

Ю. Избачков, В. Петров

ИНФОРМАЦИОННЫЕ СИСТЕМЫ

2-е издание

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



300.piter.com

Издательская программа

**300 лучших учебников для высшей школы
в честь 300-летия Санкт-Петербурга**

осуществляется при поддержке Министерства образования РФ

 **ПИТЕР®**

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

2006

ББК 32.233.02я7
УДК 004.65(075)
ПЗО

Рецензенты:

Волков Ю. И., зав. кафедрой UNESCO Московского государственного института электронной техники, к. т. н.

Чечурин С. Л., ректор Санкт-Петербургского института ЮНЕСКО по информационным технологиям в образовании при СПбГТУ, д. т. н.

Избачков Ю. С., Петров В. Н.

ПЗО Информационные системы: Учебник для вузов. 2-е изд. — СПб.: Питер, 2006. — 656 с.: ил.

ISBN 5-469-00641-7

Основное внимание в книге уделяется вопросам разработки клиентской части информационных систем с использованием приложений Delphi. В то же время в ней содержится большое количество практического материала, посвященного вопросам проектирования и создания баз данных, в частности рассматривается методология проектирования информационных систем, приводится подробное описание стандарта ANSI SQL-92, излагаются теоретические сведения о реляционной модели данных. Одна из частей книги полностью посвящена современным технологиям программирования — COM, ActiveX и Интернет-технологиям.

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

ББК 32.233.02я7
УДК 004.65(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.

Краткое содержание

Введение	13

Часть I. Анализ и проектирование информационных систем

Глава 1. Информационные системы	22
Глава 2. Жизненный цикл информационных систем	39
Глава 3. Методология и технология разработки информационных систем	60
Глава 4. Реляционные базы данных	116
Глава 5. Управление реляционными базами данных	147
Глава 6. Проектирование структуры базы данных	176

Часть II. Delphi — система быстрой разработки приложений

Глава 7. Object Pascal и объектно-ориентированное программирование	200
Глава 8. Средства быстрой разработки приложений	240
Глава 9. Компоненты для ввода и редактирования данных	275
Глава 10. Создание форм для ввода и редактирования данных	316

Часть III. Выборка данных и отображение ее результатов

Глава 11. Выборка данных	342
Глава 12. Создание отчетов	381

Часть IV. Компоновка приложения и управление проектом

Глава 13. Система меню и панель инструментов приложения	392
Глава 14. Управление проектом и создание приложения	406
Глава 15. Коллективная разработка приложений	420
Глава 16. Справочная система приложения	444

Часть V. Технология COM

Глава 17. Доступ к данным из приложений Microsoft Office	486
Глава 18. Создание COM-объектов и элементов ActiveX	529

Часть VI. Программирование для Интернета

Глава 19. Особенности Интернет-приложений	574
Глава 20. Разработка Интернет-приложений	605

Заключение	636
Рекомендуемая литература	637
Алфавитный указатель	639

Содержание

Введение	13
Информационные системы	13
База данных	14
CASE-средства	15
Средства разработки	15
Для кого предназначена эта книга	16
Как составлена книга	17
Часть I. Анализ и проектирование информационных систем	17
Часть II. Delphi — система быстрой разработки приложений	18
Часть III. Выборка данных и отображение ее результатов	19
Часть IV. Компоновка приложения и управление проектом	19
Часть V. Технология COM	20
Часть VI. Программирование для Интернета	20
От издательства	20

Часть I. Анализ и проектирование информационных систем

Глава 1. Информационные системы	22
Основные понятия	22
Факторы, влияющие на развитие корпоративных информационных систем	22
Основные составляющие корпоративных информационных систем	24
Соотношение между составляющими информационной системы	24
Классификация информационных систем	25
Области применения и примеры реализации информационных систем	33
Бухгалтерский учет	34
Управление финансовыми потоками	34
Управление складом, ассортиментом, закупками	34
Управление производственным процессом	34
Управление маркетингом	35
Документооборот	35
Оперативное управление предприятием	35
Предоставление информации о фирме	35

Требования, предъявляемые к информационным системам	36
Гибкость	36
Надежность	36
Эффективность	36
Безопасность	37

Глава 2. Жизненный цикл информационных систем 39

Общие сведения об управлении проектами	40
Понятие проекта	40
Классификация проектов	42
Основные фазы проектирования информационной системы	42
Процессы, протекающие на протяжении жизненного цикла информационной системы	45
Основные процессы жизненного цикла	46
Вспомогательные процессы жизненного цикла	47
Организационные процессы	47
Структура жизненного цикла информационной системы	48
Начальная стадия	49
Стадия уточнения	49
Стадия конструирования	49
Стадия передачи в эксплуатацию	49
Модели жизненного цикла информационной системы	50
Каскадная модель жизненного цикла информационной системы	50
Спиральная модель жизненного цикла	56

Глава 3. Методология и технология разработки информационных систем 60

Методология RAD	62
Основные особенности методологии RAD	62
Объектно-ориентированный подход	63
Визуальное программирование	65
Событийное программирование	66
Фазы жизненного цикла в рамках методологии RAD	66
Ограничения методологии RAD	69
Профили открытых информационных систем	69
Понятие профиля информационной системы	70
Принципы формирования профиля информационной системы	71
Структура профилей информационных систем	73
Стандарты и методики	77
Виды стандартов	77
Методика CDM фирмы Oracle	78
Международный стандарт ISO/IEC 12207: 1995-08-01	82
Универсальный язык моделирования	86

Глава 4. Реляционные базы данных 116

Общие сведения о базах данных	117
Основные функции систем управления базами данных	117
Эволюция систем управления базами данных	122

Реляционная модель данных	126
Базовые понятия реляционной модели данных	127
Связанные отношения	133
Основные свойства отношений	137
Реляционная система управления базами данных	137
Нормализация данных	140
Цели нормализации	141
Нормальные формы	142

Глава 5. Управление реляционными базами данных ... 147

Краткая история языка SQL	147
Типы команд SQL	148
Типы данных SQL/92	149
Строковые типы	149
Числовые типы	150
Типы для представления даты и времени	151
Управление объектами базы данных	152
Создание, модификация и удаление таблиц	152
Задание ограничений	154
Значения по умолчанию	161
Индексы	161
Представления	163
Хранимые процедуры	166
Триггеры	168
Манипулирование данными	169
Добавление в таблицу новой информации	169
Изменение данных, хранящихся в таблице	171
Удаление данных из таблицы	172
Безопасность базы данных	173
Привилегии пользователей	173
Управление доступом к базе данных	174

Глава 6. Проектирование структуры базы данных 176

Концептуальное моделирование структуры данных	176
Концептуальные модели данных	177
Модель сущность-связь	178
Общие сведения о CASE-средствах	179
Создание концептуальной модели информационной системы	182
База данных Премьер	182
Создание нового проекта в Power Designer	184
Создание сущностей	186
Создание доменов	186
Определение атрибутов сущностей	187
Определение связей между сущностями	189
Проверка модели	191
Документирование модели базы данных	192
Создание физической модели	193
Создание структуры базы данных	195
Модификация структуры базы данных	196

Часть II. Delphi — система быстрой разработки приложений

Глава 7. Object Pascal и объектно-ориентированное программирование 200

Основы языка Object Pascal	201
Структура программы в Object Pascal	201
Заголовок программы	202
Раздел объявления модулей	202
Раздел объявления меток	202
Раздел описания типов	203
Раздел переменных	203
Раздел констант	204
Типы данных в Object Pascal	204
Простые типы	205
Структурные типы	208
Указательные типы	213
Вариантные типы	214
Операторы языка Object Pascal	216
Оператор присваивания	216
Оператор безусловного перехода	217
Условный оператор	217
Операторы цикла	218
Составной оператор	220
Процедуры и функции	220
Процедуры	221
Функции	223
Модули Object Pascal	224
Основы объектно-ориентированного программирования	226
Основные понятия и отличительные черты ООП	226
Основные концепции ООП	228
Поля, свойства и методы	231
Вложенные типы данных	236
Области видимости	237
Обработка исключительных ситуаций	238

Глава 8. Средства быстрой разработки приложений ... 240

Средства визуального программирования	241
Платформа Microsoft .NET	241
Технологии COM и .NET	243
Прогноз внедрения Microsoft .NET	243
Структура .NET	244
Среда разработки Delphi	246
Главное окно	248
Главное меню	248
Панели инструментов	257
Палитра компонентов	258
Инспектор объектов	260
Редактор форм	261

Основные компоненты Delphi и построение простых приложений	261
Библиотеки компонентов	261
Основные компоненты для построения простых приложений	264

Глава 9. Компоненты для ввода и редактирования данных 275

Стандартные компоненты Delphi для ввода и редактирования данных	275
Многострочные текстовые поля	276
Списки	276
Комбинированные списки	277
Изображения	277
Стандартные окна диалога Delphi	278
Окна диалога для работы с файлами	278
Окно диалога для установки и настройки шрифтов	281
Окно диалога для выбора цвета	282
Окна диалога для работы с принтером	282
Работа с базами данных в Delphi	283
Доступ к данным с использованием BDE	283
Работа с полями	289
Подключение базы данных к BDE	292
Компоненты Delphi для отображения и редактирования данных	294
Класс TDataSource	295
Модули данных	296
Класс TDBGrid	297
Компоненты для доступа к отдельным полям	298
Навигация по набору данных	300
Создание новых компонентов	301
Модификация существующих классов	302
Создание нового класса	302
Последовательность создания компонента	303
Особенности создания компонентов для управления данными	311
Основные свойства и методы классов для связи с данными	311

Глава 10. Создание форм для ввода и редактирования данных 316

Формы в Delphi	316
Свойства класса TForm	317
Фреймы	321
Использование базовых классов для создания форм ввода	321
Размещение и удаление элементов управления	321
Выравнивание компонентов на форме	322
Изменение размеров и перемещение компонентов	324
Порядок обхода элементов	325
Настройка внешнего вида формы	325
Простые формы для ввода данных	325
Пример создания простой формы	326
Табличные формы	329
Формы с вкладками	335
Многотабличные базы данных	338

Часть III. Выборка данных и отображение ее результатов

Глава 11. Выборка данных 342

Выборка данных из таблиц с помощью SQL-запросов	342
Компоненты Delphi, работающие с базами данных через SQL-запросы	343
Компонент TQuery	343
Пример использования компонентов, работающих с SQL-запросами	344
Язык запросов к данным	348
Простейшая форма оператора SELECT	348
Задание условий при выборке данных	350
Упорядочение данных	358
Вычисляемые поля	359
Псевдонимы полей	361
Функции агрегирования	362
Группировка данных	363
Выборка данных из нескольких таблиц	367
Подзапросы	371
Объединение запросов	372
Оператор UNION	373
Оператор UNION ALL	373
Упорядочение и группировка данных в составных запросах	374
Представления	374
Создание представлений	375
Удаление представлений	377
SQL-запросы с параметрами	377

Глава 12. Создание отчетов 381

Рекомендации по созданию отчетов	381
Типы отчетов	382
Генератор отчетов Rave Reports	382
Структура генератора отчетов Rave Reports	383
Пример создания отчета	384
Использование отчетов в Delphi	389

Часть IV. Компоновка приложения и управление проектом

Глава 13. Система меню и панель инструментов приложения 392

Планирование приложения	392
Создание главного меню	394
Класс TMenuItem	395
Работа с редактором меню	396
Задание реакции на выбор команды меню	398
Создание контекстного меню	400

Панель инструментов	400
Класс TToolBar	401
Класс TToolButton	401
Обработка щелчка на кнопке	403
Контейнеры для панелей инструментов	404

Глава 14. Управление проектом и создание приложения 406

Структура проекта	406
Модуль формы проекта	407
Главный файл проекта	408
Файл описания формы проекта	409
Добавление в проект форм и модулей	410
Класс TApplication	411
Управление формами проекта	412
Работа с группой проектов	414
Создание группы проектов	414
Управление группой проектов	414
Окно настройки параметров проекта	415
Вкладка Application	416
Вкладка Compiler	416
Вкладка Linker	418
Компиляция и запуск приложения	418
Команды компиляции проекта	419
Команды запуска приложения	419

Глава 15. Коллективная разработка приложений 420

Системы контроля версий	420
Идентификация	421
Хранение файлов и контроль за их изменением	421
Блокировки	421
Последовательность работы	422
Программа TeamSource	422
Структура программы TeamSource	423
Идентификация проекта и его составляющих в TeamSource	423
Хранилище TeamSource	424
Работа с программой TeamSource	424

Глава 16. Справочная система приложения 444

Основные компоненты справочной системы	444
Создание всплывающих подсказок	445
Создание строки состояния приложения	447
Создание файла справки в формате WinHelp 4	448
Основные элементы справочной системы WinHelp 4	448
Создание файла справки	451
Создание файла справки в формате HTML Help	465
Основные элементы справочной системы HTML Help	466
Создание файла справки в формате HTML	467
Компиляция и тестирование файла справки	477

Использование справочной системы в приложениях	477
Подключение к приложению справочных файлов формата WinHelp	478
Подключение к приложению справочных файлов формата HTML Help	481

Часть V. Технология COM

Глава 17. Доступ к данным из приложений

Microsoft Office	486
Основные понятия технологии автоматизации	486
Структура пакета Microsoft Office	487
Методы взаимодействия с сервером автоматизации	489
Доступ к объекту автоматизации путем позднего связывания	489
Доступ к объекту автоматизации путем раннего связывания	491
Компоненты Delphi для взаимодействия с серверами автоматизации MS Office	494
Взаимодействие с приложениями MS Office	496
Взаимодействие с MS Word	496
Взаимодействие с MS Excel	511
Взаимодействие с MS PowerPoint	523

Глава 18. Создание COM-объектов и элементов

ActiveX	529
Основы технологии COM	529
COM-серверы	530
COM-клиенты	530
Идентификация COM-объектов	531
Интерфейс IUnknown	531
Библиотека COM	534
Фабрика классов	534
Создание COM-объектов в Delphi	535
Создание внутреннего COM-сервера	536
Разработка клиентского приложения для внутреннего сервера	549
Основы автоматизации	553
Сервер автоматизации	553
Контроллер автоматизации	554
Библиотеки типов	554
Интерфейс IDispatch	554
Создание серверов автоматизации в Delphi	554
Разработка клиента автоматизации	562
Элементы ActiveX	562
ActiveX и компонентное программирование	563
Использование и создание элементов ActiveX в Delphi	564

Часть VI. Программирование для Интернета

Глава 19. Особенности Интернет-приложений

574

Основные сведения об Интернете	574
Многоуровневая сетевая модель	575
Уровень сетевого доступа	576
Межсетевой уровень	576

Транспортный уровень	577
Уровень приложений	578
Адресация в Интернете	578
Доменная система имен	578
Порты и службы	579
Унифицированный указатель ресурсов	579
Основы веб-программирования	580
Основные понятия и термины	580
Веб-дизайн и веб-программирование	581
Протокол HTTP	582
Запрос клиента	583
Ответ сервера	584
Язык HTML	586
Структура HTML-документа	587
Теги форматирования текста	588
Гиперссылки	591
Формы	592
Поля ввода	592
Флажки	593
Переключатели	593
Кнопки	593
Язык XML	595
Типы веб-приложений	596
CGI-сценарии	596
ISAPI-расширения	597
ASP-страницы	598
Доступ к базам данных через Интернет	603
Глава 20. Разработка Интернет-приложений	605
Разработка CGI-сценариев	605
Запуск CGI-приложения	606
Разработка простейшего CGI-приложения	606
Строка передаваемых параметров	609
Методы передачи и получения строки параметров	609
Разработка веб-приложений специальными средствами Delphi	615
Компонент TWebModule	615
Компоненты для формирования ответа в формате HTML	621
Компоненты Indy	632
Заключение	636
Рекомендуемая литература	637
Алфавитный указатель	639

Введение

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

- ❑ 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 — гражданские области применения компьютеров гораздо более распространены, чем военные и научные, 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 весьма значительно 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 ориентированы на конечного пользователя, не обладающего высокой квалификацией в области вычислительной техники. Поэтому клиентские приложения информационной системы должны обладать простым, удобным, легко осваиваемым интерфейсом, который предоставляет конечному пользователю все необходимые для работы функции и в то же время не дает ему возможность выполнять какие-либо лишние действия.

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

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

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

База данных

Система управления базой данных (СУБД) является неотъемлемой частью любой информационной системы. Тип используемой СУБД обычно определяется масштабом информационной системы — малые информационные 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, которые хорошо отлажены, развиты и к тому же поддерживают 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 снизить загрузку сети, применяются трехзвенные архитектуры клиент-сервер. В этой архитектуре, помимо клиентской части системы и сервера базы данных, вводится промежуточный сервер приложений. На стороне клиента выполняются только интерфейсные действия, а вся логика обработки информации поддерживается в сервере приложений.

При разработке базы данных необходимо учитывать специфику той СУБД, для которой эта разработка проводится. Несмотря на существование стандарта ANSI SQL 92, 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 формальное описание 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. Аббревиатура CASE (Computer Aided Software/System Engineering) используется в настоящее время в весьма широком смысле. Первоначальное ее значение, ограниченное вопросами автоматизации разработки только лишь программного обеспечения, 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 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 ориентированы исключительно на создание приложений для вполне определенной СУБД и не предназначены для разра-

ботки обычных приложений, не использующих базы данных. Примером средств такого рода может служить система Power Builder фирмы Sybase.

- ❑ Универсальные средства 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 и Visual Basic 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.

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

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

С выходом платформы Microsoft .NET достоинства и недостатки языков программирования стали сглаживаться, появилась возможность межязыковой интеграции. Создавать программное обеспечение для .NET можно с помощью восьмой версии Delphi.

Для кого предназначена эта книга

Книга в первую очередь предназначена для начинающих программистов, не имеющих большого опыта разработки информационных систем. Основное 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 SQL-92 ANSI, 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 двадцать глав, которые сгруппированы в шесть 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 приводятся основная терминология и базовые понятия, знание которых необходимо для эффективного восприятия материала 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 двухмерных 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 сокращения времени разработки используются специальные программные средства проектирования баз данных, 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. Кроме того, рассматриваются наиболее типичные области применения информационных систем.
- ❑ **Глава 2. «Жизненный цикл информационных систем».** Как ясно из названия, здесь рассматриваются понятие жизненного цикла информационной системы и основные процессы, его сопровождающие. Также рассматриваются основные модели жизненного цикла информационных систем.
- ❑ **Глава 3. «Методология и технология разработки информационных систем».** В этой главе приводятся сведения о методологии быстрой разработки приложений (Rapid Application Development, RAD), 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 развития этих систем. 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 дан-

ных, так и структурой самой базы данных. Рассматриваются средства языка управления базами данных SQL, предусмотренные стандартом SQL 92 института ANSI.

- ❑ **Глава 6. «Проектирование структуры базы данных».** В данной главе рассматриваются понятия концептуальной и физической моделей 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, объектно-ориентированное программирование оказалось востребованным сравнительно недавно — в 90-х годах. Несколько позже стали выходить средства визуальной разработки приложений, позволяющие быстро разрабатывать 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. «Object Pascal 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 Delphi фирмы Borland. В языке Object Pascal в полной мере реализованы все принципы объектно-ориентированного программирования. Строгость и ясность этого языка делают его идеальным для изучения концепций объектно-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. «Средства быстрой разработки приложений».** Данная глава содержит начальные сведения о платформе Microsoft .NET и системе проектирования Delphi, а также подробное описание интегрированной среды системы визуальной разработки приложений Delphi фирмы Borland. 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 предыдущей. Однако если в главе 9

рассматривались отдельные компоненты для ввода и редактирования данных, то здесь обсуждаются различные варианты компоновки компонентов для ввода и редактирования данных на формах, обеспечивающие наиболее эффективный и наглядный доступ к информации, хранящейся в базе данных.

Часть III. Выборка данных и отображение ее результатов

Помимо редактирования данных, хранящихся в базе данных, важной функцией клиентских приложений является их выборка по какому-либо критерию. Причем только выборкой проблема не исчерпывается — данные, полученные в результате выборки, 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 разного рода выборок данных из таблиц базы данных. Также здесь рассматриваются компоненты библиотеки Borland Delphi, предназначенные для организации взаимодействия с базой данных с помощью операторов языка SQL.
- ❑ **Глава 12. «Создание отчетов».** В этой главе рассматриваются вопросы создания отчетов — форматированного представления данных, выводимого на экран, принтер или в файл. В поставку Borland Delphi входят специальные компоненты, предназначенные для создания отчетов. Подробному их рассмотрению и посвящена данная глава.

Часть IV. Компоновка приложения и управление проектом

В предыдущих частях книги, посвященных разработке приложений, затрагивались лишь вопросы создания 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 13. «Система меню и панель инструментов приложения».** В этой главе рассматриваются вопросы создания основных элементов интерфейса пользователя приложения — меню и панели инструментов.
- ❑ **Глава 14. «Управление проектом и создание приложения».** Здесь рассматриваются структура проекта в Borland Delphi, основные свойства проекта, способы компиляции и управления приложением.
- ❑ **Глава 15. «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 16. «Справочная система приложения».** В данной главе излагаются вопросы создания справочной системы приложения и ее взаимодействия с приложением, а также вопросы создания контекстной справочной системы. Здесь вы познакомитесь с методами создания файлов справки как в формате WinHelp, так и в формате HTML Help.

Часть V. Технология COM

Технология COM и основанная на ней технология ActiveX, являвшаяся основной технологией взаимодействия приложений до появления платформы .NET, широко применяются и будут применяться в приложениях, функционирующих под управлением операционной системы Windows. Данные технологии позволяют легко обеспечить взаимодействие между различными приложениями, дают возможность многократного использования кода при разработке собственных приложений, упрощают модификацию приложений. Разработчики платформы .NET отказались от многих решений, применяемых в технологии COM, однако ввиду широкого распространения последней мы приводим ее основы.

- ❑ **Глава 17. «Доступ к данным из приложений Microsoft Office».** Из этой главы вы узнаете, как организовать взаимодействие программы, разработанной с помощью Borland Delphi, с различными приложениями, входящими в состав Microsoft Office.
- ❑ **Глава 18. «Создание COM-объектов и элементов ActiveX».** В этой главе рассматриваются вопросы создания собственных COM-объектов и элементов и ActiveX.

Часть VI. Программирование для Интернета

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.

- ❑ **Глава 19. «Особенности Интернет-приложений».** В этой главе рассматриваются базовые технические особенности организации Интернета, а также основные понятия и термины веб-программирования. Излагаются основы протокола HTTP и языка разметки гипертекста (HTML).
- ❑ **Глава 20. «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!

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, относящиеся к операционным системам, определяются их типы и базовые свойства, формулируются задачи, решаемые при разработке таких систем, и возникающие при этом проблемы.

Основные понятия

Под информационной системой обычно понимается прикладная программная подсистема, 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, реализуют:

- ☐ 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 информационных систем.

Рассмотрим эти факторы более подробно.

Развитие методик управления предприятием

Теория управления предприятием представляет собой довольно обширный предмет для изучения и совершенствования. Это обусловлено постоянным изменением и разнообразием ситуаций 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 совершенной 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 программно-технологической реализации информационных систем. 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 Интернета* расширило возможности работы с удаленными подразделениями, открыло широкие перспективы электронной коммерции, обслуживания покупателей через Интернет и многое другое. Более того, определенные преимущества дает использование Интернет-технологий во внутренних сетях предприятий (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, 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 целиком относится к прикладной области и в значительной степени 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, эти две составляющие в определенном смысле независимы. Например, организация сети и протоколы, используемые для обмена дан-

ными между компьютерами, абсолютно не зависят от того, какие методы и программы планируется применять 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. Хотя, 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, ее функциональности, распределения функций, принятых в организации финансовых технологий и схем, существующей технологии документооборота и множества других факторов.

Разрабатывать и внедрять функциональные подсистемы можно 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.

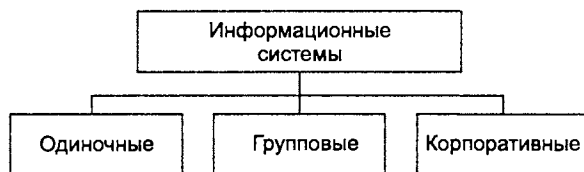


Рис. 1.1. Деление информационных систем по масштабу

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 архитектура. 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 поддержки принятия решений;
- ☐ информационно-справочные системы;
- ☐ офисные информационные системы.

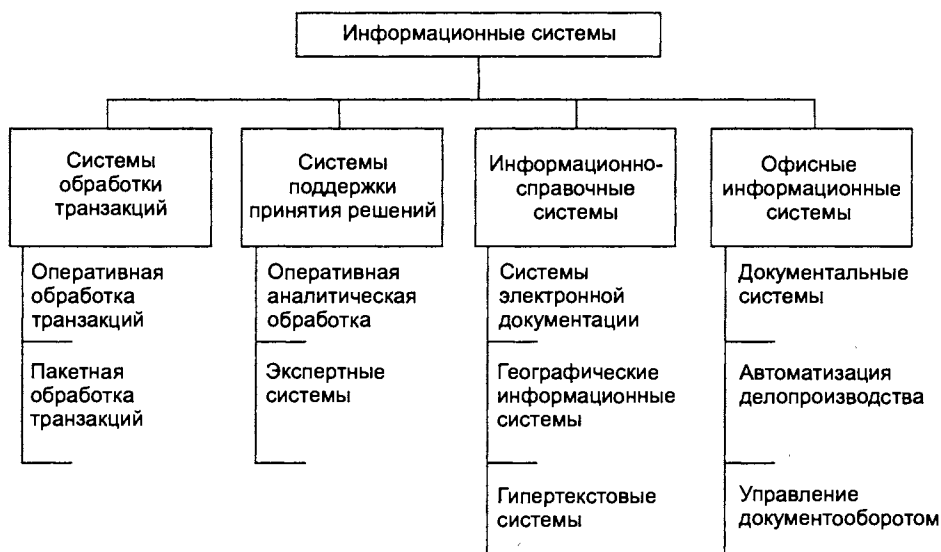


Рис. 1.2. Деление информационных систем по сфере применения

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 (OnLine Transaction Processing, OLTP) для отражения актуального состояния предметной области в любой момент времени, а пакетная обработка занимает весьма ограниченную часть. Для систем OLTP характерен регулярный (возможно, интенсивный) поток довольно простых транзакций, играющих роль заказов, платежей, запросов 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 (Decision Support System, DSS) представляют собой другой тип информационных систем, в которых с помощью довольно сложных запросов производится отбор и анализ данных в различных разрезах: временных, географических, по другим показателям.

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

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.

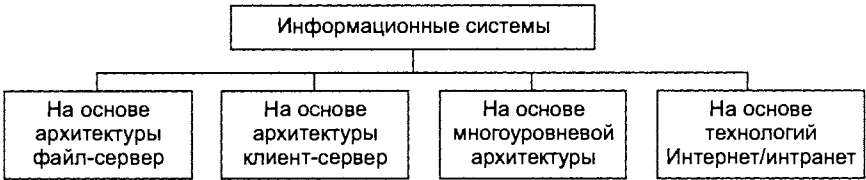


Рис. 1.3. Деление информационных систем по способу организации

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)	Обслуживает пользовательский ввод и отображает то, что сообщает ему компонент логики представления (PL), с использованием соответствующей программной поддержки

Обозначение	Наименование	Характеристика
PL	Presentation Logic (логика представления)	Управляет взаимодействием между пользователем и ЭВМ. Обработывает действия пользователя при выборе команды в меню, щелчке на кнопке или выборе пункта в списке
BL	Business 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 (реализуемые SQL-операторами), которые нужно выполнить для реализации прикладной логики управления данными
DS	Data Services (операции с базой данных)	Действия СУБД, реализующие логику управления данными, такие как манипулирование данными, определение 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

В *архитектуре файл-сервер* сетевое разделение компонентов диалога **PS** и **PL** отсутствует, а компьютер используется для функций отображения, что облегчает построение графического интерфейса. Файл-сервер только извлекает данные из файлов, так что дополнительные пользователи и приложения лишь незначительно увеличивают нагрузку на центральный процессор. Каждый новый клиент добавляет вычислительную мощность к сети.

Объектами разработки в файл-серверном приложении 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, загружая сеть и приводя к непредсказуемости времени реакции. Значительный сетевой трафик особенно 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. Тем не менее эти инструменты привлекают своей простотой, удобством применения 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 наличие выделенных серверов баз данных, понимающих запросы на языке структурированных запросов (Structured Query Language, SQL) и выполняющих поиск, сортировку и агрегирование информации.

Отличительная черта серверов БД — наличие справочника данных, в котором записаны 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 реализовать графический интерфейс. Компоненты управления данными **DS** и **FS** размещаются на сервере, а диалог (**PS, PL**) и логика (**BL, DL**) — на клиенте. В двухуровневом определении архитектуры клиент-сервер используется именно этот вариант: приложение работает на клиенте, СУБД — на сервере (рис. 1.4).

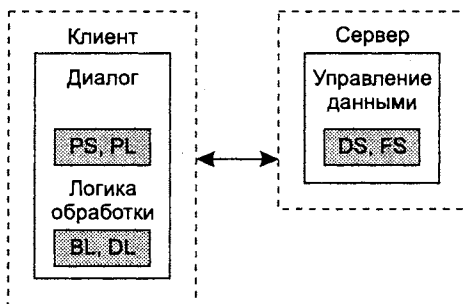


Рис. 1.4. Классический вариант клиент-серверной системы

Поскольку эта схема предъявляет наименьшие требования к серверу, она обладает наилучшей масштабируемостью. Однако сложные приложения, активно взаимодействующие с БД, могут жестко загрузить как клиент, так и сеть. Результаты SQL-запроса должны вернуться клиенту для обработки, потому что там реализована 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 **VL** можно разместить на сервере. При этом вся логика принятия решений оформляется в виде *хранимых процедур* и выполняется на сервере БД.

Хранимая процедура — процедура с SQL-операторами 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 коллективных операций и вычислений. Улучшается сопровождение таких процедур, а также безопасность (нет прямого доступа к данным).

ПРИМЕЧАНИЕ

Следует помнить, что перегрузка 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 разделении логики 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 применение 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 представляет собой сервер приложений, на котором выполняется прикладная логика **BL** и с которого логика обработки данных **DL** выполняет операции с базой данных **DS**;
- ❑ верхний уровень представляет собой удаленный специализированный сервер базы данных, выделенный для услуг обработки данных **DS** и файловых операций **FS** (без 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 реализовывать 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 реализовываться 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 систем клиент-сервер необходимость трех уровней становится все более очевидной. Продукты для трехуровневой 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 NT с вызовом 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 уровней. 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 Интернет/интранет-технологий 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-технологий 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

Бухгалтерский учет — классическая и наиболее часто реализуемая на сегодняшний день область применения информационных технологий. Такое положение вполне объяснимо. Во-первых, ошибка бухгалтера может стоить очень дорого, поэтому очевидна выгода автоматизации бухгалтерии. Во-вторых, задача бухгалтерского учета довольно легко формализуется, так что разработка систем автоматизации бухгалтерского учета не представляет технической 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, 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 двадцать процентов ассортимента, которые приносят восемьдесят процентов 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 просмотреть перспективный товар, вовремя не вложив в него деньги.

Управление производственным процессом

Оптимальное управление производственным процессом представляет собой очень трудоемкую задачу. Основным механизмом здесь является планирование.

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. Решения большинства этих задач могут быть формализованы и представлены в виде информационной системы, позволяющей существенно повысить эффективность маркетинга.

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 по автоматизации бизнес-процессов, имеющих место на конкретном предприятии.

Предоставление информации о фирме

Активное развитие Интернета 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 подразумевают 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 сохранится возможность 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 в чужих решениях, не всегда на первый взгляд логичных. Поэтому рекомендуется фазу сопровождения системы доверять 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 подбору разработчиков информационных систем.

Надежность

Надежность информационной системы подразумевает ее функционирование без искажения информации, потери данных по «техническим причинам». Требование надежности обеспечивается созданием резервных копий хранимой информации, выполнения операций протоколирования, 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 является 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 проектированию 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 потребности заказчика. Часто встречается ситуация, когда заказчик чего-то хочет, но сам не знает чего именно. Чем раньше будут учтены дополнения заказчика, тем с меньшими затратами и в более короткие сроки система будет создана.

Кроме того, заказчик, не являясь специалистом в области разработки информационных систем, может не знать о новых информационных технологиях. Контакты с заказчиком во время разработки для него информационной системы могут подтолкнуть заказчика к модернизации его аппаратных средств, применению новых методов ведения бизнеса, что отвечает потребностям как заказчика, так и проектировщика. Заказчик получает рост эффективности своего предприятия, проектировщик — 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 и 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

Имеет смысл еще до сдачи информационной системы в эксплуатацию предоставить разработчикам возможность познакомиться в роли конечных пользователей. Встречались случаи, когда такой подход приводил к отказу от использования на рабочем месте оператора манипулятора типа «мышь», что, в свою очередь, приводило к многократному повышению производительности оператора.

Безопасность

Под безопасностью, прежде всего, подразумевается свойство системы, в силу которого посторонние лица не имеют доступа к информационным ресурсам органи-

зации, 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, может причинить ущерб интересам заказчика, прежде всего имущественным.

ВНИМАНИЕ

В этой связи 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 ненадлежащее обеспечение безопасности информационной системы заказчика в худшем случае обернется для исполнителя судебным преследованием, в лучшем — потерей клиента и утратой деловой репутации.

Помимо 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, человеческим фактором, несовместимостью с используемыми программами сторонних разработчиков в случае модификации этих программ 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 информационной системы курс доллара США в одной из процедур разработчики обозначили константой. На момент ввода в эксплуатацию этой системы курс доллара был стабилен, поэтому ошибка никак себя не проявляла, а была выявлена только через некоторое время в период роста курса.

Требование безопасности обеспечивается современными средствами разработки информационных систем, современной аппаратурой, методами защиты информации, применением паролей и протоколированием, постоянным мониторингом 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, оказывают влияние на структуру информационной системы, но в то же время структуры разных предприятий в целом схожи 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 разработки проекта (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 означает завершение проекта, а также с установленными требованиями к срокам, результатам, риску, рамкам расходования средств и ресурсов, 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 осознавать, что речь идет об управлении неким динамическим объектом. Поэтому си-

стема 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», на входе которого располагаются технические требования и условия финансирования, а на выходе — требуемый результат (рис. 2.1). Выполнение работ обеспечивается наличием необходимых ресурсов:

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

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

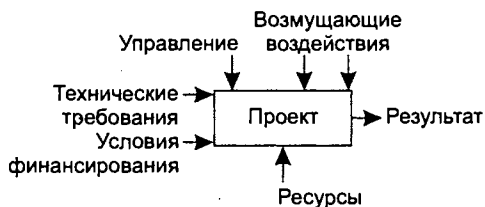


Рис. 2.1. Представление проекта в виде «черного ящика»

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 выполнения 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 значительно отличаться по сфере приложения, составу, предметной области, масштабам, длительности, составу участников, степени сложности, значимости результатов и т. п. Проекты могут быть классифицированы по самым 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 мультипроектного 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. У таких проектов две особенности. Во-первых, главная цель проекта четко определена, но отдельные цели должны уточняться по мере достижения частных результатов. Во-вторых, 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 размером бюджета и количеством участников:
 - мелкие проекты;
 - малые проекты;
 - средние проекты;
 - крупные проекты.

Можно также рассматривать масштабы проектов в более конкретной форме — отраслевые, корпоративные, ведомственные проекты, 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 отличия, поскольку эти характеристики во многом зависят от условий осуществления 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;
- ☐ подготовка технического задания;
- ☐ проектирование;
- ☐ разработка;
- ☐ 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 концептуальной фазе является определение 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, требуемых материальных, финансовых и трудовых ресурсов;
- ☐ сравнительную оценку альтернатив;
- ☐ представление предложений, их экспертизу и утверждение.

Подготовка технического предложения

Главным содержанием фазы подготовки технического предложения является уточнение технического предложения в ходе переговоров с заказчиком о заключении контракта. Общее содержание работ этой фазы:

- ☐ разработка основного содержания, 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 средств контроля за ходом работ.

Проектирование

На фазе проектирования 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, экспертизы и утверждение.

Разработка

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

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

Ввод системы в эксплуатацию

На фазе ввода системы в эксплуатацию проводятся испытания, идет 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 в 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 разработки, 30 % — фаза сдачи объекта и завершения проекта.

Следует иметь в виду, что 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 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 создании информационной системы и заканчивающийся в момент полного изъятия ее из эксплуатации.

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

ПРИМЕЧАНИЕ

ISO расшифровывается как International Organization of Standardization (международная организация по стандартизации), IEC — как International Electrotechnical Commission (международная комиссия по электротехнике).

Стандарт ISO/IEC 12207 определяет структуру жизненного цикла, включая 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 тестирования разработанных программных продуктов;
- ☐ разработку материалов, необходимых для 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 решения поставленных перед ней задач, причем ошибки обслуживающего персонала могут приводить к явным или скрытым финансовым потерям, сопоставимым со стоимостью самой информационной системы.

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

- ❑ выделение наиболее ответственных узлов системы и определение для них критичности простая (это позволит выделить наиболее критичные составляющие информационной системы и оптимизировать распределение ресурсов для технического обслуживания);
- ❑ определение задач технического обслуживания и их разделение на внутреннее, 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 не только решать каждодневные задачи администрирования, но и быстро восстанавливать работоспособность системы при сбоях.

Вспомогательные процессы жизненного цикла

Среди вспомогательных процессов одно из главных мест занимает управление конфигурацией. Это тот вспомогательный процесс, который поддерживает 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 исходным 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 — одной из ведущих фирм на рынке программного обеспечения средств разработки информационных систем (среди которых большой популярностью заслуженно пользуется универсальное CASE-средство Rational Rose).

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.

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, следовательно, достигать определенных ключевых целей.

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

При принятии любых решений, касающихся архитектуры системы, необходимо принимать во внимание 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 факторов риска в проекте.

Стадия конструирования

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

По окончании этой стадии определяется работоспособность разработанного программного обеспечения.

Стадия передачи в эксплуатацию

На стадии передачи в эксплуатацию разработанное программное обеспечение передается 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 передачи в эксплуатацию 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 выполнения действий и решения 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 две основные модели жизненного цикла:

- ☐ каскадная модель, иногда также называемая моделью водопада (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 70-х и первой половине 80-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 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 каскадной модели разбиение работ на стадии и названия этих стадий менялись. Кроме того, наиболее разумные методики и стан-

дарты 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.

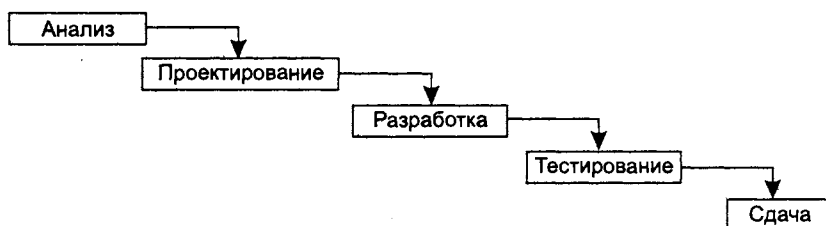


Рис. 2.2. Каскадная модель разработки

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.

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 выполнены 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 длиннее. 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 ее основные достоинства.

- ❑ На каждом этапе формируется законченный набор проектной документации, отвечающей 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 при разработке 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 либо делают ее полностью неприменимой, либо приводят к увеличению сроков разработки и стоимости проекта. В настоящее время многие неудачи программных проектов объясняются именно последовательным процессом 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 в получении результатов;
- ❑ ошибки и недоработки на любом из этапов проявляются, как правило, на последующих этапах работ, что приводит к необходимости возврата назад;
- ❑ сложность параллельного ведения работ по проекту;
- ❑ чрезмерная информационная перенасыщенность каждого из этапов;
- ❑ сложность управления проектом;
- ❑ высокий уровень риска и ненадежность инвестиций.

Задержка в получении результатов обычно считается главным недостатком каскадной схемы. Данный недостаток проявляется в основном в том, что из-за последовательного подхода к разработке согласование результатов с заинтересованными сторонами производится только после завершения очередного этапа работ. Может оказаться, что разрабатываемая информационная система не соответствует требованиям пользователей, причем такие несоответствия могут возникать на любом этапе разработки — искажения могут непреднамеренно вноситься и проектировщиками-аналитиками, и программистами, так как они необязательно 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 целом 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 этапы работы.

Самым же неприятным является то, что недоработки предыдущего этапа могут обнаруживаться не сразу на последующем этапе, а позднее (например, на стадии опытной эксплуатации могут проявиться ошибки в описании предметной области). Это означает, что часть проекта должна быть возвращена на начальный этап работы. Вообще, работа может быть возвращена с любого этапа на любой предыдущий этап, поэтому в реальности каскадная схема разработки выглядит так, как показано на рис. 2.3.

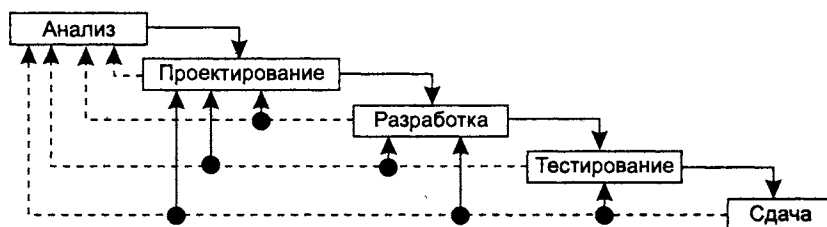


Рис. 2.3. Реальный процесс разработки по каскадной схеме

Одной из причин данной ситуации является то, что в качестве экспертов, участвующих в описании предметной области, часто выступают будущие пользова-

тели системы, которые иногда 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 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 зависят 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 загружены. Кроме того, при последовательной разработке крайне сложно внести изменения в проект после завершения этапа и передачи 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 бы использовать эту часть в своей работе. Когда система состоит из большого количества взаимосвязанных подсистем, то синхронизация внутренней документации становится важной самостоятельной задачей. При этом синхронизация документации на каждую часть системы — 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 требовать 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 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, помимо изучения нового материала, не отпадает необходимость в изучении старой информации. Это связано с тем, что вполне

вероятна ситуация, когда в процессе 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 согласования сроков работы и состава 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 приводит к существенным потерям рабочего времени.

Упростить взаимодействие между группами разработчиков и уменьшить информационную перенасыщенность документации можно, сокращая 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, запоздалая оценка порождает серьезные 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 в предметной области или в требованиях заказчика за время разработки. Причем возврат проекта на разработку вследствие этих причин 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 «зациклится» и система никогда не дойдет до сдачи в эксплуатацию. Расходы на проект будут постоянно расти, а сроки сдачи готового продукта постоянно откладываться.

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

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

ПРИМЕЧАНИЕ

Помимо рассмотренных 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. Конфликт обусловлен тем, что возврат части проекта на предыдущую стадию обычно сопровождается поиском причин и виновных. А так как однозначно персонифицировать ответственного за ошибки можно далеко не всегда, 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 их исполнением. 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. Такое положение вещей может привести к тому, что наиболее одаренные кадры со временем покинут коллектив.

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 на следующем витке 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 текущего — 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.

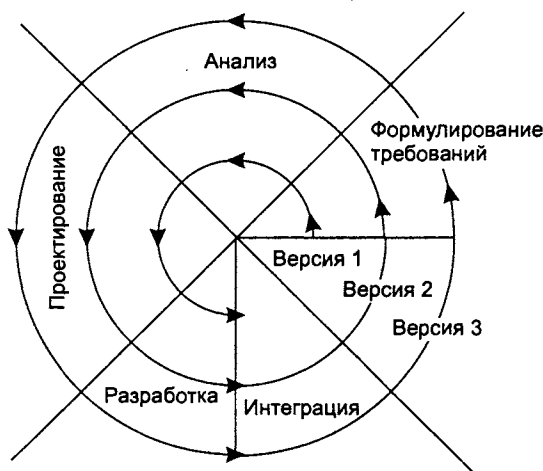


Рис. 2.4. Спиральная модель жизненного цикла информационной системы

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 разработки интеграция занимает до 40 % всех затрат в конце проекта).
- ❑ 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 уровень рисков снижается. Данное утверждение справедливо при любой модели разработки, однако при исполь-

зовании спиральной модели снижение уровня рисков происходит с наибольшей скоростью. Это связано с тем, что при итерационном подходе интеграция выполняется уже на первой итерации и на начальных итерациях выявляются многие аспекты проекта, такие как пригодность используемых инструментальных средств и программного обеспечения, квалификация разработчиков и т. п. На рис. 2.5 приведены в сравнении графики зависимости уровня рисков от времени разработки для каскадной и итерационного подходов.

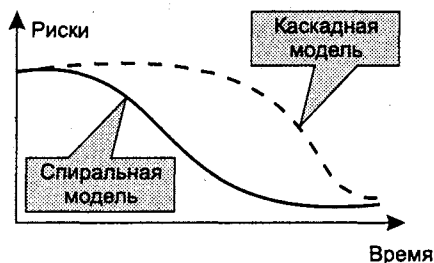


Рис. 2.5. Зависимость рисков от времени разработки

- ❑ Итерационная разработка обеспечивает большую гибкость в управлении проектом, давая возможность внесения тактических изменений в разрабатываемое изделие. Например, можно сократить сроки разработки за счет снижения 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 (реализует 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.
- ❑ 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 в случае каскадной модели доступно только перед внедрением системы.
- ❑ Итерационный подход дает возможность 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.

Недостатки 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 построения информационной системы и в управлении 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:

- ☐ соответствие создаваемой информационной системы целям и задачам предприятия, а также предъявляемым к ней требованиям по автоматизации бизнес-процессов;
- ☐ гарантирование 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;
- ☐ соответствие создаваемой корпоративной информационной системы 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. Методо-

логия 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, информационными и людскими ресурсами:

- ☐ данными, полученными на 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 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 (декомпозиции) крупных проектов на ряд подсистем — составных частей, разрабатываемых группами исполнителей ограниченной численности, с последующей интеграцией этих 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.

- ❑ обеспечивать возможность ведения работ по проектированию отдельных подсистем небольшими группами (3–7 человек), что обусловлено 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 отдельными подсистемами. Реализация же всей системы в сжатые сроки может потребовать привлечения большого числа разработчиков, при этом эффект может оказаться менее значимым, чем при реализации отдельных подсистем в более короткие сроки меньшим числом разработчиков.

- ❑ предусматривать возможность управления конфигурацией проекта, ведения версий проекта и его составляющих, 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

На начальном этапе существования компьютерных информационных систем их разработка велась на традиционных языках программирования. Однако по мере возрастания сложности разрабатываемых систем и запросов пользователей (чему в значительной степени способствовал прогресс в области вычислительной техники, а также появление удобного графического интерфейса пользователя в системном программном обеспечении) требовались 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 создания информационных систем, основанная на использовании средств быстрой разработки приложений, получила в последнее время широкое распространение и приобрела название *методологии быстрой разработки приложений* (Rapid Application Development, RAD). Данная методология охватывает все этапы жизненного цикла современных информационных систем.

Методология RAD — это комплекс специальных инструментальных средств, позволяющих оперировать с определенным набором графических объектов, функ-

ционально 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 следующим:

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

Объектно-ориентированный подход

Средства RAD позволили реализовать совершенно иную по сравнению с традиционной технологию создания приложений: информационные объекты формируются как некие действующие модели (прототипы), чье функционирование согласуется с пользователем, а затем разработчик может переходить 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. Эти принципы позволяют преодолеть одну из главных трудностей, возникающих

при разработке сложных систем, — колоссальный разрыв между реальным миром (предметной областью) и имитирующей средой.

Использование объектно-ориентированных принципов позволяет создать описание (модель) предметной области в виде совокупности объектов — сущностей, объединяющих данные и методы обработки этих данных (процедуры). Каждый объект обладает собственным поведением и моделирует некоторый объект реального мира. С этой точки зрения объект является вполне осязаемым и 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 (ОПП) получило с появлением средств визуального программирования, 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 используются множество готовых объектов, сохраняемых в общедоступном хранилище. Однако имеется также 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, решение которой отнимает 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 бизнес-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 многократного использования в будущем.

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 исключительно на разработку 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.

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 разработчиков.

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 средствами RAD, является событийно-ориентированной. Это означает, что каждый объект, входящий в состав приложения, может генерировать события и реагировать на события, генерируемые другими объектами. Примерами событий могут быть открытие и закрытие окон, щелчок на кнопке, 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 «щелчок на кнопке» 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 выполнении операций 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:

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

Рассмотрим каждую из них более подробно.

Фаза анализа и планирования требований

На фазе анализа и планирования требований определяются:

- ☐ функции, которые должна выполнять разрабатываемая информационная система;
- ☐ наиболее приоритетные функции, требующие разработки в первую очередь;
- ☐ информационные потребности;

ПРИМЕЧАНИЕ

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

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

- ❑ 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, а также предварительные функциональные и информационные модели системы.

Фаза проектирования

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

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, если требуется, корректировка функциональной модели системы. Детально рассматривается каждый процесс системы. При необходимости для каждого элементарного процесса создается частичный прототип: экран, диалоговое окно 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 проектной информации 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, диалоговых окон 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. В противоположность этому при традиционном подходе использовались средства прототипирования, не предназначенные для построения реальных приложений, поэтому разработанные прототипы не годились для последующих фаз и просто «выбрасывались» после того, как решали 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 средствами визуального 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 команды 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 повышения производительности;
- ☐ завершается разработка документации проекта.

Результатом реализации 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 начальных условий, при которых начинается разработка (например, разрабатывается совершенно новая система или на предприятии уже существует некоторая информационная система). Во втором случае существующая система может либо использоваться в качестве прототипа новой системы, либо интегрироваться в новую разработку в качестве одной из подсистем.

Ограничения методологии RAD

Несмотря на все свои достоинства, методология RAD (как, впрочем, и любая другая методология) не может претендовать на универсальность. Ее применение наиболее эффективно при создании сравнительно небольших систем, разрабатываемых для конкретного заказчика.

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 адаптироваться к различным программно-аппаратным платформам, системам управления базами данных, коммуникационным средствам, а также интегрироваться с существующими разработками. Поэтому для такого рода проектов необходимы высокий уровень планирования и жесткая дисциплина проектирования, строгое следование заранее разработанным протоколам и интерфейсам, что снижает скорость разработки.

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

Методология RAD не может быть использована для разработки приложений, в которых интерфейс пользователя является вторичным, то есть отсутствует 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 атомными электростанциями. Это обусловлено тем, что итеративный подход, являющийся одной из основ RAD, предполагает, что первые версии системы не будут полностью работоспособными, 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. *От-*

крытые информационные 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 связаны с применением стандартов на основе методологии функциональной стандартизации информационных технологий.

Понятие профиля информационной системы

При создании и развитии сложных, распределенных, тиражируемых информационных систем требуются 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 профиле выделяются и устанавливаются допустимые возможности и значения параметров каждого базового стандарта и/или нормативного документа, входящего в профиль.

Профиль не должен противоречить входящим в него базовым стандартам и нормативным документам. Применяемые в соответствии с профилем необязательные возможности и значения параметров, выбранные из альтернативных вариантов, должны оставаться в допустимых пределах.

На базе одной совокупности базовых стандартов могут формироваться и утверждаться различные профили для разных проектов информационных систем. Ограничения базовых документов профиля и их согласованность, контролируемая 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, регламентирующих:

- ☐ архитектуру и структуру информационной системы;
- ☐ 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

Профили информационных систем призваны решить следующие задачи:

- ☐ снижение трудоемкости проектов;
- ☐ повышение качества компонентов информационных систем;
- ☐ 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 потребностям 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 поддерживаны и регламентированы только самые простые объекты и рутинные, массовые процессы (телекоммуникации, программирование, документирование программ и данных), а наиболее слож-

ные 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 позволяют 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 утвержденные международные и национальные стандарты. Использование стандартов де-факто и нормативных фирменных документов 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 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 прикладные программные интерфейсы (Application Programming 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 части конкретного профиля, в которых 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 важно обеспечить корректность 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 в максимальной степени 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 должны соответствовать система и ее компоненты.

