEST_METHOD</code>	Метод передачи информации от пользователя: <code>GET</code> или <code>POST</code>
<code>SERVER_NAME</code>	IP-адрес или имя сервера
<code>SERVER_PORT</code>	Номер порта, используемый при обращении к серверу
<code>SERVER_PROTOCOL</code>	Название и версия протокола, по которому был передан запрос
<code>PATH_INFO</code>	Строка параметров, расположенная в запросе после имени сценария, но до данных запроса
<code>REMOTE_ADDR</code>	IP-адрес узла, с которого был послан запрос
<code>REMOTE_HOST</code>	Доменное имя узла, с которого поступил запрос

Перед получением строки параметров обычно следует проверить, какой метод передачи информации использован. Это обеспечит правильность считывания передаваемой информации в любом случае.

Проиллюстрируем это на примере. Объединим фрагменты программ, приведенные в разделах «Считывание строки параметров при использовании метода GET» и «Считывание строки параметров при использовании метода POST» таким образом, чтобы обеспечить корректное получение данных при использовании любого метода. В качестве результата работы сценария выведем имя используемого метода и полученную строку параметров:

```
program console;
{$APPTYPE CONSOLE}
{$E CGI}

uses
  SysUtils, Windows;

var
  buff: PChar;
  ContentLength, i: Integer;
  St1, St2: String;
  C: Char;

begin
  GetMem(buff, 50);
  GetEnvironmentVariable('REQUEST_METHOD', buff, 50);
  St1:=StrPas(buff);
  FreeMem(buff);
  i:=Length(St1);
  while i>0 do
    begin
      St1[i]:=UpCase(St1[i]);
      dec(i)
    end;
  if St1='GET'
  then begin
    GetMem(buff, 200);
    GetEnvironmentVariable('QUERY_STRING', buff, 200);
    St2:=StrPas(buff);
    FreeMem(buff);
  end;
  if St1='POST'
  then begin
    GetMem(buff, 50);
    GetEnvironmentVariable('CONTENT_LENGTH', buff, 50);
    St2:=StrPas(buff);
    FreeMem(buff);
    ContentLength:=StrToInt(St2);
    St2:='';
    for i:=1 to ContentLength do begin
      Read(C);
      St2:=St2+C;
    end;
  end;
end;
```

продолжение ➤

```

St1:='Method '+St1;
St1:='<H2 ALIGN=CENTER>'+St1+'</H2>';
St2:='Query string: '+St2;
St2:='<H2 ALIGN=CENTER>'+St2+'</H2>';
writeln('Content-Type: text/html');
writeln;
writeln('<HTML>');
writeln('<HEAD>');
writeln('<TITLE>Пример CGI-приложения</TITLE>');
writeln('</HEAD>');
writeln('<BODY>');
writeln('<H2 ALIGN=CENTER>Hello, World!</H2>');
writeln(St1);
writeln(St2);
writeln('</BODY>');
writeln('</HTML>');
end.

```

Для проверки работоспособности приведенной программы создадим HTML-документ, содержащий две одинаковые формы ввода, различающиеся лишь способом передачи информации. Чтобы не усложнять пример, разместим на форме только два элемента управления — кнопку **SUBMIT** и поле ввода. Текст HTML-документа в этом случае будет выглядеть примерно так:

```

<HTML>
<HEAD>
  <TITLE> Пример CGI-приложения </TITLE>
</HEAD>
<BODY>
  <H2> Метод GET </H2>
  <FORM METHOD="GET" ACTION="/scripts/console.cgi">
    <INPUT TYPE="TEXT" NAME="Edit1" VALUE="Test"><BR><BR>
    <INPUT TYPE="SUBMIT">
  </FORM>
  <BR><BR>
  <H2> Метод POST </H2>
  <FORM METHOD="POST" ACTION="/scripts/console.cgi">
    <INPUT TYPE="TEXT" NAME="Edit1" VALUE="Test"><BR><BR>
    <INPUT TYPE="SUBMIT">
  </FORM>
</BODY>
</HTML>

```

Напомним, что данный документ должен называться **default.htm** и располагаться в каталоге **Wwwroot**, а откомпилированный файл CGI-сценария — в каталоге **Scripts**.

Если теперь запустить веб-браузер и набрать в строке адреса имя вашего компьютера (**localhost** или **127.0.0.1**), то в окне браузера отобразится документ, показанный на рис. 17.2, *а*. При щелчке на кнопке **SUBMIT**, относящейся к форме, использующей метод **GET**, в окне браузера отобразится документ, приведенный на рис. 17.2, *б*, а при щелчке на кнопке, относящейся к форме, использующей метод **POST**, — документ, приведенный на рис. 17.2, *в*.

Обратите внимание, что во втором случае (при использовании метода **POST**) в поле ввода был введен русский текст (см. рис. 17.2, *в*).

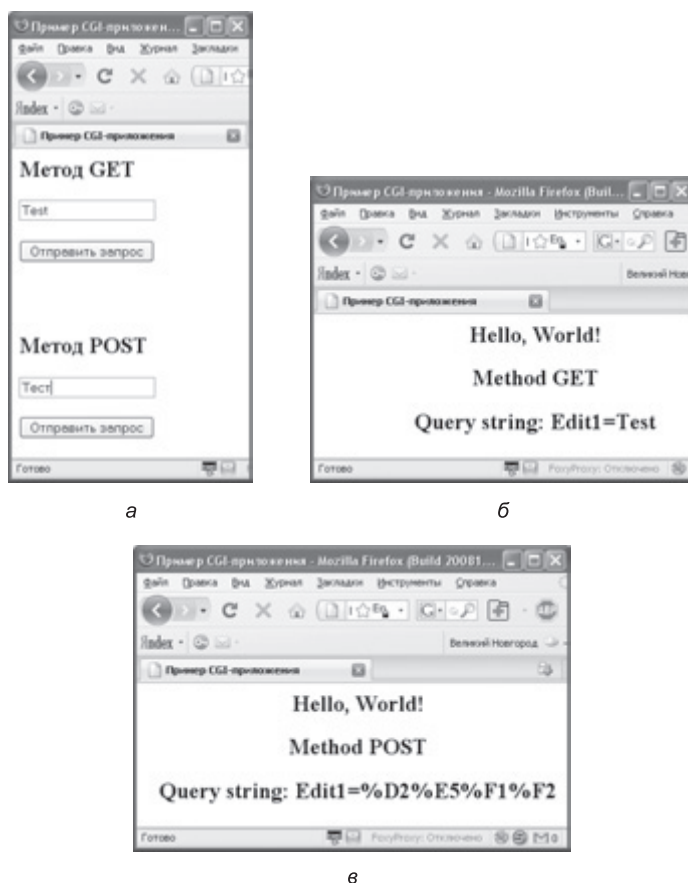


Рис. 17.2. Пример получения данных, введенных пользователем в CGI-сценарии: а — исходный HTML-документ, из которого вызывается сценарий; б — результат выполнения сценария при использовании метода GET; в — результат выполнения сценария при использовании метода POST

Рассмотренный пример позволяет лишь получить строку параметров в CGI-сценарии. Как правило, этого недостаточно. Полученную строку необходимо декодировать и затем обработать данные, полученные от пользователя. Такая задача достаточно проста и может быть решена без использования каких-либо специальных средств.

Использование специальных средств Delphi для разработки веб-приложений

Итак, в предыдущих разделах этой главы нами были рассмотрены основные вопросы разработки CGI-приложений, включая:

- передачу информации от клиента CGI-сценарию;

- ❑ особенности получения этой информации в приложении CGI в зависимости от используемого метода передачи;
- ❑ формирование сценарием ответа, посылаемого клиенту.

При рассмотрении этих вопросов был приведен ряд простых примеров создания CGI-приложений на языке Object Pascal. В принципе на базе этой информации можно выполнить разработку CGI-сценария для любой операционной системы на любом языке программирования. Это связано с тем, что способы передачи и получения данных пользователя везде абсолютно одинаковы.

Однако использование специальных средств позволяет в значительной степени упростить разработку веб-приложений, особенно в тех случаях, когда осуществляется разработка приложений для работы с базами данных.

Delphi Web Module

Компонент TWebModule является основой любых веб-приложений, разрабатываемых в Delphi, будь то CGI или ISAPI. С помощью этого компонента приложение выполняет интерпретацию HTTP-запросов.

Основное свойство компонента TWebModule — свойство Action, которое содержит список действий, являющихся обработчиками запросов, поступающих от клиента. Каждый элемент этого списка имеет тип TWebActionItem и характеризуется следующими свойствами:

- ❑ PathInfo: String — указывает, при какой строке параметров (расположенной в запросе после имени сценария, но до данных запроса) будет вызываться данное действие;
- ❑ MethodType: TMethodType — указывает метод, используемый клиентом для передачи запроса, на который данное действие может ответить. Может принимать следующие значения: mtGet, mtPost, mtHead, mtPut, mtAny. В зависимости от значения свойства MethodType действие будет обрабатывать запросы, переданные методами GET, POST, HEAD, PUT, или отвечать на запрос любого вида;
- ❑ Default: Boolean — используется для задания обработчика по умолчанию. Если данное свойство установлено равным true, то действие будет обрабатывать запросы со строками параметров, для которых не заданы обработчики;
- ❑ Enabled: Boolean — указывает, может (true) или нет (false) данное действие обработать HTTP-запрос с параметрами PathInfo и MethodType, соответствующими свойствам данного действия;
- ❑ Producer: TCustomContentProducer — указатель на специальный компонент, используемый для формирования ответа веб-приложения. Такие компоненты будут рассмотрены подробнее несколько ниже.

Каждый элемент списка Actions может обрабатывать всего одно событие — OnActions. Обработчик этого события и выполняет формирование ответа серверу на принятый запрос клиента.

```
property OnAction: THTTPMethodEvent;  
type THTTPMethodEvent = procedure (Sender: TObject; Request: TWebRequest;  
Response: TWebResponse; var Handled: Boolean) of object;
```

С помощью параметра `Request` передается запрос, полученный от клиента. Параметр `Response` используется для формирования ответа. Параметр `Handle` применяется в том случае, когда требуется указать, что запрос не обработан. Для этого данному параметру следует присвоить значение `false`.

Ввиду большой значимости параметров `Request` и `Response` рассмотрим их более подробно.

Параметр Request

Параметр `Request` является экземпляром класса `TWebRequest` — базового класса для передачи информации веб-приложениям. Это довольно сложный класс, обладающий большим количеством свойств и методов. Мы рассмотрим лишь несколько его основных свойств:

- ❑ property `Content`: `String` — строка параметров, переданная с помощью метода `POST`. Фактически это строка, содержащая тело HTTP-запроса, полученного от клиента;
- ❑ property `ContentFields`: `TStrings` — «разобранная» строка параметров, переданная с помощью метода `POST`. Каждый элемент коллекции `ContentFields` представляет собой строку, соответствующую одному элементу управления, расположенному на форме, и представляет собой имя управляющего элемента и его значение, разделенные знаком равенства (идентификатор=значение);
- ❑ property `Query`: `String` — строка параметров, переданная клиентом с помощью метода `GET`;
- ❑ property `QueryFields`: `TStrings` — «разобранная» строка параметров, переданная методом `GET`. Формат строк коллекции `QueryFields` полностью аналогичен формату коллекции `ContentFields`;
- ❑ property `RemoteAddr`: `String` — IP-адрес клиента, пославшего запрос;
- ❑ property `RemoteHost`: `String` — доменное имя клиента, пославшего запрос;
- ❑ property `Method`: `String` — метод, используемый для передачи данных серверу.

Таким образом, используя объект `Request`, можно легко получить все данные, введенные пользователем на форме, а также определить ряд параметров клиента. Причем для этого нет необходимости обращаться к переменным окружения и «вручную» декодировать и интерпретировать строку параметров, полученную от клиента.