Стандарты, важные с точки зрения заказчика, должны задаваться в техническом задании на проектирование системы и составлять ее первичный профиль. То, что не задано в техническом задании, первоначально остается на усмотрение разработчика системы, который, руководствуясь требованиями технического задания, 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 приме-

няется как основной инструмент сопровождения системы при эксплуатации, модернизации и развитии.

Формирование и применение профилей конкретных информационных систем реализуется на основе 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 профиля;
- ☐ точное перечисление функций объекта или процесса стандартизации, определяемого профилем;
- ☐ формализованные сценарии применения базовых стандартов и спецификаций, включенных в 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 профиля;
- ☐ информационные ссылки на все исходные документы.

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

- ☐ прикладного программного обеспечения;
- ☐ среды информационной системы;
- ☐ защиты информации в информационной системе;
- ☐ 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 интерфейсов 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 следующим функциональным областям эталонной модели OSE/RM:

- ☐ графического пользовательского интерфейса;
- ☐ реляционных или объектно-ориентированных СУБД (например, стандарта языка SQL-92 и спецификации доступа к разным базам данных);
- ☐ операционных систем с учетом сетевых функций, выполняемых на уровне операционной системы;
- ☐ 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 802.3 u), 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 техническом задании 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, 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 недеklarированных возможностей. Профиль должен также включать указания на методы и средства резервного копирования информации 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 выбранных 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, связанные с:

- ☐ контролем производительности и корректности функционирования системы в целом;
- ☐ конфигурированием прикладного программного обеспечения, тиражированием версий;
- ☐ управлением доступом пользователей к ресурсам системы и конфигурированием ресурсов;
- ☐ перенастройкой приложений в связи с изменениями прикладных функций информационной системы;
- ☐ настройкой пользовательских интерфейсов (экранных форм и отчетов);
- ☐ ведением баз данных системы;
- ☐ восстановлением работоспособности системы после сбоев и аварий.

Дополнительные ресурсы, необходимые для функционирования встроенных инструментальных средств, такие как минимальный и рекомендуемый объемы 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 сопровождения и развития системы и дол-

жны 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. Корпоративные стандарты представляют собой соглашение о единых правилах организации технологии или управления. При этом за основу корпоративных стандартов могут приниматься отраслевые, национальные и даже международные стандарты.

Однако динамика развития информационных технологий приводит к быстрому устареванию существующих стандартов и методик разработки информационных систем. Так, 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.

Полезны в этом отношении 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 разработки информационных систем, включая программное обеспечение и базы данных, традиционно поддерживаются ногами 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 вида стандартов:

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

Виды стандартов

Существующие на сегодняшний день стандарты можно условно разделить на несколько групп.

- ☐ *По предмету стандартизации.* К этой группе можно отнести функциональные стандарты (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 ведомственные национальные стандарты (например, ГОСТ, ANSI, IDEF0/1), стандарты международных консорциумов и комитетов по стандартизации (например, 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 адаптируемости к конкретным условиям.

Далее мы рассмотрим методику CDM (Custom Development Method) фирмы Oracle 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) в качестве стандарта описания программного обеспечения.

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

Методика CDM фирмы Oracle

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

Ниже перечислены основные составляющие CASE-технологии и инструментальной среды фирмы Oracle.

- ❑ 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, с поддержкой в клиентской части всех современных стандартов и требований к графическому интерфейсу конечного пользователя.
- ❑ Наличие централизованной базы данных — репозитария. Репозитарий предназначен для хранения спецификаций проекта прикладной системы на всех этапах ее разработки. Он 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 ситуаций, в которых 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 разработке 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 CDM определяет следующие фазы жизненного цикла информационной системы:

- ❑ стратегию;
- ❑ анализ (формулирование детальных требований к прикладной системе);
- ❑ проектирование (преобразование требований в детальные спецификации системы);
- ❑ реализацию (написание и тестирование приложений);

- ☐ внедрение (установка новой прикладной системы, подготовка к началу эксплуатации);
- ☐ эксплуатацию (поддержка и сопровождение приложения, 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 совершенствования. Этот этап не является обязательным в случае, когда существующие 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

Генераторы приложений, входящие в состав CASE-средства DESIGNER/2000, позволяют полностью автоматизировать этот этап, существенно сократить сроки разработки системы и повысить ее качество и надежность.

Методика 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 реализацию модулей;

- ☐ конвертирование данных;
- ☐ документирование;
- ☐ тестирование;
- ☐ обучение;
- ☐ переход к новой системе;
- ☐ поддержку и сопровождение.

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

Особенности методики CDM

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

- ☐ Степень адаптивности CDM ограничивается тремя моделями жизненного цикла:
 - *классическая* модель предусматривает все этапы;
 - *быстрая разработка* ориентирована на использование инструментов моделирования и программирования Oracle;
 - *облегченный подход* рекомендуется в случае малых проектов и возможности быстро прототипировать приложения.
- ☐ 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 предложенной последовательности выполнения задач.
- ☐ Все модели жизненного цикла являются по сути каскадными. Даже «облегченный подход», несмотря на итерационность действий по прототипированию, сохраняет общий последовательный и детерминированный порядок выполнения задач.
- ☐ Методика не является обязательной, но может считаться фирменным стандартом. При формальном применении степень обязательности полностью соответствует ограничениям возможностей адаптации.
- ☐ 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 отсутствуют.
- ☐ CDM теснейшим образом опирается на инструментарий Oracle, несмотря на утверждения о простоте адаптации CDM к проектам, в которых используется другой комплект инструментальных средств.
- ☐ Методика 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 г. подкомитетом SC7 (Проектирование программного обеспечения) объединенного технического комитета JTC1 (Информационные технологии) ISO/IEC.

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 создания 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, протекающие во время жизненного цикла ПО, должны быть совместимы с процессами, протекающими во время жизненного цикла автоматизированной системы.

Согласно ISO 12207, система — это объединение одного или нескольких процессов, аппаратных средств, программного обеспечения, оборудования и людей для удовлетворения определенным потребностям или целям.

ПРИМЕЧАНИЕ

В отличие от CDM фирмы Oracle, 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 относятся к 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 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 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.
- ❑ *Процесс поставки* определяет действия предприятия-поставщика, которое снабжает покупателя системой, программным продуктом или службой.
- ❑ *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 отличие от действий, перечисленных 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 текущего состояния и функциональной пригодности; сюда же относятся установка программного изделия на вычислительной системе и его удаление.

Помимо 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:

- ❑ решение проблем;
- ❑ документирование;
- ❑ управление конфигурацией;
- ❑ обеспечение качества;
- ❑ верификация;
- ❑ аттестация;
- ❑ совместная оценка;
- ❑ аудит.

В стандарте ISO 12207 также определяются четыре организационных процесса:

- ❑ управление;
- ❑ создание инфраструктуры;
- ❑ усовершенствование;
- ❑ обучение.

ПРИМЕЧАНИЕ

Под процессом усовершенствования в стандарте ISO 12207 понимается не усовершенствование 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 некоторые особенности стандарта ISO 12207.

- ❑ Стандарт ISO 12207 имеет динамический характер, обусловленный способом определения последовательности выполнения процессов и решения 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 выполняются (решаются) в ходе разработки, функционирования и сопровождения программного продукта в течение всей жизни системы, от определения требований до завершения ее использования.

- ❑ 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, неприменимых 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.

- ❑ 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 предоставлять услуги или решать задачи, включенные в процессы. Данный стандарт не предписывает имена, форматы или точное содержание получаемой документации. Решения такого типа принимаются сторонами, использующими стандарт.
- ❑ Качество обеспечивается с помощью различных процессов, выполняемых с разной степенью 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 пожеланиям потребителя.

- ❑ Степень обязательности рассматриваемого стандарта следующая: после решения организации о соответствии торговых отношений стандарту ISO 12207 в качестве условия оговаривается ее ответственность за минимальный набор процессов и задач, которые обеспечивают согласованность с этим стандартом.
- ❑ 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 обходиться без них.

Ценность стандарта ISO 12207 в том, что в нем представлены наборы задач, характеристики качества, критерии оценки и т. п., обеспечивающие всесторонний охват проектных ситуаций. Например, при анализе требований к системе предусматривается, что:

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

Далее, при анализе требований к программному обеспечению предусмотрено 11 характеристик качества, позволяющих обеспечить заданный уровень 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;
- ☐ за адаптацию процессов и задач стандарта к этой модели;
- ☐ за выбор и применение методов разработки программного обеспечения;
- ☐ за выполнение действий и решение задач, подходящих для проекта программного обеспечения.

Универсальный язык моделирования

Универсальный язык моделирования (UML), разработка которого началась с середины 90-х годов прошлого века на базе нескольких объектно-ориентированных методов и нотаций описания информационных систем, в настоящее время является общепринятым стандартом документирования процесса создания информационных систем и программного обеспечения. В качестве стандарта 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); за счет объединения методик каждого из них, собственно, и возник язык UML.

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

Первоначально язык UML создавался, чтобы упростить описание информационных систем на базе объектно-ориентированной технологии. Сейчас средствами языка можно описывать также 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 ин-

формационных систем, которая, в свою очередь, диктуется усложнением решаемых задач.

Когда количество объектов информационной системы не превышает 6–8 (то есть психологического критерия, до которого человек еще способен оперировать информацией без записи и дополнительной тренировки), сложности, возникающие при проектировании системы, преодолимы и без 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 — миллионов, ни один специалист, каким бы образованным и опытным он ни был, не способен охватить систему в целом. К примеру, система навигации должна размещаться на тысячах автомобилей, морских и речных судах, железнодорожных составах, десятках искусственных спутников Земли, использовать тысячи компьютеров и всевозможных сетей связи.

Такие информационные системы под силу создавать только группам разработчиков, как правило, с разделением обязанностей на аналитиков, проектировщиков, программистов, тестеров и, конечно, руководителей. 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 языком 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 практически невозможно разобраться в коде программ, написанных не им самим, а часто и им самим, но давно.

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 UML-диаграммами, показывает разработчику результат, который необходимо достичь в процессе проектирования.

Проблема коммуникации внутри коллектива разработчиков, требующая документирования результатов проектирования, чтобы иметь возможность в дальнейшем вносить в систему усовершенствования, является не единственной побудительной причиной создания UML.

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

Выявление объектов-сущностей, их свойств, построение их иерархии является в большинстве случаев нетривиальной задачей, которая решается методами 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 системы, авторам которых удалось найти «изящную» комбинацию, неординарный ход, «изюминку» в построении иерархии объектов информационной системы.

Построение иерархии объектов требует опыта и усердной работы аналитиков и системных проектировщиков, для облегчения интеллектуального труда которых весьма кстати пришлось возможности нотации (лат. *notatio* — обозначение, система записи), позволяющей легко и понятно фиксировать возникающие идеи. Этой цели служат всевозможные кружочки, квадратики, стрелки и линии, с помощью которых человек пытается зафиксировать свои мысли 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 восьмидесятых годов ведутся разработки методик, позволяющих автоматизировать процесс построения иерархий объектов. Некоторые из методик, например CRC-карточки, не потеряли своей актуальности.

Словарь предметной области

В словарь предметной области включаются термины, используемые специалистами, для которых разрабатывается система. Словарь создается с целью наиболее полного понимания поставленных задач проектировщиками, которые, как правило, специалистами в этой области не являются.

Словарь представляет собой список терминов с их краткой расшифровкой. 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

Построение диаграмм сущность-связь основано на выявлении форм взаимосвязи и взаимодействия сущностей. Подробнее эти диаграммы описаны в главе 6 на примере проектирования структуры базы данных.

Метод Аббота

Метод Аббота 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 — на методы этих сущностей.

Метод Аббота поддается автоматизации, в частности соответствующие системы были построены Токийским технологическим институтом и фирмой Фуджи.

CRC-карточки

Аббревиатура CRC означает Class-Responsibilities-Collaborators (класс-ответственность-участники). CRC-карточки впервые предложили Кент Бек (Kent Beck) и Уорд Каннингхэм (Ward Cunningham) для обучения объектно-ориентированному программированию.

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

В ходе анализа появляются новые карточки, в старые вносятся изменения. Может возникнуть ситуация, когда один из классов (объектов) окажется слишком большим, и на стадии реализации системы это приведет к необходимости его постоянного использования. В этом случае целесообразно разбить его на несколько классов или передать часть его функций другому классу.

ПРИМЕЧАНИЕ

Возможна ситуация, когда одни и те же сообщения будут передаваться между разными классами. Можно выделить такие сообщения в отдельный класс для удобства отладки и внесения изменений. Так поступают, например, при обращениях к базе данных с помощью SQL-запросов, выделяя 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 стал основой UML. Предложенная Бучем графическая нотация достаточно распространена и наряду с UML используется в системах автоматизации процесса разработки программного обеспечения, в частности в системе Rational Rose.

Применяемые в методе Буча обозначения несколько отличаются от обозначений, принятых в UML. Так, класс в нотации UML представляет собой прямоугольник, в нотации Буча — облако, каким его рисуют дети, что по замыслу автора символизирует абстрактность этого понятия. Кроме того, язык UML более формализован за счет 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 пр.), а также методов расширения ядра UML. Описания используемых терминов в общем совпадают с определениями, приводимыми в этой книге, поэтому мы не будем подробно на них останавливаться.

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 UML-диаграмм, основными разновидностями которых являются диаграммы:

- ☐ прецедентов, или вариантов использования (use case diagram);
- ☐ классов (class diagram);
- ☐ состояний (statechart diagram);
- ☐ активности, или деятельности (activity diagram);
- ☐ взаимодействия (interaction diagrams), к которым относятся диаграммы последовательности (sequence diagram) и кооперации, или сотрудничества (collaboration diagram);
- ☐ компонентов (component diagram);
- ☐ 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 UML-диаграмм.

ПРИМЕЧАНИЕ

Юридически к категории программы для ЭВМ относятся не только код на каком-либо языке программирования и исполняемый машинный код, но и все материалы, полученные в ходе разработки программ, в первую очередь, это касается документированных решений, применяемых при проектировании системы и написанных как на естественном человеческом языке, так и на языке UML. Это еще один аргумент в пользу активного использования UML при разработках информационных систем, так как в случае возникновения спора UML-диаграммы могут стать 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. Язык 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 на ее основе возможна автоматическая генерация части программного кода будущей информационной системы с описаниями классов и объектов (предварительное объявление в разделе типов и переменных), а также заголовки методов объектов, реализацию которых необходимо писать вручную. То же можно сказать в отношении диаграмм последовательности и кооперации, которые являются взаимно обратимыми, то есть их можно автоматически преобразовать 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 UML-диаграмм общепринято использование языка объектных ограничений (Object Constraint Language, OCL), разработанного фирмой IBM. Язык OCL похож на SQL, но создан специально для навигации и получения данных от 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 она описывает, какие возможности будет предоставлять система конечному пользователю, какая информация необходима для обработки запроса пользователя. При этом механизм функционирования системы от пользователя скрыт и на диаграмме прецедентов не показывается.

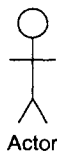
Конечный пользователь, в роли которого может выступать человек (например, покупатель или оператор) или техническое устройство (например, мобильный телефон), изображается в виде стилизованной фигурки человека (рис. 3.1).

Если же в качестве пользователя выступает сама система, что возможно, например, при предоставлении каких-либо функций определенному классу, вместо пользователя рекомендуется применять соответствующее обозначение класса.

Пользователь задействует систему определенным образом. Соответствующий прецедент (вариант использования) обозначается на диаграмме овалом, внутри которого пишется наименование варианта использования (рис. 3.2).

Для пояснения содержания диаграмм используют примечания, обозначаемые на диаграммах 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.



Actor

Рис. 3.1. Графическое изображение конечного пользователя

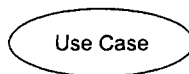


Рис. 3.2. Графическое изображение прецедента

ПРИМЕЧАНИЕ

Используемые на диаграммах обозначения являются общими для всех видов диаграмм.

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. Интерфейс определяет только описание (спецификации) операций класса или компонента, но он никогда не определяет физические реализации операций.

Интерфейс представляет собой сущность, которая дает 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 операции. Графически интерфейс обозначается небольшим кружком, рядом с которым указывается его наименование (рис. 3.4).

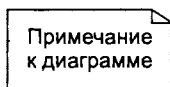


Рис. 3.3. Графическое изображение примечания

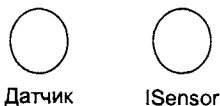


Рис. 3.4. Графическое изображение интерфейса

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 английскими, 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 прецедентами.

Ниже перечислены определенные в нотации UML виды отношений между компонентами на диаграммах прецедентов.

- ❑ *Отношение ассоциации* (association relationship) устанавливает роль пользователя в 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, говорит, что при определенных условиях прецедент В может быть дополнен прецедентом А. На практике это может означать, например, дополнительные (помимо обычных) меры по 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 другой прецедент в качестве его составной части. 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).

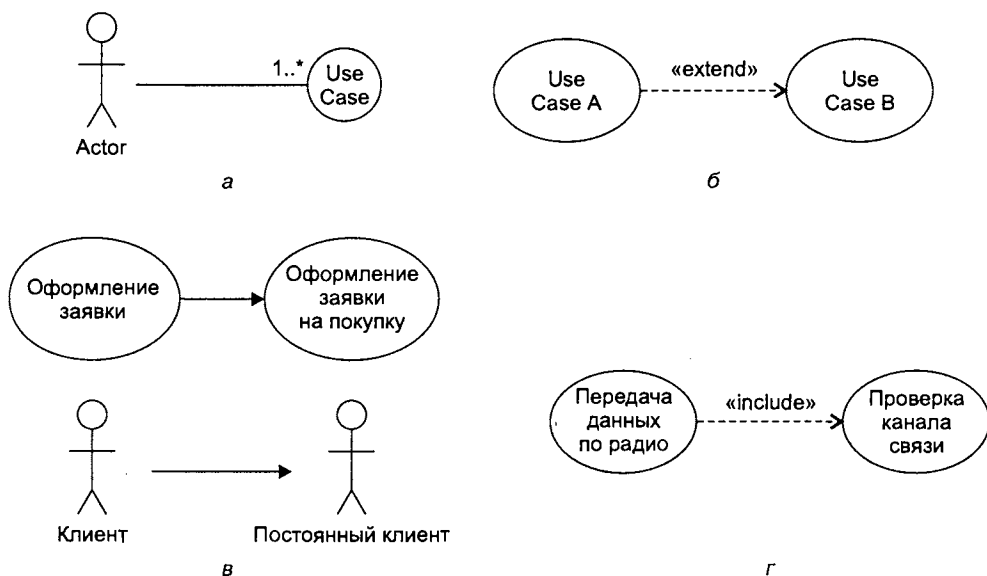


Рис. 3.5. Графическое изображение отношений на диаграммах прецедентов

Цифры над стрелкой (см. рис. 3.5, а) обозначают кратность (multiplicity) отношения и показывают количество возможных компонентов данного отношения. Случай на рисунке означает, что один и тот же пользователь может взаимодействовать с системой данным образом любое количество (обозначается звездочкой) раз.

ПРИМЕЧАНИЕ

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

Пакеты

Для сложных систем возникает необходимость разбиения 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).

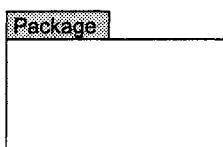


Рис. 3.7. Графическое изображение пакета

Компоненты, относящиеся к определенному пакету, могут быть доступны вне пакета, если указать имя компонента и его принадлежность определенному пакету через двойное двоеточие, например:

Имя_Пакета : : Имя_Компонента

Эта запись означает, что пакет образует собственное 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 представляется в виде прямоугольника с четырьмя секциями (на рис. 3.8, *а* показан формат, на рис. 3.8, *б* — пример). Любая из секций, кроме секции 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. Абстрактные классы весьма удобно использовать при конструировании иерархий в качестве промежуточных классов. Все, что касается абстрактных классов, в языке UML выделяется курсивом (в нотации Буча абстрактный класс помечается буквой *A* в треугольнике, обращенном вершиной вниз).

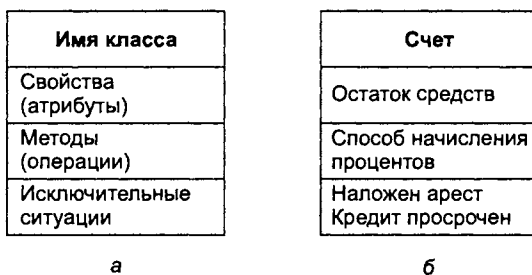


Рис. 3.8. Графическое изображение класса

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

Свойства класса 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

Ниже перечислены аргументы этой строки.

- видимость — видимость свойства для других классов. Допустимые значения:
 - public (или знак +) — свойство класса доступно любому другому классу;
 - protected (или знак #) — свойство класса доступно только экземплярам этого класса и потомкам данного класса;
 - private (или знак -) — 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, разные.

- тип_свойства — тип свойства. Этот тип выбирается, исходя из языка реализации проекта (C++, Delphi, Java и других).
- кратность — 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, например, следующие значения: Иван Петров, Уильям Джефферсон Клинтон, Мехти Асадулла оглы Мамедов, то есть значения, содержащие не более 5 слов. Если же в качестве кратности свойства доход указан диапазон 0..* и тип Integer, значение этого свойства может принимать любое положительное значение в денежном формате.

- **значение_по_умолчанию** — начальное значение свойства, которое в последующем можно изменить программно. Если в строке определения свойства вместо одного знака равенства (=) указать два (==), то значение по умолчанию программно изменить будет нельзя.

Например, следующая запись для свойства класса Менеджер означает, что по умолчанию всем сотрудникам, замещающим эту должность, первоначально устанавливается заработная плата 700 долларов:

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

ПРИМЕЧАНИЕ

Язык UML помимо проектирования информационных систем, нашел применение в сфере аналитики бизнеса. Отображенная на диаграммах структура фирмы и ее деятельности позволяет выявить, например, перегруженность определенных отделов, участки, тормозящие принятие управленческих решений. На основе полученных данных вырабатываются рекомендации по совершенствованию деятельности организаций. UML-диаграммы в этом случае служат инструментом анализа и иллюстрациями сделанных выводов.

Методы класса служат для обработки свойств класса и характеризуют поведение экземпляров класса.

Формат записи метода выглядит так:

видимость имя_метода (список параметров) : тип_значения {строка-свойство}

Ниже перечислены аргументы этой строки.

- **видимость** — этот аргумент определяется аналогично видимости свойств.
- **имя_метода** — идентификатор метода.
- **список параметров** — перечень разделенных запятой формальных параметров. Наличие круглых скобок обязательно. Каждый из формальных параметров может быть представлен в следующем виде:

вид_параметра имя_параметра : тип_параметра = значение_по_умолчанию

Здесь

- **вид_параметра** — входной (передаваемый методу), выходной (возвращаемый методом) или универсальный (значение параметра может изменяться методом) параметр (обозначается соответственно in, out или inout);
- **тип_параметра** — тип параметра, определяемый языком реализации класса.
- **тип_значения** — тип значения. Определяется языком реализации класса.
- **строка-свойство** — значения свойств, которые могут относиться к данному параметру. Метод, не изменяющий состояния объекта, обозначается строкой-свойством {query}. Строка-свойство {concurrency = sequential} указывает на необходимость обращения к методу во время вызова другого метода, строка-свойство {concurrency = concurrent} — на 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 данном случае 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 может служить запись

+ open ()

Этот метод не возвращает никакого результата своего выполнения.

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

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

В нотации UML между компонентами диаграммы классов могут существовать различные отношения.

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

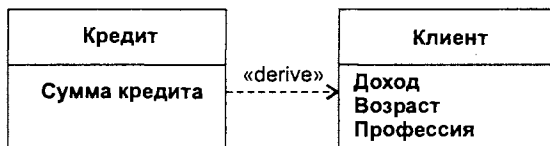


Рис. 3.9. Графическое изображение отношения зависимости

Ключевое слово (стереотип) над стрелкой означает, что свойства класса Кредит, в частности сумма кредита, выдаваемая банком заемщику, могут быть определены, исходя из характеристик заемщика (свойств класса Клиент): ежемесячного дохода, возраста и других. Ключевое слово является необязательным элементом. Помимо значения «derive», оно может принимать следующие 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).

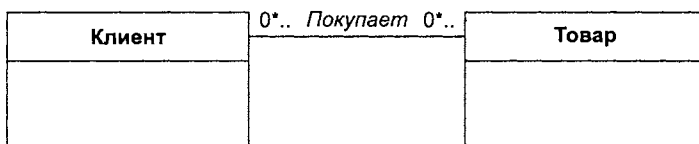


Рис. 3.10. Графическое изображение отношения ассоциации

Пример на рисунке означает, что клиент делает 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).

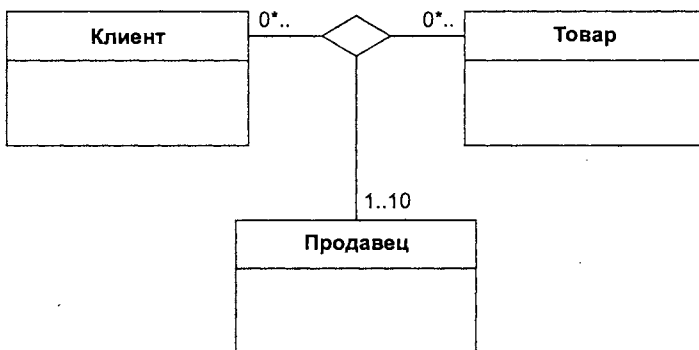


Рис. 3.11. N-арная ассоциация

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

Частными случаями отношений ассоциации являются исключаящая ассоциация, агрегация и композиция:

- *исключаящая ассоциация* (xor-association) указывает на возможность связи определенного класса только с одним из нескольких классов (рис. 3.12);
- *отношение агрегации* означает включение нескольких классов в другой класс (графически отношение агрегации показано на рис. 3.13 и означает, что в состав класса **Сервис** в качестве самостоятельных единиц могут входить тарелки и другие столовые приборы);

ПРИМЕЧАНИЕ

Не все языки программирования поддерживают такие конструкции.

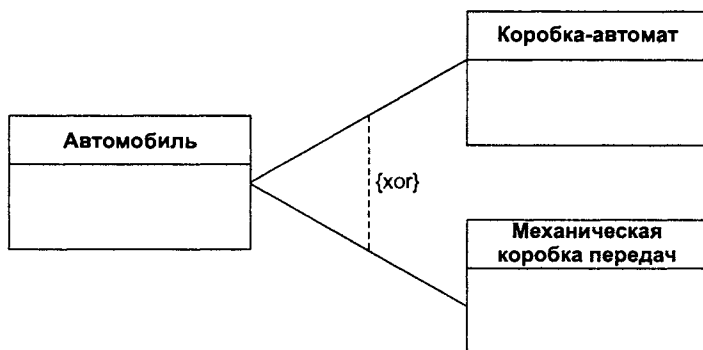


Рис. 3.12. Графическое изображение исключяющей ассоциации

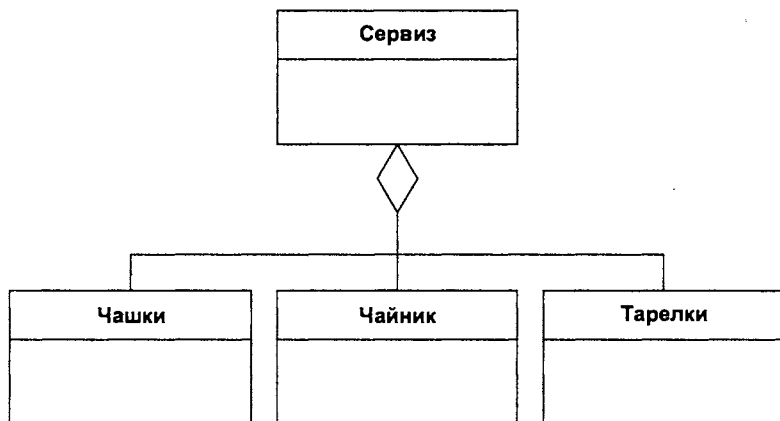


Рис. 3.13. Графическое изображение отношения агрегации

- *отношение композиции* показывает, что компонент состоит из нескольких частей, которые, в отличие от отношения агрегации, не могут использоваться отдельно — графическое изображение отношения композиции (рис. 3.14) отражает его сходство, по сути, с отношением агрегации (так, составными частями класса Газета являются редакционные статьи, фотоиллюстрации, реклама, которые не могут существовать отдельно от самой газеты, если под газетой понимать лист бумаги). Отношение композиции может иметь кратность.
- *Отношение обобщения* (generalization relationship) показывает, что компонент (пользователь или прецедент) является частным случаем другого компонента. Графически это отношение обозначается непрерывной стрелкой от частного к общему (рис. 3.15). Отношениями обобщения иллюстрируется наследование классов.

В приведенном примере класс Рыбные консервы наследует свойства и методы более общего класса Товар. Язык UML является средством документирования

и иллюстрирования удачных идей и решений в области проектирования информационных систем. Так, диаграмма, представленная на рис. 3.16, выражает идею множественного наследования, реализованную в некоторых языках программирования.

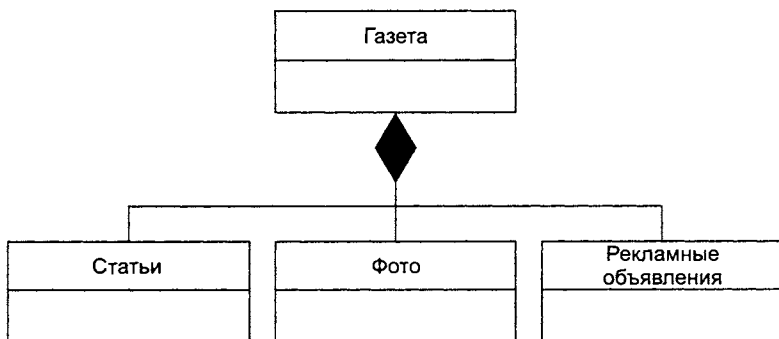


Рис. 3.14. Графическое изображение отношения композиции



Рис. 3.15. Графическое изображение отношения обобщения

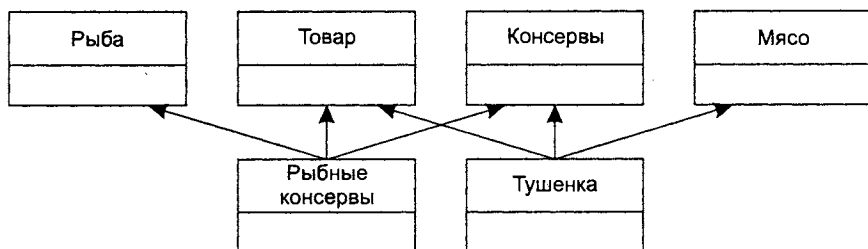


Рис. 3.16. Множественное наследование

Классы Рыбные консервы и Тушенка наследуют свойства и методы более общих классов Товар и Консервы, в то же время они наследуют, 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++ наследование поддерживается, в языке Object Pascal — нет. Отказ от поддержки множественного наследования обусловлен возможностью возникновения ситуации, когда два или несколько классов, имея единого предка, по-разному реализуют одноименный метод, поэтому при множественном наследовании возникнет проблема выбора той или иной реализации метода (рис. 3.17), которую придется разрешать «вручную» (как это делается в C++).

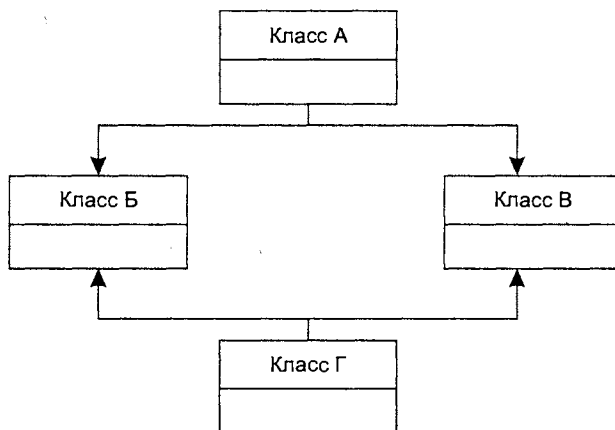


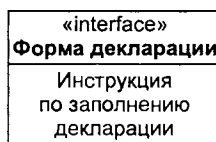
Рис. 3.17. Проблема множественного наследования классов

❑ Отношение обобщения может содержать поясняющий его идентификатор:

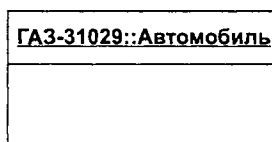
- {complete} — на диаграмме показаны все классы-потомки;
- {incomplete} — на диаграмме указаны не все классы-потомки;
- {disjoint} — множественное наследование не допускается;
- {overlapping} — множественное наследование допускается.

Помимо классов, на диаграммах классов могут присутствовать интерфейсы, объекты (экземпляры классов) и параметризованные классы (метаклассы, шаблоны).

Интерфейс на диаграмме классов показывается прямоугольником с двумя секциями (рис. 3.18, а). В первой записываются имя интерфейса, ключевое слово «interface» и служебная информация. Вторая секция предназначена для записи методов интерфейса.



а



б

Рис. 3.18. Графическое изображение интерфейса и объекта на диаграмме классов

Под объектом в языке UML понимается отдельный экземпляр, или пример, класса, структура и поведение которого полностью определяется порождающим этот объект классом. Объекты показываются прямоугольниками, как и классы, при этом имя объекта подчеркивается и содержит указание на класс объекта (рис. 3.18, б).

Параметризованный класс (метакласс, шаблон) представляет собой семейство классов, каждый член которого может отличаться от других каким-либо параметром, указываемым в пунктирной рамке в правом верхнем углу изображения класса (рис. 3.19, а).

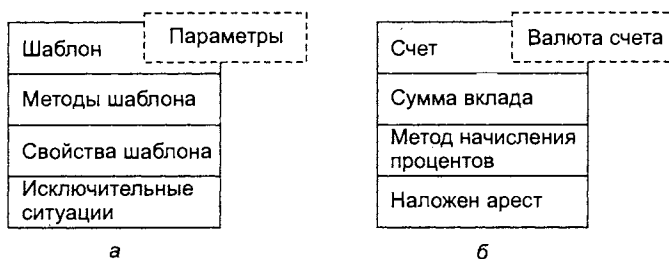


Рис. 3.19. Графическое изображение метакласса

В примере на рис. 3.19, б метакласс может служить основой для создания классов Счет в рублях, Счет в долларах США, Счет в евро или Мультивалютный счет.

Диаграмма состояний

В процессе функционирования информационной системы свойства объектов системы будут принимать различные значения, а объекты системы — различные состояния. Состояния объектов, переходы объектов из одного состояния в другое, сообщения, которыми обмениваются объекты, образуют 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 отражаются состояния и переходы между состояниями объекта под воздействием внешних факторов.

Состояния (states) на диаграмме состояний показываются прямоугольниками с закругленными вершинами. В каждом таком прямоугольнике может быть одна (обязательная) секция, в которой записывается имя состояния (для имени рекомендуется использовать глаголы, причастия и деепричастия), и несколько других секций, обозначающих действия объекта в этом состоянии.

Начальное состояние изображается черным кружком, конечное — черным кружком в белом кольце. На диаграмме, показанной на рис. 3.20, объект из начального состояния переходит в некоторое другое состояние, в котором выполняется действие 1, а затем — в конечное состояние.

Начальное и конечное состояния могут отсутствовать. Примером отсутствия конечного состояния может служить система, которая запускается один раз, а даль-

ше функционирует в непрерывном режиме. Примером отсутствия начального состояния может служить функционирующая система, неизвестно когда возникшая. Примером отсутствия как конечного, так и начального состояния могут служить циклические изменения какого-либо объекта.

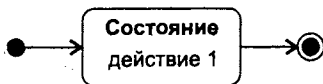


Рис. 3.20. Простейшая диаграмма состояний

Внутренние действия состояния описываются в следующем формате:

метка / действие

Здесь аргумент 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 — обращение к подсостоянию (substate) данного состояния объекта.

Составное состояние (composite state), или суперсостояние, — это 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. Два варианта графического изображения составного состояния показаны на рис. 3.21.

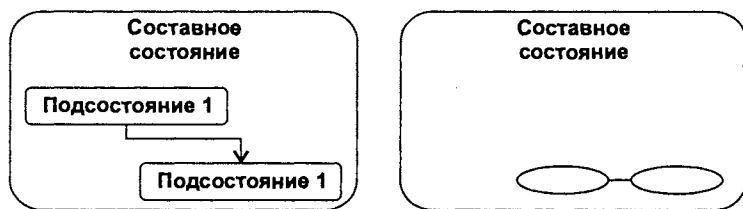


Рис. 3.21. Графическое изображение составного состояния

Частными случаями составного состояния являются последовательные состояния (на рис. 3.22 это состояния А, Б, В) и параллельные состояния (на рис. 3.22 это состояние Г по отношению к последовательности состояний А, Б, В).

Историческое состояние (history state) — это такое составное состояние, последующие переходы в которое означают переход к последнему подсостоянию объекта. В качестве примера можно привести систему ведения журнала сбоя (рис. 3.23). При втором и последующем обращении к этому состоянию нет не-

обходимости еще раз создавать журнал сбоев, поэтому переход приведет сразу к подсостоянию **Запись в журнал**. Историческое состояние обозначается символом **H** в кружке.

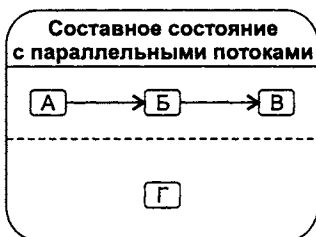


Рис. 3.22. Графическое изображение составного состояния



Рис. 3.23. 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 изображаются на диаграммах состояний стрелками, над которыми могут указываться событие (event), вызвавшее этот переход, условие допустимости этого перехода (сторожевое условие) и действия, сопровождающие этот переход, в формате:

событие(список параметров) [сторожевое условие] выполняемые действия

События на диаграммах состояний исполняют роль триггеров — переход не произойдет, пока не случится соответствующее событие. Если рядом со стрелкой, обозначающей переход, не указано событие, его специфицирующее, то такой переход называется нетриггерным. Нетриггерный переход происходит сразу после выполнения действий в этом состоянии.

Сторожевое условие (guard condition) — некоторое логическое выражение, являющееся дополнительным условием перехода. Переход не произойдет, если сторожевое условие принимает значение **false**, даже в случае наступления соответствующего события. Сторожевые условия удобно использовать, когда из одного состояния при наступлении одного и того же события возможен переход в несколько других состояний. Например, пусть клиент в виртуальном магазине выбрал товары (состояние — выбор) и по окончании послал команду оплатить покупку (событие). Тогда товары доставляются клиенту только в том случае, если на его счете есть достаточная сумма (сработает переход в состояние, в котором выполняется действие доставки), если же необходимая сумма отсутствует, возможны переходы в другие состояния.

Переходы могут быть простыми и сложными. Простой переход — это переход в другое или в то же состояние объекта. Переход в то же состояние на диаграммах изображается стрелкой, начинающейся и заканчивающейся у этого состояния (рис. 3.24). При каждом переходе такого рода выполняются соответствующие входные и выходные действия.



Рис. 3.24. Графическое изображение перехода в то же состояние

Сложные переходы — это переходы с несколькими исходными (или конечными) состояниями объекта, а также переходы между подсостояниями разных состояний. Переход с несколькими исходными или несколькими конечными состояниями объекта (параллельный переход) обозначается жирной вертикальной или горизонтальной линией (линией синхронизации), как показано на рис. 3.25.

Пример перехода между подсостояниями разных состояний показан на рис. 3.26 (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 * внутри окружности. В примере на рис. 3.27 объект не перейдет в подсостояние B, пока не произойдет переход от C к D, который «синхронизирует» переход от A к B.

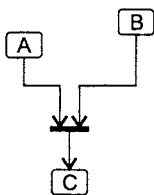


Рис. 3.25. Графическое изображение параллельного перехода

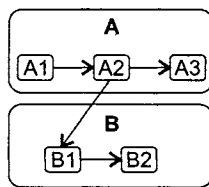


Рис. 3.26. Пример перехода между подсостояниями разных состояний

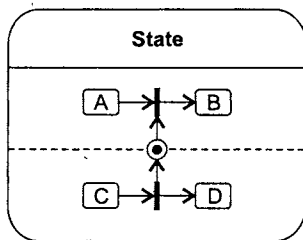


Рис. 3.27. Пример использования синхронизирующего состояния

СОВЕТ

Нотация UML предоставляет широкие возможности для иллюстрирования процесса разработки информационных систем, что видно на примере диаграмм состояний, на которых существует возможность показать составные состояния, сложные переходы и т. д. В то же время злоупотребление сложными конструкциями в конечном итоге приводит только к трудностям в понимании диаграмм — цели, прямо противоположной заявленной нами. Такой же эффект дает перегрузка диаграмм объектами и связями между ними. Отсюда следует общая рекомендация к построению диаграмм: без необходимости не усложняйте диаграммы, не пытайтесь на одной диаграмме показать сразу все аспекты функционирования системы.

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

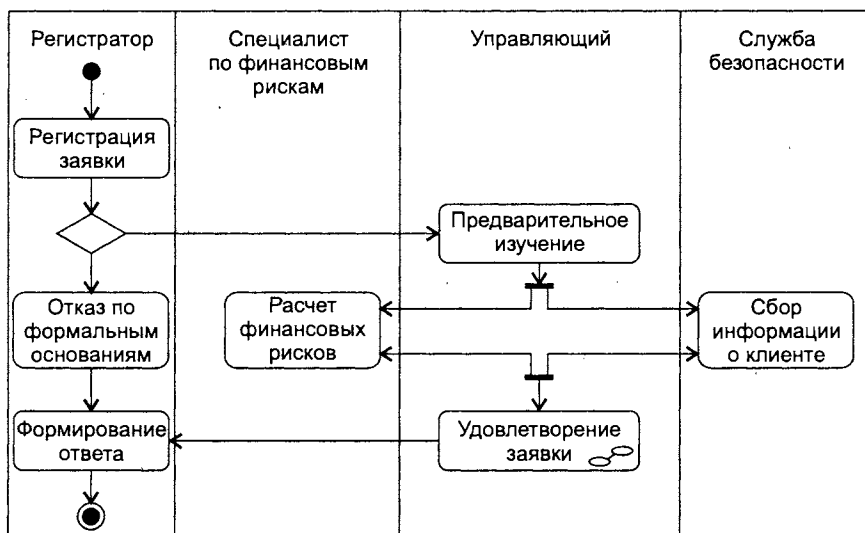


Рис. 3.28. Пример диаграммы активности

Поясним применяемые обозначения. Начальное и конечное состояния изображаются так же, как 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.).

Действие может содержать несколько поддействий. Если на диаграмме их показывать нецелесообразно, используют обозначение вложенности действий. Так, в при-

веденном примере действие Удовлетворение заявки включает в себя несколько поддействий, среди которых могут быть оценка достоверности данных о клиенте, оценка правильности расчетов, непосредственно принятие решения.

Движение данных (переходы) изображается сплошными стрелками. Возле стрелок может быть указано сторожевое условие. Если движение данных предусматривает ветвление, то указание условия обязательно (в примере на рис. 3.28 оно не показано). Символ ветвления графически изображается ромбом, из которого выходят две или более стрелки. Разделение потока данных и слияние параллельных потоков данных изображается сплошной жирной чертой, связывающей 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, передачу заказа между отделами организации иллюстрирует рис. 3.29. В качестве передаваемых данных может быть указан пользователь, например, при построении карты веб-сайта.

Вектор времени на диаграммах активности в явном виде не воспроизводится.

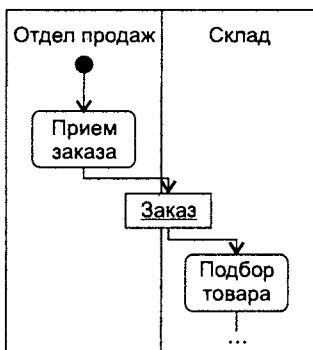


Рис. 3.29. Движение заказа между отделами

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 последовательности являются пользователь, объект, линия жизни объекта (object lifeline), сообщение (message) и фокус управления (focus of control). Объект графически изображается, как и на других UML-диаграммах, прямоугольником с подчеркнутым именем, линия жизни объекта — вертикальной пунктирной линией, сообщение — горизонтальной стрелкой, фокус управления — прямоугольником на линии жизни объекта.

Примеры диаграмм последовательности приведены на рис. 3.30. На рис. 3.30, а пользователь создает объект, который через некоторое время уничтожается (символ уничтожения объекта — жирный крест). Рисунок 3.30, б наглядно демонстрирует идею и параметры замера температурно-влажностного режима в помещении хранилища музея. Фокус управления показывает, какой именно элемент находится в активном состоянии (действует). Фокус управления может быть одновременно как у одного, так и у нескольких объектов системы. Фокус управления может передаваться от одного объекта другому. На рис. 3.31 приведен пример передачи фокуса управления от объекта А объекту Б или В в зависимости от условия $x = 0$ или $x > 0$, записываемого возле символа ветвления.



Рис. 3.30. Примеры диаграмм последовательности

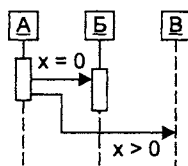


Рис. 3.31. Передача фокуса управления

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

- «call» — вызов объектом другого объекта;
- «return» — возврат значения вызвавшему объекту;
- «create» — создание объекта;
- «destroy» — уничтожение объекта, которому передается это сообщение;
- «send» — посылка асинхронного сигнала.

Рекурсия (самовывоз) объекта на диаграммах последовательности может быть показана как сообщение вызова, обращенное объектом к самому себе (рис. 3.32, а), или специальным символом на изображении фокуса управления (рис. 3.32, б).

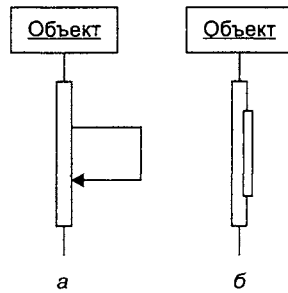


Рис. 3.32. Варианты изображения рекурсии

Диаграмма сотрудничества

Диаграмма сотрудничества, как и диаграмма последовательности, является разновидностью диаграмм взаимодействия. Современные программные средства построения диаграмм обеспечивают автоматическое преобразование 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. 3.33. Окружность — это кооперация, пунктирная линия — роль пользователя в кооперации, стрелка — отношение обобщения, общее для всех UML-диаграмм. 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 UML-диаграммах, — прямоугольниками. Однако на диаграммах данного вида имя объекта может дополняться его ролью в сотрудничестве. На рис. 3.34, *а* показан образец, на рис. 3.34, *б* — пример обозначения имени объекта, любая из трех частей которого может отсутствовать.

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 мультиобъекта, изображаемого графически двумя прямоугольниками, наложенными друг на друга (рис. 3.35).

В отличие от мультиобъекта, составной объект действительно состоит из других объектов (рис. 3.36).

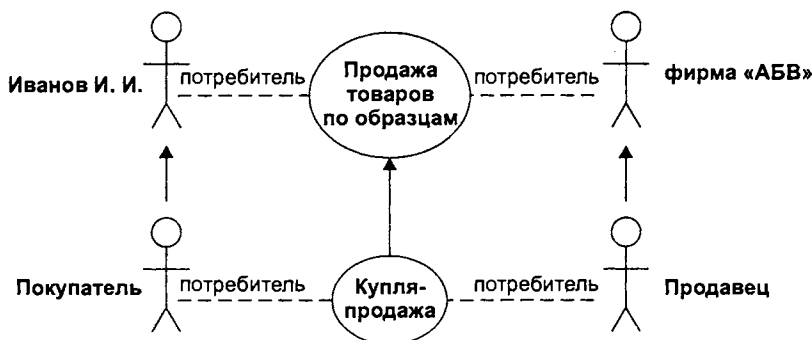


Рис. 3.33. Пример диаграммы сотрудничества уровня спецификаций

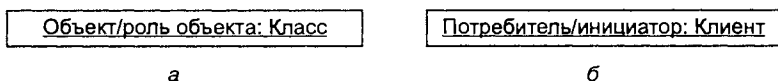


Рис. 3.34. Графическое изображение объектов на диаграммах сотрудничества

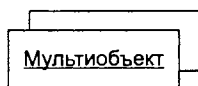


Рис. 3.35. Графическое изображение мультиобъекта

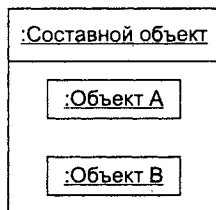


Рис. 3.36. Графическое изображение составного объекта

Между объектами диаграммы сотрудничества существуют связи (links), по которым объекты посылают друг другу сообщения. Связи не имеют названий (в терминах UML они являются анонимными), но могут быть специфицированы ключевыми словами (стереотипами):

- ☐ «association» — связь означает некую зависимость объектов;
- ☐ «parameter» — объект полагается параметром метода;
- ☐ «local» — локальная переменная метода, область видимости которой ограничена соседним объектом;
- ☐ «global» — глобальная переменная, область видимости которой ограничена диаграммой сотрудничества;
- ☐ «self» — связь объекта с самим собой.

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 объекту. Например, объект **Отдел продаж** передает объекту **Склад** некоторый принятый в организации документ (переменную), в котором сообщает о необходимости выделения продукции той или иной номенклатуры. 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 смысловую нагрузку в виде определения ролей взаимодействующих объектов. В зависимости от этого сообщения графически изображаются:

- ❑ сплошной линией с треугольной стрелкой — такое сообщение означает вызов процедуры (метода объекта) или вызов другого потока управления (рис. 3.37, а);
- ❑ сплошной линией с обычной стрелкой — такое сообщение означает простой поток управления, то есть просто передачу данных (рис. 3.37, б);
- ❑ сплошной линией с полустрелкой — такое сообщение не имеет заранее обусловленного времени передачи, являясь, как правило, асинхронным (рис. 3.37, в);
- ❑ пунктирной линией с обычной стрелкой — такое сообщение означает возврат значения из процедуры (рис. 3.37, г).

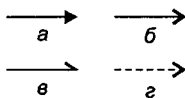


Рис. 3.37. Графическое изображение сообщений на диаграммах сотрудничества

Сообщение записывается в определенном формате. Например, показанная ниже запись 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 оно будет занимать место между сообщениями 3.1 и 3.3, при этом возможна параллельная передача сообщения с другими сообщениями, имеющими номер 3.2. Само сообщение вызывает метод нахождения сведений о человеке по фамилии, имени и отчеству (список аргументов), предполагая предоставление карточки по форме 1А на запрашиваемое лицо (возвращаемое значение):

1, 2 / [пароль] 3.2 Форма_1А := найти_сведения (Фамилия, Имя, Отчество)

В заключение приведем пример диаграммы сотрудничества. На рис. 3.38 показана диаграмма, иллюстрирующая отношения по расчетам чеками, широко используемые в хозяйственной деятельности предприятий (см. параграф 5 в главе 46 Гражданского кодекса Российской Федерации).

ПРИМЕЧАНИЕ

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

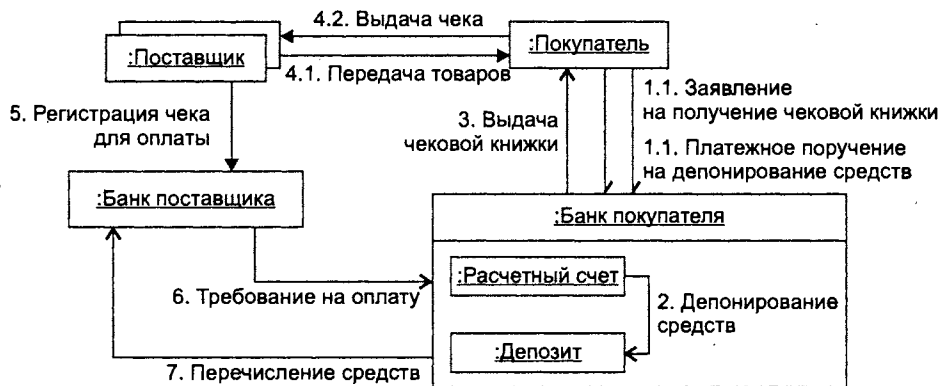


Рис. 3.38. Расчеты чеками

Диаграмма компонентов

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

ПРИМЕЧАНИЕ

Диаграммы компонентов играют существенную роль при оптимизации быстродействия системы. С их помощью выявляются наиболее часто используемые компоненты, определяется их оптимальное распределение по модулям. На завершающих стадиях разработки информационной системы может возникнуть ситуация, когда незначительные изменения в 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 приложений с числом модулей от тысячи и выше.

Основным элементом диаграмм компонентов является компонент (component), который графически изображается прямоугольником со встроенными слева прямоугольными секциями. Внутри прямоугольника указывается имя компонента, которым может быть имя исполняемого файла, базы данных и т. д., а также служебная информация: версия, язык реализации, разработчик (рис. 3.39).

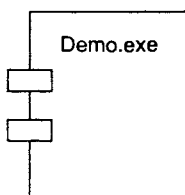


Рис. 3.39. Графическое изображение компонента

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. К таким компонентам относятся динамические библиотеки (рис. 3.40, а), справочные файлы (рис. 3.40, б), исходные тексты программ (рис. 3.40, в), веб-документы (рис. 3.40, г). Эти обозначения вместе с обозначениями исполняемых модулей иногда называют *артефактами*.

Между компонентами и другими элементами диаграммы компонентов существуют отношения зависимости (рис. 3.41), показывающие, что один компонент зависит от другого и при его изменении тоже должен меняться, а также отношения реализации, изображаемые сплошной линией (3.42, а), либо указанием на соответствующие элементы внутри обозначения компонента (3.42, б).

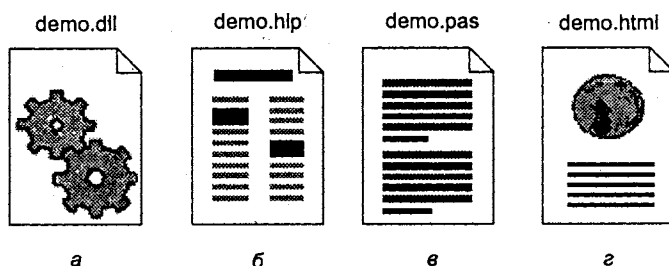


Рис. 3.40. Графическое изображение артефактов

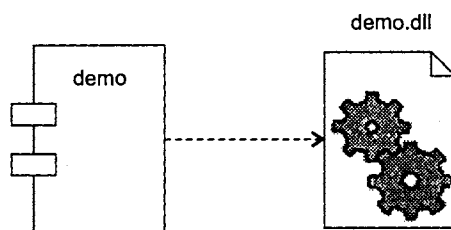


Рис. 3.41. Графическое изображение отношения зависимости

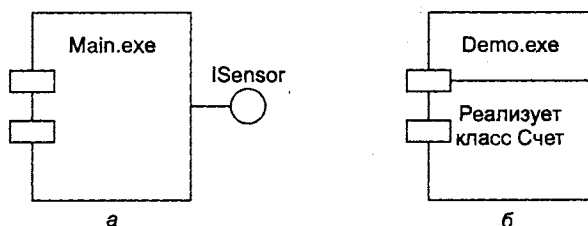


Рис. 3.42. Графическое изображение отношений реализации

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), который графически изображается аксонометрической проекцией куба (рис. 3.43). Внутри обозначения узла указываются его имя (например, Сервер) и развертываемые на нем компоненты, изображаемые так же, как на диаграмме компонентов.

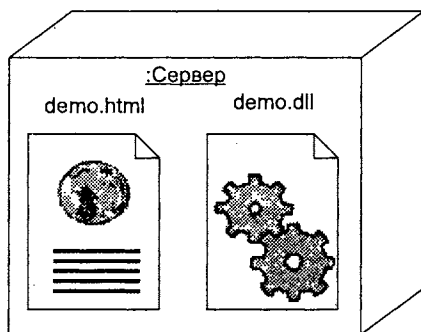


Рис. 3.43. Графическое изображение узла

Между узлами сплошными линиями показывают соединения, которые могут содержать пояснения относительно характера реализации связи между компонентами (рис. 3.44).

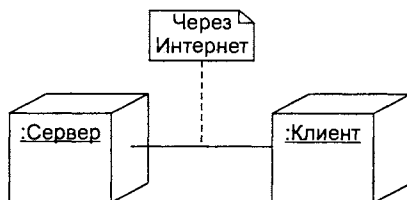


Рис. 3.44. Графическое изображение соединения

ГЛАВА 4 Реляционные базы данных

По мере развития вычислительной техники изменялись основные направления ее использования. Первоначально средства вычислительной техники предназначались для разного рода математических вычислений, которые невозможно было выполнить «вручную» за разумное время. Развитие этого направления привело к появлению разделов математики, связанных с численными методами вычислений, и 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 расширения возможностей и снижения стоимости вычислительных средств, получило развитие второе направление, связанное с использованием средств вычислительной техники в автоматизированных информационных системах. Здесь вычислительные возможности компьютеров отходят на второй план — основные функции вычислительных средств в информационных системах направлены на надежное хранение информации, выполнение специфических для данного приложения преобразований информации и/или вычислений, предоставление 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 выше. Это связано с тем, что объемы обрабатываемой информации, как правило, достаточно велики, а сама информация имеет сложную структуру. Этим же объясняется существенное возрастание требований к объему оперативной памяти 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 1400–1500 МГц) при решении сложных научных задач могут даже уступать по вычислительным возможностям «большим» компьютерам 15–20-летней давности.

Общие сведения о базах данных

Развитие компьютерных технологий, связанных с хранением и обработкой данных, привело к появлению в конце 60-х — начале 70-х годов специализированного программного обеспечения, получившего название *систем управления базами данных*, или 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:

- ☐ поддержанием логически согласованного набора файлов;
- ☐ предоставлением языка манипулирования данными;
- ☐ восстановлением информации после разного рода сбоев;
- ☐ обеспечением параллельной работы нескольких пользователей.

Основные функции систем управления базами данных

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

- ☐ непосредственное управление данными во внешней памяти;
- ☐ управление буферами оперативной памяти;
- ☐ управление транзакциями;
- ☐ 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 предоставление необходимых структур внешней памяти (постоянных запоминающих устройств, 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 пользователям базы данных в общем случае не нужно знать,

применяет ли СУБД файловую систему, а если применяет, 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 требуется обмен с внешней памятью, то вся система будет работать со скоростью функционирования устройства внешней памяти.

Повышения скорости обмена данными можно достичь, используя буферизацию данных в оперативной памяти. При этом даже если операционная система выполняет общесистемную буферизацию (как в случае ОС UNIX), этого недостаточно для целей СУБД, у которой гораздо больше информации, позволяющей судить 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 лишней.

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 ее *откат* (rollback) с отменой всех изменений данных, произведенных в ходе выполнения транзакции, и возвращением базы данных к состоянию до начала выполнения транзакции.

Управление транзакциями необходимо для поддержания логической целостности базы данных.

Поддержка механизма транзакций является обязательным условием работы даже однопользовательских, а тем более многопользовательских СУБД. Поскольку каждая транзакция начинается при целостном состоянии базы данных и оставляет это состояние целостным после своего завершения, понятие транзакции очень удобно

использовать как единицу активности пользователя по отношению к базе данных. При должном 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* и *плана сериализации* смеси транзакций. Под сериализацией понимается параллельное выполнение смеси транзакций, результат которого эквивалентен результату их последовательного выполнения. План сериализации смеси транзакций — это такой план, который приводит к сериализации транзакций. Понятно, что если удастся добиться реальной сериализации смеси транзакций, то для каждого пользователя, по инициативе которого начата транзакция, выполнение других транзакций будет незаметным (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 синхронизированных 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 обработки транзакций других пользователей.

Протоколирование

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 компьютера и 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 надежного хранения данных требуется их избыточность. 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 недоступную пользователям и поддерживаемую с особой тщательностью часть базы данных, в которую поступают записи обо всех изменениях основной части базы данных. Иногда используются две копии журнала, располагаемые на разных физических дисках.

В разных СУБД изменения базы данных протоколируются 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* (Write Ahead Log, WAL). Несколько утрируя, можно сказать, что эта стратегия подразумевает внесение в журнал записи об изменении любого объекта базы данных до того, как будет выполнено и зафиксировано изменение этого объекта. Если в СУБД поддерживается протокол WAL, то с помощью журнала можно решить все проблемы восстановления базы данных после любого сбоя.

Самая простая ситуация восстановления — откат отдельной 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. В некоторых СУБД так и делают, но в большинстве систем локальные журналы не поддерживают, а откат отдельной 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 очистки буферов оперативной памяти при мягком сбое). При соблюдении протокола WAL во внешней памяти журнала должны гарантированно находиться записи, относящиеся к операциям модификации обоих видов объектов. Целью восстановления после мягкого сбоя является приведение внешней памяти основной части базы данных в такое состояние, которое возникло бы при фиксации во внешней памяти изменений всех завершившихся транзакций и 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 сохранились во внешней памяти.

Для восстановления базы данных после жесткого сбоя используют журнал и архивную копию базы данных. Архивная копия — это полная копия базы данных, полученная к началу заполнения журнала (хотя имеются и другие варианты трактовки этого термина). Естественно, для нормального восстановления базы данных после жесткого сбоя необходимо, чтобы журнал не пропал. Тогда восстановление базы

данных состоит в том, что на основе данных архивной копии по журналу повторно выполняются все транзакции, которые закончились 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, носящие 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 — язык структурированных запросов). 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 контроль 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, хранящая

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 тоже производится на основе специального набора SQL-операторов. Идея состоит в том, что для выполнения разных SQL-операторов пользователь должен обладать разными 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 рассматриваются далее.

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 основные причины этой тенденции:

- ☐ цена персональных компьютеров значительно ниже, чем больших ЭВМ;
- ☐ по функциональным возможностям персональные компьютеры превосходят большие ЭВМ;
- ☐ существенно сократился разрыв между производительностью персональных компьютеров и больших ЭВМ, кроме того, для многих задач обработки данных 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 больших вычислительных машин (mainframes) 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 с собственными моделями, форматами представления данных и языками доступа к данным, к распределенной обработке данных в неоднородной вычислительной среде, состоящей из объединенных 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 совершенствования 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, такие как 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, не обеспечивалась переносимость прикладных программ.

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

- ☐ 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 70-х годов ряда статей Э. Кодда (Coad), 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 — *relationship*, 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, 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 обращения к файлам. 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, вполне доступных пользователям, не являющимся 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 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 получил SQL, который в 1986 г. был принят институтом ANSI (American National Standards Institute — Американский национальный институт стандартов) в качестве стандарта языков реляционных баз данных. Последнее обновление этого стандарта было принято в 1992 г., 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. Следующим этапом этого развития стало появление объектно-ориентированных баз данных, для которых характерны объектно-ориентированный подход, распределенность данных, наличие активного сервера баз данных, языки программирования четвертого поколения, фрагментация и параллельная обработки запросов, технологии тиражирования данных, многопоточная архитектура и другие революционные достижения 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 таких элементов объектного подхода, как наследование и полиморфизм;
- ❑ возможность объектного моделирования системы, позволяющего 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 по таблицам. Типы данных определяются разработчиком и не ограничены набором предопределенных 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 может использоваться. 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 оправданно. Нередко декомпозиция данных объекта не вызывает никаких проблем и вполне логична. В этом случае применение реляционной модели может быть более эффективным. 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 уже упоминавшимся Э. Коддом, известным американским специалистом в области баз данных. Основные концепции этой модели были впервые опубликованы в 1970 г. 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 стало 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 70-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 70-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 80-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 — средства 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» указывает, прежде всего, на то, что такая модель хранения данных построена на взаимоотношении составляющих ее частей, которые удобно представлять в виде двухмерной таблицы. Как показал Кодд, 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	28.02.1979	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 *типа данных* в реляционной модели данных полностью эквивалентно соответствующему понятию в алгоритмических языках. Набор поддерживаемых типов данных определяется СУБД и может значительно 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 — большой двоичный объект).

ПРИМЕЧАНИЕ

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

В рассматриваемом примере (см. табл. 4.1) используются три типа данных: строковый (столбцы Имя и Специальность), временной (столбец Дата_рождения) и целочисленный (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 для каждого столбца таблицы можно 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 представлять 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 их значений могут входить только те целые числа, которые позволяют обозначить 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 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, если 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, хотя и основаны на одном типе данных — целочисленном, сравнимыми не являются.

ПРИМЕЧАНИЕ

Понятие домена характерно далеко не для всех СУБД. В качестве примера реляционных баз данных, использующих это понятие, 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, ..., 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 также 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 Курс (так как абитуриенты еще не поступили в вуз и, следовательно, не имеют студенческого билета и не могут быть отнесены к какому-либо курсу). Кроме того, иногда при вводе информации в строку реляционной таблицы некоторые данные могут быть неизвестными и выясняться позже (при поступлении на подготовительные курсы абитуриент может еще не определиться 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 $K = (A_1, A_2, \dots, A_k)$ 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 кортежа отношения R не имеют одного и того же значения для $A_1, A_2, \dots, A_k$;
- ☐ *минимальность* — ни один из атрибутов $A_1, A_2, \dots, A_k$ 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 также условие минималь-

ности. 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 — это 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 — это ключ, созданный самой СУБД или пользователем с помощью некоторой процедуры, но сам по себе не содержащий информации. Искусственный ключ используется для создания уникальных идентификаторов строк, когда сущность должна быть описана полностью, чтобы однозначно идентифицировать конкретный элемент. Искусственный ключ часто задействуют вместо значимого сложного ключа, который является слишком громоздким, чтобы применяться в реальной базе данных. Система поддерживает искусственный ключ, но он никогда не виден пользователю.

Естественный ключ — это 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 является естественным, так как несет вполне определенную информацию.

Каждый из типов первичных ключей имеет свои достоинства 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 требуются лишь для связи между отношениями. Иными словами, естественные ключи позволяют получить более компактную форму таблиц (без

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 меняться. 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, в которой для сущности невозможно определить неизменяемый естественный ключ, является любая область с человеком в качестве сущности.

Второй довольно существенный недостаток естественных ключей состоит в том, что, как правило, уникальные естественные ключи являются составными и содержат строковые атрибуты. Как уже отмечалось, максимальная скорость выполнения операций над данными обеспечивается при использовании простых целочисленных ключей. Таким образом, с точки зрения быстрой работы 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, не зависящий ни от предметной области, ни, тем более, от структуры отношения, которое он идентифицирует. Таким образом можно обеспечить уникальность и неизменность ключа (раз он каким 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 допустимо 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 обязательно 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* (relation). Таким образом, еще одним важным понятием реляционной модели является связь между отношениями.

Для анализа связанных отношений воспользуемся рассмотренным ранее примером — отношением **СТУДЕНТЫ** (см. табл. 4.1). Данное отношение может быть связано с отношением **УСПЕВАЕМОСТЬ**, в котором содержатся сведения об успеваемости студентов по разным предметам. Фрагмент такого отношения может выглядеть так, как показано 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 и с помощью вторичных ключей. Кроме того, при установлении связей между таблицами необязательно требование уникальности ключа, по которому устанавливается связь.

ПРИМЕЧАНИЕ

Атрибуты внешнего ключа необязательно 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 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. Такие *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 реально-

го 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 целостности и кратко формулируется следующим образом: никакой атрибут первичного ключа строки не может быть пустым.

Рассмотрим теперь понятие *ссылочной целостности*.

Если две таблицы связаны между собой, то внешний ключ таблицы должен содержать только значения, уже имеющиеся 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 категорийной 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 здесь обеспечение целостности 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, можно использовать один из трех подходов, обеспечивающих ссылочную целостность:

- ❑ 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 удалении кортежа, на который имеются ссылки, во всех ссылающихся кортежах значение внешнего ключа должно автоматически становиться неопределенным;
- ❑ при удалении кортежа из отношения, на которое ведет ссылка, из ссылающегося отношения должны автоматически удаляться все ссылающиеся кортежи (это называется *каскадным удалением*).

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 такая целостность не поддерживается. 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 таблица 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 связями типа «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 связь типа «один ко многим». Если же выбрать в качестве главной таблицу УСПЕВАЕМОСТЬ, получится связь типа «многие к одному».

Основные свойства отношений

Рассмотрим 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 понятие как порядковый номер кортежа не имеет никакого смысла. Отсутствие упорядоченности кортежей отношения также является следствием определения отношения как множества кортежей. Поскольку не требуется поддержание порядка на множестве кортежей отношения, СУБД получает 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 явная упорядоченность набора атрибутов отношения не требуется, часто для неявного упорядочения атрибутов используется их порядок 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 двухмерных связанных таблиц. Набор средств управления подобным хра-

нилищем 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 рассматривали. 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

Ранее 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 алфавитным 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 требуемые страницы. 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 это положение 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 сокращает 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. Во-первых, обращение к индексу выполняется быстрее, чем к таблице, во-вторых, 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 чаще 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 в индексе, а это увеличивает время, требующееся СУБД для проведения таких операций. Поэтому, как правило, не следует индексировать столбцы, данные в которых подвержены частому изменению; столбцы, содержащие большое количество пустых значений; столбцы, содержащие небольшое количество уникальных значений; небольшие таблицы и столбцы 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 снижение 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 (непротиворечивости информации) в базе данных. Обычно различают следующие проблемы, возникающие при наличии 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 некой 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, данную 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 имеют место, 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 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. Очевидно, что аномалии обновления, удаления и ввода крайне нежелательны. Чтобы свести к минимуму возможность появления такого рода аномалий, выполняется 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 (1 Normal Form, 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 свойств 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, если 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 их должностях и зарплате вынесем в другую таблицу, которой присвоим имя СОТРУДНИКИ. Схема приведения таблицы ко второй нормальной форме приведена на рис. 4.3.

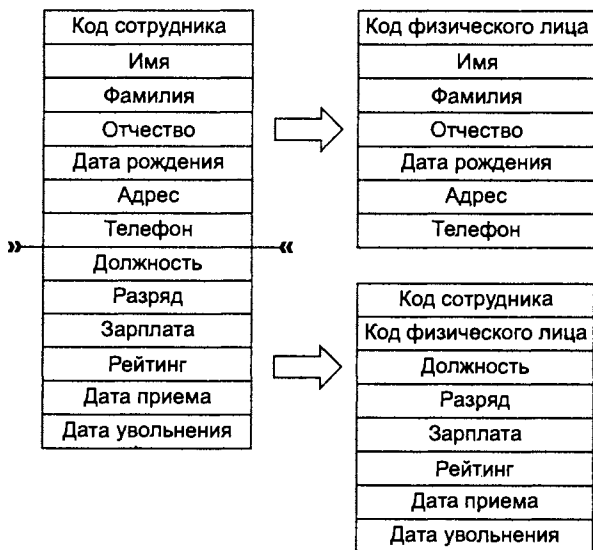


Рис. 4.3. Приведение таблицы ко второй нормальной форме

Полученные две таблицы связаны между собой по полю Код физического лица, которое является первичным ключом для таблицы ФИЗИЧЕСКИЕ ЛИЦА и внешним ключом

чом 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 функциональная 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, 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 в 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).

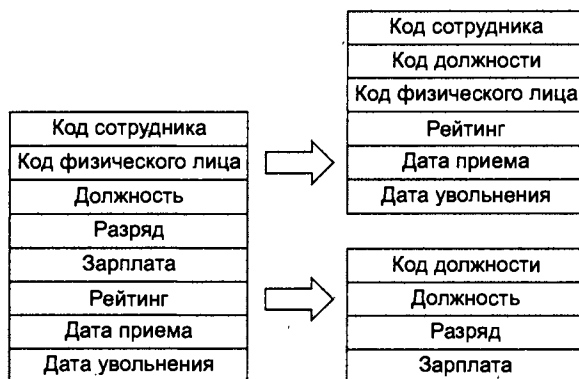


Рис. 4.4. Приведение базы данных к третьей нормальной форме

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.

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).

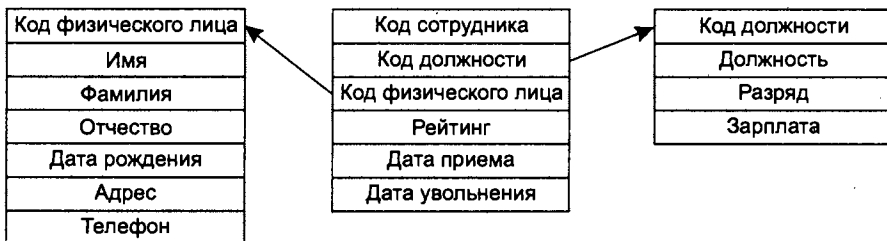


Рис. 4.5. Структура базы данных, приведенной к третьей нормальной форме

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, помимо средств построения запросов, он включает в себя и все необходимые средства управления 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 70-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 — английский язык структурированных запросов). Исходное название только частично отражало суть этого языка. Несмотря на то что язык был ориентирован главным образом на удобную и понятную англоязычным пользователям формулировку

ку запросов к реляционной базе данных, он уже являлся полноценным языком реляционной базы данных, содержащим, помимо операторов формулирования запросов и манипулирования базой данных, средства:

- ☐ для определения схемы базы данных и манипулирования ею;
- ☐ для определения ограничений целостности и триггеров;
- ☐ для создания представлений базы данных;
- ☐ для определения структур физического уровня, поддерживающих эффективное выполнение запросов;
- ☐ для авторизации доступа к отношениям и их полям;
- ☐ для поддержания точек сохранения и откатов транзакций.

В конце 70-х годов модифицированный вариант языка SEQUEL, получивший название SQL, был выпущен корпорацией Oracle в качестве языка коммерческой системы управления базами данных. В 1983 г. компания IBM выпустила SQL в качестве языка управления СУБД DB2.

Американский национальный институт стандартов (ANSI) принял язык SQL в качестве стандарта в 1986 г. С тех пор этот стандарт пересматривался два раза — в 1989 г. были внесены некоторые незначительные изменения, а в 1992 г. 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 — это стандарт на язык, а не сам язык. 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 возможно 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:

- ☐ команды языка DDL (Data Definition Language — язык определения данных) используются для создания и изменения структуры объектов базы данных (например, для создания и удаления таблиц);
- ☐ команды языка DML (Data Manipulation Language — язык манипулирования данными) служат для манипулирования информацией, содержащейся в объектах базы данных;
- ☐ команды языка DCL (Data Control Language — язык управления данными) предназначены для управления доступом к информации, хранящейся в базе данных;

- ☐ команды языка DQL (Data Query Language — язык запросов к данным), используемые наиболее часто, предназначены для формирования запросов к базе данных (*запрос* — это обращение к базе данных для получения соответствующей информации);
- ☐ команды администрирования базы данных предназначены для контроля за выполняемыми действиями и анализа производимых операций;
- ☐ команды управления транзакциями.

ПРИМЕЧАНИЕ

Язык запросов к данным (DQL) будет рассматриваться в главе 11 на примере использования в приложениях Delphi.

Типы данных SQL/92

Типы данных, используемые в стандартном языке SQL, можно подразделить на следующие группы:

- ☐ строковые типы;
- ☐ числовые типы;
- ☐ типы для представления даты и времени.

Рассмотрим эти типы данных.

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)

Здесь 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)`

Здесь n — число, определяющее максимально возможную длину строки.

В отличие от типа `CHARACTER`, тип `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;
- ☐ на вещественные типы с фиксированной точкой;
- ☐ на вещественные типы с плавающей точкой;
- ☐ на двоичные строки фиксированной и переменной длины.

Целочисленные типы

Стандартом ANSI SQL-92 устанавливаются два целочисленных типа:

- ☐ `INTEGER` — целое число со знаком размером 4 байта, может представлять числа в диапазоне от $-2\,147\,483\,648$ до $2\,147\,483\,647$;
- ☐ `SMALLINT` — короткое целое число со знаком размером 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 точкой в двоичной форме (например, при хранении значений денежных величин). Вещественные типы с фиксированной точкой, по сути, являются целочисленными типами, в которых отображается десятичная точка.

Синтаксис объявления типа с фиксированной точкой следующий:

`DECIMAL (n,m)`

Здесь n — точность (это общая длина числового значения), m — 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 использовании этих типов 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 внести серьезную ошибку, например, при суммировании большого количества значений. Поэтому типы с плавающей точкой не подходят для хранения 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)

Здесь *n* — длина строки в байтах.

Объявление строк переменной длины выглядит так:

BIT VARYING (n)

Здесь *n* — максимальная длина строки в байтах.

Типы для представления даты и времени

Очевидно, что типы для представления даты и времени используются для хранения информации, относящейся к датам и времени.

ПРИМЕЧАНИЕ

Иногда типы данных, предназначенные для хранения даты и времени, называются темпоральными.

В стандарте SQL определены следующие типы данных для хранения информации о дате и времени:

- ❑ **DATE** — хранение даты;
- ❑ **TIME** — хранение времени;
- ❑ **TIMESTAMP** — хранение даты и времени;
- ❑ **INTERVAL** — хранение промежутка 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 помощью 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 DDL-команд 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. Синтаксис этого оператора:

```
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 русскоязычные 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 допускает такую возможность, желательно все же в именах объектов базы данных ограничиваться только латинскими символами, особенно если для создания интерфейсной части информационной системы предполагается задействовать средства разработки сторонних производителей. Это позволит избежать целого ряда трудноразрешимых проблем.

Оператор ALTER TABLE

Созданная таблица может быть модифицирована с использованием оператора ALTER TABLE, который позволяет добавлять и удалять поля таблицы, изменять тип данных полей, добавлять и удалять ограничения.

ПРИМЕЧАНИЕ

Оператор ALTER TABLE не определен в стандарте ANSI, однако поддерживается в большинстве реализаций SQL, обеспечивая существенно большую гибкость в управлении структурой базы данных. Если же используемая СУБД не поддерживает этот оператор, то можно просто создать новую таблицу с измененной структурой и затем перенести в нее данные из старой таблицы, после чего старую таблицу 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 — изменяется определение поля;
- ❑ ADD — новое поле добавляется в таблицу;
- ❑ DROP — поле удаляется 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 диско-

вого пространства расходуется впустую. Целесообразнее применить для этого поля тип данных **VARCHAR**. Для изменения типа данных вызовом оператора **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.

Оператор DROP TABLE

Для удаления таблиц используется оператор **DROP TABLE**. Синтаксис этого оператора:

```
DROP TABLE имя_таблицы [RESTRICT | CASCADE]
```

Если при вызове оператора **DROP TABLE** указывается ключевое слово **RESTRICT** и 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 параллельно с удалением таблицы будут удалены все ссылающиеся на нее представления и ограничения.

Задание ограничений

Ограничения позволяют 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 ограничения **NOT NULL** запрещается ввод значений **NULL** в это поле.

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 пробел), 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 его после 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))
```

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.

1. Добавьте в таблицу поле без ограничения NOT NULL.
2. Заполните значения нового поля для всех существующих записей.
3. Измените 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.

- 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, что указание ограничения NOT NULL для поля, являющегося первичным ключом, является обязательным.

- ❑ 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 перечислить 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 содержащиеся в нем значения должны быть уникальными. Однако в отличие от первичного ключа, ограничение UNIQUE допускает наличие в поле пустых значений (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_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 группы полей уникальной — не то же самое, что объявление уникальными нескольких отдельных полей, так как уникальной обязана быть именно комбинация значений, а не просто индивидуальные значения. То есть значение каждого поля, входящего в группу, необязательно должно быть уникальным, а комбинация значений полей всегда должна быть уникальной.

Ограничение **UNIQUE** для группы полей, так же как и составной первичных 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 лежит в основе 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 ключей обязательно должны быть идентичными. А вот имена полей могут быть разными. Однако во избежание путаницы желательно и имена полей для внешнего и родительского ключей делать одинаковыми.

Внешний ключ необязательно 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 во внешнем ключе 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.

Ограничение внешнего ключа (**FOREIGN KEY**) может быть задано либо в операторе **CREATE TABLE**, либо в операторе **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)
```

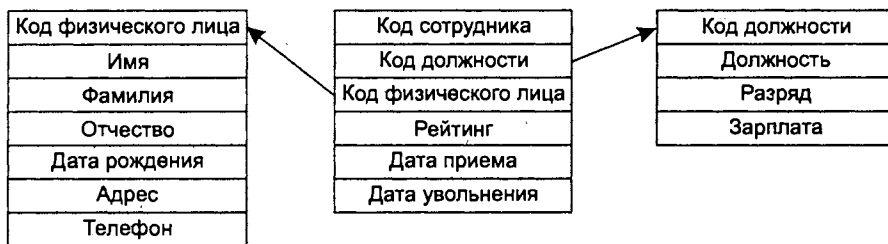


Рис. 5.1. База данных сотрудников организации

Первый список полей — это список из одного или нескольких полей таблицы, разделенных запятыми. Второй список полей — это список полей, которые будут составлять родительский ключ. Списки полей, указываемые в качестве внешнего и родительского ключей, должны быть совместимы:

- ☐ число полей должно быть одинаковым;
- ☐ порядок следования полей в списках должен совпадать, причем совпадение определяется не именами полей, которые могут быть разными, 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** для создания внешнего ключа:

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 данные о новом сотруднике, 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 таблицы ФИЗИЧЕСКИЕ ЛИЦА и 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. 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 в стандарте ANSI SQL-92 не прописаны.

Один из синтаксических вариантов задания каскадного обновления и удаления выглядит так:

```
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. Пример:

```
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 поле или вставить значение в поле, которое могло бы сделать этот предикат неверным, будет отклонена.

ПРИМЕЧАНИЕ

Проверка корректности значений, заносимых в базу данных, может также выполняться в пользовательских приложениях. Однако ограничение **CHECK** дает 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))  
...
```

Значения 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, указываемая 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,  
...)
```

Индексы

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 значительно 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 самым распространенным. Он состоит только из одного поля (столбца) таблицы, поэтому его часто называют *одно столбцовым*.

Наиболее типичный синтаксис команды создания простого индекса:

```
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:

```
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 выглядит следующим образом:

```
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 оператора удаления индекса:

```
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 повышения 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.

Представления

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 также приводит к изменению данных в таблицах, на основе которых это представление создано. На рис. 5.2 схематично поясняется процесс формирования представления.

Использование представлений незначительно отличается от использования таблиц. Выборка данных из представления выполняется точно так же, как и из обыч-

ной таблицы. Допускаются также операции манипулирования данными представления, хотя здесь имеются некоторые ограничения.



Рис. 5.2. Схема формирования представления

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 позволяют 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, с одной стороны, смогут выполнять свои служебные обязанности в полном объеме, с другой стороны, 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 примере с работниками отдела кадров можно при создании представления задать условие, исключающее из представления сотрудников, занимающих определенные должности.

ПРИМЕЧАНИЕ

Как уже отмечалось, представления строятся на основе запросов к базе данных, которые будут рассмотрены в главе 11, посвященной подмножеству DQL-команд языка SQL. Поэтому здесь мы приведем лишь общие сведения о представлениях, не вдаваясь в подробности формирования запросов.

Представления также используются для формирования итоговых результатов при подготовке отчетов. В том случае когда требуется часто распечатывать отчет, создаваемый 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 потребуются 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.

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 DDL-команд) 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. В этом случае при указании параметра `RESTRICT` появится сообщение об ошибке и удаление выполнено не будет. Если же

указать параметр **CASCADE**, то выполнение оператора **DROP VIEW** приведет к удалению всех базовых представлений и ограничений.

Типовой синтаксис оператора **DROP VIEW**:

```
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 procedures) представляют собой группы связанных SQL-операторов. Использование хранимых процедур обеспечивает дополнительную гибкость при работе с базой данных, так как выполнить хранимую процедуру обычно гораздо проще, чем последовательность отдельных SQL-операторов.

Хранимые процедуры хранятся в базе данных в откомпилированном виде, что обеспечивает более высокую скорость их выполнения.

Хранимые процедуры могут получать входные параметры, возвращать значения приложению и явно вызываться из приложения или подставляться вместо имени таблицы в инструкции **SELECT**.

Основные преимущества, которые дает использование хранимых процедур:

- ☐ хранимые процедуры позволяют вынести часть логики на сервер базы данных, что ослабляет зависимость базы данных информационной системы от клиентской части;
- ☐ хранимые процедуры обеспечивают модульность проекта, так как 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, поскольку 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, так как выполняются сервером, а не клиентом, что снижает сетевой трафик;
- ☐ скорость выполнения хранимых процедур выше, чем последовательности отдельных SQL-операторов, так как хранимые процедуры хранятся на сервере в откомпилированном виде.

Различают два вида хранимых процедур.

- ☐ *Процедуры выбора* 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* вызываются явно с использованием специального оператора. Выполняемая процедура может не возвращать результата вызываемой программе.

Создание хранимых процедур

Для создания хранимых процедур применяется оператор `CREATE PROCEDURE`. Оператор `CREATE PROCEDURE` определяет новую хранимую процедуру в базе данных. Хотя его синтаксис зависит от реализации SQL, как правило, в него могут входить все SQL-инструкции для манипулирования данными и ряд расширений, в том числе:

- ❑ условные операторы;
- ❑ различные виды операторов цикла;
- ❑ средства 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, 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).

Для вызова выполняемой процедуры следует применять 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 хранимых процедур, выполнение триггера происходит не в результате явного вызова некоторого SQL-оператора, 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 возможности:

- ☐ контроля вводимых данных, гарантирующего, что пользователь вводит в поля таблицы только допустимые значения;
- ☐ упрощения сопровождения приложений, так как изменение в триггере автоматически отражается во всех приложениях, которые используют таблицы со связанными с ними триггерами;
- ☐ автоматического документирования изменений таблицы (приложение может управлять журналом изменений с помощью триггеров, которые выполняются 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. Заголовок 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 получают права 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 SQL-операторов, выделяемая в качестве команд языка DML. С помощью DML-операторов пользователь может загружать в таблицы новые данные, модифицировать и удалять существующие данные.

В языке SQL определены только три основных DML-оператора:

- ❑ INSERT — добавление в таблицу новой информации;
- ❑ UPDATE — изменение данных, хранящихся в таблице;
- ❑ DELETE — удаление данных из таблицы.

Добавление в таблицу новой информации

Процесс ввода в таблицу базы данных новой информации обычно называется *загрузкой данных*. Для загрузки данных используется оператор INSERT.

Добавление в таблицу новой записи

Для добавления в таблицу 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. Пусть таблица `ДОЛЖНОСТИ` создана с помощью 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, Имя,  
    Фамилия, 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 списку полей.

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 в главе 11.

Объединяя операторы **INSERT** и **SELECT**, можно добавить в таблицу данные, полученные в результате выполнения запроса к другой таблице (к другим таблицам). Синтаксис оператора **INSERT** в этом случае будет следующим:

```
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 слова **SELECT** указать символ *****, то в выборку будут включены все поля). Предложение **FROM** используется для задания имени таблицы, из которой производится выборка данных. Предложение **WHERE** является необязательным и требуется 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 новыми. Оператор **UPDATE** может быть применен как к одному полю таблицы (наиболее частый случай), так и к нескольким полям. Количество изменяемых записей зависит от потребностей пользователя — с помощью оператора **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**:

```
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 модифицируемых записей.

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.

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 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 изменить 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:


```
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.

Безопасность базы данных

Одной из важнейших задач управления базами данных является обеспечение безопасности данных, то есть защиты данных от несанкционированного доступа.

ПРИМЕЧАНИЕ

Под несанкционированным 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 привилегий.

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 такие привилегии, как право:

- ☐ на создание таблицы;
- ☐ на создание представления;
- ☐ на создание хранимой процедуры;
- ☐ на удаление таблицы;
- ☐ на удаление представления;
- ☐ на 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 выполнения определенных действий над объектами базы данных пользователь должен иметь соответствующие права.

Стандартом ANSI предусмотрены следующие объектные привилегии:

- ☐ **SELECT** — разрешается производить выборку данных из указанной таблицы (представления);
- ☐ **INSERT(имя_поля)** — разрешается выполнять добавление данных в определенное поле указанной таблицы (представления);
- ☐ **INSERT** — разрешается добавление данных во все поля указанной таблицы (представления);
- ☐ **UPDATE(имя_поля)** — разрешается модифицировать данные в заданном поле указанной таблицы (представления);
- ☐ **UPDATE** — разрешается модифицировать данные во всех полях указанной таблицы (представления);
- ☐ **REFERENCE(имя_поля)** — разрешается ссылаться на заданное поле указанной таблицы (эта привилегия требуется при установке любых ограничений целостности);
- ☐ **REFERENCE** — разрешается ссылаться на все поля указанной таблицы.

ПРИМЕЧАНИЕ

Помимо 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** служит для предоставления пользователю как системных, так и объектных привилегий. Синтаксис данного оператора:

```
GRANT привилегия_1 [. привилегия_2]
ON имя_объекта
TO имя_пользователя [WITH GRANT OPTION]
```

Предоставление пользователю с именем **USER** права на выборку данных из таблицы **СОТРУДНИКИ** выполняется с помощью следующего оператора:

```
GRANT SELECT
ON Сотрудники
TO USER
```

С помощью одного оператора GRANT можно задать 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, которому 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, помимо 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 служит для отмены предоставленных пользователю привилегий. Данный оператор может вызываться с одним из двух параметров — RESTRICT или CASCADE. При указании параметра RESTRICT оператор REVOKE будет успешно выполнен только в том случае, если это 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 указании параметра 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 у пользователя USER1 в режиме CASCADE приведет к отмене привилегий и у пользователя USER2.

Синтаксис оператора REVOKE:

```
REVOKE привилегия_1 [, привилегия_2]  
ON имя_объекта  
FROM имя_пользователя [RESTRICT | CASCADE]
```

Например, для отмены права добавления данных в таблицу СОТРУДНИКИ пользователем USER требуется следующий оператор:

```
REVOKE INSERT  
ON Сотрудники  
FROM USER
```

ГЛАВА 6 Проектирование структуры базы данных

В предыдущей главе были рассмотрены вопросы создания базы данных и входящих в нее таблиц с помощью SQL-конструкций. Этот подход можно применить при конструировании небольших информационных систем, но разработка больших баз данных, содержащих сотни и тысячи таблиц и сложные связи между ними, возможна только с помощью CASE-средств. Вручную очень трудно разработать и представить графически 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 разных приложениях показывает, что реляционная модель данных достаточна для моделирования предметных областей. Однако проектирование реляционной базы данных в плане соответствия отношениям нормализации (см. главу 4) часто представляет собой очень сложный для проектировщика процесс. Это обусловлено некоторой ограниченностью реляционной модели данных, что особенно ярко проявляется в нескольких аспектах.

- ❑ Реляционная модель не предоставляет достаточных средств для раскрытия смысла данных. Проектировщик должен независимым от модели образом представлять семантику реальной предметной области. Примером 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.
- ❑ Хотя 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:

- ❑ в *объектно-ориентированных моделях* сущности реального мира представляются в виде объектов, а не записей реляционных таблиц;
- ❑ *семантические модели* отражают 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, значение семантических моделей не снижается. 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 проще 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 предоставление 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 не-

однозначности 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 П. Ченом (P. Chen) в 1976 г. 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. В настоящий момент общепринятым стандартом описания информационных систем является язык UML (см. главу 3), но часто в разработках используются диаграммы, выполненные в соответствии со стандартом IDEFX, на который ориентированы популярные CASE-системы, в частности, ERwin, Design/IDEF, Power Designer.

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).

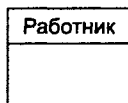


Рис. 6.1. Отображение сущности

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 только 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, обозначающего сущность (рис. 6.3). В реляционных базах данных аналогом атрибута является поле таблицы.

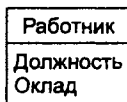


Рис. 6.3. Атрибуты сущности

Общие сведения о CASE-средствах

За последнее десятилетие в области технических средств программирования сформировалось новое направление — *CASE-технология* (аббревиатура CASE расшифровывается как Computer-Aided Software/System Engineering). *CASE-технология* представляет собой совокупность методологий анализа, проектирования, разработки и сопровождения сложных систем и поддерживается комплексом взаимосвязанных средств автоматизации.

Для методологий структурного анализа характерен ряд ограничений (сложность понимания, трудоемкость и высокая стоимость использования, неудобство внесения изменений в проектные спецификации и т. д.). CASE-технологии с самого на-

чала развивались именно с целью преодоления этих ограничений путем автоматизации процессов анализа и интеграции поддерживающих средств.

Современные CASE-средства поддерживают разнообразные технологии проектирования информационных систем: от простых средств анализа и документирования до полномасштабных средств автоматизации, охватывающих 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 важную 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 некоторыми весьма характерными особенностями.

- ❑ *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 двухмерными 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 простоту сопровождения и внесения изменений в систему.
- ❑ *Единая база данных проекта.* Основа CASE-технологии — использование базы данных проекта (репозитария) для хранения всей информации о проекте, которая может совместно использоваться разработчиками в соответствии с их правами доступа. Содержимое репозитария включает не только информационные объекты различных типов, но и отношения между их компонентами, а также правила применения или обработки этих компонентов. Репозитарий может хранить объекты различных типов: структурные диаграммы, определения экранов и меню, проекты отчетов, описания данных и логики их обработки, а также модели данных, организации и обработки, исходные коды, элементы данных и т. п.

ПРИМЕЧАНИЕ

В области проектирования информационных систем для обозначения понятия «хранилище» используют термин «репозитарий» (от английского *reposition*), в отличие от остальных видов человеческой деятельности, где принят термин «депозитарий» (от латинского *depositum*).

- ❑ *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 разработку проекта, 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 его устраивает и насколько она 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 программ-

ного обеспечения и обладающий следующими основными характерными особенностями:

- ❑ наличием мощных графических средств описания и документирования информационной системы, обеспечивающих удобный интерфейс с разработчиком и развивающих его творческие возможности;
- ❑ интеграцией отдельных компонентов CASE-средств, обеспечивающей управляемость процесса разработки информационной системы;
- ❑ использованием специальным образом организованного хранилища проектных метаданных (репозитария).

Создание концептуальной модели информационной системы

Рассмотрим создание модели информационной системы. Для этого используем базу данных Премьер в качестве примера и популярную систему моделирования данных Power Designer фирмы Sybase в качестве CASE-средства.

База данных Премьер

Строительная компания «Премьер» возводит различные здания. Для всех зданий требуются разнообразные материалы в различных количествах. На разных этапах реализации 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 бригад 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, а на рис. 6.4, б — количество материала конкретного типа, использованного в здании. 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.

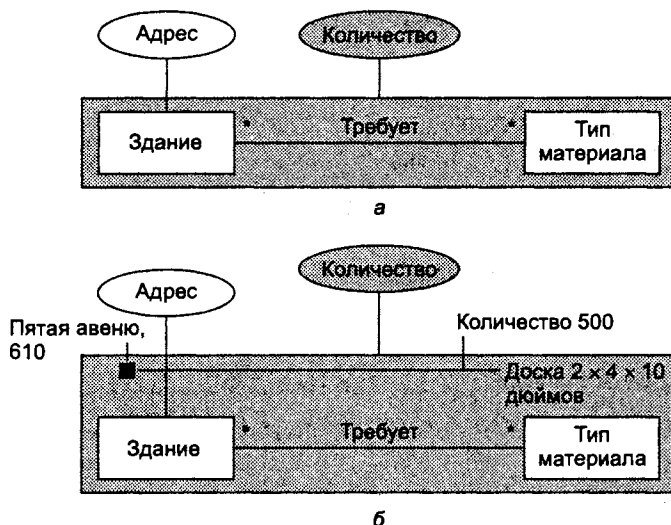


Рис. 6.4. Моделирование отношений между зданиями и материалами

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** выбираются дни работы. Например, бригаде штукатуров требуется несколько дней, чтобы оштукатурить данное здание. Таким образом, у нас есть отношение «многие ко многим» **РАБОТАЕТ-ТОГДА-ТО** между объектами **БРИГАДА** и **ДАТА**.

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.

На рис. 6.7 представлена объединенная диаграмма, представляющая полную модель данных для строительной компании «Премьер».

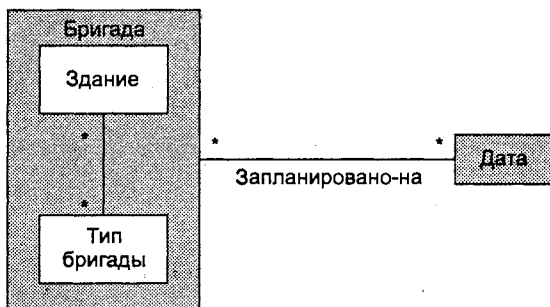


Рис. 6.5. Модель формирования бригад



Рис. 6.6. Назначение рабочих в бригады

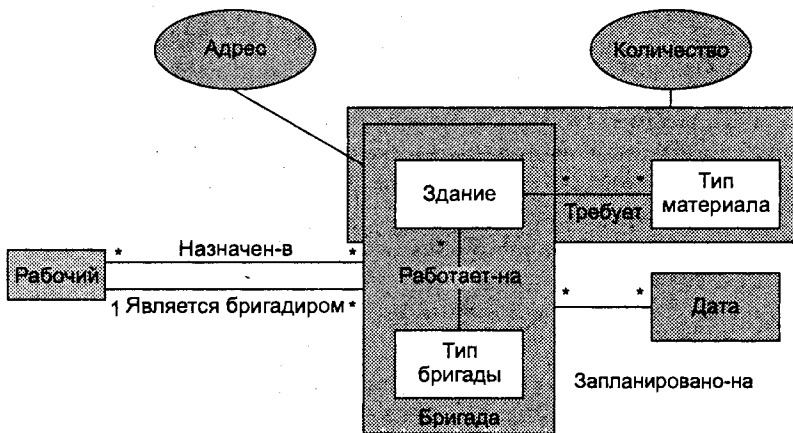


Рис. 6.7. Модель данных для строительной компании «Премьер»

Создание нового проекта в Power Designer

Для создания концептуальной модели базы данных запустите программу Power Designer и выберите в меню File команду New. Откроется основное окно программы, которое содержит область отображения модели, меню, панель инструментов и панель элементов модели (рис. 6.8).

Прежде всего определим свойства создаваемой модели, которые используются для ее идентификации, описания и отображения в отчетах по модели. Для этого выполните команду Dictionary ► Model Properties. Откроется окно диалога Model Properties (рис. 6.9). Задайте в нем наименование и идентификатор проекта, в рамках которого создается данная модель, а также наименование и идентификатор самой модели. Кроме того, вы можете указать автора модели, используемый язык, версию модели, ввести краткое и подробное описание, аннотацию.

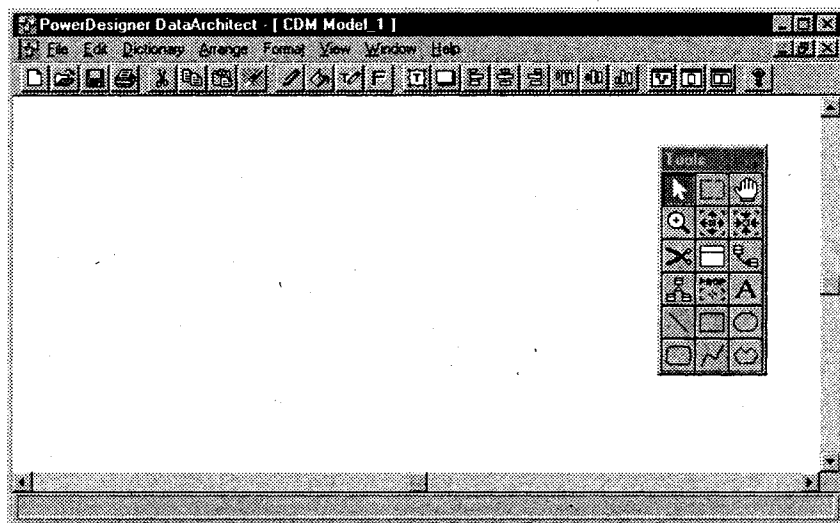


Рис. 6.8. Основное окно программы Power Designer

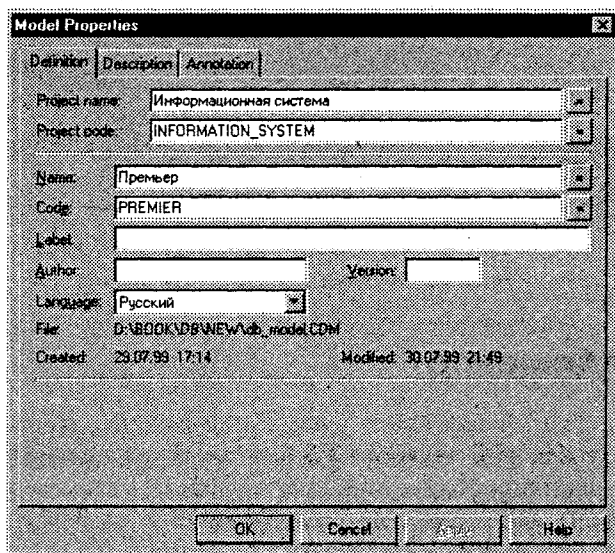


Рис. 6.9. Определение свойств модели

Создание сущностей

Для создания сущности перетащите мышью с панели элементов в область модели значок с изображением прямоугольника, содержащего в верхней части горизонтальную линию. Будет создан прямоугольник для обозначения новой сущности, которая пока содержит только наименование. Для определения свойств сущности сделайте двойной щелчок на прямоугольнике. Откроется окно диалога *Entity Properties*. Перейдите на вкладку *Definition* и введите наименование, идентификатор и краткое описание сущности (рис. 6.10). Подробное описание вводится на вкладке *Description*. При совместной разработке модели информационной системы вкладка *Annotation* может использоваться для ввода замечаний и комментариев по поводу сущности. При щелчке на кнопке *Attributes*, расположенной на вкладке *Description*, открывается окно диалога ввода атрибутов сущности. Для определения бизнес-правил сущности щелкните на кнопке *Rules* и в открывшемся окне диалога выберите одно из ранее созданных правил.

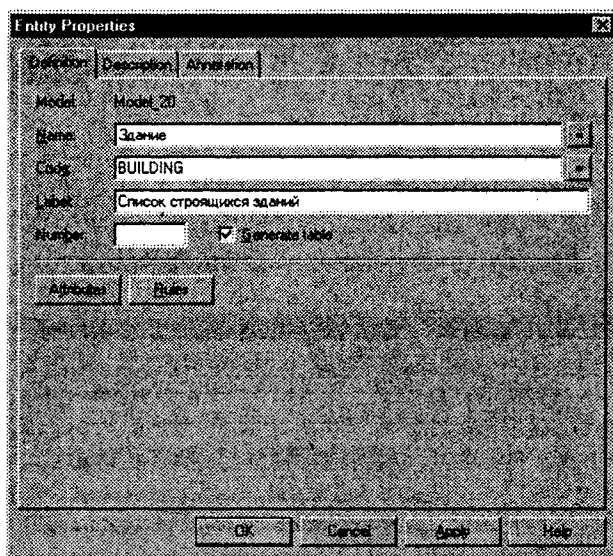


Рис. 6.10. Определение свойств сущности

Создание доменов

Прежде чем перейти к непосредственному определению атрибутов сущностей, познакомимся с созданием доменов. Домены являются аналогами пользовательских типов в реляционных базах данных и позволяют указывать типы атрибутов сущностей. Для создания домена выполните команду *Dictionary ► List of Domains*. Откроется окно диалога *List of Domains*, которое содержит таблицу со списком доменов модели (рис. 6.11).

Рассмотрим создание домена идентификаторов элементов списка, который будет использоваться при определении таких атрибутов, как **WORKER_ID** (идентификатор работника) или **BLDG_ID** (идентификатор здания). Тип данных создаваемого домена — четырехзначное число (фактически, это подтип стандартного числового типа данных **Number**) и его значение по умолчанию равно нулю. Для создания нового домена щелкните на кнопке **New** и введите в столбцы **Name** и **Code** таблицы наименование и идентификатор домена. Для определения типа данных перейдите в столбец **Data Type** и щелкните на кнопке с многоточием, расположенной с правого края ячейки. Откроется окно диалога **Standard Data Types**, в котором можно выбрать требуемый тип среди большого количества базовых типов данных. В данном случае необходимо выбрать тип **Number** и задать в поле **Length** длину 4.

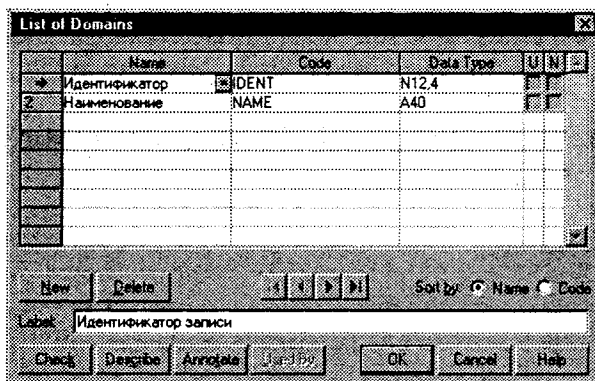


Рис. 6.11. Окно диалога со списком доменов

Для определения значения по умолчанию щелкните на кнопке **Check**. Откроется окно диалога **Check Parameters** (рис. 6.12). На отображаемой по умолчанию вкладке **Standard Parameters** в полях области **Values** определите минимальное и максимальное значения, а также значение по умолчанию. Здесь же вы можете задать формат, единицу измерения и список допустимых значений домена. Для определения бизнес-правил домена щелкните на кнопке **Rules** и выберите в открывшемся окне диалога одно из ранее созданных правил.

Для ввода подробного описания и аннотации щелкните в окне диалога **List of Domains** на кнопках **Describe** и **Annotation** соответственно. Для каждой из них будет открыто соответствующее окно диалога, содержащее поля для ввода текста.

Определение атрибутов сущностей

Для определения атрибутов сущности откройте окно ее свойств и щелкните на кнопке **Attributes**, расположенной на вкладке **Description**. Откроется окно диалога ввода атрибутов сущности (рис. 6.13).

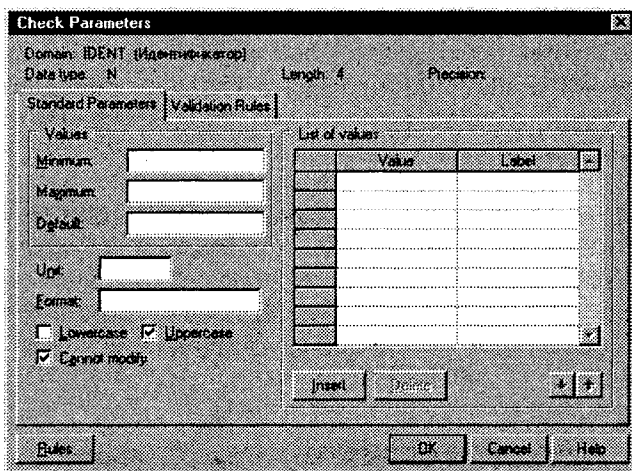


Рис. 6.12. Определение ограничений для домена

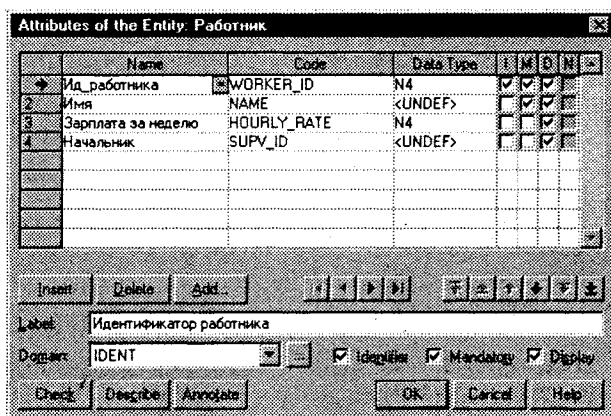


Рис. 6.13. Определение атрибутов сущности

В таблице для каждого атрибута задаются имя, идентификатор и тип данных, который может быть одним из базовых типов или доменом, созданным пользователем. В этой же таблице устанавливаются флажок идентифицирующего атрибута и признак запрета пустого значения. Для ввода относящихся к атрибуту ограничений щелкните на кнопке Check и задайте требуемые значения в окне диалога Check Parameters. Для сущности WORKER первые три атрибута имеют идентификаторы WORKER_ID (идентификатор работника), NAME (имя) и HOURLY_RATE (недельная зарплата) соответственно. Для атрибута WORKER_ID выберите в списке Domain созданный ранее домен IDENT. При этом автоматически определяются тип данных и значение по умолчанию. Атрибут WORKER_ID является идентифицирующим атрибутом, так как однозначно определяет работника, поэтому для него установите флажок в столбце I. Обратите внимание, что при этом автоматически устанавливается флажок запрета пустых значений.

Для атрибутов **BLDG_TYPE** (тип здания) и **STATUS** (статус) сущности **BUILDING** (здание) необходимо определить значения по умолчанию: 'Офис' и 1 соответственно. Кроме того, они имеют ограничения на значения атрибутов. Атрибут **BLDG_TYPE** может содержать только значения из множества ('Офис', 'Склад', 'Магазин', 'Жилой дом') (рис. 6.14), а значение столбца **STATUS** может быть от 1 до 3.

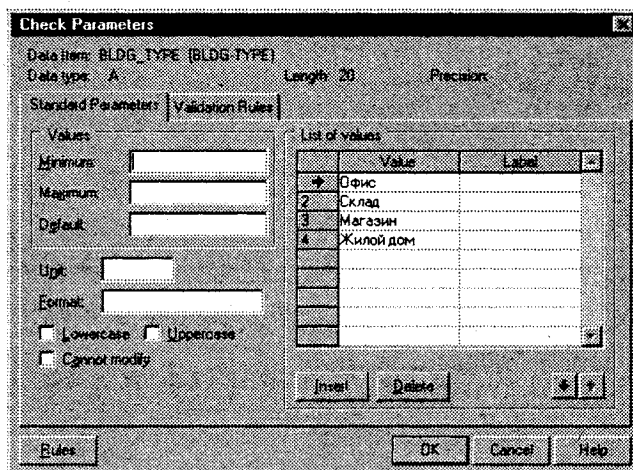


Рис. 6.14. Определение списка допустимых значений

Определение связей между сущностями

Для создания связи между двумя сущностями выполните следующие действия.

- ☐ Выберите на панели элементов кнопку, на которой показаны два прямоугольника, соединенные линией.
- ☐ Соедините линией две сущности.

В модели появится связь между выбранными сущностями, которой по умолчанию присваивается имя **Relation_n**, где **n** — порядковый номер создаваемой связи. Для определения свойств созданной связи сделайте на ней двойной щелчок мышью. Откроется окно свойств связи (рис. 6.15), в верхней части которого расположены графическое отображение связи и кнопки с наименованиями соединяемых сущностей. Введите в поля **Name**, **Code** и **Label** наименование связи, ее идентификатор и краткое описание. Затем задайте с помощью группы переключателей **Cardinality** тип связи между сущностями: «один к одному», «один ко многим», «многие к одному» или «многие ко многим».

В расположенных ниже двух областях для каждой сущности связи задаются обязательность, мощность связи и зависимость. Степень связности, то есть количество связываемых экземпляров данной сущности, указывается в полях **Min** и **Max**. Свойство **Mandatory** определяет, является ли связь обязательной, и в зависимости от типа связи отображается на линии в виде значков, перечисленных в табл. 6.1.

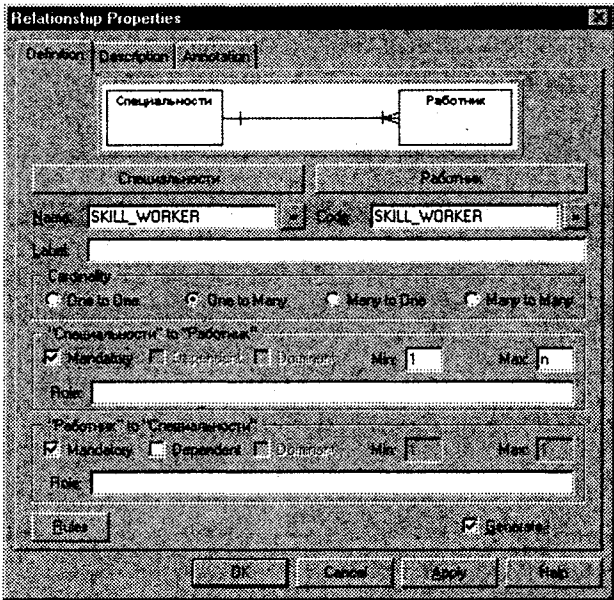


Рис. 6.15. Определение свойств связи

Таблица 6.1. Значки, идентифицирующие тип связи

Флажок Mandatory	Тип отношения	Описание	Значок
Установлен	Один	Должен существовать один и только один элемент	
Установлен	Многие	Должны существовать один или несколько элементов	
Снят	Один	Может существовать один элемент или не быть ни одного	
Снят	Многие	Может существовать один элемент, несколько или не быть ни одного	

При определении зависимой связи для идентификации сущности используются идентифицирующие атрибуты связанной сущности. Например, для сущности ASSIGNMENT обе связи с сущностями WORKER и BUILDING являются зависимыми. Каждый элемент сущности ASSIGNMENT однозначно определяется совокупностью идентифицирующих атрибутов сущностей WORKER и BUILDING. Для установки признака зависимой связи используется флажок Dependent.

Для отношения типа «один к одному» можно установить флажок Dominant, который указывает родительскую сущность.

На рис. 6.16 представлен результат создания концептуальной модели информационной системы «Премьер».

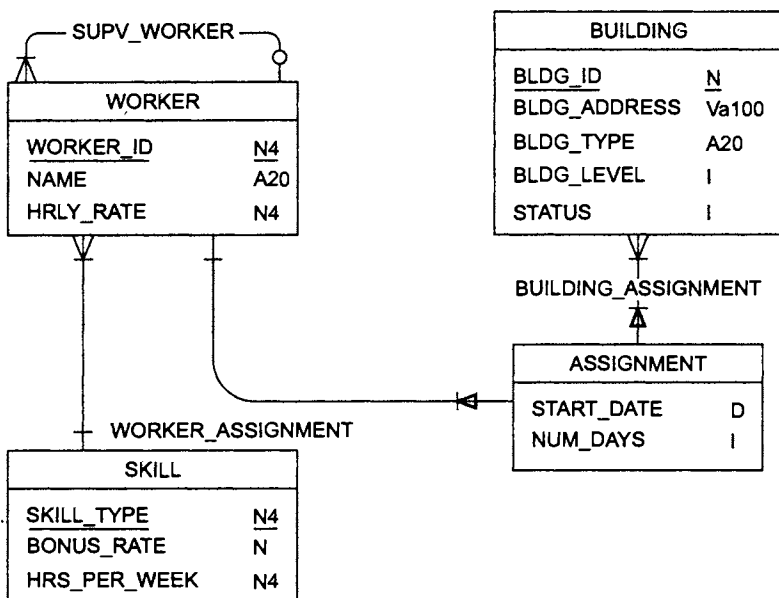


Рис. 6.16. Концептуальная модель информационной системы «Премьер»

Проверка модели

При использовании CASE-средств можно в любой момент проверить созданную модель на наличие ошибок. Для этого выполните команду Dictionary ► Check Model и установите в открывшемся окне диалога флажки проверки сущностей, атрибутов и связей. Затем щелкните на кнопке OK для запуска процесса проверки. Ее результат будет отображаться в окне Check Model Messages (рис. 6.17).

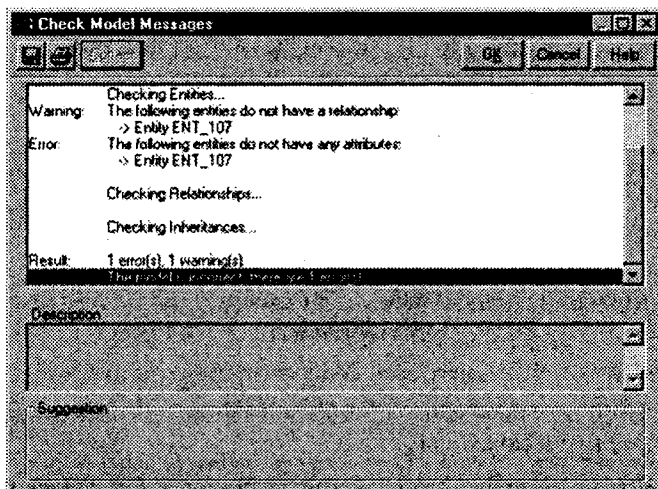


Рис. 6.17. Отображение результатов проверки

Для исправления ошибки дважды щелкните на соответствующей строке в окне сообщений. Откроется связанное с этой ошибкой окно свойств сущности, атрибута или связи.

Документирование модели базы данных

CASE-средства содержат прекрасные инструменты для описания модели базы данных. Во-первых, вы можете распечатать модель в графическом виде. Для этого в Power Designer необходимо выполнить команду **File ▶ Print Graphics** и указать в появившемся окне диалога печатаемые страницы и тип цветной или черно-белой печати.

Во-вторых, с помощью средств создания отчетов можно сформировать полное описание модели, включив в него всю необходимую информацию, которая была введена при проектировании модели. Для создания отчета выполните команду **File ▶ Create Report**. Откроется окно диалога со списком предопределенных типов отчетов. Вы можете выбрать любой из них и создать отчет, а при необходимости модифицировать выбранный тип отчета и сохранить его спецификацию под новым именем. При выборе режима модификации отчета открывается окно настройки (рис. 6.18), в левой части которого находится список возможных атрибутов отчета, а в правой — список атрибутов, выбранных для отображения в отчете.

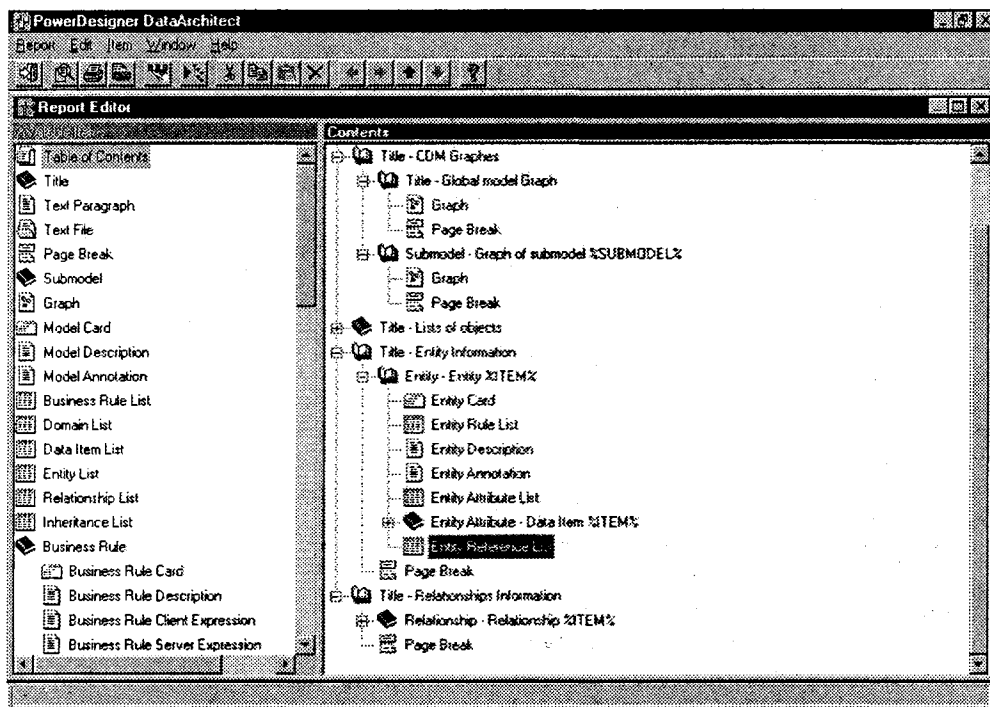


Рис. 6.18. Настройка пользовательского отчета

Путем перетаскивания вы можете выбрать любой элемент в левой части и разместить его в нужном месте шаблона отчета. Для элементов отчета можно устанавливать шрифт и параметры абзаца. Для элементов, являющихся заголовками сущностей, атрибутов и связей, можно изменять отображаемый текст. Для этого в контекстном меню элемента выберите команду Edit и измените шаблон элемента в окне редактирования (рис. 6.19).

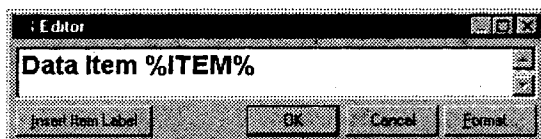


Рис. 6.19. Изменение шаблона элемента

При отображении списков сущностей, атрибутов и связей можно указать перечень отображаемых свойств, порядок их расположения в таблице, а также размеры столбцов в абсолютном и процентном выражении (рис. 6.20).

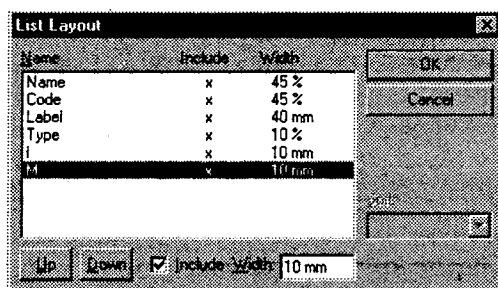


Рис. 6.20. Определение списка отображаемых свойств

Созданный отчет можно вывести на печать или сохранить в формате RTF, чтобы продолжить редактирование в текстовом процессоре.

Создание физической модели

Концептуальная модель позволяет понять суть разрабатываемой 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 сначала общие принципы преобразования:

- ☐ каждая сущность преобразуется в таблицу, и имя сущности становится именем таблицы;
- ☐ каждый атрибут становится столбцом таблицы с тем же именем, уточняется тип данных, выбирается более точный формат;

- ☐ идентифицирующие атрибуты сущности превращаются в первичный ключ таблицы, а 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);
- ☐ отношения «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 отношений «многие ко многим» создается таблица, столбцами которой являются уникальные идентификаторы связываемых сущностей (они составляют первичный ключ).

В Power Designer для преобразования концептуальной модели в физическую выполните команду Dictionary ► Generate Physical Model. Откроется окно диалога Generating Physical Data Model, в котором прежде всего укажите тип СУБД, для которой будет создаваться модель (рис. 6.21). Затем установите флажки добавления подробных описаний и аннотаций, проверки модели перед преобразованием, определите шаблоны для наименования первичных и внешних ключей. Для всех связей, имеющихся в модели, задаются единые правила удаления и изменения. Если в структуре базы данных для разных связей требуются разные правила, вы можете уточнить их в физической модели.

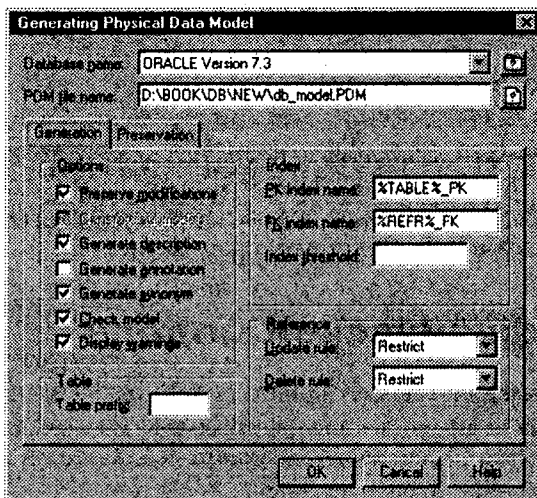


Рис. 6.21. Определение параметров преобразования концептуальной модели в физическую

Щелкните на кнопке OK. Начнется процесс преобразования, после завершения которого созданная модель появится в отдельном окне (рис. 6.22). Вы можете модифицировать физическую модель, распечатывать ее в графическом виде, создавать отчеты.

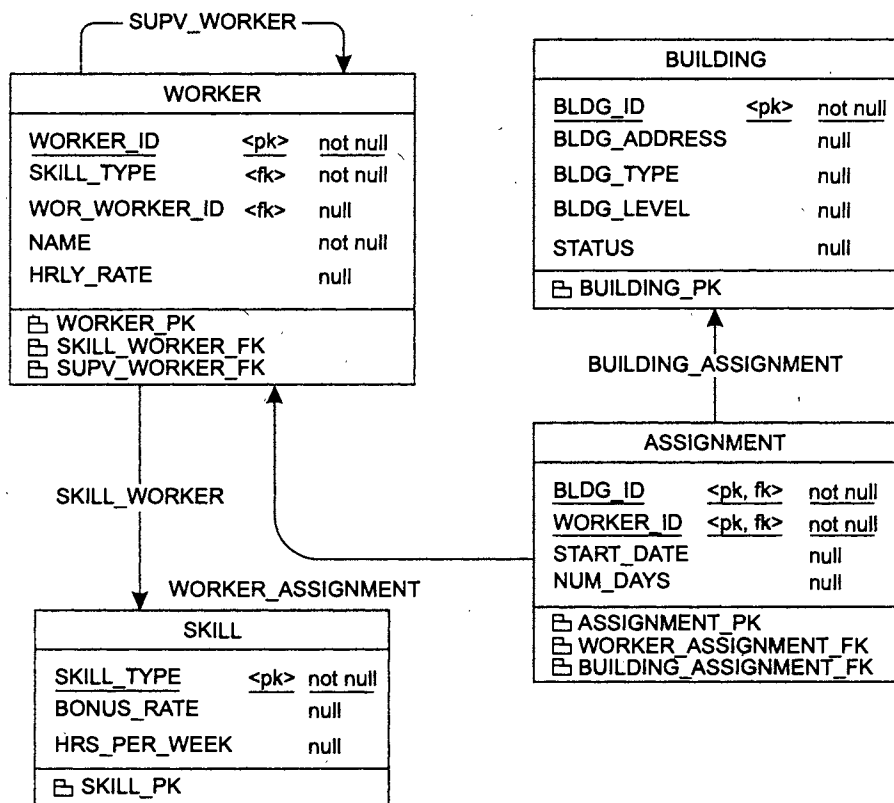


Рис. 6.22. Физическая модель информационной системы «Премьер»

Создание структуры базы данных

После создания физической модели и ее уточнения вы можете создать структуру базы данных с помощью команды Database ► Generate Database. Откроется окно диалога Parameters (рис. 6.23), в котором необходимо установить флажки создания таблиц, индексов, комментариев и т. п.

Для создания структуры базы данных, не выходя из этого окна диалога, щелкните на кнопке Create database. Откроется окно диалога установления соединения с источником данных ODBC; после соединения созданный сценарий будет выполнен сервером базы данных. Однако наиболее часто используется другой путь: с помощью кнопки Generate script создается сценарий, который затем запускается на выполнение средствами сервера базы данных.

Ниже приведен текст сценария создания таблицы BUILDING и комментариев к ней и ее столбцам:

```

create table BUILDING
(
    BLDG_ID NUMBER not null,

```

```

BLDG_ADDRESS VARCHAR2(100) null
BLDG_TYPE CHAR(20) default 'Офис' not null
constraint CKC_BLDG_TYPE_BUILDING
check (BLDG_TYPE in ('Офис','Склад','Магазин','Жилой дом')).
QLTY_LEVEL NUMBER(1) null.
STATUS NUMBER(1) default 1 not null
constraint CKC_STATUS_BUILDING check (STATUS between 1 and 3).
constraint PK_BUILDING primary key (BLDG_ID)
)
/
comment on table BUILDING is 'Список строящихся зданий'
/
comment on column BUILDING.BLDG_ID is 'Идентификатор'
/
comment on column BUILDING.BLDG_ADDRESS is 'Адрес'
/
comment on column BUILDING.BLDG_TYPE is 'Тип здания'
/
comment on column BUILDING.QLTY_LEVEL is 'Уровень качества'
/
comment on column BUILDING.STATUS is 'Статус'
/

```

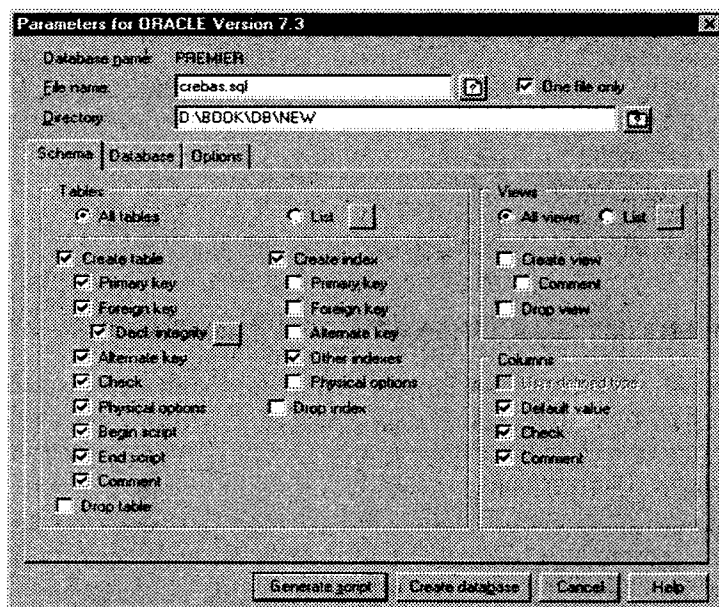


Рис. 6.23. Определение параметров создания структуры базы данных

Модификация структуры базы данных

Жизненный цикл создания и сопровождения информационной системы имеет вид спирали. Это означает, что модификация структуры базы данных практически неизбежна. CASE-средства позволяют несколько упростить поддержку нескольких

версий структуры и автоматизировать создание сценариев изменения структуры базы данных.

Закончив разработку очередной версии модели базы данных, создайте архив физической модели с помощью команды Database ► Archive Model. Затем вы можете модифицировать модель, и после завершения создания очередной версии запустить процесс модификации структуры базы данных с помощью команды Database ► Modify Database. При этом откроется окно диалога Parameters, которое, в отличие от аналогичного окна создания структуры базы данных, содержит поле для ввода имени архивной модели. После задания всех необходимых для модификации параметров щелкните на кнопке Generate script. Power Designer сравнит текущую модель базы данных с архивной моделью и создаст сценарий, содержащий команды модификации структуры базы данных. При этом учитываются особенности выбранной СУБД. Например, в ORACLE 7.3, в отличие от некоторых других систем, отсутствует команда удаления поля таблицы. В этом случае создается временная таблица, в которую переписывается вся информация из модифицируемой таблицы. После этого таблица удаляется и создается вновь — без удаленного столбца. Затем в нее добавляются записи из временной таблицы. Ниже приведен пример удаления столбца QLTY_LEVEL из таблицы BUILDING:

```
alter table ASSIGNMENT
, drop constraint FK_BUILDING_ASSIGNMENT
/
create table tmp_BUILDING
(
    BLDG_ID NUMBER not null,
    BLDG_ADDRESS VARCHAR2(100) null,
    BLDG_TYPE CHAR(20) not null,
    QLTY_LEVEL NUMBER(1) null,
    STATUS NUMBER(1) not null
)
/
insert into tmp_BUILDING (BLDG_ID, BLDG_ADDRESS, BLDG_TYPE, QLTY_LEVEL, STATUS)
select BLDG_ID, BLDG_ADDRESS, BLDG_TYPE, QLTY_LEVEL, STATUS
from BUILDING
/
drop table BUILDING cascade constraints
/
create table BUILDING
(
    BLDG_ID NUMBER not null,
    BLDG_ADDRESS VARCHAR2(100) null,
    BLDG_TYPE CHAR(20) default 'Офис' not null
    constraint CKC_BLDG_TYPE_BUILDING
    check (BLDG_TYPE in ('Офис', 'Склад', 'Магазин', 'Жилой дом')),
    STATUS NUMBER(1) default 1 not null
    constraint CKC_STATUS_BUILDING check (STATUS between 1 and 3),
    constraint PK_BUILDING primary key (BLDG_ID)
)
/
comment on table BUILDING is 'Список строящихся зданий'
/
```

```
comment on column BUILDING.BLDG_ID is 'BLDG-ID'
/
comment on column BUILDING.BLDG_ADDRESS is 'BLDG-ADDRESS'
/
comment on column BUILDING.BLDG_TYPE is 'BLDG-TYPE'
/
comment on column BUILDING.STATUS is 'STATUS'
/
insert into BUILDING (BLDG_ID, BLDG_ADDRESS, BLDG_TYPE, STATUS)
select BLDG_ID, BLDG_ADDRESS, BLDG_TYPE, STATUS
from tmp_BUILDING
/
drop table tmp_BUILDING cascade constraints
/
alter table ASSIGNMENT
  add constraint FK_BUILDING_ASSIGNMENT foreign key (BLDG_ID)
  references BUILDING (BLDG_ID) on delete cascade
/
```

Как можно видеть из данного примера, совсем небольшое изменение в структуре приводит к необходимости создания сценария, который по своим размерам превосходит даже сценарий создания таблицы. Создание такого сценария вручную уже затруднительно, так как для этого надо помнить информацию не только о данной таблице, но и обо всех таблицах, с ней связанных.

Часть II

Delphi — система быстрой разработки приложений

ГЛАВА 7 Object Pascal и объектно- ориентированное программирование

Существует две 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 процедурно-ориентированный 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 не является новой 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 три объектно-ориентированных языка: C++, Object Pascal и Visual Basic, ставших дальнейшим развитием давно известных процедурных языков C, Pascal и QuickBasic.

Основы языка Object Pascal

В данной главе рассматривается язык Object Pascal, используемый в системе визуального программирования Delphi фирмы Borland.

ПРИМЕЧАНИЕ

Строго говоря, в системе программирования Delphi 8 используется язык программирования Delphi, который создан фирмой Borland в развитие языка Object Pascal. Но в специальной литературе различия между ними не подчеркиваются благодаря тому, что Borland — единственная крупная фирма, поддерживающая язык Object Pascal. В дальнейшем мы будем использовать оба названия языка.

Object Pascal обеспечивает значительно более высокую скорость разработки программ за счет обнаружения на этапе компиляции программы ряда ошибок, которые компилятор языка C++ пропускает без предупреждения. В то же время Object Pascal в полной мере реализует концепции объектно-ориентированного программирования, в чем практически не уступает C++.

Язык Object Pascal является строгим языком, что во многом обусловлено учебным характером его предшественника языка Pascal.

Структура программы в Object Pascal

Программа, написанная на языке Object Pascal, состоит из ряда разделов (или блоков). Начало каждого раздела указывается с помощью специальных зарезервированных слов. В общем виде программа Object Pascal 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 Object Pascal 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 Object Pascal, использование меток считается плохим стилем программирования. Поэтому меток в программах, разработанных в среде Delphi, практически никогда нет.

Раздел описания типов

В Object Pascal 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 Object Pascal 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:

```
[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, в Object Pascal 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. 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_1 : integer = 20;
  // константа-переменная действительного типа:
  идентификатор_константы_2 : real = 3.14;
```

Типы данных в Object Pascal

Язык Object Pascal 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 Object Pascal, можно разделить на две группы: *простые* и *структурные*.

Несколько особняком стоят *указательные* типы, которые нельзя отнести ни к простым, ни к структурным типам. Указательные типы предназначены для работы с большими структурами данных и памятью.

В последних версиях языка Object Pascal 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 этими группами 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 Object Pascal 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 Object Pascal приведена в табл. 7.1.

Таблица 7.1. Целые типы Object Pascal

Обозначение типа	Диапазон значений	Занимаемый объем памяти
Без знака		
Byte	0...255	8 бит
Word	0...65 535	16 бит
LongWord	0...4 294 967 295	32 бита
Cardinal	0...4 294 967 295	32 бита
Со знаком		
ShortInt	-128...127	8 бит
SmallInt	-32 768...32 767	16 бит
Integer	-2 147 483 648...2 147 483 647	32 бита
LongInt	-2 147 483 648...2 147 483 647	32 бита
int64	$-2^{63} \dots 2^{63}-1$	64 бита

- *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 символьному 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 отображения символов русского алфавита, символов псевдографики и т. п. В Delphi применяется расширенный набор символов ANSI.

ПРИМЕЧАНИЕ

В последнее время довольно широкое распространение получила 16-разрядная кодировка символьной информации, называемая UNICODE. Данная кодировка позволяет применять гораздо более обширный набор символов и, вероятно, будет получать все большее распространение. В пользу этого говорит и тот факт, что она используется в последних версиях Microsoft Office. Пока это вызывает некоторые проблемы у русскоязычных пользователей, так как старые кириллические шрифты True Type не могут применяться в этих версиях MS Office. Следует отметить, что кодировка первых 256 символов UNICODE совпадает с кодировкой ANSI.

В Object Pascal 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 Object Pascal для совместимости с другими языками определены три логических типа, различающихся занимаемым объемом памяти:

- **Boolean**, или **ByteBool**, — 1 байт;
- **WordBool** — 2 байта;
- **LongBool** — 4 байта.

ПРИМЕЧАНИЕ

Слова true и false в языке Object Pascal 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 Object Pascal 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 мантиссы (с увеличением количества значащих цифр объем занимаемой памяти возрастает).

В Object Pascal используются три формата IEEE: Single, Double и Extended, предназначенные для хранения чисел с разрядностью 32, 64 и 80 бит соответственно. В более ранних версиях Object Pascal 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 Object Pascal 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 Object Pascal 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 Object Pascal.

Таблица 7.2. Действительные типы Object Pascal

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	-922337203685477.5808...922337203685477.5807	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 Object Pascal определены следующие структурные типы:

- ☐ строки;
- ☐ массивы;
- ☐ множества;
- ☐ записи;
- ☐ файлы;
- ☐ классы.

Строковые типы

В Object Pascal 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 Object Pascal 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 Object Pascal, 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 размерностью 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 обращени-

ем к динамической переменной необходимо выделить для нее память, а по окончании работы с указателем эту память освободить. Для этого используются специальные процедуры языка Object Pascal, 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 на что-нибудь 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 Object Pascal 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 Object Pascal 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 вариантного 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 Object Pascal

Операторы предназначены для контроля за порядком вычисления выражений и количеством вычислений. Операторы, используемые в языке Object Pascal, 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 Object Pascal 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 Object Pascal 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) and (B<0) or (C>=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 Object Pascal 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, следующего 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 Object Pascal 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 *фактическими*. 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 байт. 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 компиля-

тор 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 не только содержать 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 Object Pascal

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 Object Pascal 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;
```

```
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 позволяет упростить разработку сложных программ и повысить их надежность. Однако объектно-ориентированная модель построения программ принципиально отличается 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 Object Pascal.

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 Object Pascal 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 T. 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 экземпляром 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 и доступ к полям записи, с помощью одного из двух способов:

- ☐ указания имени соответствующего поля после имени экземпляра класса через точку;
- ☐ использования 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 будет общей: 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 Object Pascal 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 (о новой платформе Microsoft .NET рассказывается в следующей главе) с запретом наследования было добавлено ключевое слово `sealed`. Класс, объявленный как `sealed`, не может иметь наследников. Например:

```
TMyClass = class [abstract | sealed] (TObject)
```

Полиморфизм

Полиморфизм дает возможность определения единственного имени для действия (процедуры или функции), примененного 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 Object Pascal, 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 необязательно 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 и 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 Object Pascal 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 Object Pascal 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 Object Pascal 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 — модель составных объектов). Более подробно о технологии автоматизации см. в главах 17 и 18.

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 Object Pascal существуют специальные средства для обработки исключительных ситуаций.

Исключительные ситуации, возникающие во время выполнения программы, описываются в языке Object Pascal 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 Object Pascal 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 Средства быстрой разработки приложений

При разработке современных информационных систем возможности того или иного языка программирования далеко не всегда являются решающими для выбора его в качестве языка реализации приложений информационной системы. Гораздо важнее наличие или отсутствие удобной среды проектирования, поддерживающей этот язык. Среда проектирования должна максимально упрощать процесс проектирования, освобождая программиста от непроизводительных операций, смещая акцент к концептуальному проектированию системы.

Быстрая разработка приложений (Rapid Application Development, RAD) основывается на визуализации процесса создания программного кода. Рассматриваемая технология является инструментальным программным обеспечением, которое предоставляет программистам средства, ускоряющие разработку прикладных программ (приложений) и упрощающие их модификацию. С целью максимального упрощения перечисленных действий используются графические инструментальные средства.

Не следует сводить функции RAD только к визуальной генерации пользовательского интерфейса. Возможности этой технологии гораздо шире «простого» предоставления набора процедур, обеспечивающих помещение элементов управления на «формы» с последующей настройкой 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, 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 не визуальными; как атомарными, так и контейнерными (содержащими другие компоненты); как низкоуровневыми (системными), так и высокоуровневыми.

Средства RAD предлагают инструменты для создания не только приложений (исполняемых файлов), но и динамических библиотек, веб-страниц, а также иных

ресурсов, необходимых для создания полноценных информационных систем. Некоторые средства RAD допускают одновременное использование нескольких языков программирования.

Средства визуального программирования

При *визуальном проектировании* пользовательского интерфейса отдельные компоненты выбираются на палитре компонентов и размещаются в нужных местах основных и вспомогательных окон приложения. Среди инструментов визуального проектирования интерфейса можно назвать *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 последовательному выполнению трех операций.

- ☐ Размещение компонентов интерфейса в нужном месте.
- ☐ Задание моментов времени их появления на экране.
- ☐ Настройка связанных с ними атрибутов и событий.

В идеале среда визуальной разработки должна позволять быстро перетаскивать компоненты мышью и задавать значения изменяемых параметров. Если на выполнение каждой операции будет уходить ощутимое время, 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 каждый из инструментов (конструкторы, отладчики и т. д.) должен быть реализован 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.

Платформа Microsoft .NET

Технический прогресс, ситуация на рынке программных средств, жесткая конкуренция, борьба за безопасность программного обеспечения заставляют производителей постоянно совершенствовать свои средства, предлагать новые решения. Одним из таких новых решений является платформа Microsoft .NET Framework (произносится «дот нет» от английского «dot NET» — «точка NET»).

Microsoft .NET — это новая среда разработки с новым доступом к службам и интерфейсам API Windows, интегрированная с рядом технологий, выпущенных Microsoft в конце 90-х годов.

Помимо поддержки удачных старых решений, Microsoft .NET предлагает некоторые дополнительные возможности.

- ❑ Общеязыковая среда выполнения (Common Language Runtime, CLR) Microsoft .NET подобна виртуальной машине Java.
- ❑ Библиотеки базовых классов .NET позволяют работать на любом языке программирования, поддерживаемом .NET.
- ❑ Microsoft .NET обеспечивает межязыковую интеграцию — все языки программирования для .NET работают на одном ядре. Благодаря этому они используют общие типы данных, одинаковый набор функций. Класс, написанный на любом языке .NET, может служить основой для другого класса, также написанного на любом другом языке .NET (рис. 8.1).

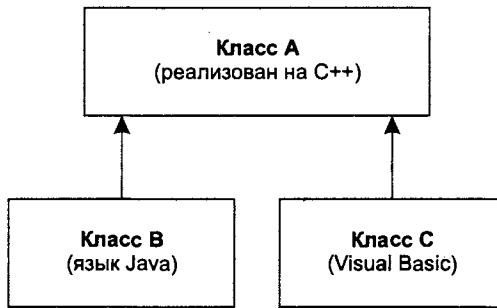


Рис. 8.1. Межязыковая интеграция в .NET

- ❑ Доступны новые методы работы с Интернетом, службы, протоколы передачи данных.
- ❑ Платформа Microsoft .NET ориентирована на использование открытых стандартов, в частности, языка XML (Extensible Markup Language — расширяемый язык разметки), стандартов Интернета.
- ❑ Доступны новые возможности обеспечения безопасности приложений, ориентированные на защиту от несанкционированного доступа к приложению и к системе в целом. В частности, .NET позволяет указывать в реализации методов атрибуты безопасности.
- ❑ Повышается надежность разрабатываемых приложений. Платформа .NET осуществляет автоматическую очистку динамической памяти (кучи) во время работы приложений, что особенно важно для непрерывно работающих приложений, например серверов. Сборщик мусора (garbage collector) работает в системе автономно, но также допускает управление им из приложений.
- ❑ Платформа .NET устраняет проблему, называемую «ад DLL» (DLL Hell), заключающуюся в том, что библиотека DLL (Dynamic Link Library — динамически связываемая библиотека), установленная позднее, может не поддерживать возможностей одноименной библиотеки, установленной ранее, из-за чего во время выполнения приложений возникают сбои. На платформе .NET исполняемый файл жестко привязывается к DLL на стадии компиляции.

- ❑ Реализована поддержка распределенных вычислений через Интернет.
 - ❑ Реализована поддержка новых устройств, например, мобильных телефонов.
- Данные преимущества представляются весьма существенными.

Технологии COM и .NET

До появления .NET самыми передовыми технологиями проектирования являлись технологии COM (Component Object Model — модель составных объектов) и COM+ корпорации Microsoft.

Среда .NET во многом отошла от технологии разработки и развертывания COM-приложений, оказавшейся слишком сложной, и предложила более простой способ построения и развертывания компонентов. В .NET не требуется изменять реестр для регистрации компонентов и исключается необходимость в дополнительном коде, присутствующем во всех COM-компонентах, включая код поддержки интерфейса IUnknown, фабрик классов, динамического связывания и прочего. В .NET все классы готовы к использованию на двоичном уровне.

Microsoft .NET поддерживает языковую интеграцию, это означает возможность наследовать от классов, перехватывать исключения и использовать преимущества межязыкового полиморфизма. Это стало возможным благодаря введению системы CTS (Common Type System — система общих типов), которую должны поддерживать все компоненты .NET. Компилятор любого языка, поддерживаемого .NET, должен соответствовать минимальному набору правил, оговоренному в спецификации CLS (Common Language Specification — общезыковая спецификация), за счет этого могут взаимодействовать классы, реализованные средствами различных языков.

Среда .NET представляет любому языку набор базовых классов, происходящих от единого предка `System.Object`, и позволяет работать с ними любому языку.

Если технология COM требовала регистрации приложения в системном реестре, то для развертывания большинства приложений в .NET достаточно простого копирования с компьютера на компьютер, как это было в DOS.

Забегая вперед, отметим, что данная книга посвящена в основном проектированию информационных систем с помощью Delphi версии 8. Однако в виду важности технологии COM она также затронута в главах 17 и 18 на примерах проектирования в Delphi версии 7.

Для платформы .NET отказ от использования подходов COM и COM+ не является абсолютным. Более того, при функционировании ядра .NET используются технологии COM. Скорее, .NET является естественной эволюцией COM.

Прогноз внедрения Microsoft .NET

В технической литературе не принято анализировать маркетинговые аспекты внедрения той или иной технологии. Однако нам представляется необходимым затронуть их, тем более что они являются ключом к пониманию многих изменений.

Корпоративные приложения на базе Интернета обещают быть самым быстрорастущим и перспективным продуктом на рынке программного обеспечения. Причин тому несколько:

- усложнение процессов производства;
- активное развитие электронной коммерции.

Для современного рынка обладание полной информацией о том или ином бизнесе и возможность максимально быстро принимать управленческие решения является решающим фактором.

Microsoft .NET предлагает технические решения, которые являются максимально удобной базой для разработки и сопровождения информационных систем уровня корпораций. Ориентация .NET на широкое использование Интернета позволяет создавать системы, объединяющие сотрудников предприятия по всему миру.

Корпорация Microsoft обладает большим опытом в создании операционных систем семейства Windows и пакетов Microsoft Office. Указанные системы нашли широкое признание предприятий, их широкое распространение значительно расширило круг предприятий, активно использующих офисные приложения.

Если у других производителей и есть подобный опыт, то финансовых возможностей, чтобы конкурировать с Microsoft в создании среды, аналогичной .NET, у них нет. Между тем, в развитие платформы .NET корпорацией Microsoft вложены колоссальные инвестиции.

Платформа .NET предоставляет технические решения, позволяющие реализовать давно созревшую идею отхода от продажи приложений к их «аренде» и подписке на предоставляемые услуги. Например, компания может не покупать пакет Microsoft Office, а использовать его на правах аренды через Интернет.

Кроме того, компоненты, созданные на основе технологии .NET и работающие в едином киберпространстве, значительно легче контролировать в плане легальности их использования. Такой подход пришелся по вкусу большинству производителей программного обеспечения, несущих колоссальные потери от несанкционированного применения их разработок.

Microsoft активно проводит политику кооперации с независимыми разработчиками и поставщиками услуг, предоставляя им поддержку. Такая политика и значительные инвестиции в разработку служат основой значительного роста рынка информационных систем на базе .NET.

Все это, а также удачные технологические решения, позволяют сделать вывод о неизбежном, в скором времени, переходе большинства пользователей на новую платформу. Этот переход хотя и связан изначально с повышением затрат, но в последующем себя окупит.

Структура .NET

Среда .NET, по сути, является системным приложением, работающим под управлением Windows или другой операционной системы (рис. 8.2). Платформа .NET распространяется бесплатно.

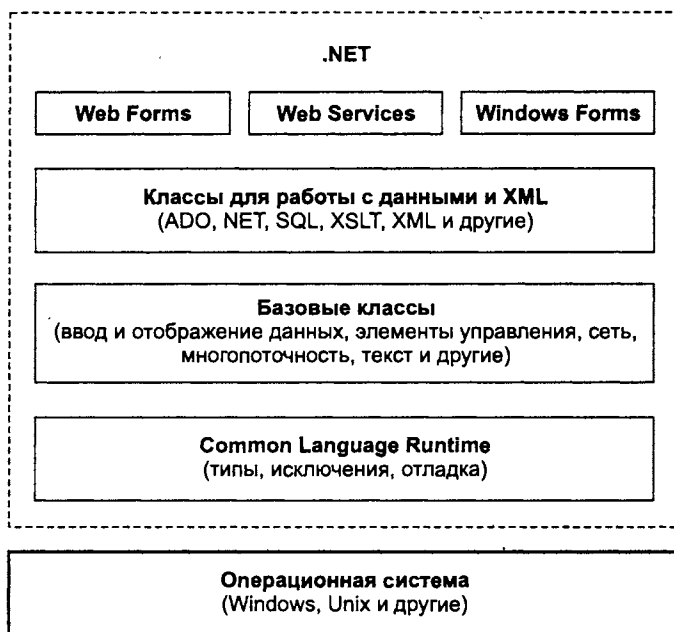


Рис. 8.2. Структура платформы .NET

Платформа .NET доступна в полной мере на операционных системах Windows NT, Windows 2000 и Windows XP.

ПРИМЕЧАНИЕ

На момент написания книги не было известно о других (помимо Windows) операционных системах, на которых работает .NET, но возможность переноса .NET на другие операционные системы в проекте заложена. В этом случае приложения, функционирующие в .NET под управлением одной операционной системы, свободно будут функционировать под управлением другой операционной системы (кроссплатформенность).

Основой .NET является общезыковая среда выполнения (CLR), которая активизирует объекты, выполняет для них проверку безопасности, располагает в памяти, использует их, обеспечивает сборку мусора.

Набор базовых классов среды .NET поддерживает элементарную функциональность ввода-вывода, манипуляций со строками, управление безопасностью и другие функции. Уровнем выше находится набор классов, расширяющих возможности базовых классов за счет поддержки управления данными, включая классы для работы с SQL, а также классы для работы с XML, позволяющие манипулировать данными в формате XML.

Классы в трех различных технологиях (Web Services, Web Forms и Windows Forms) расширяют базовые классы и классы работы с данными и XML:

- Web Services — классы для разработки облегченных распределенных компонентов;

- ❑ Web Forms — классы для разработки веб-приложений;
- ❑ Windows Forms — классы для разработки Windows-приложений (имеются в виду приложения, функционирующие только под управлением .NET).

Все языки, поддерживаемые .NET, транслируются на общий исполняемый промежуточный язык CIL. Для приложений .NET он играет ту же роль, что ассемблер для обычных Windows-приложений.

Среда разработки Delphi

Система визуального программирования Delphi (в данной книге рассматриваются восьмая и, отчасти, седьмая версии Delphi) фирмы Borland позволяет в полной мере реализовать современные концепции программирования, включая:

- ❑ объектно-ориентированный подход;
- ❑ визуальные средства быстрой разработки приложений (RAD), основанные на компонентной архитектуре;
- ❑ возможность создания приложений как с использованием библиотеки визуальных компонентов (Visual Component Library, VCL) для .NET, разработанной Borland, так и с использованием непосредственно классов .NET;
- ❑ возможность работы с базами данных;
- ❑ возможности создания Интернет-приложений различной сложности.

Сразу отметим, что Delphi 8 позволяет разрабатывать приложения только для .NET. Такие приложения не будут работать на компьютерах, на которых платформа .NET не поддерживается. Для создания обыкновенных Windows-приложений предназначена среда Delphi 7. Фирма Borland планирует поставлять обе версии совместно. Возможен перенос части кода в Delphi 8 из Delphi 7 и более ранних версий. Однако определенная часть компонентов, поддерживаемых в Delphi 7, в Delphi 8 не поддерживается. В частности, в поставке Delphi 8 нет компонентов TADOTable и TADOQuery, компонентов для работы с Microsoft Word, компонентов для реализации куба решений (decision cube) и других.

ПРИМЕЧАНИЕ

Многие методы работы с Delphi 8 аналогичны проектированию в Delphi 7. В данной книге не приводятся сверхсложные примеры приложений, поэтому большинство примеров можно использовать и в Delphi 7.

В Delphi используется язык Object Pascal, предоставляющий возможность полной реализации основных принципов ООП (инкапсуляция, наследование, полиморфизм) и обладающий встроенной обработкой исключительных ситуаций. Компонентная архитектура Delphi является прямым развитием поддерживаемой объектной модели. Все компоненты являются объектными типами (классами), обладающими возможностью неограниченного наследования. Компоненты Delphi поддерживают модель PME (Properties, Methods, Events — свойства, методы, события), позволяющую изменять поведение компонентов без создания новых классов.

Входящий в поставку Delphi процессор баз данных Borland (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.). Существуют возможности использования технологии ADO.NET (ActiveX Data Objects for .NET) — базовой технологии работы с данными в .NET.

Интерфейс интегрированной среды разработки (Integrated Development Environment, IDE) Delphi включает в себя следующие основные части (рис. 8.3):

- ☐ главное окно;
- ☐ инспектор объектов;
- ☐ редактор форм;
- ☐ редактор кода;
- ☐ палитру компонентов;
- ☐ менеджер проектов.

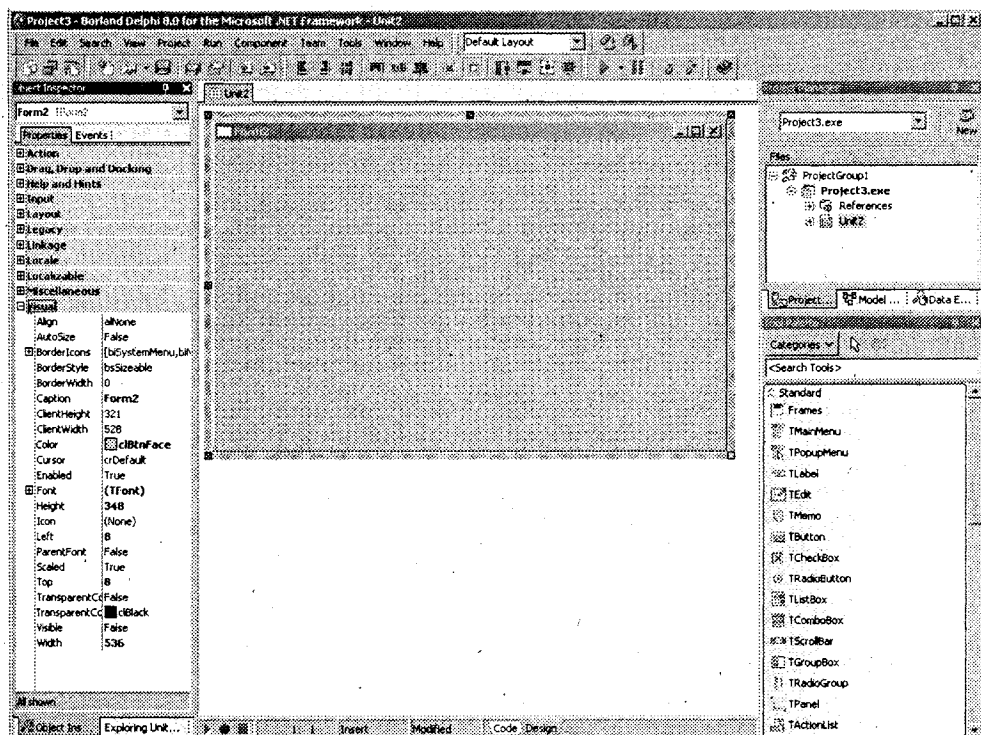


Рис. 8.3. Интерфейс Delphi

Компания Borland стремится максимально упростить работу программистам. Проявлением этого является идентичность интерфейса сред разработки Borland Delphi for the Microsoft .NET (Delphi 8) и Borland C# Builder for the Microsoft .NET, что позволяет говорить не просто о среде разработки (IDE), а об унифицированной среде разработки (Unified Development Environment, UDE).

Главное окно

Главное окно IDE Delphi состоит из нескольких разделов.

- ❑ *Главное меню* располагается непосредственно под заголовком главного окна и позволяет получить доступ ко всем функциям 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. Пользователь может выбрать стиль оформления среды разработки, характерный для более ранних версий Delphi.
- ❑ *Палитра компонентов* содержит компоненты, которые разработчик может использовать для создания приложений. В зависимости от вида разрабатываемого приложения набор компонентов меняется.
- ❑ *Инспектор объектов* — окно, расположенное в левой части экрана, в котором отображаются свойства редактируемого компонента.
- ❑ *Редактор кода* представляет собой текстовый редактор, специально приспособленный для редактирования исходных текстов программ. Редактор поддерживает возможность автоматической генерации части текстов программ, исходя из визуально спроектированной части приложения. В процессе проектирования доступен либо режим визуального проектирования (редактор форм), либо режим редактирования текста. Также поддерживаются подсветка синтаксических элементов для языков Delphi, C#, C++, XML/HTML, минимизация блоков текста, автоматический выбор 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.
- ❑ *Менеджер проектов* содержит информацию о группах проектов, проектах, их составных частях, доступных источниках данных.

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 основных команд.

СОВЕТ

Для получения справки по конкретному пункту меню достаточно выделить его и нажать клавишу F1. Это же относится к большинству компонентов, с которыми пользователь сталкивается при работе с Delphi.

Меню File

Меню File содержит команды, предназначенные для работы с файлами (рис. 8.4).

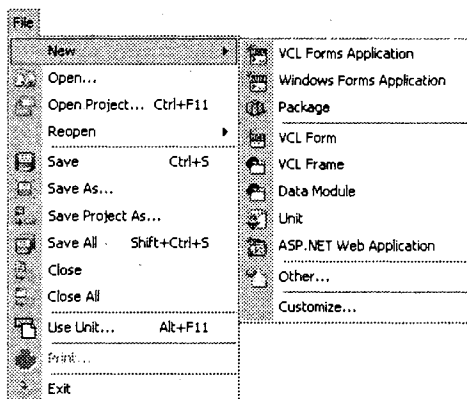


Рис. 8.4. Меню File

Часть команд, содержащихся в данном меню (Open, Save, Save As, Close, Exit), — это обычные команды для работы с файлами, 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, динамически связываемой, или динамической, библиотеки (DLL), веб-приложения и т. д.

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. Внешний вид этого подменю может меняться в зависимости от вида разрабатываемого приложения (сравните рис. 8.4 и 8.5).

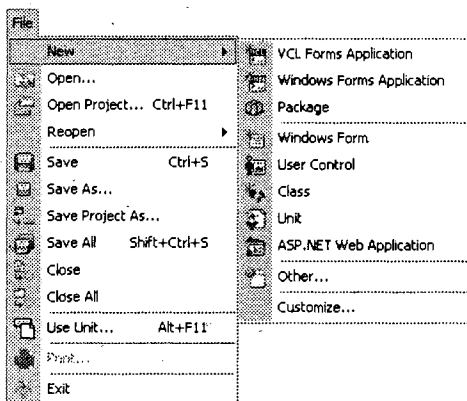


Рис. 8.5. Подменю New в случае создания приложений Windows Forms

Система визуального программирования Delphi позволяет разрабатывать различные проекты. Среди них обычные приложения, динамические библиотеки, компоненты Delphi, приложения на базе библиотеки Borland VCL for .NET и стандартной библиотеки классов FCL для платформы .NET, приложения для работы через Интернет, HTML- и XML-документы и т. п.

Если вы создаете приложение с помощью библиотеки VCL, среда Delphi предлагает добавить в проект новую форму, фрейм, модуль данных или же обыкновенный модуль. Для приложений Windows Forms (библиотека .NET) — форму на основе этой технологии, класс .NET, компонент идентификации пользователей. Эти и другие компоненты описаны в следующих главах этой книги.

Команда File ► New ► Other открывает окно диалога New Items (рис. 8.6).

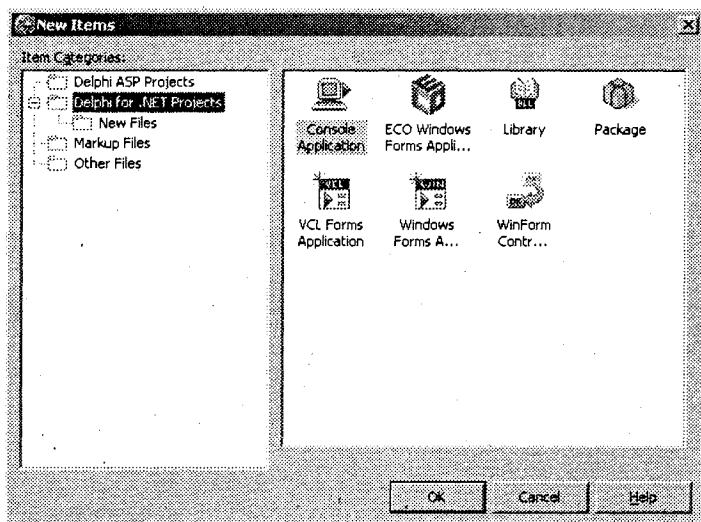


Рис. 8.6. Окно диалога New Items обеспечивает доступ к хранилищу объектов

В этом окне диалога предлагаются объекты, содержащиеся в *хранилище объектов* (object repository). 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 вкладках окна New Items.

Продолжим обсуждение команд меню File.

Команда Use Unit используется для подключения одного из модулей, уже включенных в раздел uses проекта.

Команда Print предназначена для печати формы или программы.

Меню Edit

Меню Edit содержит ряд стандартных команд, используемых в приложениях ОС Windows для редактирования (рис. 8.7): 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 (выравнивание, порядок обхода компонентов и т. д.).



Рис. 8.7. Меню Edit

Меню Search

Меню Search содержит некоторые распространенные в приложениях обработки текста команды, предназначенные для поиска и замены: Find (найти), Replace (заменить), Search Again (повторный поиск), Go to Line Number (перейти на строку с номером), а также несколько специальных команд, описываемых далее.

Команда Find in Files предназначена для поиска текстовых строк в нескольких файлах. При выборе этой команды открывается окно диалога Find Text (рис. 8.8).

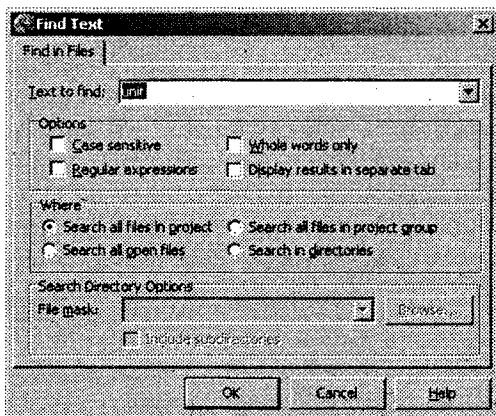


Рис. 8.8. Окно диалога Find Text

Элементы управления этого окна диалога позволяют задать параметры поиска. В разделе Options имеются следующие флажки:

- ☐ 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).
- ☐ Display results in separate tab — в зависимости от того, установлен флажок или нет, в окне сообщений будут сохраняться результаты предыдущего поиска.

Переключатели в группе Where предназначены для выбора одного из четырех альтернативных вариантов:

- ☐ Search all files in project — поиск в файлах, относящихся только к текущему проекту;
- ☐ Search all files in project group — поиск в файлах, относящихся только к текущей группе проектов;
- ☐ Search all open files — поиск во всех открытых файлах;
- ☐ Search in directories — поиск только в текущей папке (каталоге).

Раскрывающийся список File mask позволяет задать шаблон имен файлов, в которых будет производиться поиск (список доступен, если установлен переключатель Search in directories). Флажок Include subdirectories дает возможность поиска во всех вложенных папках (эта возможность также доступна, только если установлен переключатель Search in directories).

Меню View

Команды меню View предназначены для управления способом отображения информации (рис. 8.9).

Остановимся на назначении наиболее важных из команд.

Команда Project Manager делает активным одноименное окно менеджера проекта, в котором отображаются все файлы, входящие в текущую группу проектов (рис. 8.10). Менеджер проекта позволяет добавлять проекты в группу, удалять проекты из группы, выбирать активный проект, сохранять отдельные проекты или 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 щелчке правой кнопки мыши на каком-либо модуле, входящем в состав группы проектов или отдельного проекта.

ПРИМЕЧАНИЕ

Возможность объединять несколько проектов в одну группу удобно использовать, например, в тех случаях, когда одновременно с приложением разрабатываются динамические библиотеки.

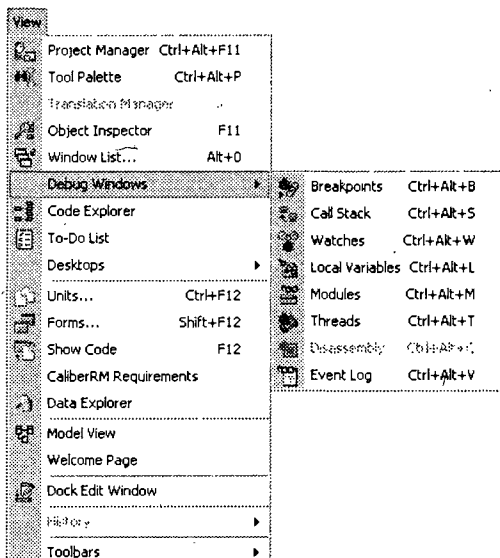


Рис. 8.9. Меню View

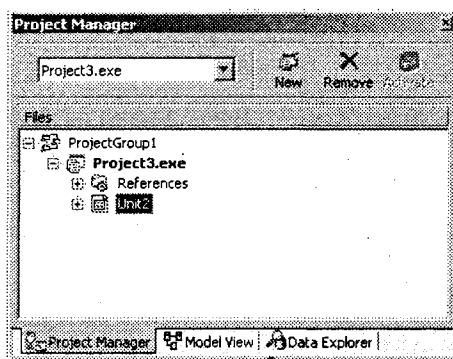
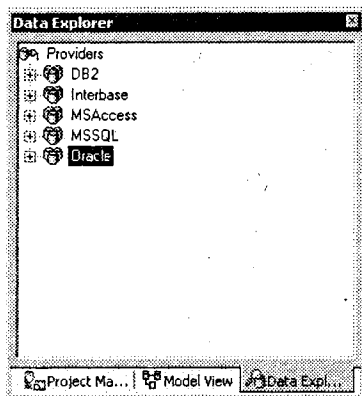


Рис. 8.10. Окно диалога Project Manager

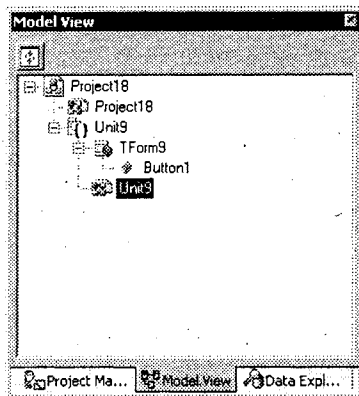
Команды Data Explorer и Model View переключают окно менеджера проектов соответственно в режим управления связями с источниками данных (рис. 8.11, а) и в режим представления проекта в виде диаграмм UML-классов (рис. 8.11, б). Вкладка Data Explorer окна Project Manager позволяет отслеживать взаимодействие между процессором баз данных (BDE) и клиентской базой данных, чтобы контролировать подключения к базе данных, транзакции, запросы и т. п.

Команда Translation Manager предназначена для разработки приложений с поддержкой нескольких языков.

Команда Object Inspector применяется для активизации окна инспектора объектов, которое используется при разработке форм. Более подробно его назначение рассматривается далее.



а



б

Рис. 8.11. Вкладки Data Explorer и Model View окна Project Manager

Команда To-Do List используется для управления списком задач, которые могут относиться как ко всему проекту, так и к отдельному модулю проекта. Все текущие задачи отображаются в окне диалогов To-Do List (рис. 8.12), которое открывается при выборе этой команды. Используя команды контекстного меню, вы можете добавлять задачи в список окна To-Do List, а также удалять и редактировать их. Слева от значка выполненной задачи устанавливается флажок.

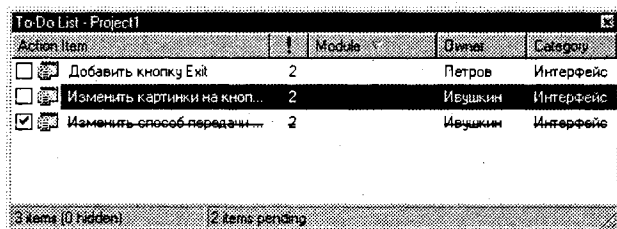


Рис. 8.12. Окно списка задач

Способ включения задачи в список зависит от ее типа. В случае если задача относится ко всему проекту, то используется команда Add контекстного меню окна диалогов To-Do List. Для отдельного модуля применяется команда Add To-Do Item контекстного меню редактора кода. При выполнении любой из этих команд открывается окно диалогов Add To-Do Item, в поле Text которого вводится описание назначения задачи, а в трех других полях может быть задан ряд дополнительных параметров: приоритет (Priority), имя исполнителя (Owner), тип задачи (Category).

ПРИМЕЧАНИЕ

Список задач никак не влияет на компиляцию проекта и используется просто как записная книжка, помогая планировать разработку программы и распределять задачи между разными программистами.

Продолжим обзор команд меню View.

Команда Code Explorer активизирует окно браузера кода, в котором отображается структура программы. Браузер кода существенно облегчает навигацию в сложных проектах, содержащих большое количество модулей и много программного кода.

Команда Window List открывает окно, содержащее список всех открытых окон Delphi и позволяющее сделать активным любое из них.

В подменю Debug Windows входят команды, предназначенные для отладки приложения.

Команды подменю Desktops позволяют сохранять текущую конфигурацию среды разработки, в частности информацию о том, какие окна открыты и где они расположены.

Команда Show Code/Show Designer используется для переключения между редактором кода и редактором форм.

Команда Units открывает окно, содержащее список всех модулей, входящих в состав текущего проекта.

Команда Forms позволяет открыть список всех форм проекта.

Команда Dock Edit Window делает окно разработки причаливаемым, то есть появляется возможность пристыковать его к краю главного окна.

Команды History ► Back и History ► Forward используются для настройки конфигурации среды проектирования путем поиска наиболее удачного расположения и размера окон и т. д.

Команды подменю Toolbars требуются для настройки панели инструментов.

Команда Welcome Page показывает стартовую страницу окна Delphi, представляющую собой Интернет-браузер, специально адаптированный для разработки приложений.

Меню Project

В меню Project содержатся команды создания и редактирования проектов.

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 выбираются те из них, которые необходимо 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 человеческих 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 одноименной команде контекстного меню менеджера проектов.

Команда **Information** открывает одноименное окно диалога, в котором отображается информация о текущем проекте:

- ☐ **Source Compiled** — количество строк программного кода в проекте;
- ☐ **Code Size** — размер исполняемого файла или динамической библиотеки без отладочной информации;
- ☐ **Data Size** — объем памяти, занимаемый глобальными переменными;
- ☐ **Initial Stack Size** — объем памяти, отведенный для хранения локальных переменных;
- ☐ **File size** — размер выходного файла;
- ☐ **Package Used** — список всех пакетов, используемых проектом.

Команда **Project Options** позволяет настраивать свойства проекта и рассматривается далее в этой и других главах.

Меню Run

Меню **Run** содержит ряд команд, предназначенных для запуска и отладки приложений. Одна часть этих команд рассматривается здесь, другая — в главе 14.

Команда **Attach to Process** открывает одноименное окно диалога со списком всех программ, работающих в текущий момент. Выбранная в этом списке программа загружается в среду Delphi в режиме отладки. При этом загружаемая программа может быть любой (не обязательно написанной в Delphi), поскольку текст программы отображается в отладчике в командах ассемблера.

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.

Команды подменю **Add Breakpoint** предназначены для установки и снятия точек останова в программе.

Меню Component

Команды меню **Component** используются для создания и настройки новых компонентов.

Команда **New VCL Component** применяется при разработке нового компонента на базе библиотеки VCL.

Команды **Install VCL Component** и **Installed .NET Components** служат для добавления компонентов в среду Delphi.

Меню Team

С помощью команд меню **Team** пользователь получает возможность интеграции с системами контроля версий разрабатываемых приложений. У Delphi версии 7

и более ранних версий встроенные возможности работы с системами поддержки версий были довольно слабыми, что приводило к необходимости использовать для решения этой задачи продукты сторонних производителей. В Delphi 8 реализована поддержка взаимодействия с системами контроля версий через Microsoft SCC API. Поддерживаются следующие популярные системы контроля версий:

- ☐ Borland StarTeam и StarTeam Microsoft SCC Integration;
- ☐ Версии GUI Concurrent Versions System (CVS), поддерживающие SCC API;
- ☐ Rational ClearCase;
- ☐ Microsoft Visual SourceSafe.

Меню Tools

Меню Tools содержит команды, вызывающие окна диалога для настройки среды Delphi, а также окна редактора кода, встроенного отладчика и хранилища объектов. Кроме того, в команды этого меню позволяют запускать некоторые внешние программы (утилиты), причем имеется возможность настройки списка вызываемых программ.

Панели инструментов

Панели инструментов содержат кнопки, обеспечивающие быстрый доступ к командам главного меню (рис. 8.13).

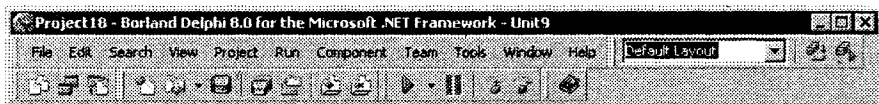


Рис. 8.13. Панели инструментов IDE Delphi

Главное окно Delphi включает пять панелей инструментов: Standard, View, Debug, Custom, Desktops, каждая из которых может закрываться и отрываться с помощью команды View ► Toolbars ► <Название панели> главного меню или соответствующей команды контекстного меню панели инструментов.

Содержание каждой панели инструментов может настраиваться пользователем с помощью окна диалога Customize (рис. 8.14), открываемого командой View ► Toolbars ► Customize главного меню или командой Customize контекстного меню.

Окно диалога Customize содержит три вкладки: Toolbars, Commands и Options.

- ☐ На вкладке Toolbars расположены шесть флажков, используемых для отображения или скрытия 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 соответствующими им значками. Настройка производится путем перетаскивания. Например, для удаления кнопки достаточно перетащить ее за пределы панели инструментов. Что-

бы добавить кнопку на 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.

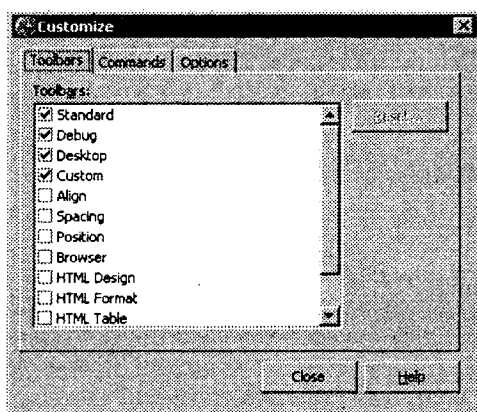


Рис. 8.14. Окно диалога *Customize* предназначено для настройки панелей инструментов

ПРИМЕЧАНИЕ

Кнопки можно размещать на любых панелях инструментов независимо от того, какие команды они выполняют. IDE Delphi не проводит никакой проверки на соответствие названия панели инструментов размещенным на ней кнопкам.

- ☐ 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 компонентов

Палитра компонентов (*component palette*) используется для отображения компонентов, содержащихся в библиотеке компонентов VCL или .NET в зависимости от типа разрабатываемого приложения (рис. 8.15).

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.

Стандартная конфигурация палитры компонентов VCL Delphi 8 содержит 19 списков, каждый из которых предоставляет разнообразные компоненты и элементы управления:

- ☐ Standard — стандартные элементы управления оконного интерфейса Windows;
- ☐ Additional — специализированные элементы управления интерфейса Windows;
- ☐ Win32 — элементы интерфейса, содержащиеся в 32-разрядных системных библиотеках Windows 95 и Win32s;
- ☐ System — специализированные системные элементы управления;
- ☐ Data Access — компоненты для настройки связей с источниками данных;
- ☐ Data Controls — компоненты для отображения и редактирования информации, хранящейся в базах данных;
- ☐ BDE — компоненты, обеспечивающие доступ к информации, хранящейся в базах данных, с использованием технологии BDE;
- ☐ InterBase и InterBase Admin — компоненты, обеспечивающие доступ к базам данных InterBase без использования технологии BDE (Borland Database Engine);
- ☐ dbExpress — компоненты для работы с SQL-серверами;
- ☐ DataSnap — компонент для подключения баз данных по технологии DCOM;
- ☐ InternetExpress — компоненты для разработки веб-серверов и веб-клиентов многоуровневых приложений управления базами данных;
- ☐ Indy Clients, Indy Misc, Indy Intcepts, Indy I/O Handlers — компоненты программирования для Интернета;
- ☐ Dialogs — стандартные окна диалога для открытия файла, сохранения файла, печати и т. п.;
- ☐ Win 3.1 — элементы управления оконного интерфейса Windows 3.1 (оставлены для совместимости с Delphi 1);

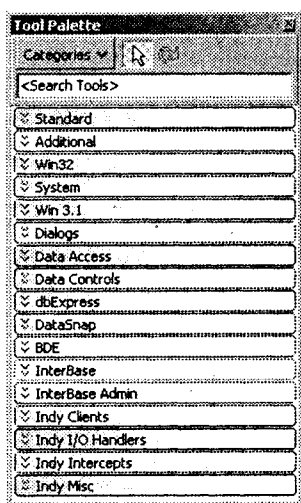


Рис. 8.15. Палитра компонентов VCL

Инспектор объектов

Инспектор объектов (object inspector) является одним из важнейших инструментов разработки приложения и используется для настройки опубликованных свойств компонента. Окно инспектора объектов показано на рис. 8.16, а.

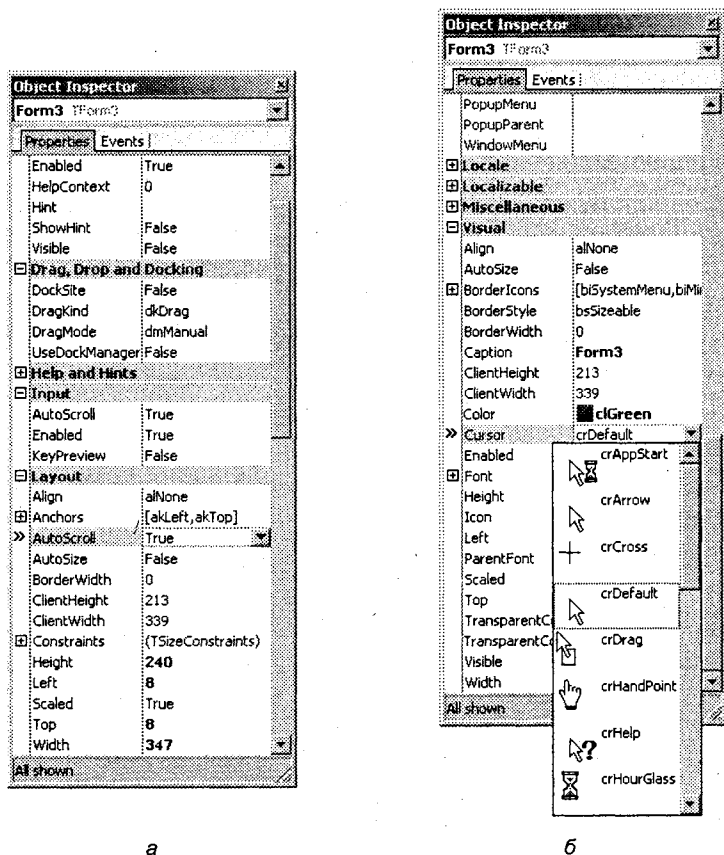


Рис. 8.16. Обычное окно инспектора объектов (а) и окно инспектора объектов в процессе выбора значения графического свойства (б)

Окно инспектора объектов содержит раскрывающийся список и две вкладки — Properties и Events. На первой из них отображается список свойств выделенного объекта, на второй — список событий, на которые реагирует объект. Раскрывающийся 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 или событий, а в правой — их значения. Значения свойств можно редактировать. Некоторые свойства имеют в поле значений собственный раскрывающийся список, в котором выбирается необходимое значение (рис. 8.16, б).

Редактор форм

Редактор форм (form designer) представляет собой инструмент визуальной разработки интерфейса приложения. С его помощью можно реализовать следующие функции:

- ☐ размещение компонентов на форме;
- ☐ модификацию свойств компонентов и самой формы;
- ☐ установку обработчиков событий.

Более подробно работа с редактором форм рассматривается в главе 10, а здесь приведем только методику установки обработчиков событий.

Установка обработчиков событий

Обработчик события — это процедура, предназначенная для обработки реакции на какое-либо воздействие. События, на которые компонент может реагировать, перечисляются на вкладке Events инспектора объектов.

Чтобы установить обработчик для какого-либо события, выполните двойной щелчок мышью в 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 писать вручную в теле процедуры-обработчика, которая ничем не отличается от обычной процедуры языка Object Pascal.

Редактор кода

Редактор кода является обычным текстовым редактором, ориентированным на написание текстов программ. Его настройка производится с помощью окна диалога, открывающегося командой Tools ► Editor Options главного меню или командой Properties контекстного меню.

Основные компоненты Delphi и построение простых приложений

Среда визуального программирования Delphi содержит базовый набор используемых в приложениях стандартных элементов управления, доступ к которым осуществляется через палитру компонентов. Применяя готовые компоненты, разработчик может создавать достаточно сложные приложения. Предусмотрены также возможности разработки собственных компонентов (или иерархии компонентов) и использования компонентов, разработанных другими программистами.

Библиотеки компонентов

Delphi предоставляет возможность выбора библиотеки компонентов, с помощью которой будет разрабатываться приложение. Можно использовать библиотеку

компонентов VCL (Visual Component Library — библиотека визуальных компонентов) фирмы Borland или воспользоваться библиотекой FCL (Foundation Classes Library) — стандартной библиотекой компонентов для платформы .NET. В последнем случае проектирование практически ничем не будет отличаться от работы со стандартными компонентами библиотеки .NET в системах проектирования, отличных от Delphi, а единственным отличием станет использование языка Delphi (Object Pascal).

По сравнению с проектированием на основе VCL изменятся свойства и события компонентов, некоторые команды главного и контекстных меню, средства дизайна, способы подключения баз данных. Так, появится возможность визуального определения порядка обхода компонентов на форме (tab order) как это принято в других средствах проектирования (рис. 8.17).

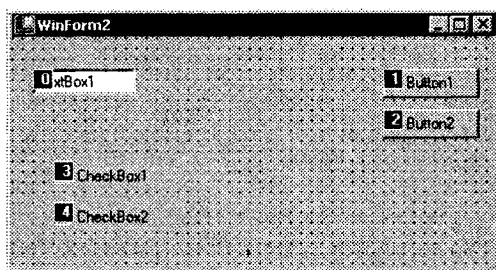


Рис. 8.17. Визуальное проектирование порядка обхода компонентов Windows Forms

Обе библиотеки допускают добавление новых компонентов и удаление существующих. Большое количество компонентов доступно через Интернет.

В настоящее время возможности разных библиотек, как, впрочем, и языков программирования, все более унифицируются. Выбор библиотеки проектирования определяется привычкой, требованиями заказчика, спецификой решаемой задачи, доступностью оригинальных компонентов, другими факторами.

Пространства имен

Следует отметить важное нововведение в языке Delphi, вызванное приспособлением его для нужд платформы .NET.

Стандартным контейнером типов в Delphi является модуль (unit). В .NET реализована другая система организации хранения типов — пространства имен. Необходимость работы с классами .NET потребовала введения в Delphi поддержки пространств имен. В отличие от модулей, пространства имен имеют иерархическую структуру. Наличие иерархии пространства имен позволяет исключить неоднозначность в определении одноименных типов данных.

В Delphi 8 файл проекта неявно объявляет новое пространство имен, называемое пространством имен проекта по умолчанию (project default namespace).

Например, следующая строка объявляет пространство имен **MyProgram**:

```
program MyProgram;
```

Модуль может иметь пространство имен, вложенное в пространство проекта, или явно объявить собственное пространство имен. Пространство имен модуля объявляется в его заголовке, например, следующий код объявляет пространство имен `MyUnit`:

```
unit MyUnit;
```

Поддержка иерархии пространства имен обеспечивается разделением точкой частей иерархии:

```
unit MyNamespace.MyUnit;
```

В данном примере объявлено пространство имен `MyUnit`, вложенное в пространство имен `MyNamespace`. При этом файл, в котором сохраняется модуль, будет называться `MyNamespace.MyUnit.pas`. Глубина иерархии пространства имен не ограничена.

По умолчанию, если имя модуля не содержит полного пути иерархии, то он считается вложенным в пространство имен проекта. При этом в имя файла модуля не добавляется полный путь пространства имен проекта. Однако результатом компиляции такого модуля станет файл, содержащий в имени полный путь иерархии и имеющий расширение `.dcul`. Рассмотрим вышесказанное на простом примере. Допустим, что у нас есть проект `MyProgram`, содержащий модуль `MyUnit`. Тогда модуль получит пространство имен `MyProgram.MyUnit`, называться файл модуля станет `MyUnit1.pas`, а компилироваться модуль будет в файл `MyProgram.MyUnit.dcul`.

Регистр символов в имени пространства имен не учитывается.

Пространство имен может иметь псевдоним (*alias*) — короткое название, объявленный после ключевого слова `as`:

```
uses MyCompany.VeryDescriptiveUnitName as MainUnit;
```

В данной строке объявлен псевдоним `MainUnit` для пространства имен `MyCompany.VeryDescriptiveUnitName`.

При сборке проекта компилятор использует следующий порядок поиска пространств имен.

1. Текущее пространство имен модуля.
2. Пространство имен проекта.
3. Пространства имен, перечисленные после ключевого слова `namespaces` в объявлении проекта.
4. Пространства имен, прописанные при настройке компилятора.

Пакеты

В Delphi библиотека визуальных компонентов состоит из файлов, содержащих откомпилированный код классов. Эти файлы обычно называются *пакетами*. Пакет представляет собой динамическую библиотеку (DLL), содержащую, помимо 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).

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 в списке и т. п. В основном визуальные компоненты применяются для создания пользовательского интерфейса.

Невизуальные компоненты видны только во время разработки. Они в основном предназначены для разработки логической структуры приложения. В то же время некоторые невидимые компоненты требуются и для построения интерфейса. Например, окна диалога не видны в процессе разработки приложения и относятся к невидимым компонентам, в то же время они являются типичным элементом интерфейса.

Формы

Форма представляет собой окно приложения на этапе разработки. Большинство приложений должно иметь, по крайней мере, одно окно, поэтому проект Delphi также должен содержать хотя бы одну форму. Формы обеспечивают создание пользовательского интерфейса, являясь своего рода контейнером, содержащим другие элементы интерфейса.

ПРИМЕЧАНИЕ

В принципе, Windows-приложение 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, 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 приложение 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.

Стандартные элементы интерфейса библиотеки VCL

В библиотеке VCL Delphi содержится ряд компонентов, которые предназначены для создания стандартных элементов интерфейса Windows-приложений. Все эти компоненты доступны в палитре компонентов и могут размещаться на формах или фреймах. Все компоненты Delphi (включая формы) являются потомками одного класса (TComponent) и имеют большое количество общих свойств и событий.

Кнопки

Кнопки (buttons) являются одним из наиболее распространенных элементов управления Windows. Свойства и методы кнопок инкапсулированы в классе TButton (табл. 8.1).

Таблица 8.1. Основные свойства компонента TButton

Свойство	Тип	Описание
Anchors	TAnchors = set of (akTop, akLeft, akRight, akBottom)	Задает привязку кнопки к верхней границе контейнера (akTop), к левой границе контейнера (akLeft), к правой границе контейнера (akRight) или к нижней границе контейнера (akBottom). Если заданы все четыре точки привязки, то при изменении размеров формы пропорционально будут изменяться размеры кнопки
Caption	TCaption	Строка текста на кнопке
Cancel	Boolean	Если это свойство равно true, то при нажатии клавиши Esc вызывается обработчик события OnClick данной кнопки. На форме может быть только одна кнопка со свойством Cancel, равным true
Default	Boolean	Если это свойство равно true, то при нажатии клавиши Enter вызывается обработчик события OnClick данной кнопки. На форме может быть только одна кнопка со свойством Default, равным true
ModalResult	TmodalResult	При щелчке на кнопке значение свойства ModalResult родительской формы устанавливается равным свойству ModalResult кнопки. Используется при создании окон диалога
Enable	Boolean	При установке данного свойства равным true кнопка становится недоступной, отображаясь серым цветом
Visible	Boolean	При установке данного свойства в true кнопка становится невидимой и недоступной
Font	TFont	Задает шрифт, которым отображается надпись на кнопке
TabOrder	TTabOrder	Определяет порядок перебора элементов управления, расположенных на форме, при нажатии клавиши Tab
TabStop	Boolean	Если данное свойство равно false, то фокус ввода никогда не будет передаваться данной кнопке при переборе элементов с помощью клавиши Tab. Однако на такой кнопке все равно можно щелкнуть мышью

Таблица 8.1 (продолжение)

Свойство	Тип	Описание
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 щелчке на кнопке и используется для программирования реакции на щелчок.

Рассмотрим пример использования кнопки. Напишем программу, в которой кнопки служат для изменения заголовка окна программы и для завершения программы.

1. С помощью команды `File ► New ► VCL Forms Application` создайте новое приложение.
2. Разместите на форме две кнопки. Затем перейдите в окне `Object Inspector` на вкладку `Properties` и, используя свойство `Caption`, измените название первой кнопки на `Change caption`, второй — на `Exit`.
3. С помощью мыши выделите кнопку `Change caption` и перейдите в окне `Object Inspector` на вкладку `Events`. Дважды щелкните мышью в поле значения события `OnClick`. После этого станет активным окно редактора кода и в него автоматически добавится 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 `Exit` и перейдите в окне `Object Inspector` на вкладку `Events`. Дважды щелкните мышью в поле значения события `OnClick`. После этого станет активным окно редактора кода и в него автоматически добавится заголовок процедуры-обработчика события `OnClick` для выделенной кнопки:

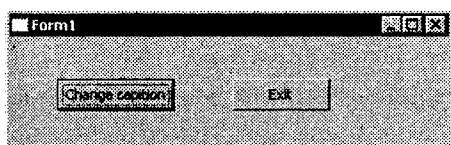
```
procedure TForm1.Button2Click(Sender: TObject);
begin

end;
```

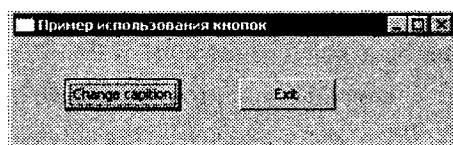
6. Напишите код завершения программы в теле процедуры `Button2Click`. Для этого воспользуйтесь методом `Terminate` класса `TApplication` (более подробно этот класс рассмотрен в главе 14):

```
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, а. При щелчке на кнопке `Change caption` в заголовке окна появляется надпись `Пример использования кнопок` (рис. 8.18, б). Щелчок на кнопке `Exit` приводит к завершению программы.



а



б

Рис. 8.18. Пример формы с кнопками

Надписи

Надписи (labels) используются для отображения на форме текста без возможностей его редактирования. Чаще всего с помощью надписей оформляют другие элементы управления. Свойства и методы надписей инкапсулированы в классе `TLabel` (табл. 8.2).

Таблица 8.2. Основные свойства компонента `TLabel`

Свойство	Тип	Описание
Alignment	<code>TAlignment = (taLeftJustify, taRightJustify, taCenter)</code>	Определяет способ выравнивания текста по горизонтали: по левому краю (<code>taLeftJustify</code>), по правому краю (<code>taRightJustify</code>) или по центру (<code>taCenter</code>)
Layout	<code>TTextLayout = (tlTop, tlCenter, tlBottom)</code>	Определяет способ выравнивания текста по вертикали: по верхнему краю (<code>tlTop</code>), по центру (<code>tlCenter</code>), по нижнему краю (<code>tlBottom</code>)
Autosize	Boolean	Если данное свойство равно <code>true</code> , то ширина и высота объекта <code>Label</code> автоматически изменяются таким образом, чтобы отобразить весь текст, заданный в свойстве <code>Caption</code>
Transparent	Boolean	Если данное свойство равно <code>true</code> , то фон надписи будет прозрачным
WordWrap	Boolean	Если данное свойство равно <code>true</code> , 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 ок-

не инспектора объектов), 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 помощью 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, добавив на форму компонент `TLabel`, и будем изменять значение его свойства `Caption` при щелчке на одной из кнопок.

1. С помощью команды `File ► New ► VCL Form Application` создайте новое приложение.
2. Разместите на форме две кнопки. Затем перейдите в окне `Object Inspector` на вкладку `Properties` и, используя свойство `Caption`, измените название первой кнопки на `Change label`, второй — на `Exit`.
3. Поместите на форму надпись (компонент `TLabel`). С помощью инспектора объектов присвойте свойству `Caption` этого компонента значение Кнопка "Change label" ни разу не была нажата.

4. Задайте следующий обработчик события `OnClick` для кнопки `Change label`:

```
procedure TForm9.Button1Click(Sender: TObject);
// константа-переменная, используется для подсчета
// щелчков на кнопке
const i : integer = 0;
begin
    inc(i);
// изменение надписи
    Label1.Caption:='Кнопка "'+Button1.Caption+'" нажата'+
        IntToStr(i)+' раз';
end;
```

5. Напишите код завершения программы в обработчике события `OnClick` кнопки `Exit`:

```
procedure TForm9.Button2Click(Sender: TObject);
begin
    Application.Terminate;
end.
```

6. 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 `Change label` надпись изменяется следующим образом: Кнопка "Change label" нажата 1 раз. При каждом последующем щелчке число в надписи увеличивается на 1. На рис. 8.19, б приведено окно программы после пяти щелчков на кнопке `Change label`. Щелчок на кнопке `Exit` приводит к завершению программы.

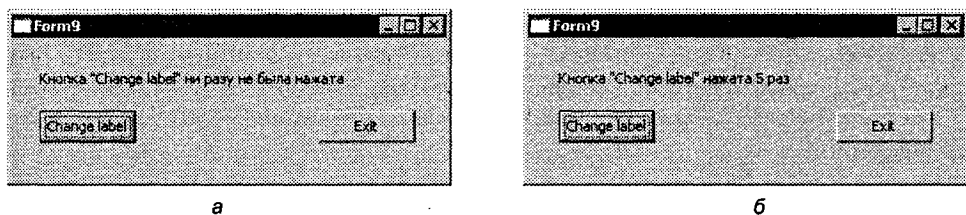


Рис. 8.19. Пример использования надписей

Флажки

Флажок (check box) может находиться в одном из двух состояний: установлен (включен) или сброшен (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.

Флажку 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 сброшен или установлен в «третье состояние».

- ❑ **State: TCheckBoxState** — определяет состояние флажка: установлен (`cbChecked`), снят (`cbUnchecked`) или установлен и закрашен серым (`cbGrayed`). Тип данных для этого свойства определен так:

```
TCheckBoxState = (cbUnchecked, cbChecked, cbGrayed)
```

- ❑ **AllowGrayed: Boolean** — определяет, может (`true`) или нет (`false`) флажок иметь «третье» состояние.

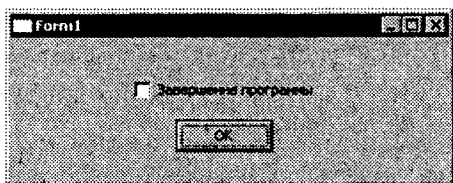
Флажки также имеют свойство `Caption`, с помощью которого может задаваться поясняющая надпись.

Рассмотрим в качестве примера использования флажка управления реакцией на щелчок на кнопке.

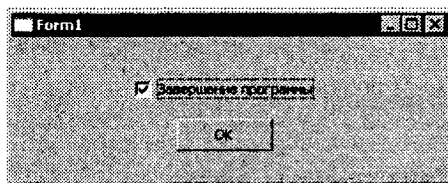
1. С помощью команды **File ► New ► VCL Forms Application** создайте новое приложение.
2. Разместите на форме одну кнопку и один флажок (элемент `TCheckBox`). Затем перейдите в окне **Object Inspector** на вкладку **Properties** и, используя свойство `Caption`, измените название кнопки на **OK**. Аналогично задайте для флажка поясняющую надпись **Завершение программы**.
3. Установите следующий обработчик события `OnClick` кнопки **OK**:

```
procedure TForm1.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 сброшенный флажок (рис. 8.20). Если флажок сброшен, то при щелчке на кнопку ОК ничего происходить не будет. Если же флажок установлен, то щелчок на кнопке приводит к завершению программы.



а



б

Рис. 8.20. Пример использования флажка

Переключатели

Переключатели (radio buttons) 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, установлен он или нет. Все переключатели, помещаемые в один контейнер (форма, фрейм и т. п.), считаются входящими в одну группу, в которой может быть установлен 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 установке переключателя. Если переключатель уже установлен, то щелчок на нем левой кнопки мыши не вызовет события `OnClick`.

Рассмотрим пример, подобный предыдущему, где для управления реакцией на щелчок на кнопке будем использовать переключатели.

1. С помощью команды `File ► New ► VCL Form Application` создайте новое приложение.
2. Разместите на форме одну кнопку и три элемента `TRadioButton`. Затем перейдите в окне `Object Inspector` на 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;
 - Завершение программы — для третьего.
3. Напишите следующий обработчик события `OnClick` кнопки `OK`:

```
procedure TForm1.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. Если установлен первый переключатель (рис. 8.21, а), то при щелчке на кнопке ОК ничего не произойдет. При установке второго переключателя (рис. 8.21, б) щелчок на кнопке вызовет звуковой сигнал. Если установлен третий переключатель, то щелчок на кнопке приведет к завершению программы.

Текстовые поля

Текстовое поле (edit box) 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).

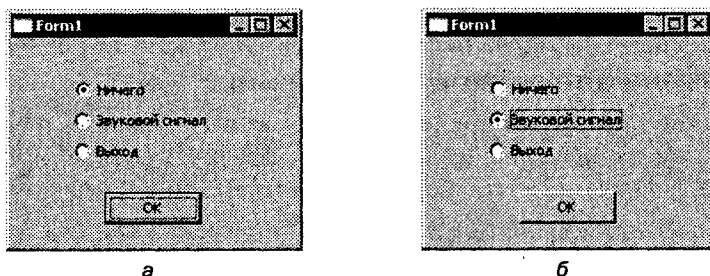


Рис. 8.21. Пример использования переключателей

Таблица 8.3. Основные свойства компонента TEdit

Свойство	Тип	Описание
AutoSelect	Boolean	Определяет, будет ли выделяться текст в поле при получении полем фокуса ввода
CharCase	TEditCharCase = (ecNormal, ecUpperCase, ecLowerCase)	Определяет регистр вводимого текста: текст может вводиться в разных регистрах (ecNormal), вводимый текст преобразуется в символы верхнего регистра (ecUpperCase), вводимый текст преобразуется в символы нижнего регистра (ecLowerCase)
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, то при изменении текста в поле `Edit1` синхронно будет изменяться и текст надписи `Label1`:

```
procedure TForm1.Edit1Change(Sender: TObject);
begin
  Label1.Caption:=Edit1.Text;
end;
```

С помощью полей ввода можно вводить только текстовую информацию. Для ввода чисел следует воспользоваться 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 сгруппированных 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.

- ☐ Компонент `TPanel` представляет собой прямоугольную область, которая может быть «вдавленной» или «выпуклой» относительно формы, на которой она размещена (это определяют свойства `BevelInner` и `BevelOuter` класса `TPanel`).
- ☐ Компонент `TGroupBox` используется для визуального выделения группы элементов управления. В отличие от панели, может иметь заголовок. Других принципиальных отличий нет.

Стандартные элементы управления Windows Forms

Далее в этой книге мы будем касаться в основном программирования с помощью библиотеки `VCL`. Для справки приведем список наиболее часто используемых компонентов библиотеки `FCL` — стандартной библиотеки `.NET` (табл. 8.4)

Таблица 8.4. Стандартные компоненты `FCL`

Элемент управления	Описание
Button	Кнопка, позволяющая выполнять определенные действия
CheckBox	Флажок

Элемент управления	Описание
CheckedListBox	Список с полосой прокрутки и флажками
ColorDialog	Стандартная палитра Windows, предназначенная для выбора цвета
ComboBox	Комбинированный список, представляющий собой поле со списком, в котором можно выбрать один или несколько элементов
ContextMenu	Контекстное меню
CrystalReportViewer	Элемент управления, обеспечивающий поддержку отчетов
DataGrid	Таблица для отображения информации из таблиц баз данных
DateTimePicker	Компонент установки даты и времени
DomainUpDown	Список с полосой прокрутки
ErrorProvider	Окно с сообщением об ошибке, появляющееся при неправильных действиях пользователя
FontDialog	Окно диалога Windows, предназначенное для установки параметров шрифта
GroupBox	Рамка для визуального выделения группы элементов управления
HelpProvider	Кнопка, связанная с HTML-файлом справки Windows-приложения
HScrollBar	Горизонтальная полоса прокрутки
ImageList	Элемент управления, хранящий коллекцию рисунков
Label	Надпись
LinkLabel	Веб-ссылка
ListBox	Список, в котором пользователь может выбрать один или несколько элементов
ListView	Список элементов, представляемый по выбору в виде таблицы, текста с маленькими или большими значками
MainMenu	Главное меню приложения
MonthCalendar	Панель в виде календаря для выбора даты
NotifyIcon	Компонент для обозначения процесса, происходящего в фоновом режиме
NumericUpDown	Список числовых значений с полосой прокрутки
OpenFileDialog	Стандартное окно диалога Windows для открытия файла
PageSetupDialog	Стандартное окно диалога Windows установки параметров страниц
Panel	Панель, используемая для группировки элементов управления
PictureBox	Панель, используемая для размещения на форме растровых рисунков
PrintDocument	Компонент для вывода документа на печать
PrintPreviewControl	Окно предварительного просмотра, задаваемое пользователем
PrintPreviewDialog	Стандартное окно предварительного просмотра Windows
ProgressBar	Индикатор процесса
RadioButton	Переключатель
RichTextBox	Поле для ввода и редактирования текста с разметкой
SaveFileDialog	Стандартное окно диалога Windows для сохранения файла

продолжение ➤

Таблица 8.4 (продолжение)

Элемент управления	Описание
Splitter	Вешка разбивки окна
StatusBar	Строка состояния
TabControl	Панель с вкладками
TextBox	Поле для ввода текстовой информации
Timer	Элемент управления, позволяющий выполнить определенные действия по истечении заданного интервала времени
ToolBar	Панель инструментов, на которой можно расположить другие элементы управления
ToolTip	Текстовая подсказка, появляющаяся при наведении указателя мыши на элемент интерфейса
TrackBar	Ползунок, используемый для задания значения из некоторого диапазона
TreeView	Иерархическая структура узлов, которую можно сворачивать и разворачивать
VScrollBar	Вертикальная полоса прокрутки

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 используется 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 невидимые 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 TМемо) предназначено для отображения и редактирования текста. Основные свойства этого элемента управления приведены в табл. 9.1.

Таблица 9.1. Основные свойства класса TМемо

Свойство	Тип	Описание
Alignment	TAlignment	Способ выравнивания текста в поле: по левому краю (taLeftJustify), по центру (taCenter), по правому краю (taRightJustify)
CaretPos	TPoint	Координаты текстового курсора в поле. Координаты указываются в строках (по вертикали) и символах (по горизонтали)
Lines	TStrings	Массив строк, содержащихся в поле

Текст в поле TМемо может быть выделен с помощью мыши или клавиатуры. Определить выделенный фрагмент текста можно с помощью свойства SelText.

Списки

Список состоит из строк (пунктов списка). Пользователь может просмотреть список и выбрать одну или несколько строк для последующей обработки. Непрямую 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
Columns	Integer	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 Columns определяет количество столбцов
ItemIndex	Integer	Порядковый номер выделенного пункта списка
Items	TStrings	Массив пунктов списка. Используется для добавления, вставки, перемещения и удаления пунктов списка
MultiSelect	Boolean	Задает возможность выбора в списке нескольких пунктов
SelCount	Boolean	Количество выбранных пунктов списка
Selected [Index: Integer]:	Boolean	Определяет, выбран пункт списка с порядковым номером Index или нет
Sorted	Boolean	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>	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 вкладке `Dialogs` палитры компонентов.

Стандартные окна диалога являются невизуальными 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 четыре компонента:

- ☐ TOpenDialog — окно диалога открытия файла;
 - ☐ TSaveDialog — окно диалога сохранения файла;
 - ☐ TOpenPictureDialog — окно диалога открытия графического файла;
 - ☐ TSavePictureDialog — окно диалога сохранения графического 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 3 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:

- ☐ OnClose — вызывается при закрытии окна диалога;
- ☐ OnFolderChange — вызывается при смене каталога;
- ☐ OnSelectionChange — вызывается при изменении списка выделенных файлов;
- ☐ OnTypeChange — вызывается при изменении фильтра;
- ☐ OnCloseQuery — 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 вы-

деления 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.

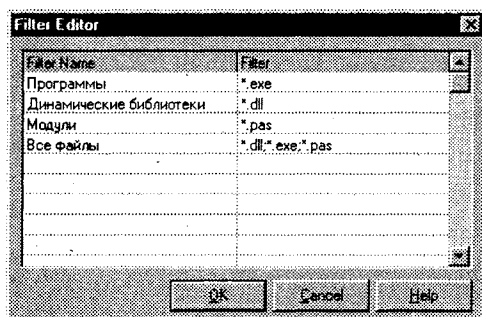


Рис. 9.1. Окно редактора фильтров

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).