Параметр Response

Параметр `Response` представляет экземпляр класса `TWebResponse` — базового класса, предназначенного для формирования ответа на HTTP-запрос. Приведем его основные свойства:

- ❑ property `ContentType`: `String` — тип данных, содержащихся в теле ответа;
- ❑ property `ContentLength`: `Integer` — число символов, содержащихся в теле ответа;
- ❑ property `Content`: `String` — содержимое тела ответа;

- ❑ `property ContentStream: TStream` — определяет объект `Stream`, который будет передан клиенту. Данное свойство обычно используется для передачи клиенту бинарных файлов. Если свойство `ContentStream` установлено, оно заменяет свойство `Content`.

События `WebModule`

Для выполнения обработки запросов и изменения содержимого ответа можно использовать действия, задаваемые в свойстве `Actions`, а также события самого компонента `TWebModule`. В этом компоненте предусмотрена возможность обработки пяти событий: `AfterDispatch`, `BeforeDispatch`, `OnCreate`, `OnDestroy` и `OnException`.

Рассмотрим каждое из этих событий более подробно:

- ❑ `property AfterDispatch: THTTPMethodEvent` — вызывается после того, как HTTP-ответ был успешно сформирован (в обработчике `OnActions` какого-либо действия из списка `Actions`), но еще не передан клиенту. Обработчик этого события можно использовать, например, для проверки сформированного HTTP-ответа;
- ❑ `property BeforeDispatch: THTTPMethodEvent` — вызывается перед тем, как диспетчер устанавливает соответствие запроса HTTP с каким-либо действием. Может использоваться для предварительной обработки HTTP-запроса;
- ❑ `property OnCreate: TNotifyEvent` — вызывается при создании экземпляра `TWebModule`. Данное событие обычно применяется для инициализации переменных и объектов, содержащихся в приложении. Например, если модуль содержит базу данных, в обработчике этого события можно выполнить подключение базы данных;
- ❑ `property OnDestroy: TNotifyEvent` — вызывается перед уничтожением `WebModule`. Обычно используется для освобождения объектов, созданных динамически. При использовании баз данных в обработчике этого события можно, например, разрывать соединение с базой данных.
- ❑ `property OnException: TWebExceptionEvent` — вызывается, когда происходит не-обработанное приложением исключение.

Пример создания CGI-приложения с использованием `WebModule`

Рассмотрим пример создания простого CGI-приложения с использованием средств, предоставляемых компонентом `WebModule`. Для этого разработаем программу, которая выполняет процедуру идентификации пользователя, то есть анализирует введенные им имя и пароль.

Разработку веб-приложения, создаваемого на основе компонента `WebModule`, можно разделить на три этапа.

1. Создание нового приложения. Именно на этом этапе определяется тип создаваемого приложения — CGI, WinCGI или ISAPI.
2. Задание действий, выполняющих обработку запросов клиента. Обработка запросов не зависит от типа веб-приложения.
3. Компиляция и тестирование созданного приложения.

Этап 1. Создание нового веб-приложения

Для создания нового CGI-приложения, использующего WebModule, необходимо выполнить следующие действия.

1. Выберите в главном меню Delphi IDE команду **Other**, затем выберите в открывшемся окне хранилища объектов в списке **Item Categories** категорию **Delphi Projects**, а в ней подкатегорию **WebBroker** и в данной категории выберите в списке справа значок **Web Server Application** и щелкните на кнопке **OK**.
2. В открывшемся окне диалога **New Web Server Application** (рис. 17.3) с помощью переключателя выберите необходимый тип приложения. Так как мы создаем CGI-приложение, выберите вариант **CGI Stand-alone executable**. Щелкните на кнопке **OK**. В результате будет создано новое CGI-приложение, содержащее компонент **TWebModule**.

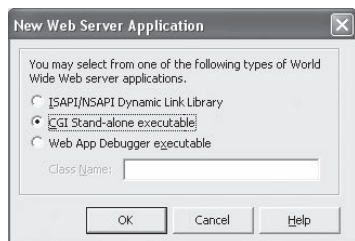


Рис. 17.3. Окно диалога New Web Server Application

Компонент **TWebModule** объединяет в себе возможности модуля данных (**TDataModule**) и диспетчера HTTP-запросов. На форму **WebModule** можно помещать компоненты доступа к данным и специальные компоненты для формирования ответа сервера на HTTP-запрос клиента.

Этап 2. Задание действий, выполняющих обработку запросов клиента

При использовании компонента **TWebModule** обязательно надо задать хотя бы одно действие (**Action**), которое будет выполнять обработку запроса клиента. Для этого выполните следующее.

1. Выберите в инспекторе объектов компонент **TWebModule** и щелкните на кнопке с многоточием в поле ввода свойства **Actions** этого компонента.
2. Создайте новое действие. Для этого следует воспользоваться либо кнопками на панели инструментов редактора действий (кнопка **Add New**), либо командой **Add** контекстного меню редактора действий, которое открывается при щелчке правой кнопкой мыши в окне редактора действий (можно также просто нажать на клавишу **Insert**).
3. Задайте необходимые свойства действия. Для этого следует воспользоваться инспектором объектов, в котором отображаются свойства действия, выделенного в окне редактора действий. Установите значение свойства **PathInfo** равным **/validate**. Это приведет к тому, что данное действие будет обрабатывать HTTP-запрос только в том случае, если при вызове сценария после его имени будет указана строка **/validate**, например: **/scripts/test.cgi/validate**. Значения всех остальных параметров оставьте без изменений.

4. Задайте обработчик запроса, то есть обработчик события `OnAction` созданного действия. Для этого выберите в инспекторе объектов вкладку `Events` и дважды щелкните в поле ввода этого события. В результате будет создан шаблон процедуры-обработчика события `OnAction`.

Далее нам требуется проанализировать информацию, введенную пользователем: соответствует она некоторым заданным данным или нет. Пользователь должен ввести два значения: имя и пароль. Пусть для их ввода используются текстовые поля с идентификаторами `login` и `password` соответственно. Необходимо получить из строки параметров, посланной клиентом, текст, введенный пользователем в каждое из этих текстовых полей, и сравнить его с заданным текстом (чтобы не усложнять задачу, будем считать, что имя пользователя и соответствующий ему пароль жестко задаются в самом веб-приложении). В зависимости от результата сравнения сформируется ответ.

В качестве ответа при точном соответствии имени пользователя и пароля заданным значениям выведем строку `Login/Password correct`, при ошибочном вводе — строку `Login/Password incorrect`. Однако перед получением параметров следует вначале определить, какой метод использовался для передачи строки параметров — `GET` или `POST`. Это необходимо, поскольку при использовании разных методов строка параметров передается по-разному: если использован метод `GET` — через свойства `Query` и `QueryFields` параметра `Request`; если же используется метод `POST` — через свойства `Content` и `ContentFields` этого же параметра. Код обработчика запроса может иметь примерно такой вид:

```
procedure TwebModule1.WebModule1WebActionItem1Action(
  Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
// Задаем корректные имя и пароль
const
  login: string = 'test';
  password: string = '123';
// Объявляем переменную, указывающую, корректны ли данные,
// введенные пользователем
var
  validate: Boolean;
begin
  validate:=false;
  // Проверяем, использован ли метод GET
  if Request.Method='GET' then
  // Проверяем, корректны или нет данные,
  // введенные пользователем
    validate:=(Request.QueryFields.Values['login']=login)
    and (Request.QueryFields.Values['password']=password);
  // Проверяем, использован ли метод POST
  if Request.Method='POST' then
  // Проверяем, корректны или нет данные,
  // введенные пользователем
    validate:=(Request.ContentFields.Values['login']=login)
    and (Request.ContentFields.Values['password']=password);
  if validate then
    Response.Content:='<H2 ALIGN=CENTER>Login/Password correct</H2>'
  else
    Response.Content:='<H2 ALIGN=CENTER>Login/Password incorrect</H2>'
end;
```

Этап 3. Компиляция и тестирование созданного приложения

Для компиляции проекта выберите в меню Delphi IDE команду Project ► Compile (или нажмите клавиши Ctrl+F9). Если в тексте приложения не содержится ошибок, то проект будет откомпилирован и в результате компиляции будет создан исполняемый файл, имеющий расширение exe. Измените расширение на cgi и перепишите его в каталог Scripts.

Для проверки разработанного сценария необходимо подготовить документ HTML, из которого будет производиться вызов сценария. Документ должен содержать форму, в которой определены два элемента управления для ввода имени и пароля. Так как созданное приложение может обрабатывать данные, переданные как методом GET, так и методом POST, то имеет смысл включить в документ две формы, одна из которых будет использовать метод GET, а вторая — POST. Исходный код HTML такого документа выглядит примерно так:

```
<HTML>
<HEAD>
  <TITLE> Пример CGI-приложения </TITLE>
</HEAD>
<BODY>
  <H2> Метод GET </H2>
  <FORM METHOD="GET" ACTION="/scripts/test.cgi/validate">
    Введите имя:&nbsp;<INPUT TYPE="TEXT" NAME="login"><BR>
    Введите пароль:&nbsp;<INPUT TYPE="PASSWORD" NAME="password"><BR><BR>
    <INPUT TYPE="SUBMIT">
  </FORM>
  <BR><BR>
  <H2> Метод POST </H2>
  <FORM METHOD="POST" ACTION="/scripts/test.cgi/validate">
    Введите имя:&nbsp;<INPUT TYPE="TEXT" NAME="login"><BR>
    Введите пароль:&nbsp;<INPUT TYPE="PASSWORD" NAME="password"><BR><BR>
    <INPUT TYPE="SUBMIT">
  </FORM>
</BODY>
</HTML>
```

Файлу, содержащему приведенный код, присвойте имя **default.htm** и поместите его в каталог **wwwroot**. После этого можно запустить веб-браузер и проверить работоспособность CGI-приложения.

Как видно из рассмотренного примера, компонент TWebModule значительно упрощает интерпретацию данных, полученных от пользователя. Однако формировать ответ приходится вручную, задавая строку Response.Content в формате HTML. Если требуется выдавать сложный ответ (хотя бы для того, чтобы результат выполнения сценария эстетично выглядел в окне веб-браузера), то такой подход не очень удобен. Поэтому в Delphi имеются специальные компоненты, упрощающие формирование сложных документов HTML.

Компоненты для формирования ответа в формате HTML

Компоненты для формирования документов HTML располагаются на странице Интернет палитры компонентов Delphi IDE. Условно их можно разделить на две группы:

- ❑ компоненты для генерирования HTML-документа на основе данных, хранящихся в базе данных;
- ❑ компоненты для генерирования HTML-документов без использования баз данных.

Первую группу мы рассмотрим несколько ниже. А пока познакомимся с единственным компонентом, относящимся ко второй группе.

Компонент TPageProducer

Это простейший компонент для генерации HTML-документа на основе некоторого заданного шаблона. Шаблон задается с помощью одного из следующих свойств:

- ❑ property HTMLDoc: TStrings — содержит код шаблона HTML-документа;
- ❑ property HTMLFile: TFileName — задает имя файла, в котором содержится шаблон HTML-документа.

Указанные свойства являются взаимоисключающими, поэтому можно задавать только одно из них.

Компонент TPageProducer выполняет обработку документа при вызове метода Content:

```
function Content: string; override;
```

Этот метод выполняет обработку шаблона и возвращает результат обработки в виде HTML-документа.

Шаблон представляет собой обычный HTML-документ, в который, кроме обычных тегов HTML, могут включаться специальные теги, имеющие следующий формат:

```
<#name param1=value1 param2=value2 ...>
```

Когда при обработке шаблона компонент TPageProducer встречается с такими тегами, вызывается событие OnHTMLTag (единственное событие TPageProducer). В обработчике этого события можно определить, какой тег обрабатывается и чем его следует заменить.