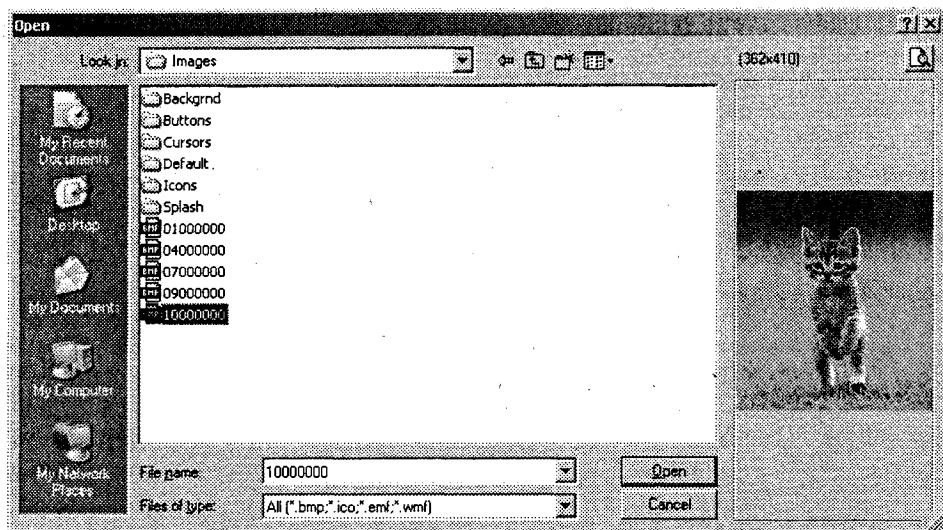


Рис. 9.2. Окно диалога для открытия графического файла с областью предварительного просмотра

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;
- ☐ 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
<code>Color</code>	<code>TColor</code>	Выбранный цвет
<code>CustomColors</code>	<code>TStrings</code>	Дополнительные цвета в шестнадцатеричном коде в формате RGB. Каждый цвет определяется строкой вида <code>ColorA = ff0000 (ColorB = 00ff00 и т. д.)</code>
<code>Options</code>	<code>TColorDialogOptions</code>	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
<code>Collate</code>	<code>Boolean</code>	Состояние флажка Разобрать по копиям
<code>Copies</code>	<code>Integer</code>	Количество копий, заданное в окне диалога
<code>FromPage</code>	<code>Integer</code>	Номер страницы документа, начиная с которой будет производиться печать
<code>MaxPage</code>	<code>Integer</code>	Максимальный номер страницы, который может быть задан в окне диалога
<code>MinPage</code>	<code>Integer</code>	Минимальный номер страницы, который может быть задан в окне диалога
<code>Options</code>	<code>TPrintDialogOptions</code>	Параметры окна диалога
<code>PrintRange</code>	<code>TPrintRange</code>	Определяет диапазон печатаемых страниц: весь документ (<code>prAllPages</code>), выделенный фрагмент (<code>prSelection</code>) или страницы с указанными номерами (<code>prPageNums</code>)
<code>PrintToFile</code>	<code>Boolean</code>	Состояние флажка Печать в файл
<code>ToPage</code>	<code>Integer</code>	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). Обращение к базам данных производится с помощью специальных компонентов, использующих функции BDE.

В Delphi есть возможность доступа к базам данных с использованием *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.

Доступ к данным с использованием BDE

Механизм BDE обеспечивает доступ как к локальным, так и к распределенным базам данных. Взаимодействие с базами данных 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.

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.

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.

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. 9.3.

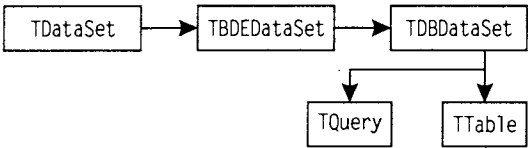


Рис. 9.3. Иерархия классов для доступа к таблицам базы данных

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. 9.9.

Таблица 9.9. 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
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

Свойство	Тип	Описание
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 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
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 полей индекса (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 группы практически идентичен. Различие заключается только в моменте возникновения события и имени обработчика собы-

тия. 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. 9.10.

Таблица 9.10. 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
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

Свойства	Тип	Описание
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. 9.4), который вызывается командой Field 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. 9.5). Для создания вычисляемого поля установите в группе Field type переключатель Calculated.

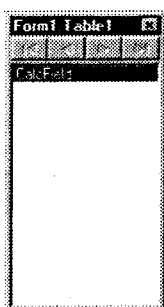


Рис. 9.4. Окно редактора полей

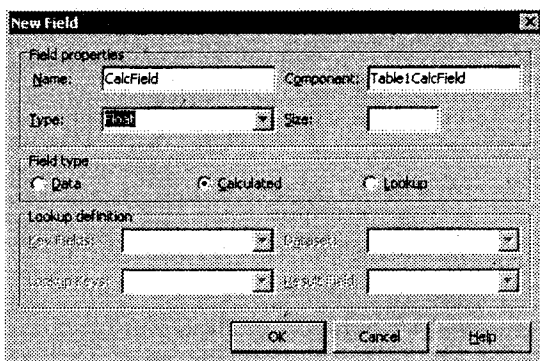


Рис. 9.5. Окно диалога New Field предназначено для создания нового поля

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

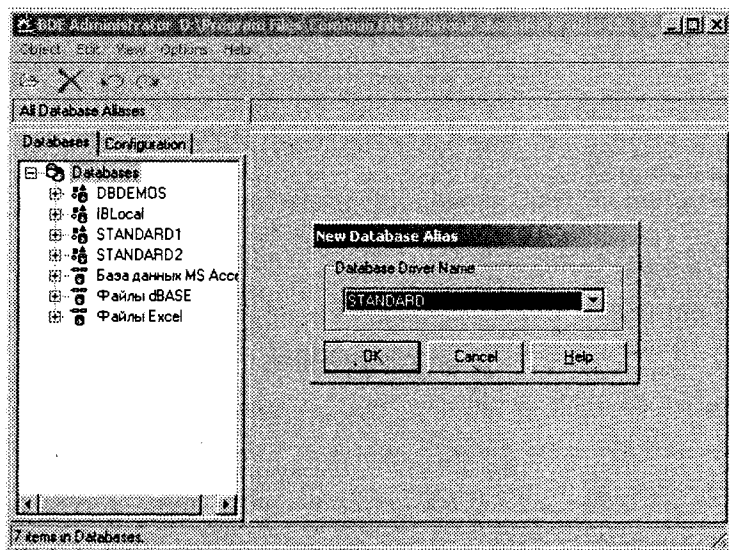
Delphi позволяет работать с таблицами базы данных, не зарегистрированными в BDE. Хотя для этого достаточно указывать полные пути поиска в свойствах соответствующих компонентов, такой подход в корне неправилен. Он не позволяет работать с базой данных как с целостным объектом, напрямую обращаясь к ней 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.

Для добавления новой базы данных необходимо выполнить команду Object ► New и в списке появившегося окна выбрать пункт STANDARD (рис. 9.6, а).

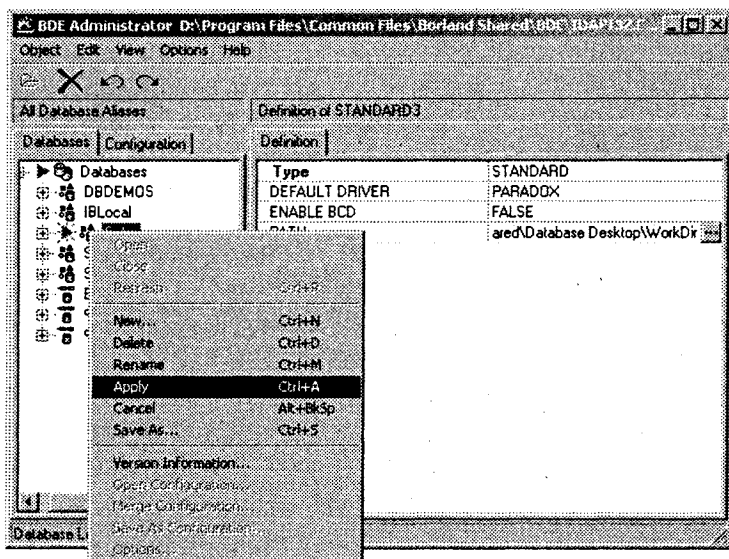
После щелчка на кнопке OK в списке появится новый элемент, помеченный треугольником. Это означает, что регистрация базы данных не завершена. По умолчанию формируется имя базы данных Standard1; имеет смысл изменить его на значащее имя, например на Sales (продажи). Нужно убедиться, что свойства базы данных 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 (рис. 9.6, б).



а

Рис. 9.6. Регистрация базы данных в BDE

Далее база данных в программах будет доступна под именем Sales. Это имя выбирается в списке возможных значений свойства DatabaseName подключаемых таблиц данных (рис. 9.7).



6

Рис. 9.6. Регистрация базы данных в BDE

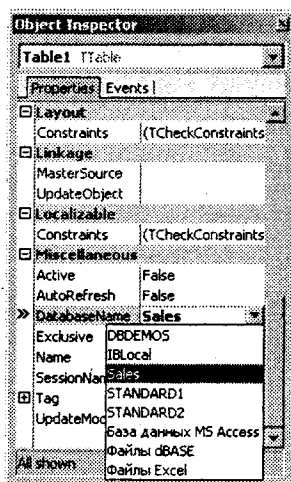


Рис. 9.7. Установка свойства DatabaseName

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 эти компоненты 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 данных 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. 9.11.

Таблица 9.11. 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

Ниже перечислены методы класса 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. Новый модуль данных создается с помощью команды **File** ▶ **New** ▶ **Data Module** главного меню (рис. 9.8).

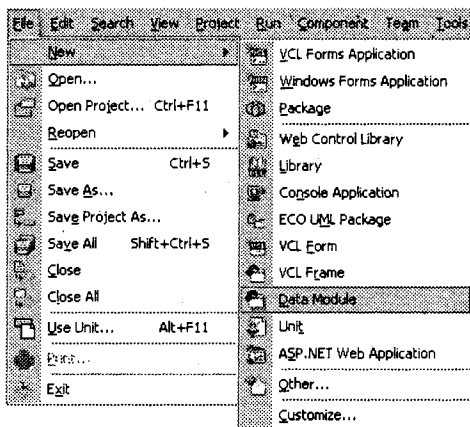


Рис. 9.8. Создание модуля данных

Модули данных представляют собой контейнеры, в которые можно помещать не-визуальные компоненты для доступа к данным: TTable, TQuery, TDataSource и т. п., а также визуально устанавливать связи между ними (рис. 9.9).

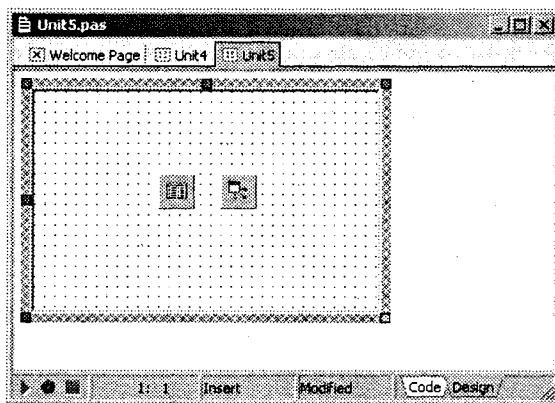


Рис. 9.9. Окно редактора модуля данных

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. 9.12.

Таблица 9.12. 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
SelectedIndex	Integer	Номер текущего столбца

В таблице не обязательно отображать все поля, содержащиеся в наборе данных. Нужные поля выбираются с помощью редактора столбцов (рис. 9.10), вызываемого двойным щелчком на компоненте TDBGrid.

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. 9.13.

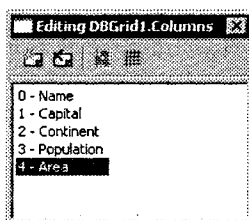


Рис. 9.10. Окно редактора столбцов

Таблица 9.13. Основные свойства класса TColumn

Свойство	Тип	Описание
Alignment	TAlignment	Способ выравнивания данных в столбце
Color	TColor	Цвет фона столбца
DropDownRows	Cardinal	Количество строк в раскрывающемся списке столбца
FieldName	String	Имя поля набора данных, связанного со столбцом
Font	TFont	Шрифт, используемый для отображения данных в столбце
PickList	TStrings	Раскрывающийся список столбца, используемый при редактировании данных
PopupMenu	TPopupMenu	Контекстное меню, связанное со столбцом
Showing	Boolean	Определяет, отображается (true) столбец или нет (false)
Title	TColumnTitle	Заголовок столбца и его параметры
Visible	Boolean	Видимость столбца
Width	Integer	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;
- ☐ отмену 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 данных.

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.

Создание новых компонентов

К настоящему времени для Delphi версии 7 и более ранних версий было создано громадное количество VCL-компонентов. Для Delphi 8 доступных компонентов намного меньше по причине поддержки в этой версии платформы .NET, которая еще не получила широкого распространения.

Большое количество компонентов входит в стандартную поставку Delphi, кроме того, в Интернете можно найти множество компонентов практически на любой вкус (как коммерческих, так и бесплатных). Тем не менее в некоторых случаях может потребоваться создание собственных компонентов:

- ☐ требуемый компонент отсутствует вследствие специфичности выполняемых функций (очень маловероятная ситуация);
- ☐ ни один из существующих компонентов в полной мере не удовлетворяет необходимым требованиям;
- ☐ подходящий компонент существует, но является коммерческим, и цена на него настолько высока, что экономически целесообразнее создать собственный.

Кроме того, необходимо учитывать следующий факт. Компоненты, загруженные из Интернета, представляют собой своего рода «черный ящик» и могут стать источником ошибок (правда, следует отметить, что значительная часть компонентов распространяется вместе с исходными кодами, что позволяет устранять ошибки, возникающие «внутри» компонента).

Создание нового объекта в Delphi является существенно более сложной задачей, чем разработка приложений, и требует знаний WinAPI, технологии .NET, деталей объектно-ориентированного программирования, а также иерархии классов VCL и Windows Forms.

При создании нового компонента можно идти двумя путями:

- ☐ модифицировать уже существующий компонент (класс);
- ☐ создавать новый компонент «с нуля».

Модификация существующих классов

Модификация существующего класса является самым простым путем создания нового компонента (хотя в этом случае, наверное, о *создании* можно говорить лишь условно). Как правило, у большинства VCL-компонентов есть абстрактный предок, у которого объявлены и в основном реализованы все свойства. Обычно имя такого класса начинается с префикса `Custom`, а все его свойства находятся в разделе `protected` и недоступны пользователю. Существующий компонент (компонент, который вы хотите изменить) создается путем опубликования необходимых свойств этого абстрактного предка, то есть переноса их в раздел `published` или `public`. Для изменения существующего компонента следует соответствующим образом переопределить набор доступных методов и свойств нового компонента.

Таким образом, при модификации существующего компонента выполняются следующие действия.

- ☐ Необходимые свойства и методы абстрактного предка компонента делаются доступными, для чего они просто помещаются в раздел `public` или `published` описания класса (в первом случае доступ к свойствам и методам возможен только во время работы приложения, а во втором — как во время работы, так и в среде Delphi с помощью инспектора объектов).
- ☐ Старые свойства и методы переопределяются, а также добавляются новые свойства и методы.

Создание нового класса

Создание нового класса может потребоваться в том случае, если не существует близкого по функциональности VCL-компонента. Тогда в качестве предка выбирается один из базовых классов VCL. Выбор предка и в этом случае определяется функциями создаваемого компонента. Можно выделить четыре наиболее типичных случая:

- ☐ создание оконного элемента управления — в качестве предка следует выбирать `TWinControl`;
- ☐ создание нестандартного оконного элемента управления — `TCustomControl`;
- ☐ создание графического элемента управления — `TGraphicControl`;
- ☐ создание невидимого компонента — `TComponent`.

Кратко рассмотрим особенности перечисленных выше классов.

Все стандартные элементы управления являются потомками класса `TWinControl`, поэтому при создании обычных оконных элементов управления удобнее всего именно его выбирать в качестве предка. Основу функциональности данного класса составляет свойство `Handle`. Наличие этого свойства означает, что компонент может:

- ☐ принимать фокус ввода;
- ☐ передавать дескриптор окна (свойство `Handle`) при вызове функций WinAPI.

Дескриптор необходим системе для определения окна, которое запрашивает тот или иной ресурс. Все кнопки и тому подобные элементы являются потомками клас-

са TWinControl. Класс TWinControl и его потомки не имеют свойства Canvas, поэтому на них нельзя рисовать.

В том случае, если требуется создать компонент, который способен отображать графику, следует в качестве предка выбрать класс TGraphicControl. Этот класс является абстрактным, имеет канву для вывода информации и сообщение WM_Paint для вызова метода рисования. Для прорисовки компонента необходимо просто перекрыть метод Paint в новом компоненте. Компоненты этого класса не могут принимать фокус ввода и, соответственно, Windows не получает от них сообщений, но зато они требуют меньше системных ресурсов. Типичным примером компонента, созданного на базе TGraphicControl, является TLabel. Следует иметь в виду, что, хотя классы-потомки TGraphicControl и имеют свойство Canvas, необходимое для рисования, в действительности используется Canvas компонента, на котором они располагаются. Класс TGraphicControl не имеет свойства Handle.

Класс TCustomControl и его наследники объединяют свойства классов TWinControl и TGraphicControl, то есть обладают свойствами и Handle, и Canvas.

Класс TControl предназначен для создания компонентов, которые не отображаются во время работы приложения (невизуальных компонентов).

Последовательность создания компонента

Независимо от того, каким образом выполняется создание компонента («с нуля» или на основе существующего VCL-компонента), можно выделить общую последовательность действий.

1. Создание нового проекта.
2. Создание свойств.
3. Создание методов.
4. Создание событий.
5. Регистрация компонента.
6. Установка компонента.

Рассмотрим все этапы создания нового компонента более подробно.

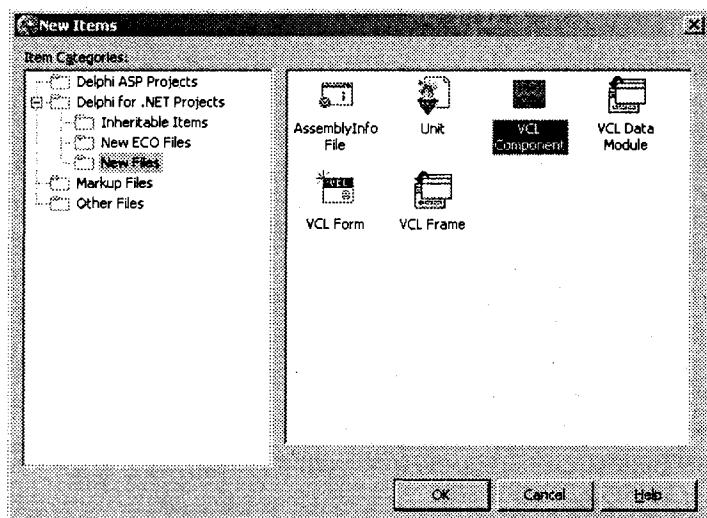
Создание нового проекта

Для создания нового проекта в главном меню IDE Delphi выберите команду File ► New, в открывшемся окне диалога New Items, выделите значок VCL Component (рис. 9.11, а) и щелкните на кнопке ОК. Можно также сразу выбрать команду Component ► New VCL Component в главном меню IDE Delphi (рис. 9.11, б).

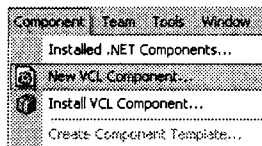
После щелчка на кнопке ОК в окне диалога New Items или выбора команды Component ► New VCL Component откроется окно диалога настройки параметров создаваемого компонента, в котором задаются следующие параметры (рис. 9.12):

- ☐ класс-предок выбирается в списке Ancestor type;
- ☐ имя класса компонента вводится в поле Class Name;
- ☐ страница палитры компонентов, на которую будет устанавливаться создаваемый компонент, выбирается в списке Palette Page;

- ☐ имя файла модуля компонента вводится в поле Unit file name;
- ☐ путь поиска файлов модулей Delphi вводится в поле Search path (обычно это поле заполняется автоматически).



а



б

Рис. 9.11. Создание нового VCL-компонента

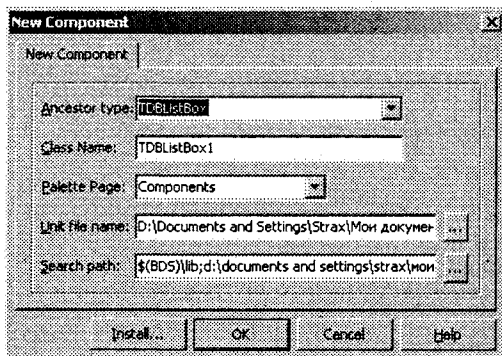


Рис. 9.12. Окно настройки параметров создаваемого компонента

После щелчка на кнопке OK будет создан новый файл модуля, содержащий описание нового класса компонента и процедуру регистрации компонента:

```
unit DBListBoxImg;
```

```
interface
```

```
uses
```

```
Windows, Messages, SysUtils, Classes, Graphics.
```

Controls, Forms, Dialogs, StdCtrls, DBCtrls:

```
type
  TDBListBoxImg = class(TDBListBox)
  private
    { Private declarations }
  protected
    { Protected declarations }
  public
    { Public declarations }
  published
    { Published declarations }
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Samples', [TDBListBoxImg]);
end;

end.
```

Создание свойств

Свойства компонента обычно образуют интерфейс, предоставляющий доступ к внутренним полям. Внутренние поля обычно объявляются в разделе `private` класса компонента и их имена принято начинать с буквы `F`.

Свойства компонента имеют следующие особенности:

- ☐ их можно модифицировать во время конструирования программы с помощью инспектора объектов;
- ☐ свойства скрывают от пользователя свою реализацию;
- ☐ изменение свойства дает немедленный эффект, так как при этом вызывается связанный со свойством метод.

Свойства могут создаваться с использованием нескольких типов данных. Обычно различают свойства следующих типов: простые, перечисляемые, а также свойства-множества, свойства-объекты и свойства-массивы.

Простые свойства

Простые (simple) свойства имеют поля числовых, символьных и строковых типов данных. Такие свойства отображаются в окне инспектора объектов и могут изменяться без специальных редакторов. Например, объявление простых свойств может выглядеть следующим образом:

```
TDBList = class(TCustomListBox)
private
  { Private declarations }
  FNumber: double;
  FChar: char;
```

```

FString: string;
protected
{ Protected declarations }
public
{ Public declarations }
property NumberProp: double read FNumber write FNumber;
published
{ Published declarations }
property CharProp: char read FChar write FChar;
property StringProp: string read FString write FString;
end;

```

Свойства, объявленные в разделе `published`, отображаются в инспекторе объектов и могут изменяться во время разработки приложения. Свойства из раздела `public` могут модифицироваться только во время выполнения программы.

Перечисляемые свойства

Перечисляемые (enumerated) свойства имеют поля перечисляемых типов данных. Такие свойства представляются в окне инспектора объектов в виде раскрывающегося списка. Если свойство имеет перечисляемый тип, определяемый пользователем, то объявление этого типа должно предшествовать объявлению класса компонента:

```

TEnum = (epFirst, epSecond, epThird);
TDBList = class(TCustomListBox)
private
{ Private declarations }
FNumber: double;
FChar: char;
FString: string;
FEnum: TEnum;
protected
{ Protected declarations }
public
{ Public declarations }
property NumberProp: double read FNumber write FNumber;
published
{ Published declarations }
property CharProp: char read FChar write FChar;
property StringProp: string read FString write FString;
property EnumProp: string read FEnum write FEnum;
end;

```

Свойства-множества

Свойства-множества (set) имеют поля с типом данных «множество». В инспекторе объектов слева от имени такого поля отображается символ плюс (+), при щелчке на котором разворачивается список всех возможных элементов множества с указанием значения `true` или `false`, которое показывает, включено данное значение в множество или нет. Примером такого свойства может служить свойство `Anchor` компонента `TForm` (рис. 9.13).

Так же как и в случае перечисляемого свойства, тип множества должен быть описан прежде класса компонента.

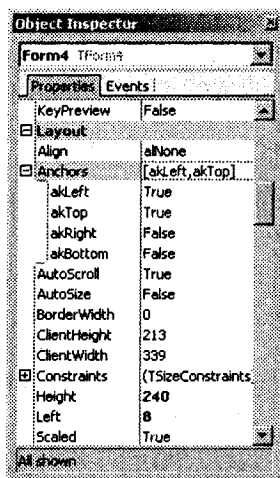


Рис. 9.13. Пример отображения свойства-множества в окне инспектора объектов

Свойства-объекты

Свойства-объекты (object) включают отдельные классы и объекты. Так же как и свойства-множества, свойства-объекты помечаются в окне инспектора объектов символом плюс (+) слева от имени свойства (примером может служить свойство Font), а в поле ввода значения для таких свойств в инспекторе объектов отображается кнопка с многоточием. Щелчок на кнопке приводит к открытию редактора свойств для данного поля.

Свойства-массивы

К свойствам-массивам (которые также называются векторными свойствами) можно обращаться как к обычным массивам языка Object Pascal. Опубликованные свойства-массивы нельзя редактировать непосредственно в инспекторе объектов — необходим специальный редактор свойств. В поле ввода значений таких свойств в инспекторе объектов так же, как и для свойств-объектов, отображается кнопка с многоточием, с помощью которой и производится обращение к редактору свойств. При объявлении свойства-массива после ключевых слов read и write нельзя просто указать имя поля — для доступа к внутренним полям необходимо применять методы доступа, которые будут рассмотрены далее.

Чтение и запись значений свойств

Как уже отмечалось, свойства не являются полями класса, а представляют собой лишь интерфейс для доступа к полям. Способ взаимодействия с полями определяется с помощью ключевых слов read и write. Если после этих слов указывается просто имя соответствующего поля (в приводимых выше примерах использовалась именно такая форма обращения к полям), то обращение к свойству эквивалентно прямому обращению к полю. Однако часто для записи и чтения значений полей используют специальные методы, которые называются *методами доступа*. Такие методы представляют собой функции (методы чтения) и процедуры (методы записи).

Обычно имена методов чтения начинаются с префикса `Get`, а имена методов записи — с префикса `Set`. Методы чтения вызываются без параметров, а методы записи имеют один параметр, который может передаваться как по ссылке, так и по значению.

При использовании методов доступа для обращения к полю их имена просто указываются после ключевых слов `read` и `write` в объявлении свойства:

```
TDBList = class(TCustomListBox)
private
{ Private declarations }
FString: string;
function GetStringProp: string; virtual;
procedure SetStringProp(const Value: string); virtual;
protected
{ Protected declarations }
public
{ Public declarations }
published
{ Published declarations }
property StringProp: string read GetStringProp
write SetStringProp;
end;
...
function TDBList.GetStringProp: string;
begin
    result:=FString;
end;
function TDBList.SetStringProp(const Value: string);
begin
    FString:=Value;
end;
...
```

Инициализация значений свойств

Инициализация значений свойств может выполняться двумя способами: с помощью конструктора `Create` или метода `Loaded`. Различие между этими двумя способами состоит в очередности вызова. При создании компонента (помещении его на форму) выполняется следующая последовательность действий.

1. Создается экземпляр компонента, то есть для него выделяются необходимые ресурсы.
2. Вызывается конструктор.
3. Из `dfm`-файла считываются сохраненные данные компонента.
4. Вызывается метод `Loaded`.
5. Выполняется отображение компонента на форме.

Таким образом, конструктор вызывается до считывания сохраненных данных, а метод `Loaded` — после.

ПРИМЕЧАНИЕ

При переопределении конструктора `Create` и метода `Loaded` всегда необходимо в начале вызывать унаследованные методы с использованием директивы `inherited`. Только в этом случае все наследуемые свойства будут инициализироваться корректно.

Рассмотрим пример инициализации свойств путем переопределения конструктора. При переопределении необходимо объявить конструктор в классе компонента:

```
TDBList = class(TCustomListBox)
private
{ Private declarations }
FString: string;
function GetStringProp: string; virtual;
procedure SetStringProp(const Value: string); virtual;
protected
{ Protected declarations }
public
{ Public declarations }
constructor Create(AOwner: TComponent); override;
published
{ Published declarations }
property StringProp: string read GetStringProp
write SetStringProp;
end;
```

А затем в разделе `implementation` задать код реализации конструктора:

```
...
constructor TDBList.Create(AOwner: TComponent);
begin
    // вызываем унаследованный конструктор
    inherited Create(AOwner);
    // инициализируем свойство
    FString := 'Initial Value';
end;
...
```

Создание методов

Методы компонентов ничем не отличаются от обычных методов Object Pascal. Можно создавать методы пяти видов: статические, динамические, виртуальные, обработки сообщений и абстрактные.

Виртуальные и динамические методы могут перекрываться в классах-потомках. Статические методы не могут быть переопределены. При переопределении наследуемых методов, как правило, необходимо вызывать унаследованный метод.

Создание событий

Событие — это обычное свойство процедурного типа. Стандартные события имеют тип `TNotifyEvent`. Однако при создании события пользователь может задействовать любой процедурный тип, объявленный ранее класса компонента.

Само событие обрабатывается с помощью специального метода, который называется *методом отправления уведомления о событии*. Данный метод и выполняет обработку события, а также проверяет, назначен ли пользователем соответствующий обработчик события. Если обработчик назначен, то метод отправления уведомления вызывает его. По соглашению имена свойств-обработчиков событий начинаются с префикса `On`, а имена методов отправления уведомления о событии имеют то же имя, что и свойство, но без префикса (например, `OnDbClick` — имя свойства-обработчика, `DbClick` — имя метода).

Методы отправления уведомления о событии представляют собой процедуры, которые вызываются в том случае, когда компонент получает сообщение о произошедшем событии. Как правило, эти методы являются виртуальными и объявляются в разделе `protected` класса компонента.

Таким образом, при создании события необходимо создать метод отправления уведомления о событии, а также объявить поле и свойство, которые будут соответствовать обработчику события, задаваемому пользователем:

```
TDBList = class(TCustomListBox)
private
{ Private declarations }
FString: string;
FOnChange: TNotifyEvent;
function GetStringProp: string; virtual;
procedure SetStringProp(const Value: string); virtual;
protected
{ Protected declarations }
procedure Change; virtual;
public
{ Public declarations }
constructor Create(AOwner: TComponent); override;
published
{ Published declarations }
property StringProp: string read GetStringProp
    write SetStringProp;
property OnChange: TNotifyEvent read FOnChange
    write FOnChange;
end;
```

Стандартный код метода отправления уведомления о событии выглядит следующим образом:

```
procedure TDBList.Change;
begin
    // Обработка события
    ...
    // Вызов обработчика события, заданного пользователем:
    if Assigned(FOnChange)
    then FOnChange(Self);
end;
```

Регистрация и установка компонента

Чтобы компонент можно было использовать в приложениях, необходимо включить его в палитру компонентов.

1. Выберите команду **Component ► Install VCL Component** в главном меню IDE Delphi. При этом откроется окно диалога **Install Component** (рис. 9.14).
2. Если компонент устанавливается в уже существующий пакет, то все остальные настройки будут выполняться на вкладке **Into existing package**. Если вы хотите установить компонент в новый пакет, то следует перейти на вкладку **Into new package** и в списке **Package file name** указать имя файла пакета (или задать новое имя файла), в который будет установлен компонент.

3. В поле Unit file name укажите имя модуля, содержащего описание компонента.
4. Щелкните на кнопке OK, а затем в открывшемся окне диспетчера проекта — на кнопке Install. После этого произойдет компиляция пакета, и все содержащиеся в нем компоненты будут установлены в палитру компонентов.

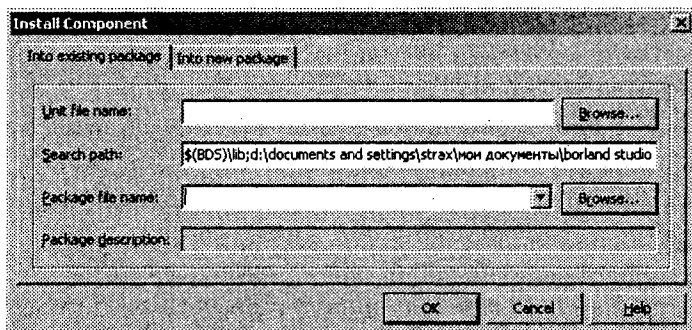


Рис. 9.14. Окно диалога Install Component

Особенности создания компонентов для управления данными

Создание компонентов для управления данными в целом аналогично созданию обычных компонентов, но имеет одну особенность. Эта особенность заключается в том, что при создании компонентов управления данными необходимо использовать *объект связи с данными* (data-link object). Данный объект обеспечивает взаимодействие между компонентом и классом TDataSource. Базовым классом для объектов связи с данными является класс TDataLink. На его основе создан ряд других классов, которые обычно и применяются при создании полей компонента, обеспечивающих связь с источником данных:

- ❑ TGridDataLink — используется в компонентах, работающих со многими полями набора данных (например, в компоненте TDBGrid);
- ❑ TMasterDataLink — служит для задания главного набора данных при работе со связанными таблицами базы данных;
- ❑ TFieldDataLink — используется в компонентах, работающих только с одним полем из набора данных;
- ❑ TNavDataLink — применяется в компонентах навигации по набору данных;
- ❑ TDataSourceLink, TListSourceLink — используются в компонентах синхронного просмотра данных.

Основные свойства и методы классов для связи с данными

Базовый класс TDataLink имеет ряд важных свойств и методов, которые необходимо знать при создании компонентов для управления данными.

- ❑ Метод `UpdateRecord` используется для уведомления объекта связи с данными о том, что ему необходимо обновить значения данных в элементе управления, в который он входит. Это свойство вызывается автоматически при потере элементом управления фокуса ввода.
- ❑ Свойство `Active` позволяет определить, открыт или нет набор данных, к которому подключен объект связи с данными.
- ❑ Свойство `ActiveRecord` служит для считывания или установки номера текущей записи в наборе данных.
- ❑ Свойство `DataSet` представляет собой ссылку на компонент набора данных, с которым связан объект связи с данными.

Помимо класса `TDataLink` большое значение также имеет один из его наследников — класс `TFieldDataLink`. Рассмотрим его основные методы и свойства.

- ❑ Метод `Edit` используется для переключения в режим редактирования поля данных. Этот метод возвращает результат логического типа, который указывает, было выполнено переключение в режим редактирования (`true`) или нет (`false`). Режим редактирования не включается в том случае, если набор данных предназначен только для чтения.
- ❑ Метод `Modified` вызывается для внесения изменений значения поля в базу данных.
- ❑ Метод `Reset` восстанавливает исходные значения поля данных. Изменения, внесенные в компоненте управления данными, при этом теряются.
- ❑ Свойство `CanModify` определяет, может ли объект связи с данными изменять информацию в базе данных.
- ❑ Свойство `Field` является ссылкой на поле (класс `TField`), с которым соединен объект связи с данными.
- ❑ Свойство `FieldName` используется для считывания или установки имени поля, с которым соединен объект связи с данными.
- ❑ Событие `OnActiveChange` вызывается при изменении свойства `Active` набора данных, с которым соединен объект связи с данными.
- ❑ Событие `OnDataChange` вызывается при изменении данных в поле, с которым соединен объект связи с данными. Это событие используется для обновления значений, отображаемых в элементе управления.
- ❑ Событие `OnEditingChange` вызывается при изменении состояния редактирования набора данных.
- ❑ Событие `OnUpdateData` вызывается перед внесением изменений из набора данных в базу данных.

В заключение рассмотрим пример создания элемента управления данными. Создадим список (компонент `ListBox`), в котором порядковый номер выделенного пункта списка должен соответствовать числу, записанному в поле связанной со списком таблицы базы данных. А при выборе пользователем другого пункта в списке значение поля базы данных будет заменяться порядковым номером выбранного пункта списка. В случае если числу, хранимому в поле таблицы базы данных, не

может быть поставлен в соответствие пункт списка, список должен стать недоступным для редактирования. Проще всего создать такой элемент на базе класса TListBox.

Приведем полный текст модуля, содержащего код компонента:

```
unit DBList;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Db, DBCtrls;

type
  TDBList = class(TListBox)
  private
    { Private declarations }
    FDataLink: TFieldDataLink;
    function GetDataField: string; virtual;
    function GetDataSource: TDataSource; virtual;
    procedure SetDataField(const AValue: string); virtual;
    procedure SetDataSource(const AValue: TDataSource); virtual;
    procedure DataChange(Sender: TObject);
    procedure UpdateData(Sender: TObject);
  protected
    { Protected declarations }
    procedure Click; override;
  public
    { Public declarations }
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    { Published declarations }
    property DataSource: TDataSource read GetDataSource
      write SetDataSource;
    property DataField: string read GetDataField
      write SetDataField;
  end;

procedure Register;

implementation

constructor TDBList.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FDataLink:=TFieldDataLink.Create;
  FDataLink.OnDataChange:=DataChange;
  FDataLink.OnUpdateData:=UpdateData;
end;

destructor TDBList.Destroy;
begin
  FDataLink.OnDataChange:=nil;
  FDataLink.OnUpdateData:=nil;
  FDataLink.Free;
  inherited Destroy;
end;
```

```
function TDBList.GetDataField: string;
begin
    result:=FDataLink.FieldName;
end;

function TDBList.GetDataSource: TDataSource;
begin
    result:=FDataLink.DataSource;
end;

procedure TDBList.SetDataField(const AValue: string);
begin
    FDataLink.FieldName:=AValue;
end;

procedure TDBList.SetDataSource(const AValue: TDataSource);
begin
    FDataLink.DataSource:=AValue;
end;

procedure TDBList.DataChange(Sender: TObject);
begin
    if FDataLink.Field<>nil
    then if FDataLink.Field.AsInteger<=Items.Count
    then begin
        Enabled:=true;
        ItemIndex:=FDataLink.Field.AsInteger
    end
    else Enabled:=false;
end;

procedure TDBList.UpdateData(Sender: TObject);
begin
    if Enabled
    then FDataLink.Field.Value:=ItemIndex;
end;

procedure TDBList.Click;
begin
    if FDataLink.Edit
    then begin
        inherited Click;
        FDataLink.Modified;
        FDataLink.UpdateRecord;
    end;
end;

procedure Register;
begin
    RegisterComponents('Samples', [TDBList]);
end;

end.
```

Прокомментируем некоторые фрагменты приведенного кода. Процедуры `DataChange` и `UpdateData` представляют собой обработчики событий `OnDataChange` и `OnDataUpdate` класса `TFieldDataLink`. Первый из них (`DataChange`) используется для выбора соот-

ветствующего пункта в списке при изменении данных в таблице базы данных (или при переходе на другую запись таблицы). Второй обработчик события (UpdateData) требуется для занесения в базу данных информации при смене выделенного пункта в списке.

Процедура Click — это метод отправления уведомления о событии, заданный в родительском классе (TListBox). Мы переопределяем его, дополняя новыми функциями — обновлением информации в базе данных.

Переопределение конструктора здесь необходимо для создания объекта TFieldDataLink. Кроме того, в конструкторе мы задаем обработчики событий OnDataChange и OnDataUpdate класса TFieldDataLink.

Переопределение деструктора требуется для освобождения ресурсов, занятых объектом TFieldDataLink.

ПРИМЕЧАНИЕ

В настоящее время широкое применения находят специальные средства, такие как ModelMaker (для Delphi 7) и Component Studio One (для Delphi 8). Они позволяют визуализировать процесс создания класса, причем не обязательно элемента управления.

ГЛАВА 10 Создание форм для ввода и редактирования данных

Большинство приложений, разрабатываемых в среде Delphi, должны содержать, по крайней мере, одну форму. Исключения составляют консольные приложения и ряд других.

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, упрощающего создание форм для отображения и редактирования данных как для одной таблицы, так и для нескольких связанных таблиц.

Формы в Delphi

Форма представляет окно приложения, 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 оконный интерфейс двух типов: *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 и нет. *Опубликованные свойства* отображаются в окне инспектора объектов во время разработки приложения, где их можно менять, а неопубликованные свойства 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 автоматического создания 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, необходимость отображения строки меню и 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
Ctl3D	Boolean	Определяет внешний вид окна: «трехмерное» или «плоское»
Cursor	TCursor	Вид указателя мыши над формой
FormStyle	TFormStyle = (fsNormal, fsMDIChild, fsMDIForm, fsStayOnTop)	Стиль окна: обычное окно SDI (fsNormal); окно SDI, всегда остающееся поверх всех других окон (fsStayOnTop); главная форма MDI (fsMDIForm); дочерняя форма MDI (fsMDIChild)

Таблица 10.1 (продолжение)

Свойство	Тип	Описание
Height	Integer	Высота окна в пикселах
Icon	TIcon	Значок формы, отображающийся в верхнем левом углу окна
Left	Integer	Левая граница формы в координатах экрана
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 во весь экран (wsMaximized), обычное состояние (wsNormal)
Tag	LongInt	Не имеет никакого предопределенного 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 (наличие кнопок в заголовке определяется свойством **BorderIcons**);
- ☐ **bsDialog** — форма для создания окон диалога, не меняющая размеры во время выполнения программы (независимо от значения свойства **BorderIcons** в заголовке выводится только кнопка 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)

Свойство	Тип	Описание
ModalResult	TmodalResult	Величина, возвращаемая функцией ShowModal при закрытии модального окна
MDIChildCount	Integer	Количество дочерних окон (только для чтения)
MDIChildren[i : integer]	Tform	Указывает на i-е дочернее окно (только для чтения)

Помимо свойств, класс TForm включает ряд методов, которые могут быть полезны при разработке приложения. Ниже перечислены основные методы класса TForm.

procedure Close;

Вызывает метод CloseQuery и, если он возвращает true, закрывает форму.

function CloseQuery: Boolean;

Позволяет определить, 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. Рассмотрим основные из них:

- ☐ OnActivate — вызывается при передаче форме фокуса ввода;
- ☐ OnClick — вызывается при одиночном щелчке на форме;
- ☐ OnDblClick — 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 клавиши;
- ❑ 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 на форме.

С помощью методов-обработчиков событий можно выполнить ряд действий, не применяя дополнительных элементов управления. Для иллюстрации 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.

1. Выберите команду File ► New ► VCL Forms Application.
2. Перейдите в окне Object Inspector на вкладку Events и дважды щелкните в 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;
```

```
end;
```

4. Перейдите в окне Object Inspector на вкладку Events и дважды щелкните в поле значения события OnDblClick. Окно редактора кода станет активным и в него будет автоматически добавлен заголовок процедуры-обработчика события OnDblClick:

```
procedure TForm1.FormDblClick(Sender: TObject);  
begin
```

```
end;
```

5. Напишите код завершения программы в теле процедуры FormDblClick. Для этого воспользуйтесь методом Terminate класса TApplication:

```
procedure TForm1.FormDblClick(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

В состав пятой версии Delphi введен новый компонент — фрейм. *Фреймы* (frames) 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 ► VCL Frame (эта команда становится доступной после выполнения команды File ► New ► VCL Forms Applications). 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 в списке 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.

Фреймы предназначены для более гибкого и эффективного объединения и хранения объектов. Фрейм может быть использован как шаблон для быстрого создания новых фреймов нужной структуры. Допустимы неограниченные вложения фреймов друг в друга. При этом если вносятся изменения в шаблон, то автоматически изменяются все фреймы, созданные на его основе.

Использование базовых классов для создания форм ввода

В Delphi при создании форм используется класс TForm. Класс TForm фактически является контейнерным классом, позволяющим включать в себя любое количество элементов управления, а также другие контейнеры, такие как TGroup, TPanel и т. п.

Размещение и удаление элементов управления

Для создания новой формы используется команда File ► New ► New VCL Forms Applications 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 указателя мыши при этом задает место вставки левого верхнего угла

элемента управления, а размер размещаемого компонента определяется параметрами, используемыми в Delphi по умолчанию. Однако можно одновременно с размещением элемента задавать 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, затем кнопку мыши отпустите. После размещения компонента его размеры можно изменять либо с помощью инспектора объектов, либо мышью.

ПРИМЕЧАНИЕ

Для добавления на форму нескольких одинаковых компонентов при выборе компонента в палитре нажмите клавишу Shift. После этого при каждом щелчке левой кнопки мыши на форме будет добавляться экземпляр выбранного компонента.

Для удаления компонента с формы выделите его щелчком мыши и нажмите клавишу Del. Можно также воспользоваться командой Edit ► Delete главного меню.

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 выборе команды View ► Alignment Palette главного меню, или через окно диалога Alignment, открывающееся при выполнении команды Align контекстного меню редактора форм.

Команды выравнивания, как правило, применяются к группе выделенных компонентов. Для выделения компонентов можно воспользоваться двумя способами:

- ☐ удерживая клавишу Shift, последовательно щелкайте на нужных компонентах;
- ☐ удерживая левую кнопку мыши нажатой, обведите область формы, на которой расположены выделяемые компоненты, контуром выделения (рис. 10.1).

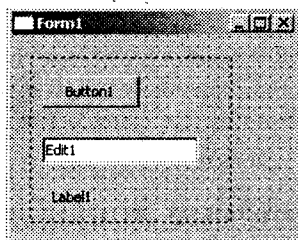


Рис. 10.1. Выделение компонентов мышью

Для выравнивания компонентов можно использовать либо окно диалога Alignment, открывающееся командой Align контекстного меню редактора форм (рис. 10.2), либо панель инструментов Align (рис. 10.3).

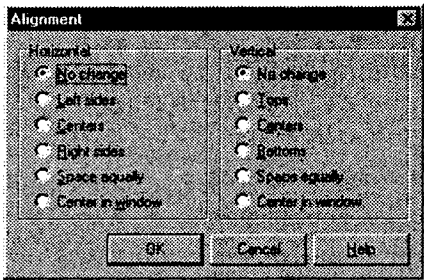


Рис. 10.2. Окно диалога Alignment



Рис. 10.3. Панель инструментов Align

Окно диалога Alignment содержит две группы переключателей, управляющих выравниванием выделенных компонентов по горизонтали (Horizontal) и по вертикали (Vertical). Переключателям группы Horizontal соответствует верхний ряд кнопок панели инструментов Align, переключателям группы Vertical — нижний. Описание команд выравнивания приводится в табл. 10.3.

Таблица 10.3. 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 выделить компонент. Для этого достаточно щелкнуть на нем, после чего контур выделенного компонента отображается 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. Однократное нажатие любой из них приводит к перемещению компонента на один пиксел в соответствующем направлении.

Для одновременного изменения размеров группы выделенных компонентов используются элементы управления окна диалога Size (рис. 10.4), открываемого с помощью команды Position ► Size контекстного меню редактора форм.

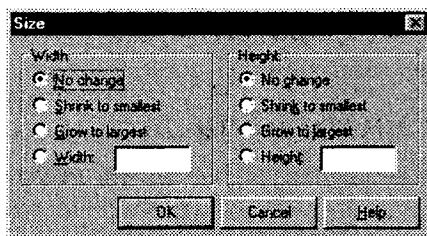


Рис. 10.4. Окно диалога Size

Данное окно диалога содержит две группы переключателей, используемых для одновременного изменения ширины (Width) и высоты (Height) группы выделенных элементов. Назначение переключателя приведено в табл. 10.4.

Таблица 10.4. Переключатели окна диалога Size

Переключатель	Описание
No change	Размеры элемента не меняются
Shrink to smallest	Ширина (или высота) всех выделенных элементов устанавливается равной минимальной ширине (или высоте)

Переключатель	Описание
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 задействовать 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 клавиши 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 значение свойства TabOrder равно нулю, получит фокус ввода при открытии формы.

Свойство TabStop определяет, может элемент получить фокус ввода (true) или нет (false).

Порядок обхода компонентов может задаваться командой Tab Order контекстного меню.

Настройка внешнего вида формы

Настройка внешнего вида формы производится установкой свойств, определяющих ее размеры, положение на экране, заголовок, цвет фона и значок, отображаемый в левом верхнем углу формы (эти свойства можно задать с помощью инспектора объектов).

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 главе 9). 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, помимо 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 — используется в качестве первичного ключа и имеет тип `Integer`;
- ☐ Фамилия, Имя, Отчество, Телефон, Индекс, Страна, Город, Адрес — текстовые поля;
- ☐ Дата рождения — поле типа `TDateField`;
- ☐ Пол — поле типа `Boolean`.

Для отображения текстовых полей и поля *Дата рождения* будем использовать компоненты `TDBEdit`. Логические поля удобнее отображать с помощью флажков — компонентов `TDBCheckBox`. Кроме того, на форму необходимо поместить компонент `TDBNavigator` для навигации по набору данных, а также несколько компонентов `TLabel`, с помощью которых мы 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 мы работаем только с одной таблицей, то использовать модуль не имеет смысла, и компоненты доступа к данным можно поместить прямо на форму.
2. Разместите на форме компонент `TTable`. Затем перейдите в палитре компонентов на вкладку `Data Access` и установите на форму компонент `TDataSource`. Последний необходим для связи набора данных с компонентами визуализации данных.
3. Теперь подключите к компоненту `TTable` таблицу *Физические лица* базы данных *Sales*. Как это сделать, описано в главе 9.
4. Выделите на форме компонент `TTable` и 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, используется свойство `DataSet`.
5. С помощью инспектора объектов укажите в свойстве `DataSet` имя объекта `TTable` (по умолчанию — `Table1`).

6. Разместите на форме необходимые элементы управления и выполните их настройку. Примерный вариант размещения компонентов показан на рис. 10.5.

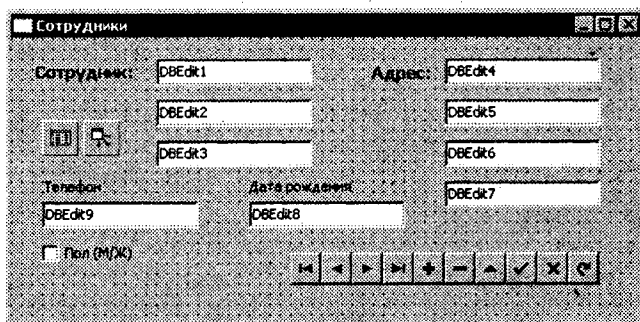


Рис. 10.5. Пример размещения элементов управления на простой форме

7. Для настройки элементов визуализации полей базы данных (девять полей DBEdit и флажок TDBCheckBox) 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 написать 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 TTable, для закрытия — метод Close того же класса.

8. Вызовите метод Open в обработчике события OnShow главной формы, а метод Close — в обработчике OnClose. Текст модуля разработанной формы приведен в листинге 10.1.
9. Откомпилируйте и запустите программу. Внешний вид окна программы приведен на рис. 10.6.