Обработчик события OnHTMLTag имеет следующий формат:

```
type
  TTag = (tgCustom, tgLink, tgImage, tgTable, tgImageMap,
          tgObject, tgEmbed);
  THTMLTagEvent = procedure (Sender: TObject; Tag: TTag;
                             const TagString: string; TagParams: TStrings;
                             var ReplaceText: string) of object;
```

Здесь параметр Tag указывает тип тега и может принимать одно из семи значений, в зависимости от имени тега:

- ❑ tgLink — тег LINK, гипертекстовая ссылка;
- ❑ tgImage — тег IMAGE, изображение;
- ❑ tgTable — тег TABLE, таблица;
- ❑ tgImageMap — тег IMAGEMAP, изображение с контекстно-чувствительными областями;

- ❑ `tgObject` — тег OBJECT, объект ActiveX;
- ❑ `tgEmbed` — тег EMBED, встраиваемая библиотека;
- ❑ `tgCustom` — тег, определяемый пользователем (имя этого тега не совпадает ни с одним из predefined).

Имя тега содержится в параметре `TagString`. Параметры тега передаются через параметр `TagParams` — здесь каждый параметр тега представлен строкой вида `name=value`. Через параметр `ReplaceText` возвращается строка, на которую должен быть заменен обрабатываемый тег.

Пример использования компонента TPageProducer

Модифицируем пример, рассмотренный в разделе «Пример создания CGI-приложения с использованием WebModule», используя компонент `TPageProducer`. В ответе дополнительно выведем информацию об IP-адресе клиента, пославшего запрос, а также отобразим имя, которое ввел пользователь. Для этого потребуется внести в проект следующие изменения.

1. Поместите на форму компонента `WebModule` компонент `TPageProducer`.
2. В свойстве `HTMLDoc` компонента `TPageProducer` задайте следующий шаблон:

```
<HTML>
<HEAD>
<TITLE> Результат проверки </TITLE>
</HEAD>
<BODY>
<H3 ALIGN=LEFT> <#RESULT> </H3>
<P>Адрес: <#HOST></P>
<P>Имя: <#NAME></P>
</BODY>
</HTML>
```

3. В разделе `private` описания класса `TWebModule1` задайте две строковые переменные: `ResultStr` и `LoginStr`.
4. Измените обработчик события `OnAction` и задайте обработчик события `OnHTMLTag` так, как показано в приведенном ниже коде:

```
unit test_unit;

interface

uses
  Windows, Messages, SysUtils, Classes, HTTPApp;

type
  TWebModule1 = class(TWebModule)
    PageProducer1: TPageProducer;
    procedure WebModule1WebActionItem1Action(
      Sender: TObject; Request: TWebRequest;
      Response: TWebResponse; var Handled: Boolean);
    procedure PageProducer1HTMLTag(Sender: TObject;
      Tag: TTag; const TagString:
      String; TagParams: TStrings;
      var ReplaceText: String);
  private
```

```

    { Private declarations }
    ResultStr: string;
    LoginStr: string;
    public
    { Public declarations }
    end;

var
    WebModule1: TWebModule1;

implementation

{$R *.DFM}

procedure TWebModule1.WebModule1WebActionItem1Action(
    Sender: TObject; Request: TWebRequest;
    Response: TWebResponse; var Handled: Boolean);
const
    login: String = 'test';
    password: String = '123';
var
    validate: Boolean;
begin
    validate:=false;
    if Request.Method='GET'
    then begin
        LoginStr:=Request.QueryFields.Values['login'];
        validate:=(LoginStr=login) and
            (Request.QueryFields.Values['password']=password);
    end;
    if Request.Method='POST'
    then begin
        LoginStr:=Request.ContentFields.Values['login'];
        validate:=(LoginStr=login) and
            (Request.ContentFields.Values['password']=password);
    end;
    if validate
    then ResultStr:='Имя/пароль введены правильно'
    else ResultStr:='Имя/пароль введены неверно';
    Response.Content:=PageProducer1.Content;
end;

procedure TWebModule1.PageProducer1HTMLTag(Sender: TObject; Tag: TTag;
    const TagString: String; TagParams: TStrings; var ReplaceText: String);
begin
    if TagString='RESULT'
    then ReplaceText:=ResultStr;
    if TagString='HOST'
    then ReplaceText:=(Sender as
        TCustomContentProducer).Dispatcher.Request.RemoteHost;
    if TagString='NAME'
    then ReplaceText:=LoginStr;
end;

end.

```

Чтобы протестировать полученное приложение, нет необходимости вносить изменения в код HTML-документа, из которого производится вызов сценария.

ПРИМЕЧАНИЕ

Приведенный пример достаточно прост, поэтому с его помощью трудно оценить преимущества использования компонента TPageProducer. Тем не менее он дает представление об особенностях использования этого компонента и интерпретации управляющих тегов.

Компоненты для работы с базами данных

Информацию, публикуемую в WWW, очень часто бывает удобно хранить в базах данных. Кроме того, как уже отмечалось выше, иногда при работе с базами данных в качестве клиентского приложения используется веб-браузер. В обоих случаях требуется организовать вывод хранящейся в базе данных информации в HTML-документ. Для упрощения решения такой задачи в VCL Delphi имеется ряд компонентов, специально предназначенных для генерации HTML-документов на основе информации, извлекаемой из базы данных. Таких компонентов четыре:

- ☐ TDataSetPageProducer;
- ☐ TDataSetTableProducer;
- ☐ TQueryTableProducer.
- ☐ TSQLQueryTableProducer.

Рассмотрим каждый из них более подробно.

Компонент TDataSetPageProducer

Этот компонент имеет единственное отличие от компонента TPageProducer. Оно заключается в том, что с TDataSetPageProducer можно связать набор данных с помощью свойства DataSet. При обработке шаблона теги, имена которых совпадают с именами полей набора данных, заменяются значениями этих полей из текущей записи.

Таким образом, используя этот компонент, можно очень просто создавать HTML-документы, содержащие информацию, хранящуюся в базе данных.

ПРИМЕЧАНИЕ

Для использования компонента TDataSetPageProducer необходимо, чтобы имена полей таблицы базы данных состояли только из латинских букв и не содержали пробелов.

Компонент TDataSetTableProducer

Компонент TDataSetTableProducer предназначен для вывода в окне веб-браузера информации, содержащейся в базе данных. Информация при этом представляется в табличной форме. Это своего рода генератор табличных отчетов для публикации в WWW.

Перечислим основные свойства `TDataSetTableProducer`:

- ❑ property `DataSet`: `TDataSet` — указывает набор данных, на основе которого будет формироваться выводимая таблица;
- ❑ property `Dispatcher`: `TCustomWebDispatcher` — указывает на компонент-диспетчер, выполняющий обработку HTTP-запросов. Обычно значение этого свойства устанавливается автоматически — в нем указывается имя компонента `TWebModule`, на форме которого располагается `TDataSetTableProducer`;
- ❑ property `Caption`: `String` — заголовок генерируемого HTML-документа;
- ❑ property `CaptionAlignment`: `THTMLCaptionAlignment` — местоположение заголовка. Может принимать следующие значения:
 - `caDefault` — местоположение заголовка определяется веб-браузером;
 - `caTop` — заголовок располагается над HTML-таблицей;
 - `caBottom` — заголовок размещается под HTML-таблицей;
- ❑ property `Header`: `TStrings` — текст, располагаемый перед таблицей;
- ❑ property `Footer`: `TStrings` — текст, выводимый после вывода таблицы;
- ❑ property `Columns`: `THTMLTableColumns` — используется для указания полей, включаемых в формируемую таблицу, а также для задания атрибутов отображения полей в таблице. Для установки этого свойства используется специальный редактор, открывающийся при щелчке на кнопке с многоточием в поле ввода свойства `Columns` в инспекторе объектов (либо при двойном щелчке на значке компонента `TDataSetTableProducer`, размещенного на форме);
- ❑ property `MaxRows`: `Integer` — максимальное количество строк (записей), выводимых в таблице;
- ❑ property `TableAttributes`: `THTMLTableAttributes` — задает атрибуты отображения таблицы с помощью следующих своих свойств:
 - property `Align`: `THTMLAlign` — способ выравнивания таблицы относительно окна браузера. Может принимать следующие значения: `haDefault` — способ выравнивания определяется браузером, `haLeft` — выравнивание по левому краю, `haRight` — выравнивание по правому краю, `haCenter` — выравнивание по центру;
 - property `BgColor`: `THTMLBgColor` — цвет фона таблицы. Может принимать одно из следующих значений: `Aqua`, `Black`, `Blue`, `Fuchsia`, `Gray`, `Green`, `Lime`, `Maroon`, `Navy`, `Olive`, `Purple`, `Red`, `Silver`, `Teal`, `White`, `Yellow`;
 - property `Border`: `Integer` — толщина линий (в пикселах), разделяющих ячейки таблицы. Если задано значение `-1`, то линии не отображаются;
 - property `CellPadding`: `Integer` — расстояние между ячейками таблицы (в пикселах). Если задано значение `-1`, то расстояние определяется веб-браузером;
 - property `CellSpacing`: `Integer` — расстояние от текста до границы ячейки (в пикселах). Если задано значение `-1`, то расстояние определяется веб-браузером;

- property Width: Integer — ширина таблицы в процентах от ширины окна браузера;
- property RowAttributes: THTMLTableRowAttributes — атрибуты отображения строк в таблице. Класс THTMLTableRowAttributes имеет четыре свойства, два из которых, Align и BgColor, полностью аналогичны соответствующим свойствам класса THTMLTableAttributes. Оставшиеся два свойства устанавливают следующие параметры:
 - property VAlign: THTMLVAlign — вертикальное выравнивание текста в ячейке таблицы. Может принимать одно из следующих значений: haVDefault — выравнивание определяется браузером, haTop — выравнивание по верхней границе ячейки, haMiddle — выравнивание по центру ячейки, haBottom — выравнивание по нижней границе ячейки;
 - property Custom: String — используется для задания дополнительных атрибутов при выводе строки.

Пример вывода информации из таблицы базы данных в виде HTML-таблицы

Рассмотрим пример вывода информации из таблицы базы данных в виде HTML-таблицы. Выведем данные из таблицы «физические лица» базы данных employees.mdb. Для этого потребуются выполнить следующие действия.

1. Создайте новое веб-приложение.
2. Поместите на форму WebModule два компонента: TADOTable и TDataSetTableProducer. По умолчанию им будут присвоены имена ADOTable1 и DataSetTableProducer1.
3. Выполните подключение компонента ADOTable1 к таблице «физические лица» базы данных employees.mdb.
4. Установите свойство DataSet компонента DataSetTableProducer1 равным ADOTable1.
5. Задайте свойству Caption компонента DataSetTableProducer1 следующее значение:
<H2>Список сотрудников</H2>
6. Задайте свойству Header компонента DataSetTableProducer1 следующее значение:
<P>Текст, размещаемый перед выводом HTML-таблицы</P>
7. Задайте свойству Footer компонента DataSetTableProducer1 следующее значение:
<P>Текст, размещаемый после вывода HTML-таблицы</P>
8. Откройте окно редактора столбцов HTML-таблицы. Для этого выполните двойной щелчок на значке компонента DataSetTableProducer1 либо щелкните на кнопке с многоточием в поле ввода свойства Columns этого компонента. Редактор столбцов позволяет задать поля таблицы базы данных, которые следует включить в результирующую HTML-таблицу, и установить атрибуты отображения таблицы. Результаты изменения отображаются в области

предварительного просмотра, расположенной в нижней части окна редактора столбцов (рис. 17.4). При указании полей таблицы базы данных, включаемых в HTML-таблицу, удобно вначале подключить все поля, а затем удалить лишние.

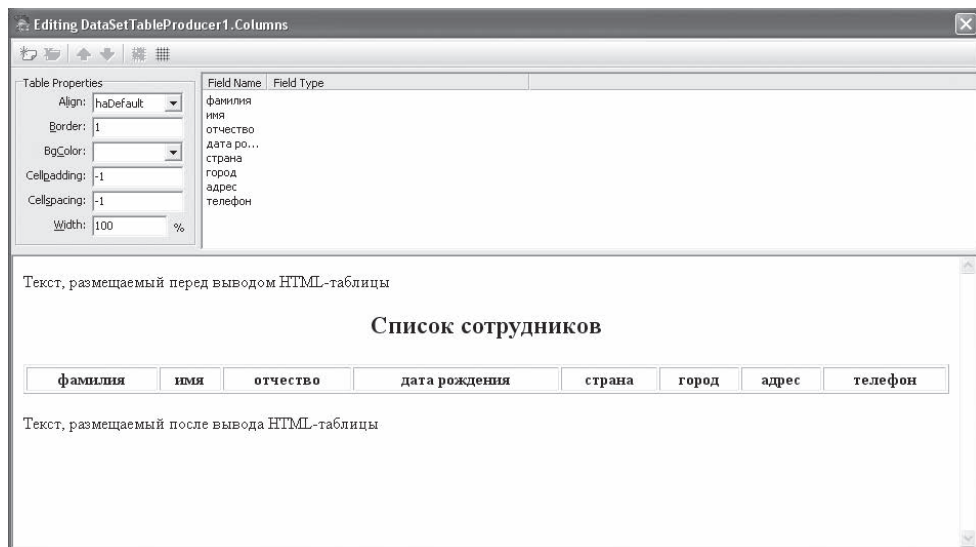


Рис. 17.4. Редактор столбцов компонента TdatasetTableProducer

9. Для включения всех полей таблицы базы данных, связанной с компонентом DataSetTableProducer1, щелкните на кнопке **Add All Fields** на панели инструментов редактора столбцов. Затем оставьте только те поля, которые указаны на рис. 17.4. Измените значение свойства **Border**, установив его равным 1 (в этом случае при выводе будут отображаться линии таблицы).
10. Откройте окно редактора действий компонента WebModule и создайте новое действие. Свойство **PathInfo** действия можно не задавать, остальные параметры также можно оставить заданными по умолчанию. Установите следующий обработчик события **OnAction** для созданного действия:

```
procedure TWebModule1.WebModule1WebActionItem1Action(
  Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content:=DataSetTableProducer1.Content;
end;
```

11. Для получения информации из базы данных необходимо вначале открыть набор данных, а по завершении работы закрыть его. Эти операции удобно выполнять в обработчиках событий **OnCreate** и **OnDestroy** компонента TWebModule. В первом из них следует открыть набор данных ADOTable1, во втором — закрыть его:

```
procedure TWebModule1.WebModuleCreate(Sender: TObject);
begin
  ADOTable1.Open;
end;
```

```

procedure TWebModule1.WebModuleDestroy(Sender: TObject);
begin
    ADOTable1.Close;
end;

```

Теперь веб-приложение, выводящее информацию из таблицы базы данных в виде HTML-таблицы, полностью готово.

Осталось только откомпилировать его, изменить расширение полученного файла на `cgi` и перенести его в каталог `Scripts`.

Тестирование созданного сценария

Для тестирования полученного сценария следует создать простую форму HTML, с помощью которой сценарий запускается на выполнение. В наиболее простом случае достаточно формы, содержащей одну кнопку `SUBMIT`:

```

<HTML>
<HEAD>
    <TITLE> Пример вывода HTML-таблицы </TITLE>
</HEAD>
<BODY>
<FORM METHOD="GET" ACTION="/scripts/test.cgi">
<INPUT TYPE="SUBMIT">
</FORM>
</BODY>
</HTML>

```

Результат выполнения разработанного сценария приведен на рис. 17.5.

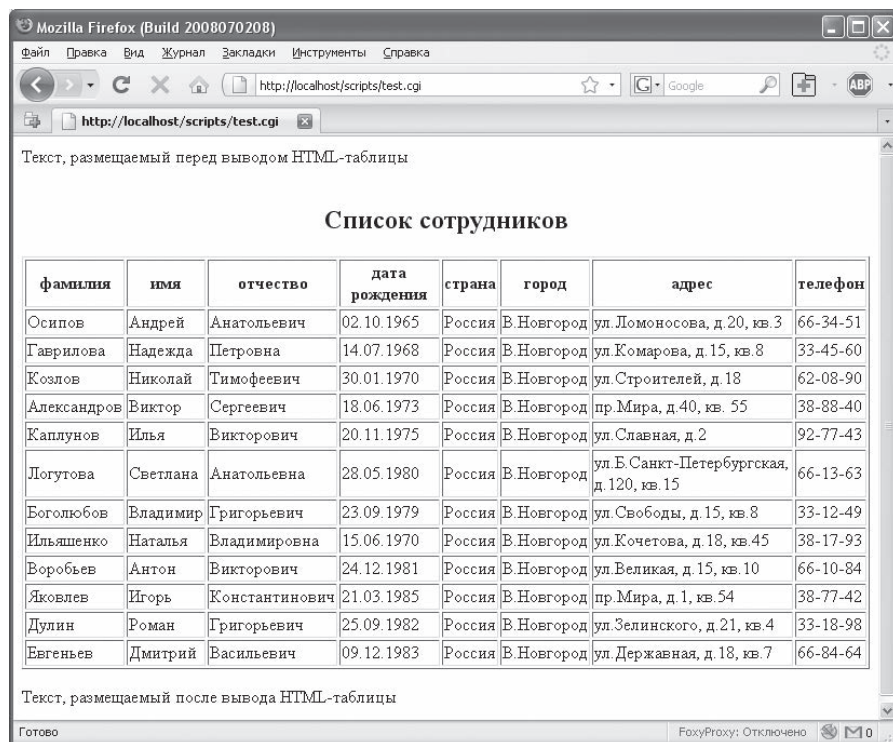


Рис. 17.5. Пример вывода информации из таблицы базы данных в окне браузера

Компонент TDataSetTableProducer может обрабатывать несколько событий, наибольший интерес из которых представляют два:

- type TCreateContentEvent = procedure (Sender: TObject; var Continue: Boolean) of object; property OnCreateContent: TCreateContentEvent — вызывается перед началом генерации HTML-документа. Позволяет запретить генерацию документа, для чего параметру Continue следует присвоить значение false. В обработчике данного события можно, например, проверять, открыт ли набор данных, на основе которого создается HTML-таблица;
- THTMLFormatCellEvent = procedure (Sender: TObject; CellRow, CellColumn: Integer; var BgColor: THTMLBgColor; var Align: THTMLAlign; var VAlign: THTMLVAlign; var CustomAttrs, CellData: string) of object; property OnFormatCell: THTMLFormatCellEvent — вызывается перед формированием каждой ячейки HTML-таблицы. Параметр CellData содержит значение, которое будет помещено в ячейку. Данный параметр можно изменять в тексте процедуры-обработчика.

Путем обработки события OnFormatCell можно улучшить внешний вид получаемой таблицы. Например, задав для этого события обработчик следующего вида, мы получим таблицу, цвет фона строк которой чередуется: нечетные строки отображаются на светло-сером фоне, четные — на белом (рис. 17.6):

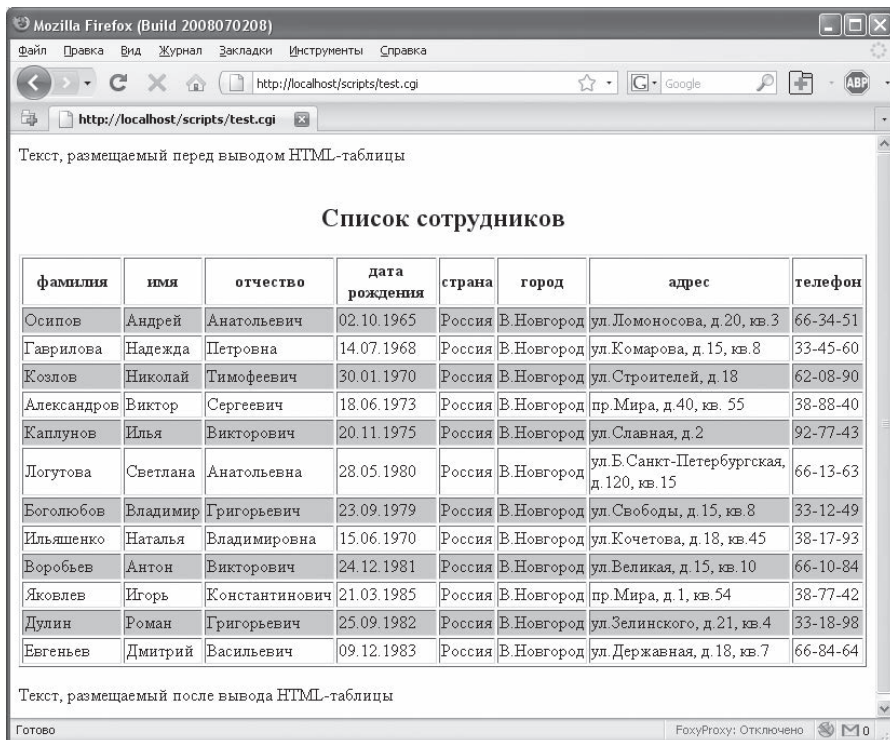


Рис. 17.6. Пример изменения внешнего вида таблицы путем обработки события OnFormatCell

```
procedure TWebModule1.DataSetTableProducer1FormatCell(  
  Sender: TObject; CellRow, CellColumn: Integer;  
  var BgColor: THTMLBgColor; var Align: THTMLAlign;  
  var VAlign: THTMLVAlign; var CustomAttrs,  
  CellData: String);  
begin  
  if Odd(CellRow)  
    then BgColor:='Silver'  
    else BgColor:='';  
end;
```

Компонент TQueryTableProducer

Данный компонент имеет только одно существенное отличие от компонента TDataSetTableProducer: он может связываться только с набором данных TQuery и позволяет настраивать параметры заданного SQL-запроса в соответствии со строкой параметров, полученной с помощью HTTP-запроса. Используя данный компонент, можно предоставить пользователю возможность задавать критерии выборки данных, на основе которых будет формироваться HTML-таблица.

Компонент TSQLQueryTableProducer

Данный компонент имеет только одно существенное отличие от компонента TDataSetTableProducer: он может связываться только с набором данных TSQLQuery и позволяет настраивать параметры заданного SQL-запроса в соответствии со строкой параметров, полученной с помощью HTTP-запроса. Используя данный компонент, можно предоставить пользователю возможность задавать критерии выборки данных, на основе которых будет формироваться HTML-таблица.