Листинг 10.1. Главный модуль приложения с простой формой для ввода данных

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls, DBCtrls, StdCtrls, Mask, Db;

type
  TfrmSimple = class(TForm)
    Label1: TLabel;
    DBEdit1: TDBEdit;
    DBEdit2: TDBEdit;
    DBEdit3: TDBEdit;
```

```

Label2: TLabel;
Label3: TLabel;
DBEdit4: TDBEdit;
DBCheckBox1: TDBCheckBox;
Label4: TLabel;
DBEdit5: TDBEdit;
DBEdit6: TDBEdit;
DBEdit7: TDBEdit;
DBEdit8: TDBEdit;
DBEdit9: TDBEdit;
Table1: TTable;
DataSource1: TDataSource;
DBNavigator1: TDBNavigator;
procedure FormShow(Sender: TObject);
procedure FormClose(Sender: TObject);
var Action: TCloseAction);
private
{ Private declarations }
public
{ Public declarations }
end;

var
  frmSimple: TfrmSimple;

implementation

{$R *.DFM}

procedure TfrmSimple.FormShow(Sender: TObject);
begin
  Table1.Open;
end;

procedure TfrmSimple.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  Table1.Close;
end;

end.

```

Рис. 10.6. 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 главе 9.

ПРИМЕЧАНИЕ

К сожалению, естественное для компонентов Delphi типа `TStringGrid` и `TDBGrid` название «таблица» начинает путаться с понятием таблицы базы данных (компонент `TTable`). Поэтому иногда мы будем использовать дословный перевод — «сетка».

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*, в которой содержатся сведения о работниках некоторой фирмы.

1. Создайте новое приложение с помощью команды **File** ► **New** ► **VCL Forms Application**.
2. Поместите на форму компонент `TTable`. Затем для создания связи между компонентом доступа к данным и элементом `TDBGrid` поместите на форму компонент `TDataSource`. Настройка этих компонентов выполняется в соответствии с описанием, приведенным в главе 9.
3. Поместите на форму компонент `TDBTable`. С помощью инспектора объектов задайте значение свойства `Align` данного компонента равным `alClient`. При этом размеры сетки 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 `TTable` или установить значение `true` для свойства `Active` данного объекта.
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. Выполнив все перечисленные действия, вы получите форму, внешний вид которой показан на рис. 10.7.

Чтобы во время работы программы в таблице отображалась информация, хранящаяся в таблице, мы воспользуемся двумя методами-обработчиками событий `OnCreate` и `OnClose`. Первый выполняется при создании 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 `OnCreate` выберите в раскрывающемся списке в инспекторе объектов компонент `Form1` и перейдите на вкладку `Events`. Выполните двойной щелчок в поле ввода для события `OnCreate`. При этом

IDE Delphi автоматически переключится в редактор кода и сгенерирует заголовок процедуры-обработчика выбранного события. В теле данной процедуры напишите код открытия набора данных (процедура FormShow в листинге 10.2).

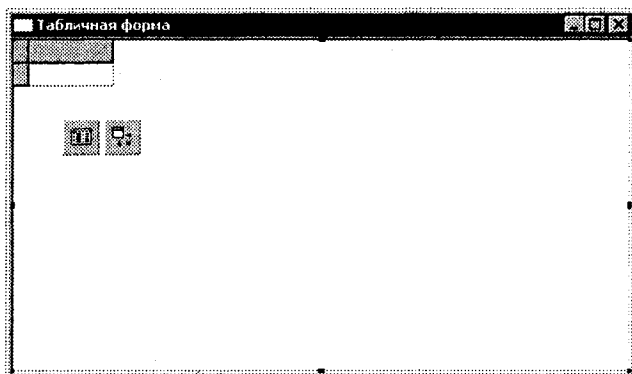


Рис. 10.7. Внешний вид табличной формы в редакторе форм

7. Аналогично задайте обработчик события OnClose, закрывающий набор данных (процедура FormClose в листинге 10.2).
8. Откомпилируйте и запустите программу.

Листинг 10.2. Модуль приложения табличной формы

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, Db, Grids, DBGrids;

type
  TForm1 = class(TForm)
    DBGrid1: TDBGrid;
    DataSource1: TDataSource;
    Table1: TTable;
  procedure FormCreate(Sender: TObject);
  procedure FormClose(Sender: TObject; var Action: TCloseAction);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}
```

```

procedure TForm1.FormCreate(Sender: TObject);
begin
  Table1.Open
end;

```

```

procedure TForm1.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  Table1.Close
end;

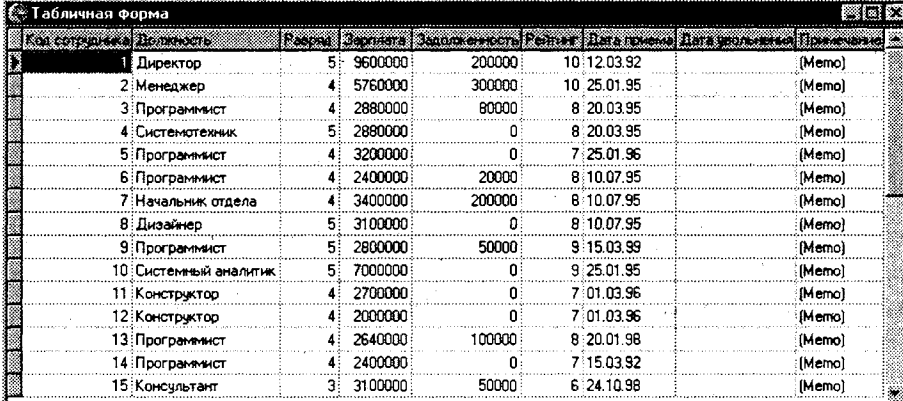
```

```

end.

```

Внешний вид окна программы приведен на рис. 10.8.



Код сотрудника	Должность	Рейс	Зарплата	Задолженность	Рейс	Дата приема	Дата увольнения	Примечание
1	Директор	5	9600000	200000	10	12.03.92		(Memo)
2	Менеджер	4	5760000	300000	10	25.01.95		(Memo)
3	Программист	4	2880000	80000	8	20.03.95		(Memo)
4	Системотехник	5	2880000	0	8	20.03.95		(Memo)
5	Программист	4	3200000	0	7	25.01.96		(Memo)
6	Программист	4	2400000	20000	8	10.07.95		(Memo)
7	Начальник отдела	4	3400000	200000	8	10.07.95		(Memo)
8	Дизайнер	5	3100000	0	8	10.07.95		(Memo)
9	Программист	5	2800000	50000	9	15.03.99		(Memo)
10	Системный аналитик	5	7000000	0	9	25.01.95		(Memo)
11	Конструктор	4	2700000	0	7	01.03.96		(Memo)
12	Конструктор	4	2000000	0	7	01.03.96		(Memo)
13	Программист	4	2640000	100000	8	20.01.98		(Memo)
14	Программист	4	2400000	0	7	15.03.92		(Memo)
15	Консультант	3	3100000	50000	6	24.10.98		(Memo)

Рис. 10.8. Табличная форма при выполнении программы

При открытии набора данных в ходе выполнения программы в сетке TDBGrid автоматически будут отображаться все поля таблицы Сотрудники. При этом сетка будет содержать столбцы, 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 столбце сетки, соответствующей полю такого типа, отображается не информация, содержащаяся в поле, а его тип — как в последнем столбце сетки из нашего примера. Для отображения информации из таких полей необходимо использовать специальные компоненты: TDBMemo, TDBGraphic и т. п.

ПРИМЕЧАНИЕ

В принципе, в ячейке сетки 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 сетке «лишних» полей, следует воспользоваться редактором столбцов, вызываемым командой Columns Editor контекстного меню компонента TDBGrid (см. главу 9). После открытия набора данных в сетке будут

отображаться только те поля, которые указаны в редакторе столбцов. Последовательность отображения столбцов также определяется в редакторе 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 только для связи с другими таблицами, а также поле Примечание, которое все равно не отображается в компоненте TDBGrid. Для реализации этого укажем в редакторе столбцов только используемые поля (рис. 10.9).

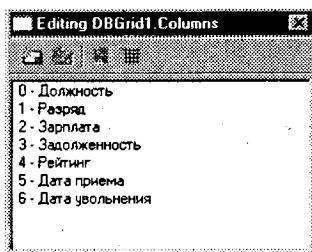


Рис. 10.9. Окно редактора столбцов

При добавлении полей в редакторе столбцов в компонент TDBGrid, помещенный на форму, автоматически добавляются соответствующие столбцы, видимые во время разработки формы (рис. 10.10). Ширина столбца (в пикселах) задается свойством Width класса TColumn.

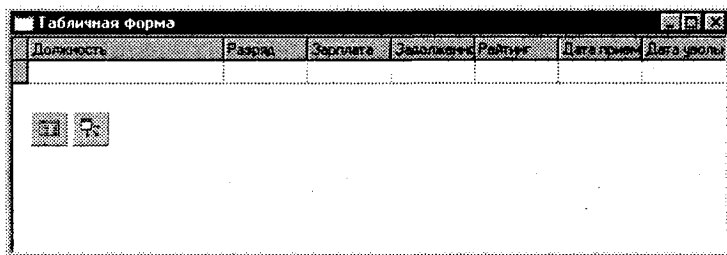


Рис. 10.10. Табличная форма с настроенными столбцами

После запуска программы в таблице будут отображаться только заданные поля (рис. 10.11).

Поскольку компонентом TDBGrid автоматически поддерживается навигация по набору данных, то при использовании табличных форм нет необходимости применять компонент TDBNavigator. Однако он в ряде случаев может быть полезным, так как обеспечивает следующие возможности навигации и редактирования набора данных:

- ☐ переход на первую и последнюю записи таблицы;
- ☐ удаление записи;
- ☐ вставку новой записи;
- ☐ отмену ошибочно введенных данных.

Должность	Разряд	Зарплата	Задолженность	Пенсия	Дата приема	Дата увольнения
Директор	5	960000	200000	10	12.03.92	
Менеджер	4	5760000	300000	10	25.01.95	
Программист	4	2880000	80000	8	20.03.95	
Системотехник	5	2880000	0	8	20.03.95	
Программист	4	3200000	0	7	25.01.96	
Программист	4	2400000	20000	8	10.07.95	
Начальник отдела	4	3400000	200000	8	10.07.95	
Дизайнер	5	3100000	0	8	10.07.95	
Программист	5	2800000	50000	9	15.03.99	
Системный аналитик	5	7000000	0	9	25.01.95	
Конструктор	4	2700000	0	7	01.03.96	
Конструктор	4	2000000	0	7	01.03.96	
Программист	4	2640000	100000	8	20.01.98	
Программист	4	2400000	0	7	15.03.92	
Консультант	3	3100000	50000	6	24.10.98	

Рис. 10.11. Табличная форма с настроенными столбцами при выполнении программы

Кроме того, компонент `TDBNavigator` позволяет включить механизм контроля за удалением записей. Для этого необходимо сделать 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. Проще всего задействовать метод-обработчик события `BeforePost` компонента доступа к данным, который выполняется перед сохранением данных в таблице. Для возврата исходных значений можно использовать свойство `TDBGGrid.SelectedField`, 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 — подтверждения изменения и отмены изменения. В качестве такого окна диалога используется обычная форма, отображаемая в модальном режиме. Примерный внешний вид окна диалога подтверждения изменения показан на рис. 10.12, а текст процедуры-обработчика события `BeforePost` приведен в листинге 10.3.

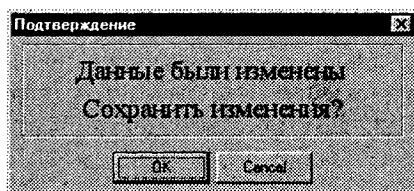


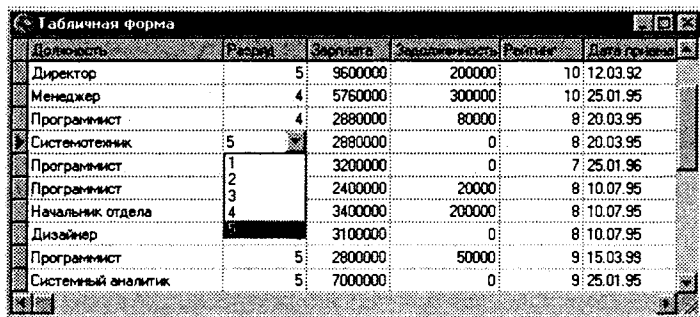
Рис. 10.12. Запрос на подтверждение модификации данных

Компонент `TDBGGrid` позволяет создавать для каждого поля, отображаемого в таблице, список возможных значений. Для раскрытия списка необходимо щелкнуть на

кнопке, появляющейся с правой стороны ячейки таблицы при переходе ячейки в режим редактирования (рис. 10.13).

Листинг 10.3. Обработчик события BeforePost

```
procedure TForm1. Table1BeforePost(DataSet: TDataSet);
begin
    if OKBottomDlg.ShowModal<mrOK
    then
        DBGrid1.SelectedField.Value:=
            DBGrid1.SelectedField.OldValue;
end;
```



Должность	Разряд	Зарплата	Задолженность	Различ	Дата приема
Директор	5	9600000	200000	10	12.03.92
Менеджер	4	5760000	300000	10	25.01.95
Программист	4	2880000	80000	8	20.03.95
Системотехник	5	2880000	0	8	20.03.95
Программист	1	3200000	0	7	25.01.96
Программист	2	2400000	20000	8	10.07.95
Программист	3	3400000	200000	8	10.07.95
Начальник отдела	4	3100000	0	8	10.07.95
Дизайнер	5	2800000	50000	9	15.03.99
Системный аналитик	5	7000000	0	9	25.01.95

Рис. 10.13. Табличная форма со списком редактирования

Список задается с помощью свойства `PickList` класса `TColumn`. Данное свойство имеет тип `TStrings` и для его редактирования во время разработки приложения вызывается специальный редактор (рис. 10.14), который открывается щелчком на кнопке с многоточием в поле ввода значения свойства `PickList` инспектора объектов.

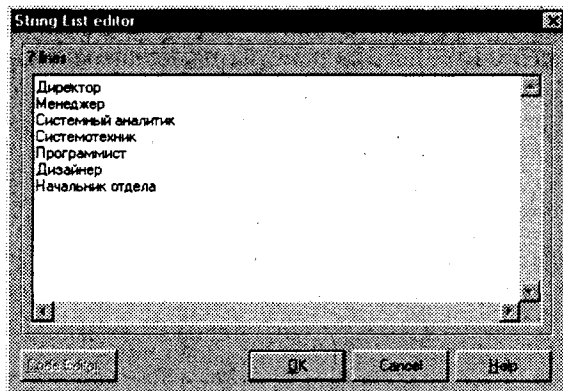


Рис. 10.14. Окно редактора строк объекта `TStrings`

Строки, заданные в этом редакторе, во время выполнения программы станут пунктами раскрывающегося списка.

ПРИМЕЧАНИЕ

Если для поля задан список возможных значений с помощью свойства `PickList`, то это не означает, что при редактировании данного поля можно только выбирать значения в списке. Можно также вводить любые значения (естественно, соответствующие типу поля) с клавиатуры.

Формы с вкладками

При разработке приложений баз данных часто требуется размещать на форме большие объемы информации. Если размещаемые данные можно разделить на несколько групп, то удобно использовать вкладки. Для создания форм с вкладками предназначен специальный элемент управления `TPageControl`.

Компонент TPageControl

Компонент `TPageControl` (страница Win32 палитры компонентов) обеспечивает создание многостраничных окон. На каждой странице данного компонента можно размещать любые другие компоненты (включая `TPageControl`).

Добавлять и удалять вкладки можно как на стадии разработки программы, так и во время выполнения программы. Для добавления вкладки используется команда `New Page` контекстного меню компонента `TPageControl`. Удаление вкладок производится командой `Delete Page` контекстного меню.

Основные свойства компонента `TPageControl` приведены в табл. 10.5.

Таблица 10.5. Основные свойства компонента `TPageControl`

Свойство	Тип	Описание
<code>ActivePage</code>	<code>TTabSheet</code>	Активная вкладка
<code>ActivePageIndex</code>	<code>Integer</code>	Порядковый номер активной вкладки
<code>PageCount</code>	<code>Integer</code>	Количество страниц в компоненте (только для чтения)
<code>Pages[Index: integer]</code>	<code>TTabSheet</code>	Массив вкладок, имеющихся в <code>TPageControl</code> . Используется для прямого доступа к вкладкам
<code>HotTrack</code>	<code>Boolean</code>	Определяет, будет (true) или нет (false) выделяться цветом заголовков, находящийся под указателем мыши
<code>MultiLine</code>	<code>Boolean</code>	Если значение данного свойства равно true, то заголовки вкладок могут размещаться в несколько строк. Используется при большом количестве вкладок
<code>Style</code>	<code>TTabStyle = (tsTabs, tsButtons, tsFlatButtons);</code>	Определяет внешний вид заголовков вкладок: обычные вкладки (<code>tsTabs</code>); отображает заголовки в виде кнопок (<code>tsButtons</code>); отображает заголовки в виде плоских кнопок (<code>tsFlatButtons</code>)

Различные виды вкладок, определяемые свойством `Style`, представлены на рис. 10.15. Каждая вкладка является объектом класса `TTabSheet`. Поскольку при использовании вкладок обращаться к свойствам и методам отдельных вкладок обычно не приходится, то здесь мы рассматривать их не будем. В случае необходимости обращайтесь к справочной системе Delphi.

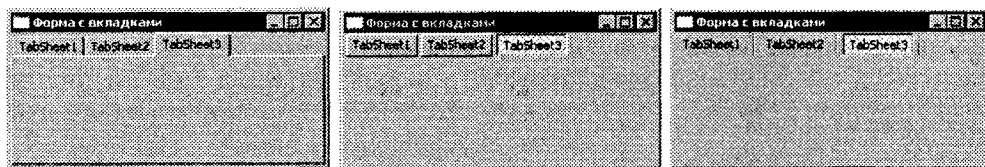


Рис. 10.15. Различные виды вкладок

Пример создания формы с вкладками

В базе данных sales, 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 разместив 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.

1. Для создания нового приложения выполните команду File ► New ► VCL Forms Application.
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 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 TTable и TDataSource.
6. Подключите к компонентам TTable таблицы Физические лица и Сотрудники базы данных Sales (подключение таблицы к компоненту набора данных было подробно рассмотрено ранее на 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.
9. Разместите на вкладке Служебная информация элементы управления для отображения и редактирования информации из таблицы Сотрудники.
Примерный вариант размещения элементов управления на обеих вкладках показан на рис. 10.16.
10. Настройте элементы визуализации полей базы данных и элементы навигации по набору данных. Свяжите элементы, расположенные на вкладке Персональ-

ная информация, с набором данных таблицы Физические лица, а на вкладке Служебная информация — с набором данных таблицы Сотрудники.

11. Добавьте в обработчик события OnCreate 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.
12. Откомпилируйте и запустите программу.

Рис. 10.16. Размещение элементов управления на вкладках Персональная информация и Служебная информация

Листинг 10.4. Обработчики событий для формы с вкладками

```
procedure TfrmTabbed.FormCreate(Sender: TObject);
begin
  Table1.Open;
  Table2.Open;
end;
```

```
procedure TfrmTabbed.FormClose(Sender: TObject;
  var Action: TCloseAction);
begin
  Table1.Close;
  Table2.Close;
end;
```

Внешний вид окна программы приведен на рис. 10.17.

Форма с вкладками

Персональная информация | Служебная информация

Сотрудник: Анциферова Ольга Анатольевна

Адрес: 603006 Россия Нижний Новгород ул. Горького д. 160

Телефон: (831)256-4216

Дата рождения: 02.12.1972

Служебная информация

Должность: Программист

Разряд: 9

Рейтинг: 3200

Зарплата: 14700

Задолженность: 0

Рис. 10.17. Вкладки Персональная информация и Служебная информация окна работающей программы

Многотабличные базы данных

В большинстве случаев база данных состоит из нескольких взаимосвязанных таблиц. При редактировании информации в базе данных часто необходимо учитывать эту взаимосвязь. В рассмотренном примере создания формы для двух связанных таблиц связь между этими таблицами не принималась во внимание, поэтому изменение положения курсора данных в первом наборе данных никак не отражалось на информации, показываемой на второй вкладке. 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 данных 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 компо-

ентов, 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 ведет к изменению положения курсора данных в главной таблице.

Для установления связи набор данных главной таблицы не требует никакой дополнительной настройки.

В подчиненном наборе данных необходимо с помощью свойства **MasterSource** указать источник данных главной таблицы. С помощью свойства **MasterFields** 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. 10.18), который открывается при щелчке на кнопке с многоточием в поле значения свойства **MasterFields** инспектора объектов.

В окне редактора связей отображаются два списка — список полей подчиненной таблицы (**Detail Fields**) и список полей главной таблицы (**Master Fields**). Для создания связи необходимо выделить 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.

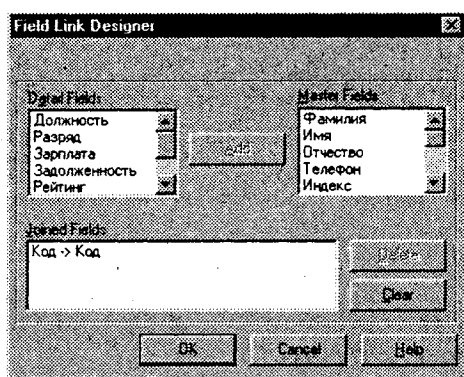


Рис. 10.18. Редактор связей полей

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 требуется 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 воспользуйтесь редактором связей полей. Общим для обеих таблиц является поле `Код`, поэтому установите связь между этими полями, как показано на рис. 10.18.
4. Откомпилируйте и запустите программу. Теперь при щелчке на кнопках элемента навигации данные будут синхронно изменяться на обеих вкладках.

ПРИМЕЧАНИЕ

При установлении связи между полями типы полей 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 и отображение ее результатов

ГЛАВА 11 Выборка данных

Одной из задач, наиболее часто возникающих при работе с базами данных, является задача выборки данных, то есть задача извлечения из базы данных информации, отвечающей ряду требований, заданных пользователем.

Выборка данных из таблиц с помощью SQL-запросов

Одним из наиболее эффективных и универсальных средств выборки данных из таблиц базы данных являются запросы языка SQL. SQL-команды подразделяются на несколько категорий. Для выборки данных используются команды, относящиеся к так называемому языку DQL (Data Query Language — язык запросов к данным).

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 задействована, так как команды языка SQL стандартизованы (стандарт ANSI SQL 92). 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 SQL-команд 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 через механизм BDE применяется собственная реализация языка SQL, называемая локальным языком 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

Для работы с базами данным через SQL-запросы в VCL Delphi используется, как правило, компонент TQuery.

ПРИМЕЧАНИЕ

В VCL Delphi есть и другие компоненты для взаимодействия с базами данных посредством SQL-запросов, например TStoredProc, TCommand, TDataSet, TStoredProc.

Компонент TQuery

При использовании компонента TQuery подготовка и диспетчеризация запросов выполняется механизмом BDE.

По своим свойствам и назначению компонент TQuery подобен компоненту TTable. Отличие заключается только в способе получения данных: в TTable используются методы, инкапсулированные в классе TTable, а TQuery получает данные как результаты выполнения SQL-запросов.

Применение языка SQL позволяет легко решать задачи, которые сложно или вообще невозможно решить в рамках класса TTable. Поэтому компонент TQuery является гораздо более мощным и гибким инструментом для работы с базами данных. Основные свойства класса TQuery приведены в табл. 11.1.

Таблица 11.1. Основные свойства класса TQuery

Свойство	Тип	Описание
Constrained	Boolean	Определяет, можно (false) или нет (true) задавать полям набора данных значения, которые не соответствуют условиям отбора
DataSource	TDataSource	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
ParamCount	Word	Текущее количество параметров в запросе
Params[Index: Word]	TParams	Массив параметров, используемых в SQL-запросе
Prepared	Boolean	Определяет, готов запрос к выполнению (true) или нет (false)
RowsAffected	Integer	Количество записей, измененных с момента последнего выполнения запроса
SQL	TStrings	Текст SQL-запроса
Text	PChar	Текст SQL-запроса, передаваемый BDE

В классе TQuery имеется ряд также 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 данные не возвращаются (например, при выполнении команд INSERT, UPDATE, DELETE и CREATE TABLE).

ПРИМЕЧАНИЕ

Для выполнения команды SELECT необходимо использовать метод Open компонента TQuery.

function ParamByName (const Value: String): TParam;

Обеспечивает доступ к заданным параметрам по имени.

procedure Prepare;

Посылает запрос механизму BDE для подготовки к выполнению. Вызов данного метода перед выполнением запроса повышает скорость выполнения.

procedure UnPrepare;

Освобождает ресурсы, занятые при подготовке запроса к выполнению.

Пример использования компонентов, работающих с SQL-запросами

Рассмотрим возможность практического использования перечисленных выше компонентов. В дальнейшем полученные результаты пригодятся нам для изучения SQL-команд. В качестве исходных данных воспользуемся уже известной нам базой данных sales.mdb.

1. Для создания нового приложения выполните команду File ► New VCL Form Application.
2. Разместите на форме компоненты TQuery 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. Поскольку эта процедура рассматривалась в главе 9 применительно к компоненту TTable, здесь мы на ней останавливаться не будем.

ПРИМЕЧАНИЕ

В отличие от TTable, класс TQuery 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 инспектора объектов. При этом откроется окно 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.

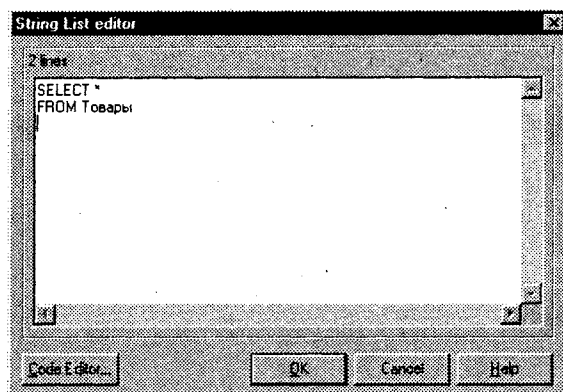


Рис. 11.1. Окно редактора SQL-запросов

5. Щелкните на кнопке Code Editor. Текст запроса появится в окне редактора кода (рис. 11.2), причем ключевые слова языка SQL будут выделены полужирным шрифтом, что снижает вероятность ошибок при написании запроса.

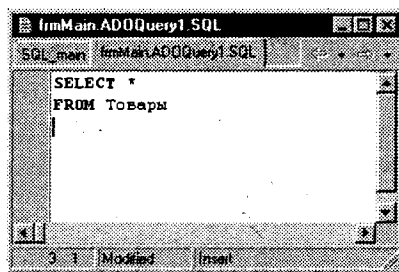


Рис. 11.2. Текст SQL-запроса в окне редактора кода

СОВЕТ

При использовании компонента доступа к данным TQuery для задания SQL-запроса можно применять визуальный редактор запросов SQL Builder, который вызывается командой SQL Builder контекстного меню компонента TQuery, помещенного на форму. Однако он плохо работает с базами данных, в которых имена полей заданы кириллицей (выдаются различные малопонятные сообщения об ошибках).

6. Выполните настройку источника данных TDataSource. Она производится так же, как в случае компонента TTable — в свойстве DataSet указывается имя объекта доступа к данным (по умолчанию — Query1).

Далее необходимо разместить на форме необходимые элементы управления и выполнить их настройку.

7. Выберите следующие элементы:

- компонент TMemo — для отображения и редактирования текста запроса;
- компонент TDBGrid — для отображения результатов выполнения запроса;
- кнопку TButton — для подачи команды на выполнение запроса.

Примерный вариант размещения на форме необходимых компонентов показан на рис. 11.3.

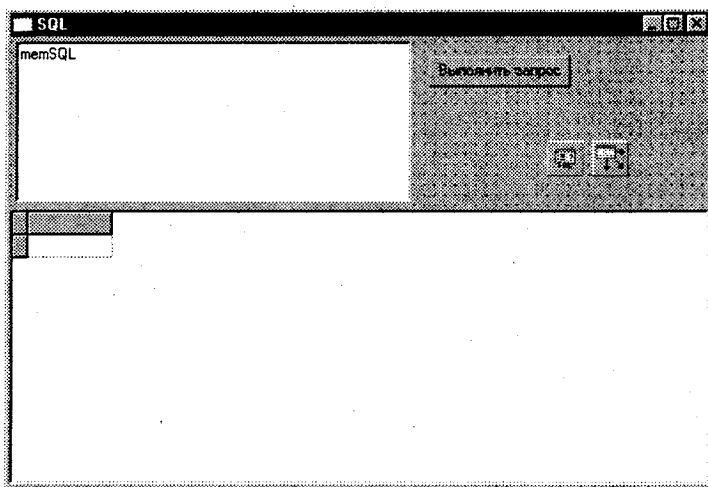


Рис. 11.3. Размещение элементов управления на форме

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.
 - Если в результате выполнения SQL-запроса возвращаются данные, для его выполнения необходимо воспользоваться методом Open класса TQuery. 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 сетке DBGrid1.
 9. 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.
 10. Осталось написать обработчик события OnClick для кнопки Выполнить запрос. Данная кнопка понадобится в дальнейшем для изменения текста запроса с последующим выполнением его без перекомпиляции программы. При щелчке на кнопке должен выполняться запрос. Будем полагать, что в результате выполнения запроса возвращаются данные. В этом случае при щелчке на кнопке следует передать текст запроса из компонента memSQL в свойство SQL компонента Query1 и открыть набор данных, вызвав метод Open компонента Query1.
 11. Откомпилируйте и запустите приложение.
- 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)
    Query1: TQuery;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    memSQL: TMemo;
    btnExecSQL: TButton;
    procedure FormShow(Sender: TObject);
    procedure FormClose(Sender: TObject);
    procedure btnExecSQLClick(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(Query1.SQL);
  Query1.Open;
end;

procedure TfrmMain.FormClose(Sender: TObject);
var Action: TCloseAction;
begin
  if Query1.Active then Query1.Close;
end;

procedure TfrmMain.btnExecSQLClick(Sender: TObject);
begin
  if Query1.Active then Query1.Close;
  Query1.SQL.Clear;
  Query1.SQL.Assign(memSQL.Lines);
  Query1.Open;
end;

end.
```

После запуска приложения в компоненте DBGrid1 на форме отобразится информация, содержащаяся в таблице Товары базы данных sales.mdb (рис. 11.4).

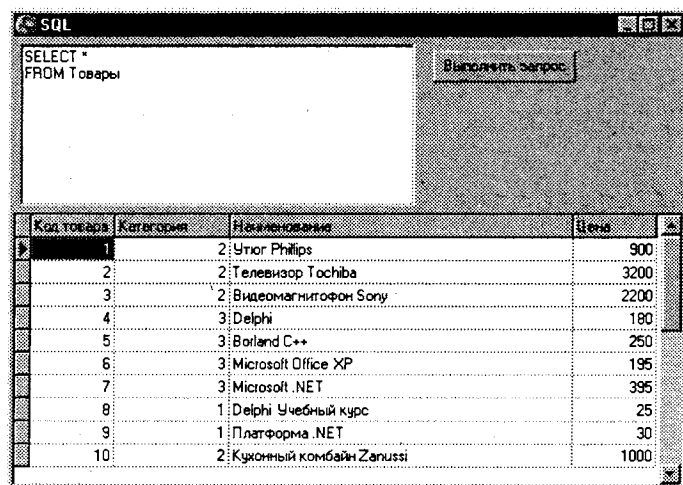


Рис. 11.4. Результат выполнения SQL-запроса

Язык запросов к данным

Синтаксис языка запросов к данным (DQL), являющегося одним из подмножеств языка SQL, состоит всего из одной команды SELECT. Эта команда вместе с множеством параметров и предложений служит 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. Эти предложения могут быть обязательными и дополнительными. *Обязательным* является только одно предложение — FROM, без которого оператор SELECT не может работать.

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 поле1, поле2, .... полеN }
FROM Таблица1 { . Таблица2, .... ТаблицаN }
```

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 *;

- ☐ параметр ALL (задан по умолчанию) означает, что результат выборки будет содержать все записи, включая дублирующиеся;
- ☐ при использовании параметра DISTINCT результат запроса не будет содержать дублирующихся строк.

Совместно с командой SELECT всегда используется предложение FROM. С помощью этого предложения указывается имя таблицы (таблиц), из которой производится выборка. Если в предложении FROM указывается несколько таблиц, то их имена разделяются запятыми.

Ранее 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
```

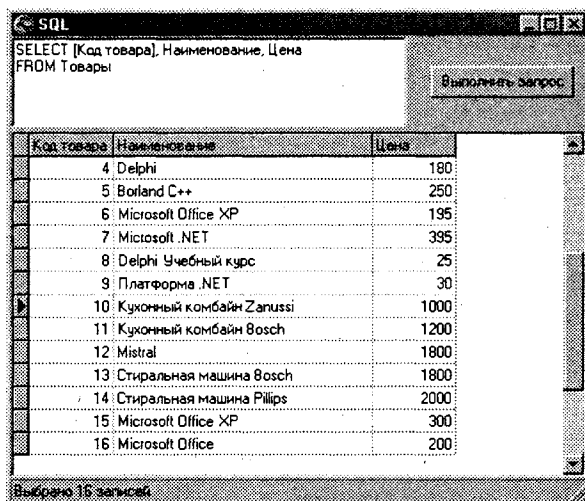
СОВЕТ

Обратите внимание, что при указании в списке оператора SELECT имен полей, содержащих пробел, их необходимо заключать в квадратные скобки. Это правило требуется выполнять и для имен таблиц, указываемых, например, в предложении FROM.

В результате выполнения данного запроса возвращаются все записи, содержащиеся в трех полях таблицы Товары (рис. 11.5).

ПРИМЕЧАНИЕ

Для выполнения запроса нет необходимости перекомпилировать программу. Достаточно во время ее выполнения ввести текст запроса в поле ввода и щелкнуть на кнопке Выполнить запрос.



Код товара	Наименование	Цена
4	Delphi	180
5	Borland C++	250
6	Microsoft Office XP	195
7	Microsoft .NET	395
8	Delphi Учебный курс	25
9	Платформа .NET	30
10	Кухонный комбайн Zanussi	1000
11	Кухонный комбайн Bosch	1200
12	Mistral	1800
13	Стиральная машина Bosch	1800
14	Стиральная машина Philips	2000
15	Microsoft Office XP	300
16	Microsoft Office	200

Выбрано 16 записей

Рис. 11.5. Результат выбора трех полей

Выясним теперь, зачем нужен параметр `DISTINCT`. Он позволяет исключить вывод повторяющихся значений, содержащихся, например, в поле Наименование:

```
SELECT DISTINCT Наименование  
FROM Товары
```

В результате выполнения этого запроса будет выбрано только 15 записей из 16, так как в выборку включается только одно значение Microsoft Office XP (рис. 11.6).

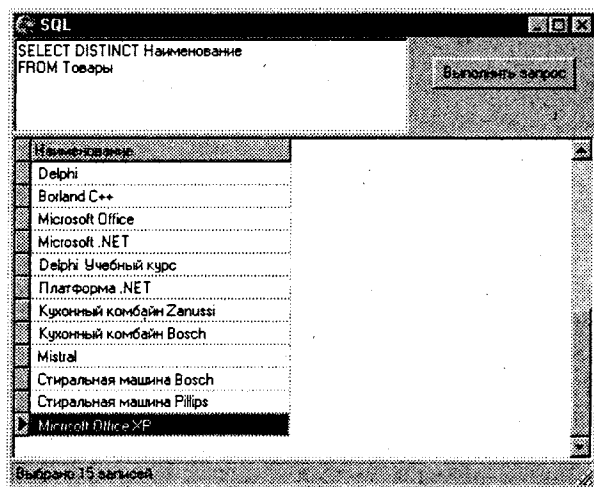


Рис. 11.6. Результат применения параметра `DISTINCT`

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 поле1, поле2, .... полеN }  
FROM Таблица1 { , Таблица2, .... ТаблицаN }  
WHERE условие
```

Специальные операторы языка `SQL`, применяемые для задания условий, 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

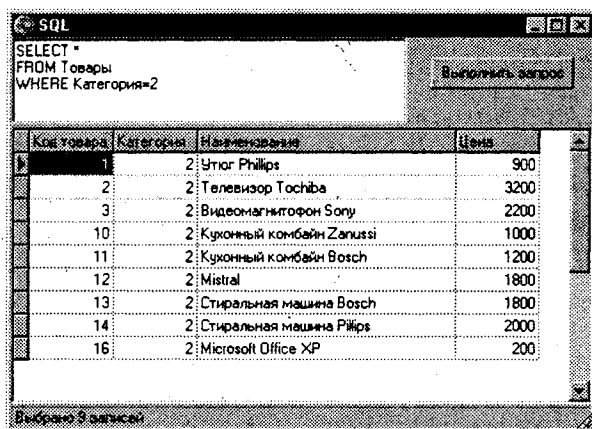
Операторы сравнения используются в SQL-запросах для наложения ограничений на информацию, возвращаемую в результате выполнения запроса. Это типичные операторы, существующие во всех алгоритмических языках:

- ☐ 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» (>) позволяют отбирать записи, в которых значение определенного поля меньше или больше некоторой заданной величины;
- ☐ операторы «меньше или равно» (<=) и «больше или равно» (>=) 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. 11.7.



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	900
2	2	Телевизор Toshiba	3200
3	2	Видеомагнитофон Sony	2200
10	2	Кухонный комбайн Zanussi	1000
11	2	Кухонный комбайн Bosch	1200
12	2	Mistral	1800
13	2	Стиральная машина Bosch	1800
14	2	Стиральная машина Philips	2000
16	2	Microsoft Office XP	200

At the bottom of the window, it says "Выбрано 9 записей".

Рис. 11.7. 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.

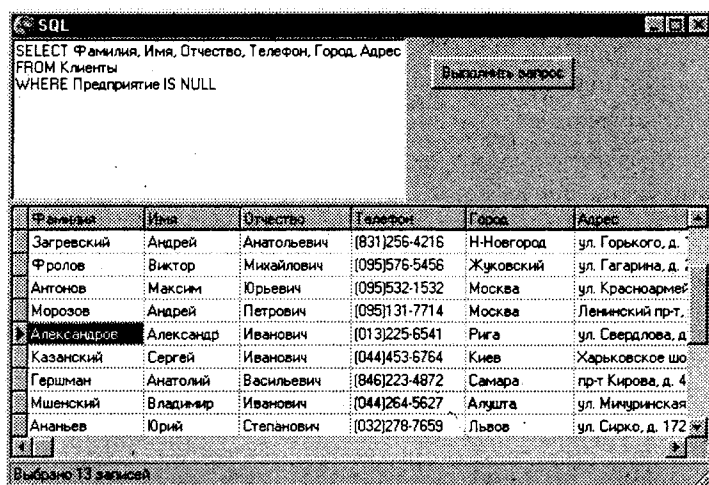
Оператор IS NULL

Оператор IS NULL предназначен для сравнения текущего значения поля со значением NULL. Он используется для отбора записей, в некоторое поле которых не занесено никакого значения.

Для иллюстрации использования этого оператора воспользуемся таблицей Клиенты. С помощью приведенного ниже запроса произведем выборку из нее записей, в которых не указано название предприятия:

```
SELECT Фамилия, Имя, Отчество, Телефон, Город, Адрес
FROM Клиенты
WHERE Предприятие IS NULL
```

Результат выполнения запроса показан на рис. 11.8.



Фамилия	Имя	Отчество	Телефон	Город	Адрес
Загребский	Андрей	Анатольевич	(831)256-4216	Н-Новгород	ул. Горького, д.
Фролов	Виктор	Михайлович	(095)576-5456	Жуковский	ул. Гагарина, д.
Антонов	Максим	Юрьевич	(095)532-1532	Москва	ул. Красноармей
Морозов	Андрей	Петрович	(095)131-7714	Москва	Ленинский пр-т.
Александров	Александр	Иванович	(013)225-6541	Рига	ул. Свердлова, д.
Казанский	Сергей	Иванович	(044)453-6764	Киев	Харьковское шо
Гершман	Анатолий	Васильевич	(846)223-4872	Самара	пр-т Кирова, д. 4
Мишенский	Владимир	Иванович	(044)264-5627	Алжир	ул. Мичуринская
Ананьев	Юрий	Степанович	(032)278-7659	Львов	ул. Сирко, д. 172

Рис. 11.8. 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 200 до 2000. Для этого сформируем следующий запрос:

```
SELECT *
FROM Товары
WHERE Цена BETWEEN 200 AND 2000
```

Результаты, возвращенные при выполнении данного запроса, приведены на рис. 11.9.

Оператор IN

Оператор IN используется для выборки записей, в которых значение некоторого поля соответствует хотя бы одному из значений заданного списка.

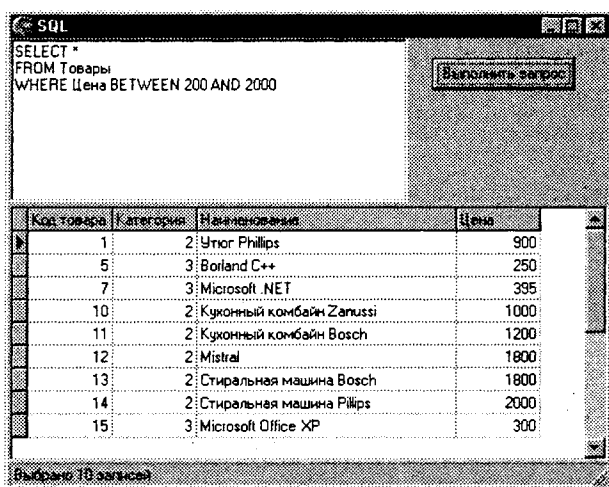


Рис. 11.9. Пример использования оператора BETWEEN...AND

Выберем из таблицы Клиенты 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. 11.10.

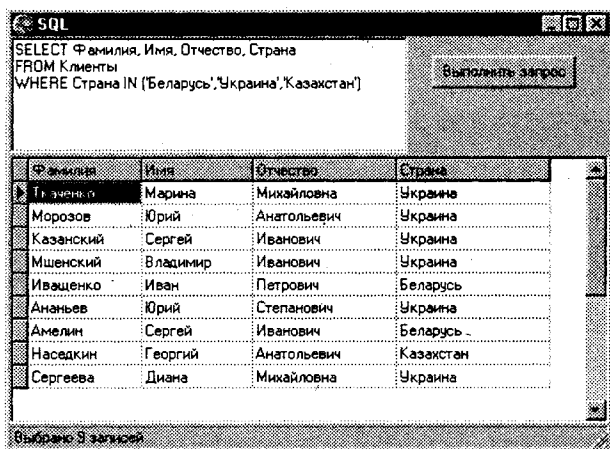


Рис. 11.10. Пример использования оператора IN

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 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 будет выбрано 4 записи (рис. 11.11).

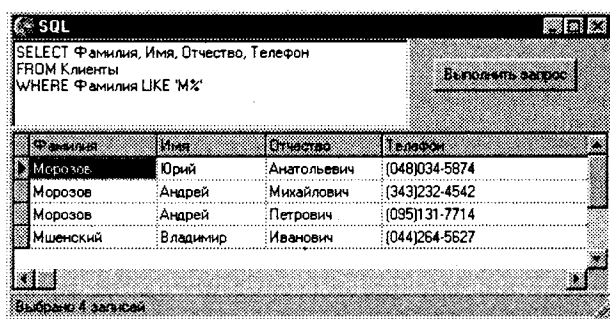


Рис. 11.11. Использование оператора LIKE с шаблоном в виде символа процента

А теперь найдем в этой же таблице записи, для которых номер телефона начинается с символов (816)025-61, а две последние цифры неизвестны:

```
SELECT Фамилия, Имя, Отчество, Телефон
FROM Клиенты
WHERE Телефон LIKE '(816)025-61__'
```

При выполнении данного запроса будут отображены две записи (рис. 11.12).

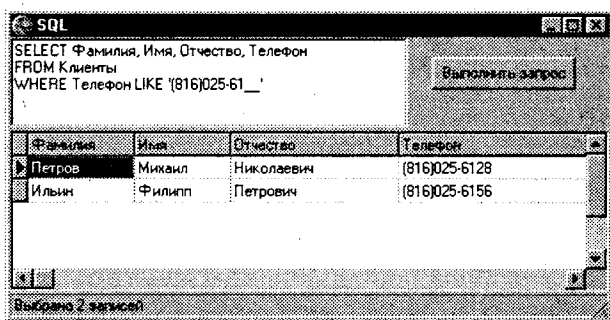


Рис. 11.12. Использование оператора LIKE с шаблоном в виде символа подчеркивания

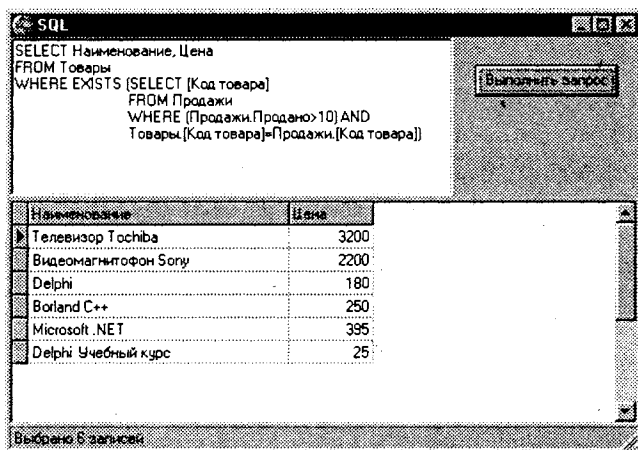
Оператор EXISTS

Оператор EXISTS используется для отбора записей, соответствующих заданному критерию.

Для иллюстрации работы оператора EXISTS рассмотрим следующий пример. Из таблицы Товары требуется отобрать товары, количество продаж которых превышает 10. Сведения о продажах содержатся в поле Продано таблицы Продажи. Для получения необходимой выборки воспользуемся оператором EXISTS:

```
SELECT Наименование, Цена
FROM Товары
WHERE EXISTS (SELECT [Код товара]
              FROM Продажи
              WHERE (Продажи.Продано>10) AND
                    Товары.[Код товара]=Продажи.[Код товара])
```

В этом запросе после ключевого слова EXISTS следует оператор SELECT, отбирающий из таблицы Продажи записи, для которых количество продаж превышает 10. Оператор EXISTS отбирает из таблицы Товары записи, в которых значение поля Код товара соответствует записям, отобранным из таблицы Продажи. Результат выполнения данного запроса приведен на рис. 11.13.



Наименование	Цена
Телевизор Toshiba	3200
Видеомагнитофон Sony	2200
Delphi	180
Borland C++	250
Microsoft .NET	395
Delphi Учебный курс	25

Рис. 11.13. Пример использования оператора EXISTS

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 далее.

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 10, используется следующий запрос:

```
SELECT *
FROM Товары
WHERE Цена>ALL (SELECT Продажи.Цена
                FROM Продажи
                WHERE Продажи.Продано>10)
```

Результат выполнения данного запроса приведен на рис. 11.14.

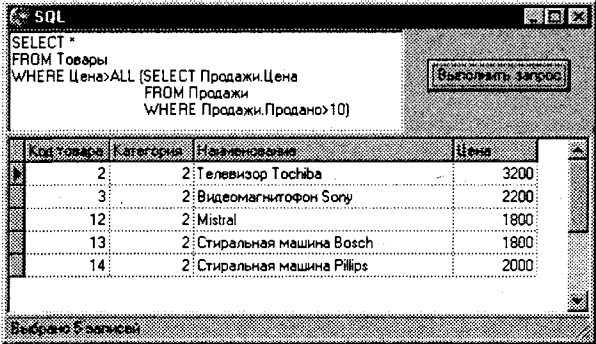


Рис. 11.14. Пример использования оператора ALL

Оператор ANY

Оператор ANY применяется для сравнения заданного значения с каждым из значений некоторого набора данных. Если в предыдущем примере заменить оператор ALL оператором ANY, то будет возвращен список товаров, цена которых больше, чем хотя бы у одного из товаров, проданных в количестве больше 10. Результат выполнения такого запроса показан на рис. 11.15.

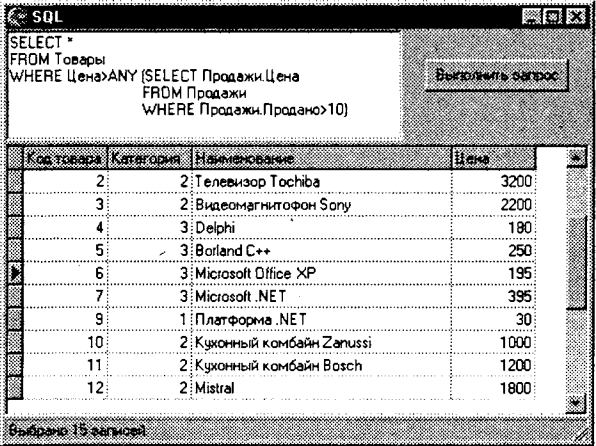


Рис. 11.15. Пример использования оператора ANY

Операторы объединения

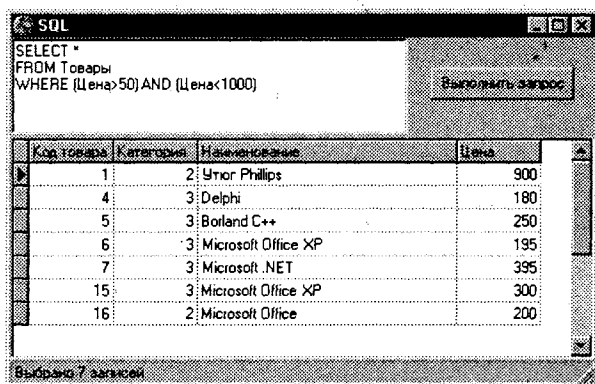
Часто при написании запроса на выборку данных требуется задать сложное условие, для которого одного оператора недостаточно. В этом случае несколько условий объединяются с помощью специальных операторов. В SQL их два — это операторы AND и OR.

- ❑ 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 50, но меньше 1000:

```
SELECT *  
FROM Товары  
WHERE (Цена>50) AND (Цена<1000)
```

Результат выполнения запроса приведен на рис. 11.16.