Алфавитный указатель

A

- abstract, 99, 240
- access, 100
- active, 112
- active object, 112
- ActiveX, 250
- ADO, 281, 306
- ALL, 352
- ALERTTABL, 155
- ALERTTABLE, 155–161
 - ADD, 155
 - DROP, 155
 - MODIFY, 155
- AND, 354
- Ansi Char, 211
- ANSI SQL-92, 150–153, 162, 340
- Ansi String, 213
- ANY, 353
- array, 214
- ASC, 164, 356
- association, 113
- automated, 242
- AVG, 360

B

- BDE, 247, 281, 306
- BETWEEN...AND, 349
- BIT, 153
- BIT VARYING, 153
- BLOB, 129
- Boolean, 211
- Borland Delphi, 66
- break, 224
- Byte, 210
- Byte Bool, 211

C

- Cardinal, 210
- CASCADE, 167, 177
- CASCADES, 162
- case, 217
- CASE-средства, 61, 64, 68, 69, 179–185, 197, 200
- CASE-технология, 183, 185
- CGI-приложение, 496–498, 505, 508
 - запуск, 496
- CGI-сценарии, 492, 497, 500, 505, 506
 - передача данных серверу, 500
- Char, 211
- CHARACTER, 151
- CHECK, 156, 162
- class, 231
- COM, 242
- COMMIT, 120
- Comp, 212
- complete, 104
- component, 114
- composite state, 106
- concurrency, 99
- concurrent, 99
- Const, 207
- CONSTRAINT, 162
- constructor, 241
- continue, 224
- COUNT, 360
- CRC-карточки, 90
- CREATE INDEX, 164, 165
- CREATE PROCEDURE, 169
- CREATE TABLE, 154, 157, 171
- CREATE TRIGGER, 170
- CREATE UNIQUE INDEX, 164

CREATE VIEW, 167, 372
Currency, 212, 213

D

DATE, 153
default, 237
DEFAULT, 163
DELETE, 171
DELETE FROM, 174
DELETE OF, 162
Delphi, 246, 256
Delphi IDE, 248
 главное меню, 248, 258
 меню Component, 257
 меню Edit, 251
 меню File, 249
 меню Project, 255
 меню Run, 256
 меню Search, 252
 меню Tools, 257
 меню View, 253
 палитра компонентов, 249, 258
 страница, 259
 панель инструментов, 248, 257, 258
derive, 100
DESC, 164, 356
destructor, 241
disjoint, 104
Dispose, 219
DISTINCT, 360, 362
DMT, 239
DNS-сервер, 475
Double, 212, 213
DOUBLE, 153
DROP INDEX, 165
DROP PROCEDURE, 169
DROP TABLE, 156
DROP TRIGGER, 171
DROP VIEW, 167, 168, 373
dynamic, 239

E

EAbstractError, 240
ER-диаграммы, 181
 атрибут, 182
 связь, 182
 сущность, 181

ER-модель, 181
event, 107
Exception, 243
EXECUTE, 169
EXISTS, 351, 352
Extended, 212, 213

F

finalization, 230
Finalize, 215
FLOAT, 153
FloatToStr, 267
FloatToStrF, 267
focus of control, 110
for, 223
FOREIGN KEY, 160
FreeAndNil, 219
FreeMem, 219
FULL OUTER JOIN, 367

G

GetEnvironmentVariable, 501
GetLongHint, 433
GetMem, 219
GetShortHint, 433
global, 113
goto, 221
GRANT, 176
GROUP BY, 360–363, 371
guard condition, 107
guarded, 99

H

HAVING, 363, 371
history state, 106
HtmlHelp, 466
HTML Help, 435, 452–461
 Workshop, 455, 458, 462
 основные элементы, 452
HTML-документ, 477, 482, 483, 499, 504
 абзацы, 485
 выделение фрагментов текста, 485
 заголовки, 484
 раздел заголовка, 483
 списки, 485
 тело документа, 484

HTTP-запрос, 507, 508
HTTP-протокол, 492
HTTP-сервер, 479
 запрос клиента, 479
 коды ответов, 481
 ответ сервера, 480
 сеанс взаимодействия, 479

I

implementation, 230
import, 100
IN, 350
incomplete, 104
inherited, 241
initialization, 230
INSERT, 171
INSERT INTO, 171–176
Int64, 210
Integer, 210
INTEGER, 152
interface, 104, 230
INTERVAL, 153
IntToStr, 267
IP-адрес, 472–475, 479
IP-пакеты, 473
ISAPI-расширения, 492
IS NULL, 348

L

Label, 206
LEFT OUTER JOIN, 367
library, 115
LIKE, 350
link, 112
local, 113
LongBool, 211
LongInt, 210
LongWord, 210

M

MAX, 360
message, 110, 112
MIME, 478, 480, 481
MIN, 360
New, 219

N

nil, 218
node, 116
NOT, 355
NOT NULL, 156
Null, 219

O

object lifetime, 110
ObjectPascal, 205, 246
ODBC, 66
OLEDB, 66
OMG-ObjectManagementGroup, 87
OR, 354
ORDER BY, 355, 356, 361, 362, 368, 371
OUTER JOIN, 367
overlapping, 104
overload, 241
override, 239

P

parameter, 113
pointer, 218
PRIMARY KEY, 158
private, 98, 242
procedure, 226
Program, 206
property, 236
protected, 98, 242
public, 97, 242
published, 242

Q

query, 99

R

raise, 244
Rational Rose, 92
read, 236, 237
Real, 212, 213
record, 216
REFERENCE, 176
refine, 100

repeat...until, 224
RESTRICT, 167, 177
RESTRICTED, 162
result, 228
REVOKE, 176, 177, 178
RIGHT OUTER JOIN, 367
ROLLBACK, 120

S

SELECT, 167, 168, 172, 176, 345, 346
self, 113
sequential, 99
set, 216
SetLength, 215
ShortInt, 210
ShortString, 213
Single, 212, 213
SmallInt, 210
SMALL INT, 152
SQL-запрос, 90, 296
SQL-запросы, 340
SQL-запросы с параметрами, 374
SQL-сервер, 340
String, 214
StrToFloat, 271
StrToInt, 271
StrToIntDef, 271
SUM, 360

T

TabOrder, 322
TabStop, 322
TAction, 387
 события
 OnExecute, 388
TActionList, 387
 свойства, 387
 Images, 387
 Name, 387
 Tag, 387
TApplication, 395, 398
 Active, 398
 BringToFront, 399
 CurrentHelpFile, 398
 ExeName, 398
 Handle, 398

TApplication (продолжение)

 HelpCommand, 464
 HelpContext, 464
 HelpFile, 398
 HelpJump, 464
 Hint, 399
 HintColor, 433
 HintHidePause, 433
 HintPause, 433
 Icon, 399
 MainForm, 399
 Minimize, 399
 OnActionExecute, 399
 OnActivate, 399
 OnDeactivate, 399
 OnHelp, 464
 OnMessage, 399
 OnShortCut, 399
 OnShowHint, 433
 Process Messages, 399
 Restore, 399
 ShowHint, 433
 Terminate, 399
 методы, 399
 свойства, 398
 события, 399
TApplicationEvents, 399
TArrayField, 305
TBlobField, 305, 310
TBooleanField, 304
TButton, 264, 265
 свойства
 Anchors, 264
 Cancel, 264
 Caption, 264
 Default, 264
 Enable, 264
 Font, 264
 Left, 264
 ModalResult, 264
 TabOrder, 264
 TabStop, 264
 Top, 264
 Visible, 264
 события
 OnClick, 265
TCheckBox, 268, 391
 свойства
 AllowGrayed, 269

TCheckBox *(продолжение)*

- Caption, 269
- Checked, 268
- State, 269

TColorDialog, 279

- Color, 280
- CustomColors, 280
- Options, 280
- свойства, 279

TColumn, 309

- Alignment, 309
- Color, 309
- DropDownRows, 309
- FieldName, 309
- Font, 309
- PickList, 309
- PopupMenu, 309
- Showing, 309
- Title, 309
- Visible, 309
- Width, 309
- свойства, 309

TComboBox, 275, 388

- DropDownCount, 275
- DroppedDown, 275
- ItemIndex, 275
- Items, 275
- MaxLength, 275
- SelText, 275
- Sorted, 275
- Style, 275
- свойства, 275

TComponent, 264

TCoolBand, 392

TCoolBands, 392

TCoolBar, 392

TCP-пакеты, 473

TCP-соединение, 479

TDataSet, 296

TDataSetField, 305

TDataSetPageProducer, 515

- DataSet, 515
- свойства, 515

TDataSetState, 300

TDataSetTableProducer, 515

- Caption, 516
- CaptionAlignment, 516
- Columns, 516
- DataSet, 516

TDataSetTableProducer *(продолжение)*

- Dispatcher, 516
- Footer, 516
- Header, 516
- MaxRows, 516
- OnCreateContent, 520
- OnFormatCell, 520
- RowAttributes, 517
- TableAttributes, 516
- свойства, 516
- события, 520

TDataSetTableProducer, 515, 516, 521

TDataSource, 306

- AutoEdit, 307
- DataSet, 307
- Edit, 307
- Enabled, 307
- IsLinkedTo, 307
- OnDataChange, 307
- OnStateChange, 307
- OnUpdateData, 307
- State, 307

TDBCCheckBox, 310

TDBCComboBox, 311

TDBEdit, 310

TDBGrid, 308, 330

- Columns, 308
- DataSource, 308
- DefaultDrawing, 308
- FieldCount, 308
- Fields, 308
- Options, 308
- ReadOnly, 308
- SelectedField, 308
- SelecteedIndex, 308
- свойства, 308

TDBImage, 311

TDBListBox, 311

TDBMemo, 310

TDBNavigator, 311–322, 331

- BeforeAction, 312
- OnClick, 312
- события, 312

TDBRadioGroup, 310

TDBText, 310

TeamSource

- закладки, 429
- работа с локальной копией проекта, 422
- работа с проектом, 421

TeamSource (*продолжение*)

- работа с хранилищем версий, 424, 427

- сборка проекта, 430

TEdit, 270, 271, 388

- свойства, 270

- AutoSelect, 271

- CharCase, 271

- ReadOnly, 271

- SelText, 271

- Text, 271

- события

- OnChange, 271

TField, 302

- Alignment, 302

- AsBoolean, 302

- AsCurrency, 302

- AsDateTime, 302

- AsFloat, 302

- AsInteger, 302

- Assign, 303

- AssignValue, 303

- AsString, 302

- AsVariant, 302

- Calculated, 302

- CanModify, 302

- Clear, 303

- ConstraintErrorMessage, 302

- CurValue, 302

- CustomConstraint, 302

- DataSet, 302

- DataSet, 302

- DataType, 302

- DefaultExpression, 303

- DisplayLabel, 303

- DisplayText, 303

- FieldKind, 303

- IsBlob, 303

- IsIndexField, 303

- IsNull, 303

- IsValidChar, 303

- KeyFields, 303

- Lookup, 303

- NewValue, 303

- OldValue, 303

- ReadOnly, 303

- SetFieldType, 303

- ValidChars, 303

- Value, 303

- Visible, 303

TField (*продолжение*)

- методы, 303

- свойства, 302

TFloatField, 304

TFontDialog, 279

- Device, 279

- Font, 279

- MaxFontSize, 279

- MinFontSize, 279

- OnApply, 279

- OnClose, 279

- OnShow, 279

- Options, 279

- свойства, 279

- события, 279

TForm, 314, 315, 394

- события, 316

- OnActivate, 316

- OnClick, 316

- OnClose, 316

- OnCloseQuery, 316

- OnCreate, 316

- OnDblClick, 316

- OnDeactivate, 316

- OnDestroy, 316

- OnKeyPress, 316

- OnMouseDown, 316

- OnMouseMove, 316

- OnMouseUp, 316

- OnPaint, 316

- OnShow, 316

- методы, 316

- Close, 316

- CloseQuery, 316

- FocusControl, 316

- GetFormImage, 316

- Hide, 316

- Print, 316

- Release, 316

- Show, 316

- ShowModal, 316

- свойства, 314, 315

- Active, 315

- ActiveControl, 314

- ActiveMDIChild, 315

- AutoScroll, 314

- AutoSize, 314

- BorderIcons, 314

- BorderStyle, 314, 315

TForm, свойства (*продолжение*)

- BorderWidth, 314
- Canvas, 315
- Caption, 314
- ClientHeight, 314
- ClientWidth, 314
- Color, 314
- Constraints, 314
- Ctl3D, 314
- Cursor, 314
- FormStyle, 314
- Height, 314
- Icon, 314
- Left, 314
- MDIChildCount, 315
- MDIChildren, 315
- Menu, 314
- ModalResult, 315
- Name, 314
- PopupMenu, 314
- Tag, 314
- Top, 314
- Visible, 314
- Width, 314
- WindowState, 314

TFrames, 318

TGraphic, 276

TGraphicField, 305

TGroupBox, 271, 272

TImage, 275

- AutoSize, 275
- Center, 275
- Picture, 275
- Stretch, 275

TIME, 153

TIMESTAMP, 153

TIntegerField, 304

Tlabel, 266

свойства

- Alignment, 267
- Autosize, 267
- Layout, 267
- Transparent, 267
- WordWrap, 267

TLargeField, 304

TListBox, 274

- Columns, 274
- ItemIndex, 274

Items, 274

TListBox (*продолжение*)

MultiSelect, 274

SelCount, 274

Selected, 274

Sorted, 274

свойства, 274

TMainMenu, 382

методы

Merge, 382

Unmerge, 382

методы, 382

свойства, 382

AutoHotkeys, 382

AutoMerge, 382

Images, 382

OwnerDraw, 382

TMemo, 274, 310

Alignment, 274

CaretPos, 274

Lines, 274

SelText, 274

TMemoField, 305, 310

TMenuItem, 382, 383, 387, 388

свойства, 383

Action, 383

Add, 383

Bitmap, 383

Break, 383

Caption, 383

Checked, 383

Clear, 383

Click, 384

Default, 383

Enabled, 383

Find, 384

GroupIndex, 383

ImageIndex, 383

IndexOf, 384

Insert, 384

InsertNewLineAfter, 384

InsertNewLineBefore, 384

IsLine, 384

MenuIndex, 383

OnAdvancedDrawItem, 384

OnClick, 384

OnDrawItem, 384

OnMeasureItem, 384

TMenuItem, свойства *(продолжение)*

RadioItem, 383

Remove, 384

ShortCut, 383

методы, 383

события, 384

TObject, 235

TOpenDialog, 276

TOpenPictureDialog, 277

TPageControl, 333

свойства, 333

ActivePage, 333

ActivePageIndex, 333

HotTrack, 333

MultiLine, 333

PageCount, 333

Pages, 333

Style, 333

TPageProducer, 512, 515

Content, 512

HTMLDoc, 512

HTMLFile, 512

методы, 512

свойства, 512

TPageProducer

OnHTMLTag, 512

события, 512

TPanel, 271, 272

свойства

BevelInner, 272

BevelOuter, 272

TPicture, 275

Bitmap, 276

Height, 275

Icon, 276

LoadFromClipboardFormat, 276

LoadFromFile, 276

Metafile, 276

Width, 275

методы, 276

TPopupMenu, 388

TPrintDialog, 280

Collate, 280

Copies, 280

FromPage, 280

MaxPage, 280

MinPage, 280

Options, 280

PrintRange, 280

TPrintDialog *(продолжение)*

PrintToFile, 280

ToPage, 280

свойства, 280

TPrinterSetupDialog, 280

TQuery, 296, 341, 521

методы, 341

ExecSQL, 341

свойства, 341

DataSource, 341

ParamCheck, 341

Prepared, 341

RowsAffected, 341

TQueryTableProducer, 515, 521

TRadioButton, 269, 391

свойства

Caption, 269

Checked, 269

события

OnClick, 270

try...except, 243

try...finally, 244

TSaveDialog, 276

TSavePictureDialog, 277

TSmallIntField, 304

TStatusBar, 434

TStatusPanel, 435

Bevel, 435

Style, 435

Text, 435

Width, 435

свойства, 435

TStringField, 304

TTable, 296, 341

Active, 288, 296

AddIndex, 283, 298

AfterCancel, 301

AfterClose, 301

AfterDelete, 301

AfterEdit, 301

AfterInsert, 301

AfterOpen, 301

AfterPost, 301

AfterScroll, 301

ApplyRange, 298

BeforeCancel, 301

BeforeClose, 301

BeforeDelete, 301

BeforeEdit, 301

TTable (продолжение)

- BeforeInsert, 301
- BeforeOpen, 301
- BeforePost, 301
- BeforeScroll, 301
- BOF, 288, 296
- Cancel, 298
- CancelRange, 298
- DatabaseName, 296
- DefaultIndex, 297
- DeleteTable, 298
- Edit, 298
- EditKey, 298
- EditRangeEnd, 298
- EditRangeStart, 298
- EmptyTable, 298
- EOF, 297
- Exclusive, 297
- Exists, 297
- FieldByName, 298
- FieldCount, 288, 297
- Fields, 288, 297
- FindKey, 298
- FindNearest, 298
- First, 298
- GotoKey, 299
- GotoNearest, 299
- IndexDefs, 288, 297
- IndexFieldCount, 288, 297
- IndexFieldNames, 288, 297
- IndexFields, 289, 297
- IndexFiles, 297
- IndexName, 289, 297
- Insert, 299
- KeyExclusive, 297
- KeyFieldCount, 297
- Last, 299
- Locate, 287, 299
- LockTable, 299
- MasterFields, 289, 297
- MasterSource, 289, 297
- Modified, 289, 297
- MoveBy, 299
- Next, 299
- OnCalcFields, 301
- OnNewRecord, 301
- Post, 299
- Prior, 299
- ReadOnly, 289, 297

TTable (продолжение)

- RecordCount, 289, 297
- RenameTable, 299
- SetKey, 299
- SetRange, 299
- SetRangeEnd, 300
- SetRangeStart, 300
- TableLevel, 297
- TableName, 289, 297
- TableType, 297
- UnlockTable, 300
- методы, 286, 298
- свойства, 282, 288, 296
- события, 301

TTabSheet, 334

TToolBar, 388–392

- свойства, 388
 - ButtonHeight, 389
 - ButtonWidth, 389
 - DisabledImages, 389
 - Flat, 389
 - HotImages, 389
 - Images, 389
 - Indent, 389
 - List, 389
 - OnDockOver, 389
 - OnDragDrop, 389
 - OnDragOver, 389
 - OnEndDock, 389
 - OnEndDrag, 389
 - OnStartDock, 389
 - OnStartDrag, 389
 - ShowCaptions, 389
 - Transparent, 389
 - Wrapable, 389
- события, 389

TToolButton, 389, 391

- свойства, 389
 - Action, 389
 - AllowAllUp, 390
 - AutoSize, 390
 - Caption, 389
 - Down, 390
 - DropdownMenu, 390
 - Grouped, 390
 - Hint, 390
 - ImageIndex, 390
 - Indeterminate, 390
 - Marked, 390

TToolButton (*продолжение*)

MenuItem, 390

ShowHint, 390

Style, 390

методы

CheckMenuDropDown, 390

Click, 390

методы, 390

события, 390

OnClick, 390

TWebModule, 506, 509, 511

Action, 506

AfterDispatch, 508

BeforeDispatch, 508

OnCreate, 508

OnDestroy, 508

свойства, 506

события, 508

TWebRequest, 507

Content, 507

ContentFields, 507

Method, 507

Query, 507

QueryFields, 507

RemoteAddr, 507

свойства, 507

TWebResponse, 507

Content, 507

ContentLength, 507

ContentStream, 508

ContentType, 507

свойства, 507

TWordField, 304

Type, 206, 207

UUML— универсальный язык
моделирования, 87

call, 111

create, 111

destroy, 111

return, 111

send, 111

state, 105

swimlanes, 109

абстрактный класс, 97

вариант использования, 93

UML (*продолжение*)

диаграмма активности, 92

диаграмма классов, 92, 97

диаграмма компонентов, 114

диаграмма последовательности, 92

диаграммапрецедентов, 93

диаграмма состояний, 105

диаграммы активности, 108

диаграммы последовательности, 110

диаграммы развертывания, 116

диаграммы сотрудничества, 111

диаграммы сотрудничества уровня
примеров, 112диаграммы сотрудничества уровня
спецификаций, 112

дорожки, 109

интерфейс, 93

исключающая ассоциация, 100

исторические состояния, 106

конечное состояние, 105

кратность, 98

метамодель, 91

начальное состояние, 105

отношение агрегации, 101

отношение композиции, 102

отношение реализации, 115

отношения ассоциации, 94, 100

отношения включения, 94

отношения зависимости, 99

отношения обобщения, 94, 102

отношения расширения, 94

отношение зависимости, 115

пакеты, 96

параллельное состояние, 106

прецедент, 93

синхронизирующие состояния, 108

составное состояние, 106

состояние действия, 109

состояния, 105

сторожевое условие, 107

строка-свойство, 99

узел, 116

Unassigned, 219

UNION, 369, 370

UNION ALL, 369, 370

UNIQUE, 156, 158, 352

unit, 229, 230

UPDATE, 171, 173, 174, 176

UPDATE OF, 162

URL, 475, 476
Uses, 206, 229, 230

V

Var, 207
VarArrayCreate, 220
VarArrayLock, 221
VarArrayOf, 221
VarArrayRedim, 221
VARCHAR, 152
variant, 219
VarType, 219
VCL, 262, 264
virtual, 239
VisualBasic, 66
VisualComponentLibrary, 262
VMT, 239

W

WHERE, 173, 175, 347, 362
while...do, 224
WideChar, 211
WideString, 214
WinHelp, 435, 462
WITH GRANT OPTION, 177
Word, 210
WordBool, 211
write, 236, 237
WWW-сервер, 476

X

XML, описание типа документа, 491

A

Аббота метод, 90
абсолютный URL-адрес, 487
абстрактные методы, 99
абстрактный класс, 97
активный объект, 112
алгоритм, 108
аномалии ввода, 144
аномалии обновления, 144
аномалии удаления, 144
артефакт, 115
архитектура клиент-сервер, 32, 80

архитектура файл-сервер, 31
атрибуты, 129, 131, 136

Б

база данных, 119, 121, 123, 125
безопасность, 40
библиотека визуальных компонентов, 262
блокировки, 410
браузер, 477
Буча метод, 91
быстрая разработка приложений, 69

B

вариантные записи, 217
вариантные типы, 209, 219
веб-дизайн, 478, 482
веб-документ, 477
веб-приложения, 470, 476–478
веб-программирование, 478, 480
веб-сервер, 476–481, 492, 493, 497
 CONTENT_LENGTH, 502
 PATH_INFO, 502
 QUERY_STRING, 501
 REMOTE_ADDR, 502
 REMOTE_HOST, 502
 REQUEST_METHOD, 502
 SERVER_NAME, 502
 SERVER_PORT, 502
 SERVER_PROTOCOL, 502
 переменные окружения, 502
веб-страницы, 477
 динамические, 477
 статические, 477
верификация проекта, 185
версии проекта, 409, 429
ветвление, 109
вещественные типы с плавающей
 точкой, 152, 153
вещественные типы с фиксированной
 точкой, 152
вид параметра, 99
визуальное программирование, 66
визуальные средства программирования, 69
визуальные средства разработки, 66
 специализированные, 66
 универсальные, 66

внешние соединения, 367
внешний ключ, 159, 161
всплывающие подсказки, 381, 432
вычисляемые поля, 304

Г

гибкость, 38
гиперссылки, 487
главная таблица, 138, 337
горячие клавиши, 381
группа проектов, 401, 402
 управление, 402
группировка данных, 360

Д

двоичные строки переменной длины, 152
двоичные строки фиксированной
 длины, 152, 153
действительные типы, 212, 213
действия, 387, 388
декомпозиция проекта, 62
деструктор, 241
детерминант, 145
диаграммы сущность–связь, 90, 181
диапазонные типы, 212
директивы компилятора, 404, 405
домен, 129, 130, 131, 474
доменная система имен, 474
доменный адрес, 474

Ж

жизненный цикл информационных
 систем, 41–67, 81, 83, 200
 вспомогательные процессы жизненного
 цикла, 49
 организационные процессы жизненного
 цикла, 49
 основные процессы жизненного
 цикла, 47
 разработка, 48
 сопровождение, 48
 эксплуатация, 48
структура, 47, 50
 начальная стадия, 50
 стадия конструирования, 51
 стадия перехода, 51
 стадия уточнения, 51

журнализация (протоколирование), 121,
 122
журнал изменений базы данных, 121

З

заголовок программы, 206
записи, 213, 216
значение по умолчанию, 98
значения по умолчанию, 163

И

иерархия объектов, 89
избыточность данных, 143
изображения, 275
индексы, 140
 одностолбцовый индекс, 164
 создание, 163
 составные индексы, 164
 типы индексов, 141
 удаление, 163, 165
 уникальные индексы, 164
инжиниринг бизнеса, 98
инкапсуляция, 65, 232, 233, 234, 246
инспектор объектов, 260
интегрированная среда разработки, 246
Интернет, 470
Интернет-приложения, 470–520
интерфейс, 104
 пользователя, 380, 381
интранет, 494
интрасети, 494
информационная система, 24, 61, 64,
 67, 78
 корпоративная, 41, 48
информационно-справочные системы, 30
информационные системы, 408
информационные технологии, 61
исключительные ситуации, 99, 243
итерационная модель разработки, 64
итерационный процесс разработки, 58

К

кардинальное число, 131
каскадное обновление, 162
каскадное удаление, 137

клавиатурные сокращения, 382, 385
клавиши ускоренного доступа, 385
класс, 97
классификация информационных систем, 28, 29
классы, 213, 217, 231
ключ, 129, 132
 альтернативный, 134
 внешний, 135, 136
 вторичный, 134
 естественный, 133
 искусственный, 133
 первичный, 132, 133, 145
 потенциальный, 134
 простой, 133
 сложный, 133
 составной, 133
 суррогатный, 133
кнопки, 264
Кодд, 126, 128, 129
коллективная разработка приложений, 408
коллективная разработка проектов, 184
команды запуска приложения, 407
команды компиляции, 406
комбинированные списки, 275
компиляция приложения, 406
компиляция файла справки, 444
компонент, 114
компонентный подход
 к программированию, 59
компоненты, 245, 247
компоненты Delphi, 250, 262
 визуальные, 263
 невизуальные, 263
компоненты VCL, 393
компоненты для формирования документов HTML, 511
компьютерная инфраструктура, 26
константы, 207
конструктор, 241
контекстное меню, 388
контекстно-зависимая справка, 431
контрольная сумма, 473
конфликт методов, 103
концептуальная модель, 180, 182
 объектно-ориентированная модель, 180
 семантическая модель, 180
кооперация, 112
корпоративная информационная система, 24

корпоративные стандарты, 78
кортеж, 129, 131, 134

Л

линия синхронизации, 107
логические типы, 211
локальные переменные, 228

М

макрокоманды WinHelp 438
 ALink, 438
 AppendMenu, 438
 BrowseButtons, 438
 ControlPanel, 438
 CreateButton, 438
 ExecFile, 438
 Find, 438
 InsertMenu, 438
 KLink, 438
 MPrintID, 438
 ShellExecute, 438
 ShortCut, 438
манипулирование данными, 171, 173
маршрутизатор, 472
маршрутизация, 472
массивы, 213, 214
 динамические массивы, 215
 статические массивы, 214
менеджер проектов, 253, 398, 402
меню, 380, 381
 главное меню, 381
 подменю, 386
 разделители, 386
 реакция на выбор команды, 386, 387
 редактор меню, 384
метакласс, 104
метки, 206, 223
метод, 98
метод GET, 479, 500, 501, 511
метод POST, 479, 500, 501, 504, 511
методика Oracle CDM, 79, 80, 81, 82, 83
методология RAD, 63, 64, 65, 69, 70
 фазы жизненного цикла, 67, 69
 фаза анализа и планирования требований, 67
 фаза внедрения, 70

методология RAD (*продолжение*)
 фаза построения, 69
 фаза проектирования, 68
методология быстрой разработки
 приложений, 63
методы, 238
 абстрактные, 240
 виртуальные, 239
 динамические, 239
 класса, 231
 перегружаемые, 241
 статические, 238
многодокументный интерфейс, 313
многомерные массивы, 214
многоуровневая сетевая модель, 471
 LLC, 472
 MAC, 472
 межсетевой уровень, 471, 472
 транспортный уровень, 471, 473
 уровень приложений, 471, 473
 уровень сетевого доступа, 471, 472
множества, 213, 216
множественное наследование, 103
модальные формы, 313
модели жизненного цикла, 52, 85, 87
 информационной системы, 51–59
 каскадная модель, 52–54
 возврат на более ранние стадии, 55, 57
 высокий уровень риска, 57
 задержка получения результатов, 54
 информационная
 перенасыщенность, 56
 основные этапы разработки, 52
 сложность распараллеливания
 работ, 55
 сложность управления проектом, 56
 спиральная модель, 52, 58
 итерации, 58
модель сущность–связь, 181
модули, 206, 229
 блок завершения, 230
 блок инициализации, 230
 блок интерфейса, 230
 блок реализации, 230
 данных, 308
 заголовков, 230
мощность отношения, 131
мультиобъект, 112

Н

набор данных, 281, 300
надежность, 38
надписи, 266
наследование, 127, 232, 234, 246
неключевой атрибут, 145, 147
немодальные формы, 313
нормализация данных, 142–147
нормальные формы, 144
 вторая нормальная форма, 144, 145
 нормальная форма Бойса–Кодда, 145
 первая нормальная форма, 144, 145
 пятая нормальная форма, 145
 третья нормальная форма, 145, 146
 четвертая нормальная форма, 145

О

области видимости, 242
 автоматизация, 242
 защищенные, 242
 личные, 242
 общие, 242
 опубликованные, 242
область действия метода, 99
обновление локальных копий файлов
 проекта, 413, 418
обобщение, 112
обработка исключительных ситуаций, 246
обработчик события, 67, 261
объединение запросов, 369
объект, 65, 67, 104, 232
объект базы данных, 154
объектная модель, 127
объектно-ориентированная модель, 204
объектно-ориентированное
 программирование, 65, 66, 89, 204, 230
объектно-ориентированное
 проектирование, 64
объектно-ориентированные методы, 65
объектно-ориентированные
 СУБД, 126, 127
объектно-ориентированный анализ
 CRC-карточки, 90
 метод Аббота, 90
объектные типы, 231
объекты, 232

ограничения, 156
ограничение CHECK, 156, 162
ограничение NOTNULL, 156–158
ограничение UNIQUE, 156–159
ограничения (*продолжение*)
ограничение внешнего
ключа, 156, 159–161
ограничение первичного ключа, 156, 157
ограничения целостности, 136
ограничительные условия, 136, 154
однодокументный интерфейс, 313
окна диалога, 276–278
DefaultExt, 277
Execute, 277
FileEditStyle, 277
FileName, 277
Files, 277
Filter, 277
FilterIndex, 277
HistoryList, 277
InitialDir, 277
OnClose, 277
OnCloseQuery, 277
OnFolderChange, 277
OnSelectionChange, 277
OnShow, 277
OnTypeChange, 277
Options, 277
Title, 277
методы, 277
свойства, 277
события, 277
фильтры, 278
оператор break, 224
оператор case...of, 223
оператор continue, 224
оператор if, 222
оператор безусловного перехода, 221
оператор присваивания, 221
операторы Object Pascal, 221
операторы цикла, 223
оптимизация быстродействия, 115
ответственность, 40
открытые информационные системы, 71
открытые массивы, 227, 228
относительный URL-адрес, 484, 487
отношения, 129
основные свойства, 138
ошибки процесса разработки, 40

П

пакеты, 262, 403, 470, 471
пакеты времени выполнения, 262
пакеты времени разработки, 262
панели инструментов, 380, 381, 388
параметризированный класс, 104
параметры-константы, 227
параметры-переменные, 227
параметры процедуры, 226
первичный ключ, 67
передача параметров, 226
посылке, 227
позначению, 226
переключатели, 269
перекомпиляция, 115
перекрестные ссылки, 436
перекрытие методов, 241
переменные, 207
перечисляемые типы, 211
планирование приложения, 380
повторное использование компонентов, 59
подзапросы, 368
подсистемы, 26
подчиненная таблица, 138, 337
позднее связывание, 239
поле, 302
полиморфизм, 127, 232, 235, 246
пользователь, 93
поля записи, 216
полякласса, 231, 236
порт, 475
порядковые типы, 210
предметная область, 65, 90, 134, 179
представления, 165, 371
области применения, 166
создание, 167
удаление, 167
привилегии пользователей, 175
объектные привилегии, 175, 176
системные привилегии, 175
признак оригинальности программы
для ЭВМ, 92
примесные классы, 103
примечания, 93
проблема коммуникации, 89
программа Object Pascal, 205
проект, 42
классификация, 43

проект (*продолжение*)
настройка параметров, 403
основные отличительные признаки, 42
свойство управляемости, 43
техничко-экономические показатели, 43
проект Delphi, 393
главный файл проекта, 395
добавление форм и модулей, 397
модуль формы проекта, 394
удаление форм и модулей, 397
файл описания формы, 395
файлы проекта, 394
проектирование информационной системы
основные ошибки, 47
производный класс, 234
простые индексы, 141
простые типы, 209
протокол, 470, 471
FTP, 474, 475
HTTP, 474–481, 497
IP, 470, 471, 473
POP, 474
SMTP, 474, 475
TCP/IP, 473
TCP, 473
WAL, 122
прототипы, 64–69, 184
профили жизненного цикла
информационных систем, 78
профиль информационных систем, 71
структура, 74
процедурно-ориентированная модель, 204
процедуры, 225
псевдоним, 305
псевдонимы полей, 358
публикация, 476

Р

расширения ISAPI, 493
редактор кода, 262
редактор форм, 261
рекурсия, 111
реляционная модель данных, 127, 129, 179
реляционная система управления базами данных, 139
реляционные базы данных, 118–148

реляционные СУБД, 128, 134, 137
репозитарий, 184
родительская таблица, 159
родительский ключ, 159, 161
роль, 112

С

сбои, 121
аппаратные, 121
жесткие, 121, 122
мягкие, 121, 122
программные, 121
свойства, 236
векторные, 237
значение по умолчанию, 237
только для записи, 237
только для чтения, 237
свойства класса, 231, 232
связи между таблицами, 135
многие к одному, 138
многие ко многим, 138
один ко многим, 138
связь один к одному, 138
связи между таблицами, 138
селектор, 223
семантическое моделирование, 180
символьные строки переменной длины, 151, 152
символьные строки фиксированной длины, 151
символьные типы, 210
синхронизация версий, 63
синхронизация документации, 56
система управления базами данных (СУБД), 63, 66, 119
основные свойства, 119
основные функции, 119
системы контроля версиями проектов, 409
системы обработки транзакций, 29
системы поддержки принятия решений, 30
сложность, 40
событийное программирование, 67
события, 67
содержание справочной системы, 436
соединение неравенства, 367
соединение равенства, 364
соединения таблиц, 364

сокращение избыточности, 103
сообщение, 232
составной объект, 112
составной оператор, 224
составные запросы, 369
составные индексы, 142
состояния набора данных, 300
спецификация ISAPI, 493
список, 274
справочная система, 256, 431–468
среда визуальной разработки, 246
средства RAD, 65
средства быстрой разработки приложений, 63, 245
средства визуального программирования, 65, 66, 245, 246
средства групповой разработки, 411
ссылочная целостность, 67
ссылочной целостность данных, 161
стандарты, 380
 ISO/IEC 12207 1995-08-01, 79, 83
 ISO/IEC 12207, 47, 51
 виды стандартов, 79
 на интерфейсы, 78
 открытых систем, 78
степень отношения, 131
стереотип, 100, 111
сторожевое условие, 107, 114
строка состояния, 432, 434
строки, 213
строковые типы, 213
структурные типы, 207, 209, 213
суперключ, 133
схема базы данных, 131
схема отношения, 131
сценарий, 477

Т

таблица виртуальных методов, 239
таблица динамических методов, 239
табличные формы, 327
тег, 483, 485
 A, 487
 B, 485
 BODY, 483
 BR, 485
 FORM, 488

тег (*продолжение*)

 FRAMESET, 484
 HEAD, 483
 I, 485
 INPUT, 488, 489
 LI, 485
 LINK, 484
 OL, 485
 P, 485
 U, 485
 UL, 485
текстовые поля, 270
текущая запись, 138
темы, 435
 атрибуты, 439
 изображения, 443
 кнопки, 443
 перекрестные ссылки, 440
 текст, 440
темы, 438, 453
технология проектирования, 62
тип данных, 129
типизированные константы, 208
типизированные поля, 304
типы, 206
типы данных SQL/92, 151
 строковые типы, 151
 типы для представления даты и времени, 151, 153
 числовые типы, 151, 152
транзакции, 90, 120
 откат, 120
 фиксация изменений, 120
триггеры, 67, 80, 107, 150
 создание, 170
 удаление, 171
триггеры, 170

У

указатели справочной системы, 436
указательные типы, 209, 218
уникальные индексы, 142
унифицированный указатель ресурсов, 475
управление безопасностью баз данных, 175, 177
управление доступом к базе данных, 176

- управление конфигурацией, 64
- управление объектами базы данных, 154–169
- управление проектами, 42, 43, 45
- управление реляционными базами данных, 149, 178
- управление транзакциями, 120
- условный оператор, 222
- устройства внешней памяти, 120

Ф

- фазы проектирования информационной системы, 44
 - ввод системы в эксплуатацию, 46
 - концептуальная фаза, 45
 - проектирование, 46
 - разработка, 46
 - разработка технического предложения, 45
- файловый тип, 217
- файлы, 213
- фактические параметры, 226
- физическая модель, 182, 198
- физическое проектирование системы, 69
- флажки, 268
- фокус управления, 110
- формальные параметры, 226, 227
- формы, 263, 313, 399
 - HTML, 488
 - для ввода, 322
 - со вкладками, 333
 - управление формами, 399
- функции, 228
- функции агрегирования, 359
- функциональная зависимость, 145
- функциональная связь, транзитивная, 147

Х

- хранилище версий, 427
- хранилище объектов, 250, 409, 412
- храняемая процедура, 32, 80
- хранимые процедуры, 168
 - выполнение, 169
 - выполняемые процедуры, 169

- хранимые процедуры (*продолжение*)
 - процедуры выбора, 168
 - создание, 169
 - удаление, 169

Ц

- целостность данных, 136
 - категорийная целостность, 136
 - ссылочная целостность, 136, 137
- целочисленные типы, 152
- целые типы, 210
- цикл for...do, 223
- цикл repeat...until, 224
- цикл while...do, 224

Ч

- члены класса, 233

Э

- экземпляр класса, 232
- эффективность, 39

Я

- язык
 - HTML, 478–489
 - QBE, 126, 149
 - QUEL, 149
 - SQL, 123, 126, 149, 296, 340, 341, 345
 - команды администрирования базы данных, 151
 - команды управления транзакциями, 151
 - OCL, 92
 - UML, 87
 - XHTML, 491
 - XML, 491
 - баз данных, 123
 - манипулирования данными, 123, 150, 171
 - определения схем данных, 123
 - запросов, 151
 - определения данных, 150
 - управления данными, 150

*Юрий Сергеевич Избачков, Владимир Николаевич Петров,
Александр Алексеевич Васильев, Ирина Сергеевна Телина*

Информационные системы: Учебник для вузов
3-е издание

Заведующий редакцией
Руководитель проекта
Ведущий редактор
Редактор
Художественный редактор
Корректоры
Верстка

*А. Кривцов
А. Юрченко
Ю. Сергиенко
В. Смартышев
Л. Адуевская
В. Листова, В. Нечаева
Л. Егорова*

Подписано в печать 18.06.10. Формат 70х100/16. Усл. п. л. 43,86. Тираж 2000. Заказ 0000.
ООО «Лидер», 194044, Санкт-Петербург, Б. Сампсониевский пр., д. 29а.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Отпечатано по технологии СІР в ОАО «Печатный двор» им. А. М. Горького.
197110, Санкт-Петербург, Чкаловский пр., д. 15.

ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
предлагают эксклюзивный ассортимент компьютерной, медицинской,
психологической, экономической и популярной литературы

РОССИЯ

Санкт-Петербург м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

Москва м. «Электрозаводская», Семеновская наб., д. 2/1, корп. 1, 6-й этаж
тел./факс: (495) 234-38-15, 974-34-50; e-mail: sales@msk.piter.com

Воронеж Ленинский пр., д. 169; тел./факс: (4732) 39-61-70
e-mail: piterctr@comch.ru

Екатеринбург ул. Бебеля, д. 11а; тел./факс: (343) 378-98-41, 378-98-42
e-mail: office@ekat.piter.com

Нижний Новгород ул. Совхозная, д. 13; тел.: (8312) 41-27-31
e-mail: office@nnov.piter.com

Новосибирск ул. Станционная, д. 36; тел.: (383) 363-01-14
факс: (383) 350-19-79; e-mail: sib@nsk.piter.com

Ростов-на-Дону ул. Ульяновская, д. 26; тел.: (863) 269-91-22, 269-91-30
e-mail: piter-ug@rostov.piter.com

Самара ул. Молодогвардейская, д. 33а; офис 223; тел.: (846) 277-89-79
e-mail: pitvolga@samtel.ru


УКРАИНА

Харьков ул. Суздальские ряды, д. 12, офис 10; тел.: (1038057) 751-10-02
758-41-45; факс: (1038057) 712-27-05; e-mail: piter@kharkov.piter.com

Киев Московский пр., д. 6, корп. 1, офис 33; тел.: (1038044) 490-35-69
факс: (1038044) 490-35-68; e-mail: office@kiev.piter.com

БЕЛАРУСЬ

Минск ул. Притыцкого, д. 34, офис 2; тел./факс: (1037517) 201-48-79, 201-48-81
e-mail: gv@minsk.piter.com

 Ищем зарубежных партнеров или посредников, имеющих выход на зарубежный рынок.
Телефон для связи: **(812) 703-73-73. E-mail: fuganov@piter.com**

 **Издательский дом «Питер»** приглашает к сотрудничеству авторов. Обращайтесь
по телефонам: **Санкт-Петербург – (812) 703-73-72, Москва – (495) 974-34-50**

 Заказ книг для вузов и библиотек по тел.: (812) 703-73-73.
Специальное предложение – e-mail: kozin@piter.com

 Заказ книг по почте: на сайте **www.piter.com**; по тел.: (812) 703-73-74
по ICQ 413763617

ДАЛЬНИЙ ВОСТОК

Владивосток

«Приморский торговый дом книги»
тел./факс: (4232) 23-82-12
e-mail: bookbase@mail.primorye.ru

Хабаровск, «Деловая книга», ул. Путевая, д. 1а
тел.: (4212) 36-06-65, 33-95-31
e-mail: dkniga@mail.kht.ru

Хабаровск, «Книжный мир»
тел.: (4212) 32-85-51, факс: (4212) 32-82-50
e-mail: postmaster@worldbooks.kht.ru

Хабаровск, «Мирс»
тел.: (4212) 39-49-60
e-mail: zakaz@booksmirs.ru

ЕВРОПЕЙСКИЕ РЕГИОНЫ РОССИИ

Архангельск, «Дом книги», пл. Ленина, д. 3
тел.: (8182) 65-41-34, 65-38-79
e-mail: marketing@avfkniga.ru

Воронеж, «Амиталь», пл. Ленина, д. 4
тел.: (4732) 26-77-77
http://www.amital.ru

Калининград, «Вестер»,
сеть магазинов «Книги и книжечки»
тел./факс: (4012) 21-56-28, 6 5-65-68
e-mail: nshibkova@vester.ru
http://www.vester.ru

Самара, «Чакона», ТЦ «Фрегат»
Московское шоссе, д. 15
тел.: (846) 331-22-33
e-mail: chaconne@chaccone.ru

Саратов, «Читающий Саратов»
пр. Революции, д. 58
тел.: (4732) 51-28-93, 47-00-81
e-mail: manager@kmsvrn.ru

СЕВЕРНЫЙ КАВКАЗ

Ессентуки, «Россы», ул. Октябрьская, 424
тел./факс: (87934) 6-93-09
e-mail: rossy@kmw.ru

СИБИРЬ

Иркутск, «ПродаЛитЪ»
тел.: (3952) 20-09-17, 24-17-77
e-mail: prodalit@irk.ru
http://www.prodalit.irk.ru

Иркутск, «Светлана»
тел./факс: (3952) 25-25-90
e-mail: kkcbooks@bk.ru
http://www.kkcbooks.ru

Красноярск, «Книжный мир»
пр. Мира, д. 86
тел./факс: (3912) 27-39-71
e-mail: book-world@public.krasnet.ru

Новосибирск, «Топ-книга»
тел.: (383) 336-10-26
факс: (383) 336-10-27
e-mail: office@top-kniga.ru
http://www.top-kniga.ru

ТАТАРСТАН

Казань, «Таис»,
сеть магазинов «Дом книги»
тел.: (843) 272-34-55
e-mail: tais@bancor.ru

УРАЛ

Екатеринбург, ООО «Дом книги»
ул. Антона Валека, д. 12
тел./факс: (343) 358-18-98, 358-14-84
e-mail: domknigi@k66.ru

Екатеринбург, ТЦ «Люмна»
ул. Студенческая, д. 1в
тел./факс: (343) 228-10-70
e-mail: igm@lumna.ru
http://www.lumna.ru

Челябинск, ООО «ИнтерСервис ЛТД»
ул. Артиллерийская, д. 124
тел.: (351) 247-74-03, 247-74-09,
247-74-16
e-mail: zakup@intser.ru
http://www.fkniga.ru, www.intser.ru

ВАМ НРАВЯТСЯ НАШИ КНИГИ? ЗАРАБАТЫВАЙТЕ ВМЕСТЕ С НАМИ!

У Вас есть свой сайт?

Вы ведете блог?

Регулярно общаетесь на форумах? Интересуетесь литературой, любите рекомендовать хорошие книги и хотели бы стать нашим партнером?

ЭТО ВПОЛНЕ РЕАЛЬНО!

СТАНЬТЕ УЧАСТНИКОМ ПАРТНЕРСКОЙ ПРОГРАММЫ ИЗДАТЕЛЬСТВА «ПИТЕР»!



*Зарегистрируйтесь на нашем сайте в качестве партнера по адресу **www.piter.com/ePartners***



Получите свой персональный уникальный номер партнера



*Выбирайте книги на сайте **www.piter.com**, размещайте информацию о них на своем сайте, в блоге или на форуме и добавляйте в текст ссылки на эти книги (на сайт **www.piter.com**)*

ВНИМАНИЕ! В каждую ссылку необходимо добавить свой персональный уникальный номер партнера.

С этого момента получайте 10% от стоимости каждой покупки, которую совершит клиент, придя в интернет-магазин «Питер» по ссылке с Вашим партнерским номером. А если покупатель приобрел не только эту книгу, но и другие издания, Вы получаете дополнительно по 5% от стоимости каждой книги.

Деньги с виртуального счета Вы можете потратить на покупку книг в интернет-магазине издательства «Питер», а также, если сумма будет больше 500 рублей, перевести их на кошелек в системе Яндекс.Деньги или Web.Money.

Пример партнерской ссылки:

<http://www.piter.com/book.phtml?978538800282> – обычная ссылка

<http://www.piter.com/book.phtml?978538800282&refer=0000> – партнерская ссылка, где 0000 – это ваш уникальный партнерский номер

**Подробнее о Партнерской программе
ИД «Питер» читайте на сайте
WWW.PITER.COM**







КНИГА-ПОЧТОЙ



ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **postbook@piter.com**
- по телефону: **(812) 703-73-74**
- по почте: **197198, Санкт-Петербург, а/я 127, ООО «Питер Мейл»**
- по ICQ: **413763617**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

-  Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
-  С помощью банковской карты. Во время заказа Вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
-  Электронными деньгами. Мы принимаем к оплате все виды электронных денег: от традиционных Яндекс.Деньги и Web-money до USD E-Gold, MoneyMail, INOCard, RBK Money (RuPay), USD Bets, Mobile Wallet и др.
-  В любом банке, распечатав квитанцию, которая формируется автоматически после совершения Вами заказа.

Все посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку Ваших покупок максимально быстро. Дату отправления Вашей покупки и предполагаемую дату доставки Вам сообщат по e-mail.

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, факс, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.