Код товара	Категория	Наименование	Цена
1	2	Утюг Philips	900
4	3	Delphi	180
5	3	Borland C++	250
6	3	Microsoft Office XP	195
7	3	Microsoft .NET	395
15	3	Microsoft Office XP	300
16	2	Microsoft Office	200

Рис. 11.16. Пример использования оператора AND

Синтаксические правила использования оператора OR такие же, как и для оператора AND. Следующий запрос возвратит список товаров, цена которых меньше 50 или больше 1000 (рис. 11.17):

```
SELECT *  
FROM Товары  
WHERE (Цена<50) OR (Цена>1000)
```

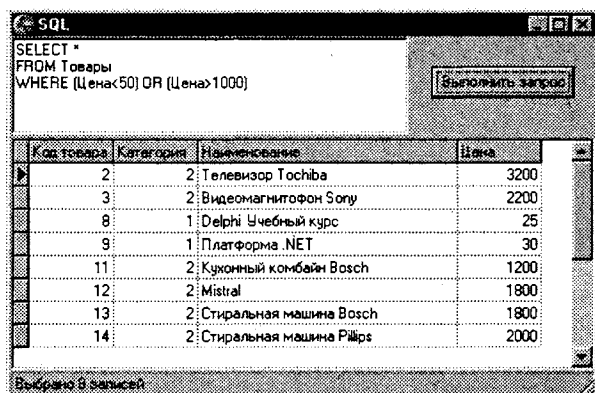


Рис. 11.17. Пример использования оператора OR

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 применения этого оператора к логическим операторам:

```
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 поле, поле2, .... полеN }
FROM Таблица1 {, Таблица2, .... ТаблицаN}
WHERE условие
ORDER BY поле {ASC | DESC}
```

После ключевых слов ORDER BY сначала указывается имя поля (полей), по которому производится сортировка, а затем — режим сортировки:

- ☐ ASC — информация располагается в порядке возрастания значения указанного поля (для текстовых полей — в алфавитном порядке), этот режим используется по умолчанию;
- ☐ DESC — информация располагается 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
```

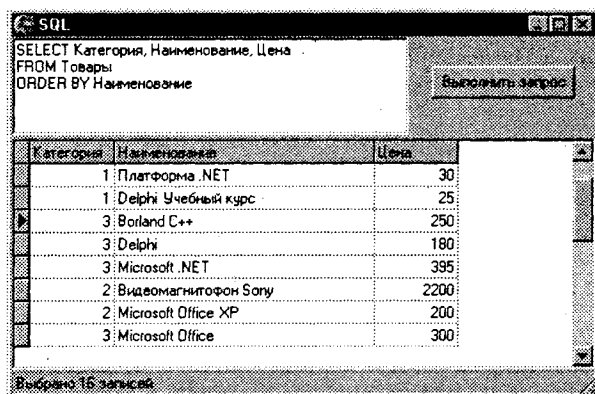


Рис. 11.18. Пример использования предложения ORDER BY

Результат выполнения данного запроса приведен на рис. 11.18.

Вместо имени поля в предложении ORDER BY можно указать целое число, определяющее порядковый номер поля в списке после ключевого слова SELECT (если производится выборка всех полей таблицы с помощью символа *, то число задает 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. 11.19.

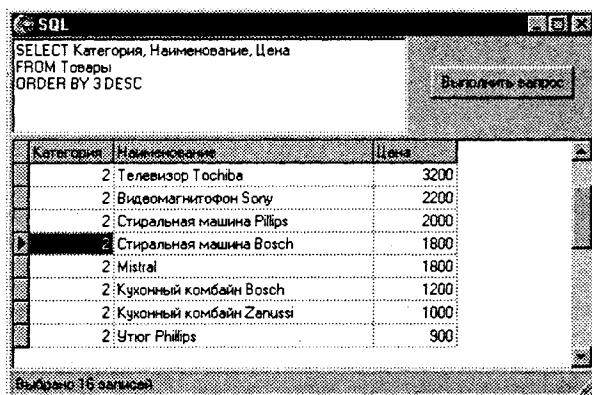


Рис. 11.19. Пример использования порядкового номера поля в предложении ORDER BY

Вычисляемые поля

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 в каче-

стве переменных. В результате выполнения запроса с вычисляемыми полями выборка будет содержать не только информацию из таблиц базы данных, но и дополнительную информацию, полученную в результате вычисления заданного выражения.

ПРИМЕЧАНИЕ

Помимо математических операций, в SQL поддерживается ряд строковых функций, выполняющий такие операции, как конкатенация строк, выделение подстроки, поиск подстроки внутри строки и ряд других. В SQL-запросах 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: сначала выполняются 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. 11.20.

Помимо арифметических операторов допускается использование ряда математических функций, например:

- ☐ ABS — вычисление абсолютного значения;
- ☐ ROUND — округление;
- ☐ SQRT — 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).

SQL

```
SELECT [Код товара], Цена, Заказано, Продано,
Цена*Продано, Цена*Заказано,
Цена*Заказано-Цена*Продано
FROM Продажи
```

Выполнить запрос

Код товара	Цена	Заказано	Продано	Expr1004	Expr1005	Expr1006
2	610	5	5	3050	3050	0
1	2600	2	2	5200	5200	0
2	625	15	15	9375	9375	0
2	630	7	7	4410	4410	0
2	645	10	10	6450	6450	0
2	605	500	450	272250	302500	30250
3	1025	10	8	8200	10250	2050
8	27	100	100	2700	2700	0
2	620	20	20	12400	12400	0
4	210	20	20	4200	4200	0
8	27	50	50	1350	1350	0
1	2670	3	3	6010	6010	0

Выбрано 41 записей

Рис. 11.20. Результат выполнения запроса с вычисляемыми полями

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 и как SQR, и как SQRT).

Псевдонимы полей

В SQL-запросах 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 вычисляемого выражения через ключевое слово AS.

ВНИМАНИЕ

Псевдоним действителен только в пределах конкретного запроса.

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. 11.21.

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.

SQL

```

SELECT [Код товара], Цена, Заказано, Продано,
Цена*Продано AS [Сумма продаж],
Цена*Заказано AS [Сумма заказов],
Цена*Заказано-Цена*Продано AS [Разность]
FROM Продажи

```

Выполнить запрос

Код товара	Цена	Заказано	Продано	Сумма продаж	Сумма заказов	Разность
2	610	5	5	3050	3050	0
1	2600	2	2	5200	5200	0
2	625	15	15	9375	9375	0
2	630	7	7	4410	4410	0
2	645	10	10	6450	6450	0
2	605	500	450	272250	302500	30250
3	1025	10	8	8200	10250	2050
8	27	100	100	2700	2700	0
2	620	20	20	12400	12400	0
4	210	20	20	4200	4200	0
8	27	50	50	1350	1350	0

Выбрано 41 записей

Рис. 11.21. Пример запроса с псевдонимами полей

Функции агрегирования

Функциями агрегирования называются функции, которые позволяют определить количество записей в таблице, количество значений в столбце таблицы, найти минимальное, максимальное и среднее значение для столбца таблицы, вычислить 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 нужен следующий запрос:

```

SELECT COUNT(Продано) AS [Всего записей],
MIN(Продано) AS min,
MAX(Продано) AS max,
SUM(Продано) AS [Всего продано],
AVG(Продано) AS [Среднее количество продаж]
FROM Продажи

```

Результат выполнения этого запроса показан на рис. 11.22.

Со всеми функциями агрегирования можно использовать параметр DISTINCT. В этом случае выполняется обобщение информации только для различающихся строк.

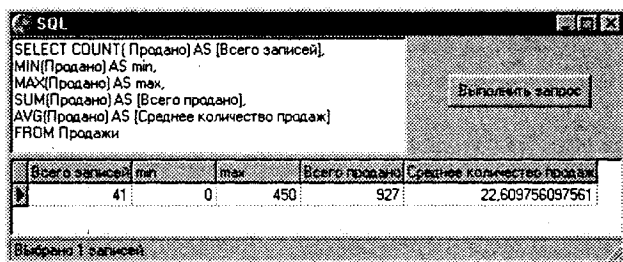


Рис. 11.22. Пример использования функций агрегирования

ПРИМЕЧАНИЕ

Как правило, использовать параметр `DISTINCT` с агрегирующими функциями не имеет смысла, поскольку при подсчете обобщенных данных обычно приходится учитывать все записи, а не только уникальные.

Группировка данных

Группировка данных — это объединение записей в соответствии со значениями некоторого 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* отбираемые 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 поле1, поле2, .... полеN
FROM Таблица1 {, Таблица2, .... ТаблицаN}
WHERE условие
GROUP BY поле1, поле2, .... полеN
ORDER BY поле1 {ASC | DESC}
```

Например, если выбрать из таблицы `Товары` два поля — `Наименование` и `Категория`, а затем сгруппировать их с помощью следующего запроса, то результат выборки будет упорядочен по значению первого поля, указанного в предложении `GROUP BY` (рис. 11.23):

```
SELECT Наименование, Категория
FROM Товары
GROUP BY Категория, Наименование
```

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 (см. рис. 11.23), но выбрать только поле `Категория`, как показано ниже, то выборка будет содержать только три записи (рис. 11.24):

```
SELECT Категория
FROM Товары
GROUP BY Категория
```

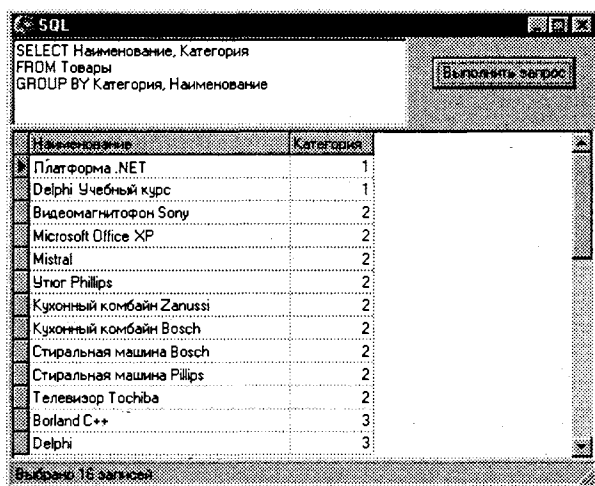


Рис. 11.23. Пример группировки данных

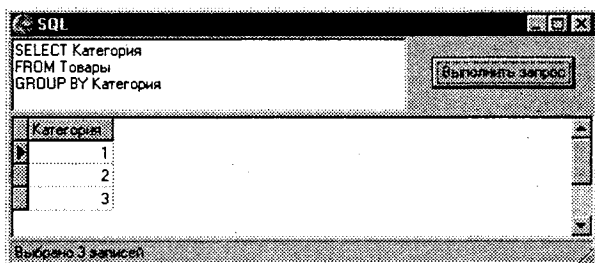


Рис. 11.24. Результат группировки одного поля

В этом случае группировка дает такой же результат, как применение оператора **SELECT** с параметром **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 служат 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. 11.25.

Результаты группировки можно упорядочить с помощью предложения **ORDER BY**, а в операторе **SELECT**, содержащем предложение группировки, использовать предложение **WHERE**. Для иллюстрации этой возможности модифицируем предыдущий запрос следующим образом: выберем только тех клиентов, которые сделали

за один раз более 10 покупок, и упорядочим результаты выборки по возрастанию:

```
SELECT [Код клиента],  
SUM(Продано) AS [Количество покупок]  
FROM Продажи  
WHERE Продано>10  
GROUP BY [Код клиента]  
ORDER BY 2 DESC
```

Результат выполнения данного запроса изображен на рис. 11.26.

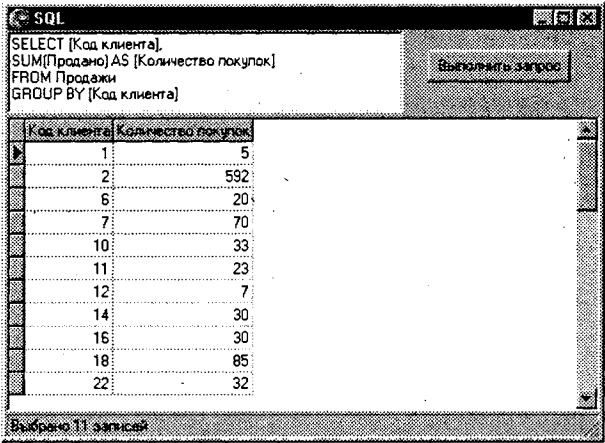


Рис. 11.25. Использование агрегирующих функций при группировке данных

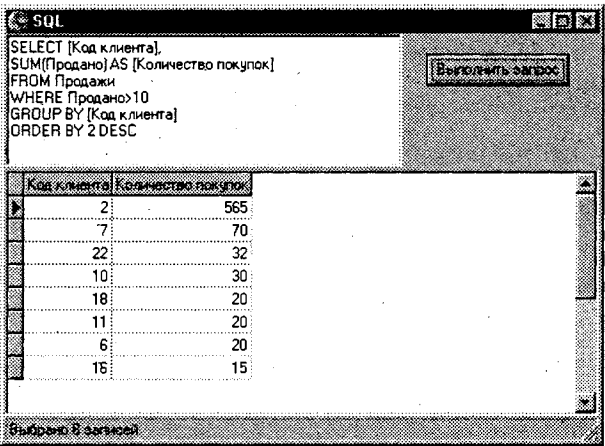


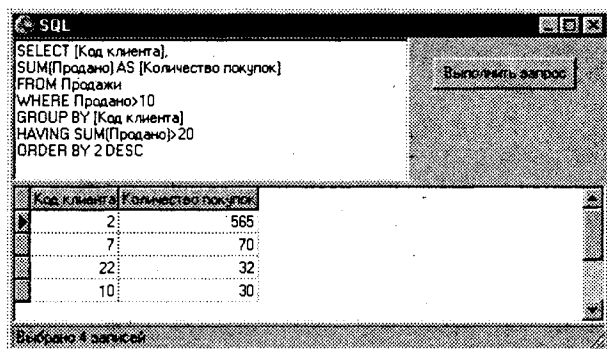
Рис. 11.26. Пример сортировки результатов выборки с помощью предложения GROUP BY

Для задания ограничений на создаваемые группы совместно с ключевым словом GROUP BY может использоваться предложение HAVING. Оно должно следовать после предложения 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 нужно применить предложение **HAVING**:

```
SELECT [Код клиента],  
SUM(Продано) AS [Количество покупок]  
FROM Продажи  
WHERE Продано>10  
GROUP BY [Код клиента]  
HAVING SUM(Продано)>20  
ORDER BY 2 DESC
```

Результат выполнения запроса приведен на рис. 11.27.



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>10  
GROUP BY [Код клиента]  
HAVING SUM(Продано)>20  
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
2	565
7	70
22	32
10	30

At the bottom of the window, it says "Выбрано 4 записей".

Рис. 11.27. Пример использования предложения **HAVING**

ПРИМЕЧАНИЕ

В предложении **HAVING** не обязательно использовать только те поля, которые заданы в списке оператора **SELECT**.

Модифицируем рассмотренный пример (см. рис. 11.27) 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)>25000  
ORDER BY 2 DESC
```

Данный запрос учитывает клиентов, купивших товаров более чем на 25 000, и отображает количество сделанных ими покупок (рис. 11.28).

ПРИМЕЧАНИЕ

В предложении **GROUP BY**, в отличие от предложения **ORDER BY**, нельзя вместо имен выбранных полей использовать их порядковые номера в списке оператора **SELECT**.

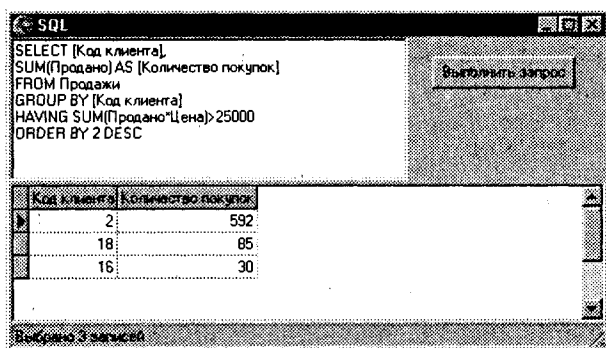


Рис. 11.28. Пример запроса с предложением HAVING

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* 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

Соединение равенства, используемое чаще других, обычно производится по общему для нескольких таблиц полю (которое, как правило, является первичным ключом).

Синтаксис оператора выборки для этого способа соединения таблиц следующий:

```
SELECT Таблица1.поле1, Таблица2.поле2, { . . . . . ТаблицаN.полеN}  
FROM поле1, поле2 { . . . . . полеN}  
WHERE Таблица1.общее_поле1 = Таблица2.общее_поле1  
{AND Таблица1.общее_поле2 = Таблица2.общее_поле2}
```

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 посредством 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. 11.29.

Фамилия	Имя	Отчество	Продано
Александров	Сергей	Иванович	5
Ткаченко	Марина	Михайловна	2
Ткаченко	Марина	Михайловна	15
Ткаченко	Марина	Михайловна	7
Ткаченко	Марина	Михайловна	10
Ткаченко	Марина	Михайловна	450
Ткаченко	Марина	Михайловна	8
Ткаченко	Марина	Михайловна	100
Петров	Михаил	Николаевич	20
Зайцева	Людмила	Николаевна	20
Зайцева	Людмила	Николаевна	50
Сухарев	Андрей	Анатолевич	3
Сухарев	Андрей	Анатолевич	15
Сухарев	Андрей	Анатолевич	15

Выбрано 41 записей

Рис. 11.29. Использование соединения равенства для выборки из двух таблиц

При связывании таблиц можно использовать предложение группировки. Изменим рассмотренный запрос (см. рис. 11.29) 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. 11.30.

Выборка из трех таблиц проводится аналогичным образом, только в предложении WHERE необходимо указать условие связи с третьей таблицей. Для примера дополним предыдущий запрос (см. рис. 11.30) 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

Результат выполнения данного запроса показан на рис. 11.31.

SQL

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

Выполнить запрос

Фамилия	Имя	Отчество	Количество покупок
Александров	Сергей	Иванович	5
Зайцева	Дарья	Ивановна	23
Зайцева	Людмила	Николаевна	70
Морозов	Андрей	Петрович	85
Мишенский	Владимир	Иванович	32
Петров	Михаил	Николаевич	20
Райский	Максим	Сергеевич	30
Сукарев	Андрей	Анатольевич	33
Ткаченко	Марина	Михайловна	592
Уханов	Максим	Сергеевич	30
Фролов	Виктор	Михайлович	7

Выбрано 11 записей

Рис. 11.30. Пример группировки результатов выборки из двух таблиц

SQL

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

Выполнить запрос

Фамилия	Имя	Отчество	Количество покупок	Наименование
Александров	Сергей	Иванович	5	Телевизор Toshiba
Зайцева	Дарья	Ивановна	20	Delphi
Зайцева	Дарья	Ивановна	1	Видеомагнитофон Sony
Зайцева	Дарья	Ивановна	1	Утюг Philips
Зайцева	Дарья	Ивановна	1	Телевизор Toshiba
Зайцева	Людмила	Николаевна	50	Delphi Учебный курс
Зайцева	Людмила	Николаевна	20	Delphi
Морозов	Андрей	Петрович	10	Платформа .NET
Морозов	Андрей	Петрович	40	Borland C++
Морозов	Андрей	Петрович	5	Microsoft .NET

Выбрано 35 записей

Рис. 11.31. Пример выборки из трех таблиц

Соединение неравенства

В случае 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 $\neq$, <, > 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}
```

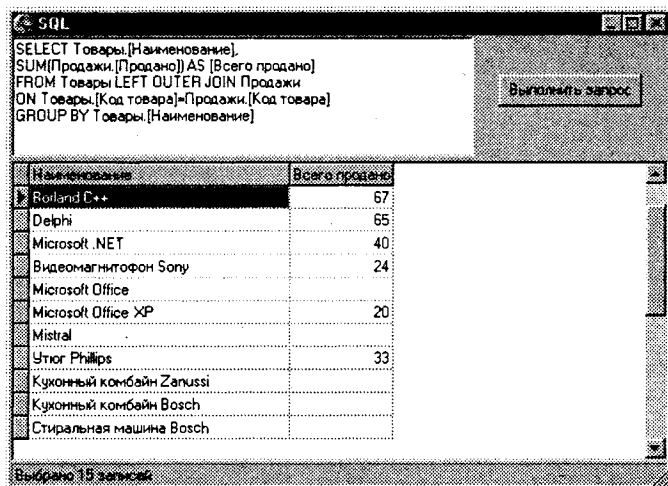
Соединения неравенства используются довольно редко. В частности, для базы данных, применяемой нами в качестве практической модели, довольно трудно привести осмысленный пример такого соединения.

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 оператором OUTER JOIN.

Внешние соединения подразделяются на три группы:

- ☐ LEFT OUTER JOIN — левое внешнее соединение (в выборку включаются все записи таблицы, имя которой указано слева от оператора OUTER JOIN);
- ☐ RIGHT OUTER JOIN — правое внешнее соединение (в выборку включаются все записи таблицы, имя которой указано справа от оператора OUTER JOIN);
- ☐ FULL OUTER JOIN — полное внешнее соединение (в выборку включаются все записи из правой и левой таблиц).



The screenshot shows an SQL window with the following query:

```
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]
```

Below the query is a table with the results:

Наименование	Всего продано
Borland C++	67
Delphi	65
Microsoft .NET	40
Видеомагнитофон Sony	24
Microsoft Office	
Microsoft Office XP	20
Mistral	
Утюг Philips	33
Кухонный комбайн Zanussi	
Кухонный комбайн Bosch	
Стиральная машина Bosch	

At the bottom, it says "Выбрано 15 записей".

Рис. 11.32. Результат внешнего соединения двух таблиц

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 были проданы (рис. 11.32).

Подзапросы

Подзапрос представляет собой запрос, выполняемый в рамках другого запроса. Подзапросы служат 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 поле1, поле2, .... полеN}  
FROM таблица1 {, таблица2, .... таблицаN}  
WHERE условие (SELECT поле1{, поле2, .... полеN}  
FROM поле1 {, поле2, .... полеN}  
WHERE условие)
```

Для иллюстрации технологии составления подзапроса воспользуемся следующим примером. Выберем из таблицы Продажи информацию о продажах товара с наименованием Delphi 5:

```
SELECT [Код клиента], Заказано, Продано, Цена  
FROM Продажи  
WHERE [Код товара]=(SELECT [Код товара]  
FROM Товары  
WHERE Наименование='Delphi 5')
```

Поскольку в таблице Продажи не содержится наименований товара, то сначала с помощью подзапроса мы обращаемся к таблице Товары и определяем код товара заданного наименования, а затем в основном запросе выбираем интересующие нас поля из таблицы Продажи, в которых код товара совпадает с полученным в результате выполнения подзапроса. Результат показан на рис. 11.33.

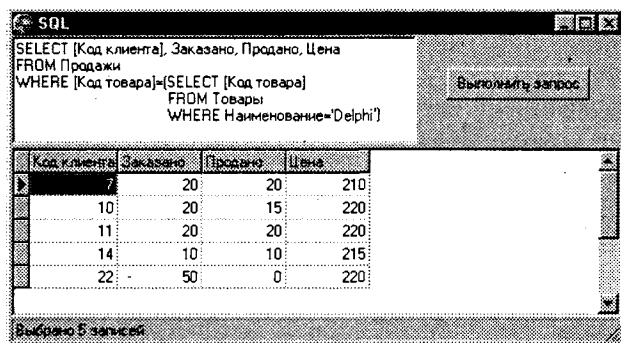


Рис. 11.33. Пример использования подзапроса

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 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 простых запросов делает текст запроса более понятным. Как правило, проще написать составной запрос, состоящий из нескольких простых запросов, чем 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 и другие операторы объединения, такие как 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 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 200 или больше 2000. Такую выборку можно сделать путем объединения логических операторов в предложении WHERE с помощью оператора OR или путем объединения запросов:

```
SELECT *
FROM Товары
WHERE Цена<200
UNION
SELECT *
FROM Товары
WHERE Цена>=2000
```

Здесь первый запрос отбирает товары, цена которых меньше 200, а второй — товары, цена которых превышает 2000. С помощью оператора UNION результаты, возвращаемые отдельными запросами, объединяются в один набор данных (рис. 11.34).

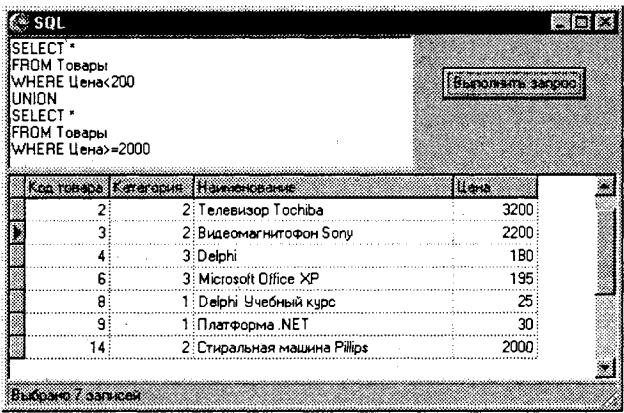


Рис. 11.34. Результат объединения запросов с помощью оператора UNION

Оператор UNION ALL

Оператор UNION ALL аналогичен оператору UNION, за исключением того, что в результирующую выборку включаются дублирующие записи. Если в предыдущем примере (см. рис. 11.34) заменить оператор UNION оператором UNION ALL, то результат не изменится, так как в нем не содержится дублирующих записей. Однако если задать запрос таким образом, чтобы одни и те же записи попали в результаты обоих запросов, входящих в объединение, то в результирующей выборке они также будут присутствовать два раза. Например, при выполнении следующего запроса результат будет содержать 23 записи, хотя в таблице Товары содержатся всего лишь 16 записей:

```
SELECT *
FROM Товары
WHERE Цена>100
UNION ALL
SELECT *
FROM Товары
WHERE Цена<1000
```

Это объясняется тем, что часть записей выбрана и в первом, и во втором запросе (это те товары, цена которых больше 100, но меньше 1000), поэтому в результирующей выборке содержится несколько одинаковых записей (рис. 11.35).

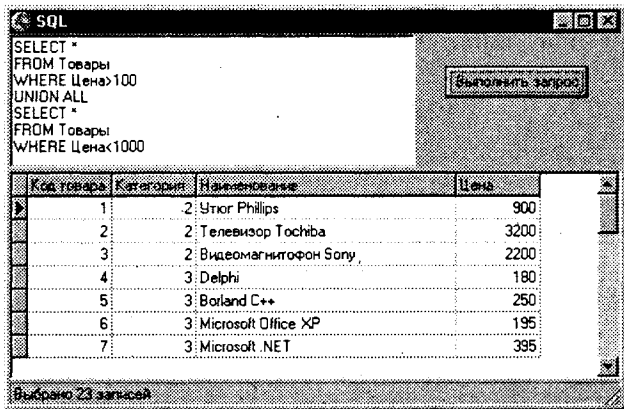


Рис. 11.35. Объединение запросов с помощью оператора UNION ALL

Упорядочение и группировка данных в составных запросах

В составном запросе для упорядочения данных можно использовать предложение ORDER BY. Независимо от того, сколько запросов входит в объединение, допускается только одно предложение ORDER BY. Для задания полей, по которым производится сортировка, в этом предложении разрешается указывать как имена полей, так и их порядковые номера в списке оператора SELECT.

В отличие от предложения ORDER BY, предложение GROUP BY можно задействовать в каждом из запросов, входящих в объединение. Вместе с предложением GROUP BY допускается применение оператора HAVING. Предложение GROUP BY можно использовать и для группировки результатов выполнения составного запроса.

Представления

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 пол-

ностью 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, 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 познакомиться с примером 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 вместо метода `Open`, который мы задействовали для выполнения оператора `SELECT`, требуется метод `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 выполнения операторов, не возвращающих данных, — на кнопке `Exec SQL`. Обработчик щелчка на кнопке `Exec SQL` приведен в листинге 11.2.

Листинг 11.2. Обработчик события `OnClick` кнопки `Exec SQL`

```
procedure TfrmMain.btnExecSQLClick(Sender: TObject);  
begin  
  if Query1.Active then Query1.Close;  
  Query1.SQL.Clear;  
  Query1.SQL.Assign(memSQL.Lines);  
  Query1.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 этого запроса (рис. 11.36) аналогичен результату, который возвратил бы запрос, следующий после ключевого слова AS в операторе CREATE VIEW. При создании представлений допускается использование вычисляемых полей. Например, можно создать представление, подобное рассмотренному, 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. 11.37.

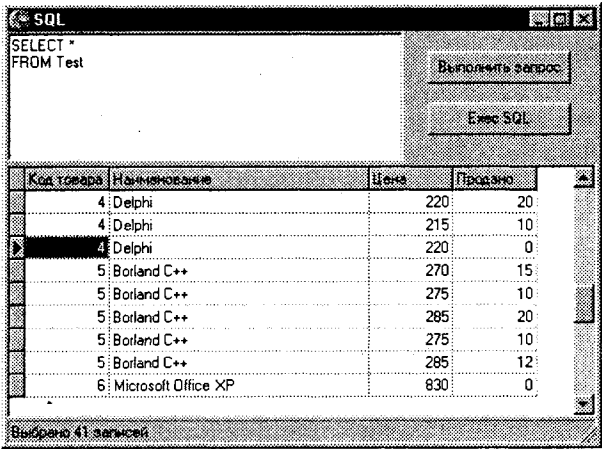


Рис. 11.36. Результат выборки всех записей из представления Test

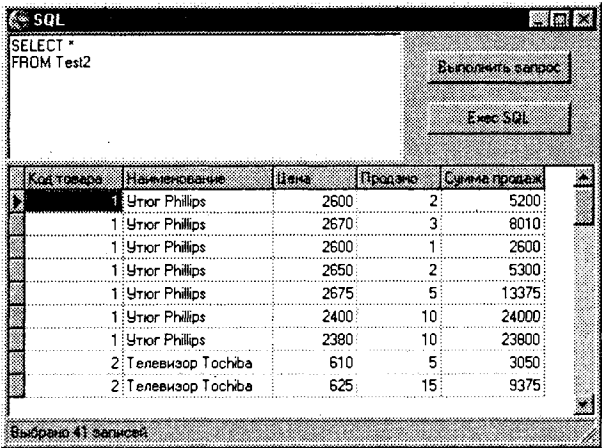


Рис. 11.37. Представление с вычисляемым полем

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
```

SQL-запросы с параметрами

При задании SQL-запроса можно использовать параметры — переменные, включаемые в SQL-оператор, значения которых определяются во время выполнения программы. Параметры в значительной степени повышают гибкость SQL-запросов, обеспечивая возможность при выборке данных запрашивать у пользователя численные значения критериев выборки.

Параметры задаются в тексте SQL-запроса. Для определения параметра перед его именем указывается символ двоеточия (:), 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` инспектора 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 воспользоваться либо 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 практической иллюстрации концепции параметров 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

```
procedure TFormMain.btnExecQueryClick(Sender: TObject);
begin
    if Query1.Active then Query1.Close;
    Query1.SQL.Clear;
    Query1.SQL.Assign(memSQL.Lines);
    if Query1.Params.Count>0
    then begin
        Query1.Params.ParamByName('P_P1').Value:=
            StrToInt(edtParam1.Text);
        Query1.Params.ParamByName('P_P2').Value:=
            StrToInt(edtParam2.Text);
    end;
    Query1.Open;
end;
```

Теперь, если в запросе имеются параметры, их значения будут считываться из полей ввода edtParam1 и edtParam2.

После запуска программы зададим следующий запрос, в котором определены два параметра (P_P1 и P_P2):

```
SELECT *
FROM Товары
WHERE (Цена>:P_P1) AND (Цена<:P_P2)
```

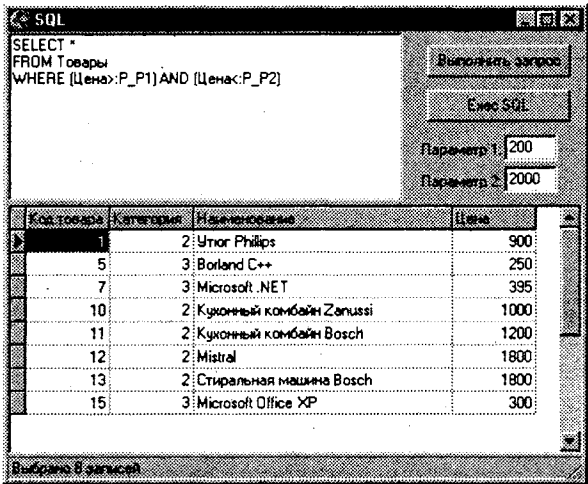


Рис. 11.38. Результат выполнения запроса с параметрами

Зададим значения параметров с полей `edtParam1` и `edtParam2` — в первом укажем значение 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. 11.38).

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. Данный механизм позволяет организовать связь 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 ► VCL Form Applications`.
2. Поместите на форму по два экземпляра компонентов `TQuery`, `TDataSource` и `TDBGrid`. Затем переименуйте шесть размещенных на форме компонентов следующим образом: `Master`, `Detail`, `dsMaster`, `dsDetail`, `DBGMaster`, `DBGDetail`.
3. Для каждого компонента доступа к данным (`Master` и `Detail`) установите связь с базой данных `sales`.
4. Свяжите источник данных `dsMaster` с компонентом доступа к данным `Master`, а источник данных `dsDetail` — с компонентом `Detail`.
5. Свяжите элементы отображения данных `DBGMaster` и `DBGDetail` с источниками данных `dsMaster` и `dsDetail`, соответственно.
6. Задайте в свойстве `SQL` компонента `Master` следующий запрос:

```
SELECT *  
FROM Товары
```

7. Задайте в свойстве `SQL` компонента `Detail` 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.

8. Выберите на форме компонент `Detail` и щелкните в инспекторе объектов на кнопке с многоточием в поле свойства `Params`. В открывшемся окне редактора параметров выберите единственный параметр Код товара и задайте ему тип `Integer` (свойство `DataType` в инспекторе объектов).

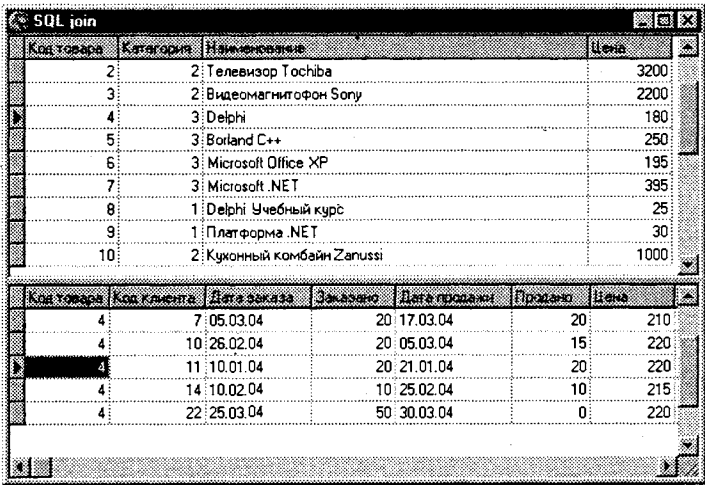
9. Задайте обработчики событий формы OnShow и OnClose. В первом вызовите метод Open для обеих таблиц:

```
procedure TForm1.FormShow(Sender: TObject);
begin
    Master.Open;
    Detail.Open;
end;
```

Во втором обработчике вызовите для двух таблиц метод Close:

```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
    Master.Close;
    Detail.Close;
end;
```

10. Откомпилируйте и запустите программу. В одной из двух таблиц в окне программы будет выводиться вся информация из таблицы Товары, во второй — те записи из таблицы Продажи, для которых значение поля Код товара равно значению одноименного поля выбранной записи в таблице Товары (рис. 11.39).



Код товара	Категория	Наименование	Цена
2	2	Телевизор Toshiba	3200
3	2	Видеомагнитофон Sony	2200
4	3	Delphi	180
5	3	Borland C++	250
6	3	Microsoft Office XP	195
7	3	Microsoft .NET	395
8	1	Delphi Учебный курс	25
9	1	Платформа .NET	30
10	2	Кухонный комбайн Zanussi	1000

Код товара	Код клиента	Дата заказа	Заказано	Дата продажи	Продано	Цена
4	7	05.03.04	20	17.03.04	20	210
4	10	26.02.04	20	05.03.04	15	220
4	11	10.01.04	20	21.01.04	20	220
4	14	10.02.04	10	25.02.04	10	215
4	22	25.03.04	50	30.03.04	0	220

Рис. 11.39. 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.

ГЛАВА 12 Создание отчетов

В предыдущих главах мы выяснили, как получить доступ к информации, хранящейся в таблицах базы данных, и как создать форму для отображения и редактирования этой информации. Используя эти знания, можно создавать приложения баз данных, обладающие удобным и понятным интерфейсом пользователя. Наряду с этим, на практике требуется выводить полученную из базы данных информацию на печать.

Информацию, выводимую на печать (или в файл) и представленную в удобном для восприятия виде, принято называть *отчетом*. Создание отчетов является одной из основных функций приложений баз данных.

Для создания отчетов используются специальные компоненты, которые значительно облегчают эту процедуру, выполняя все основные функции, включая форматирование, предварительный просмотр и вывод на печать информации из некоторого набора данных. Поэтому разработка отчетов, как правило, сводится лишь к форматированию структуры и внешнего вида отчета.

Рекомендации по созданию отчетов

Разработка отчета в какой-то степени подобна разработке формы для ввода данных: выполнение основных функций уже «встроено» в соответствующие компоненты, поэтому требуется только оптимальным образом эти компоненты разместить. Многие компоненты, используемые при создании отчетов, подобны компонентам визуализации данных, применяемым при разработке форм. Поэтому перед разработкой отчета, так же как и перед разработкой формы, следует определить:

- ☐ информацию, которая должна быть представлена в отчете;
- ☐ таблицы, содержащие необходимые данные;
- ☐ внешний вид создаваемого отчета;
- ☐ поля, по которым производится упорядочивание и группировка данных в отчете;
- ☐ содержание итоговой части отчета (если она присутствует);
- ☐ дополнительную информацию, отображаемую в отчете: заголовки, поясняющие надписи, разделительные линии, рисунки и т. п.

Эти детали лучше всего уточнить, дополнительно встретившись с конечными пользователями системы. При этом следует иметь в виду, что для абсолютного большинства пользователей правильное и аккуратное оформление отчета имеет весьма существенное значение. Положительное или отрицательное мнение конечных пользователей об отчетах может предопределить отношение ко всей системе целиком.

Перед разработкой отчета может быть полезно нарисовать эскиз отчета на бумаге. Это позволит сразу определить перечень компонентов, которые потребуются для создания отчета, и сократит время его разработки.

ПРИМЕЧАНИЕ

Набор данных, на основе которого создается отчет, удобно формировать с помощью SQL-запросов, особенно в том случае, когда в отчете должна содержаться информация из нескольких таблиц. В этом случае с помощью SQL-запроса производится выборка требуемой информации, а компоненты создания отчета обеспечивают форматирование и вывод этой информации.

Типы отчетов

Компоненты для создания отчетов, входящие в поставку средств разработки приложений, позволяют формировать отчеты различных видов. Условно все отчеты можно разделить на две группы:

- ☐ табличные отчеты;
- ☐ отчеты в свободной форме.

В случае табличного отчета информация представляется в виде таблицы, то есть упорядочивается по строкам и столбцам. Такие отчеты фактически повторяют структуру таблиц базы данных (или выборку из таблиц). Так как табличное представление является наиболее простой и естественной формой представления данных, этот тип отчетов применяется наиболее часто.

Отчеты, выполненные в свободной форме, содержат информацию, не упорядоченную в виде таблицы, а располагающуюся в произвольной форме. Примером отчетов такого типа могут служить почтовые адреса и письма, где поля таблицы базы данных должны располагаться в специально отведенных местах. Следует отметить, что при реализации отчетов в свободной форме данные все равно представляются в упорядоченном виде, но результат упорядочения не является таблицей.

Помимо значений полей из таблиц базы данных (или вычисляемых полей), отчет может содержать и другие объекты, например графики и диаграммы, рисунки, поясняющие надписи.

Генератор отчетов Rave Reports

В стандартную поставку Delphi входит набор компонентов для создания отчетов, разработанных фирмой Nevrona. Данные компоненты, расположенные на странице Rave палитры компонентов (рис. 12.1) и входящие в генератор отчетов Rave Reports for Delphi 8, позволяют настроить связь между приложением и отчетом, а сам отчет

создается с помощью реактора, вызываемого из контекстного меню компонента TRvProject. Можно сначала создать отчет, а затем подключить его к приложению.

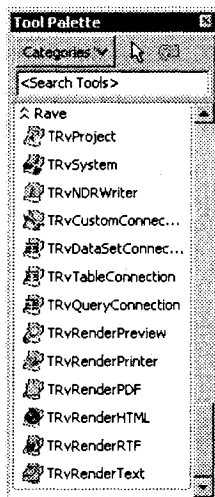


Рис. 12.1. Страница Rave палитры компонентов

ВНИМАНИЕ

Генератор отчетов Rave Reports после установки автоматически интегрируется в среду Delphi 8, но устанавливать его нужно отдельно.

Генератор отчетов Rave Reports позволяет создавать как табличные отчеты, так и отчеты в свободной форме. Отчет, созданный с помощью Rave Reports, можно не только распечатать, но и просмотреть на экране, а также записать в файл.

Все компоненты генератора отчетов Rave Reports, доступные в Delphi, можно разделить на группы:

- ☐ базовый компонент TRvProject связывает приложение и отчет;
- ☐ компоненты TRvCustomConnection, TRvDataSetConnection, TRvTableConnection и TRvQueryConnection служат для связывания отчета с набором данных;
- ☐ компоненты TRvNDRWriter, TRvRenderPrinter, TRvRenderPreview, TRvRenderPDF, TRvRenderHTML, TRvRenderRTF и TRvRenderText используются для печати, предварительного просмотра и сохранения отчета в различных форматах.

Структура генератора отчетов Rave Reports

Формой для будущего отчета в Rave Reports служит виртуальный образ листа бумаги (в центре экрана). В общем случае страниц отчета может быть несколько, каждая из них располагается на отдельной вкладке.

Слева от образа отчета расположен инспектор объектов, посредством которого можно задавать свойства компонентов отчета. Справа расположен менеджер проекта, в котором отображается структура отчета, подключенные к нему данные.

ПРИМЕЧАНИЕ

Форму будущего отчета можно настроить с помощью команды Edit ► Preferences главного меню Rave Reports.

Над формой отчета расположены страницы компонентов, которые можно использовать для вывода информации в отчет и графического оформления отчета:

- ☐ Drawing — графические средства оформления отчета (линии, геометрические фигуры);
- ☐ Bar Core — различные варианты штриховки;
- ☐ Standard — компоненты для отображения статичной информации (надписей, картинок, многострочных текстовых полей, номеров страниц отчета);
- ☐ Report — компоненты связи с данными;
- ☐ Zoom — средства масштабирования виртуальных страниц;
- ☐ Color — средства управления цветом отчета;
- ☐ Lines — средства построения линий отчета;
- ☐ Fills — компоненты заливки фона отчета;
- ☐ Fonts — средства управления шрифтами в отчете;
- ☐ Alignmet — средства выравнивания компонентов отчета.

Пример создания отчета

Покажем, как использовать генератор отчетов Rave Reports, на примере создания простого отчета — списка сотрудников некоторой фирмы из базы данных Sales.

1. Создайте новое приложение, выполнив команду File ► New ► VCL Forms Application.
2. На форму поместите компонент TTable со страницы BDE палитры компонентов. Выполните подключение базы данных к приложению, как это описано в главе 9. Свойству DataBaseName компонента TTable присвойте значение псевдонима базы данных. Выберите соответствующую таблицу с персональными данными сотрудников в базе данных Sales и присвойте это значение свойству TableName. Свойству Active присвойте значение true.
3. Поместите на форму компоненты TRvProject и TRvTableConnection (рис. 12.2). Свойству Table присвойте значение Table1.
4. Вызовите генератор отчетов командой Rave Visual Designer контекстного меню компонента TRvProject. Окно генератора отчетов показано на рис. 12.3.
5. Далее подключим к отчету источник данных. Для этого выполните команду File ► New Data Object, в списке открывшегося окна выберите пункт Direct Data View (рис. 12.4) и щелкните на кнопке Next.
6. В списке появившегося окна выберите пункт RvTableConnection1, как показано на рис. 12.5. Так мы связываем отчет с определенной таблицей базы данных. Отчет можно также связать с набором данных, полученным в результате выполнения SQL-запроса (компонент TRvQueryConnection), или с данными из обычных файлов либо массивов (компонент TRvCustomConnection).

После щелчка на кнопке Finish в окне менеджера проекта появится значок представления данных и каждого поля таблицы. Теперь мы можем помещать на форму компоненты и связывать их с данными путем задания свойств DataView и DataField.

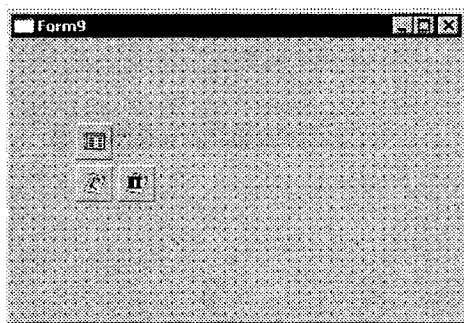


Рис. 12.2. Размещение компонентов на форме при создании отчета

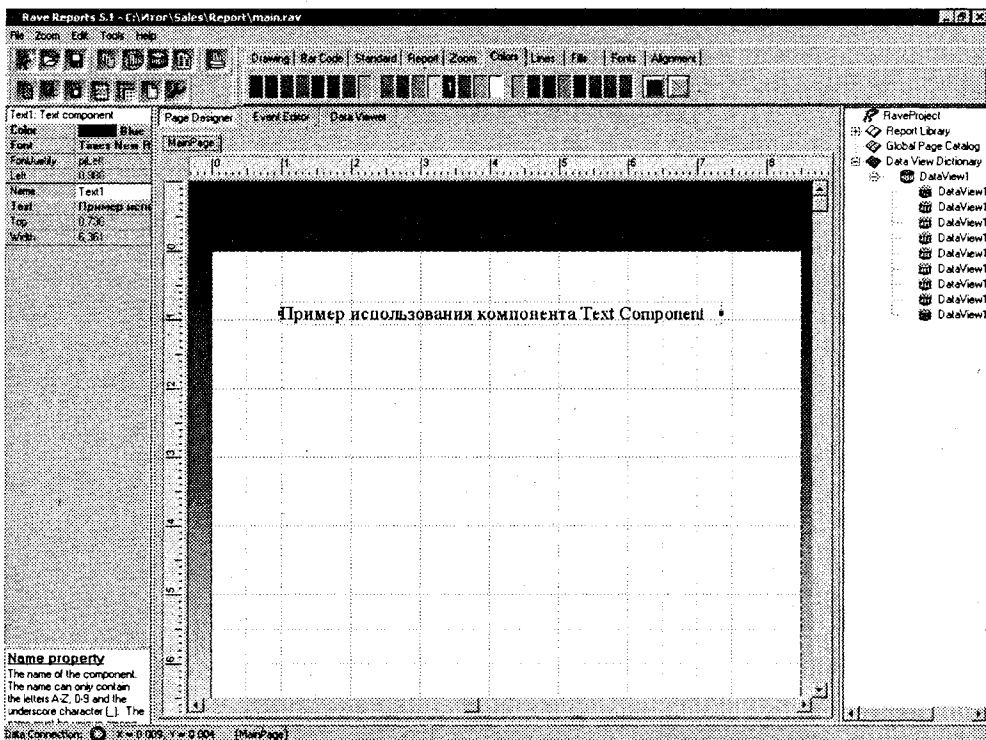


Рис. 12.3. Внешний вид генератора отчетов Rave Reports

7. Вручную задавать все полосы и поля отчета достаточно долго и трудоемко. Воспользуемся средством автоматической генерации отчетов. Для этого выберите команду Tools ► Report Wizards ► Simple Table (для создания отчета с использованием данных из нескольких таблиц нужно было бы выбрать команду Tools ► Report

Wizards ► Master/Detail Reports). Rave Reports предложит указать для отчета набор данных (рис. 12.6, а), поля таблицы базы данных (рис. 12.6, б), порядок группировки данных (рис. 12.6, в), заголовок и отступы отчета (рис. 12.6, г).

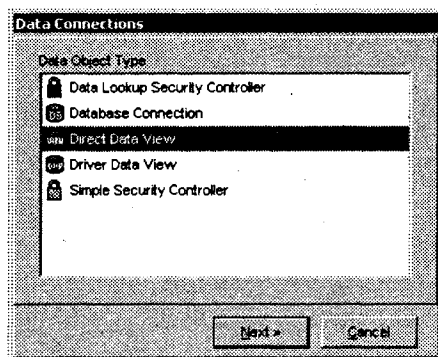


Рис. 12.4. Установление связи с базой данных

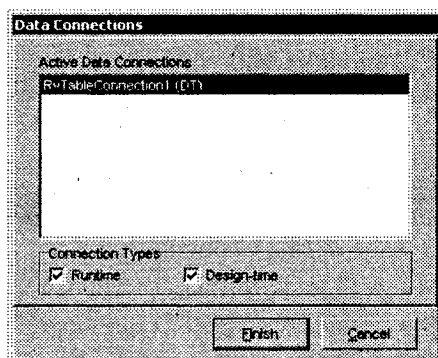
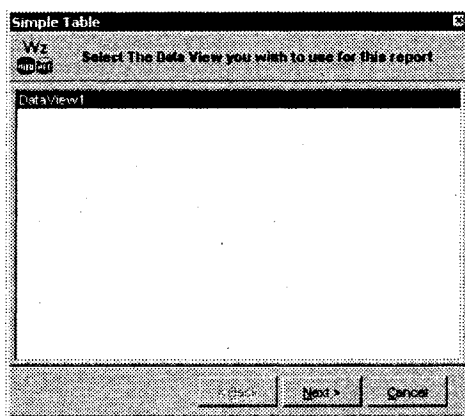


Рис. 12.5. Соединение с таблицей базы данных



а

Рис. 12.6. Автоматическая генерация отчета

Simple Table

Wz

Select The Fields you wish to use for this report

☐ NUMBER
☐ TABNUMBER
☒ NAME
☐ VATERNAME
☒ FAMILIE
☒ JOB
☐ ADRES
☐ TELEFON
☐ COST

< Back Next > Cancel

6

Simple Table

Wz

Arrange Fields In The Order You Wish Them Printed

↑ ↓

NAME
FAMILIE
JOB

< Back Next > Cancel

8

Simple Table

Wz

Report Layout Options

Report Title
Список сотрудников в фирмы "Елена"

☒ Print Report Title

Report Margins

Left: 1.0
Top: 1.0
Right: 1.0
Bottom: 1.0

< Back Next > Cancel

8

Рис. 12.6. Автоматическая генерация отчета

8. Задавая соответствующие параметры, щелкайте на кнопке Next, чтобы получить искомый отчет (рис. 12.7).

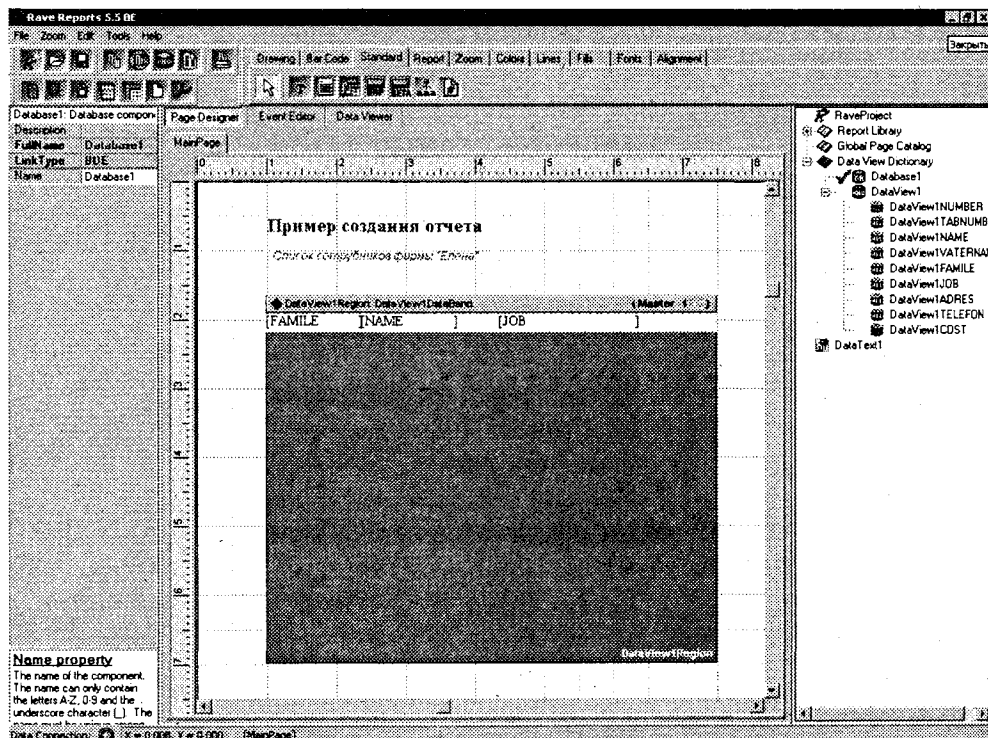


Рис. 12.7. Пример простого отчета

9. Полученный отчет можно протестировать, щелкнув на кнопке предварительного просмотра (седьмая слева кнопка на панели инструментов Rave Reports) или выбрав соответствующую команду в контекстном меню отчета. Откроется окно диалога, показанное на рис. 12.8.

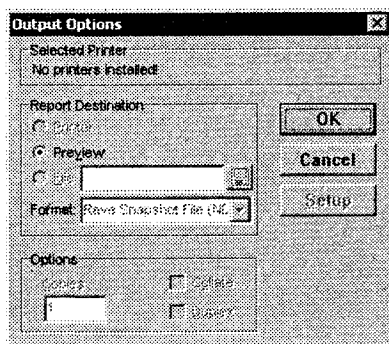


Рис. 12.8. Задание параметров предварительного просмотра отчета

10. Установите переключатель Preview и щелкните на кнопке ОК, чтобы увидеть проектируемый отчет в области предварительного просмотра (рис. 12.9).

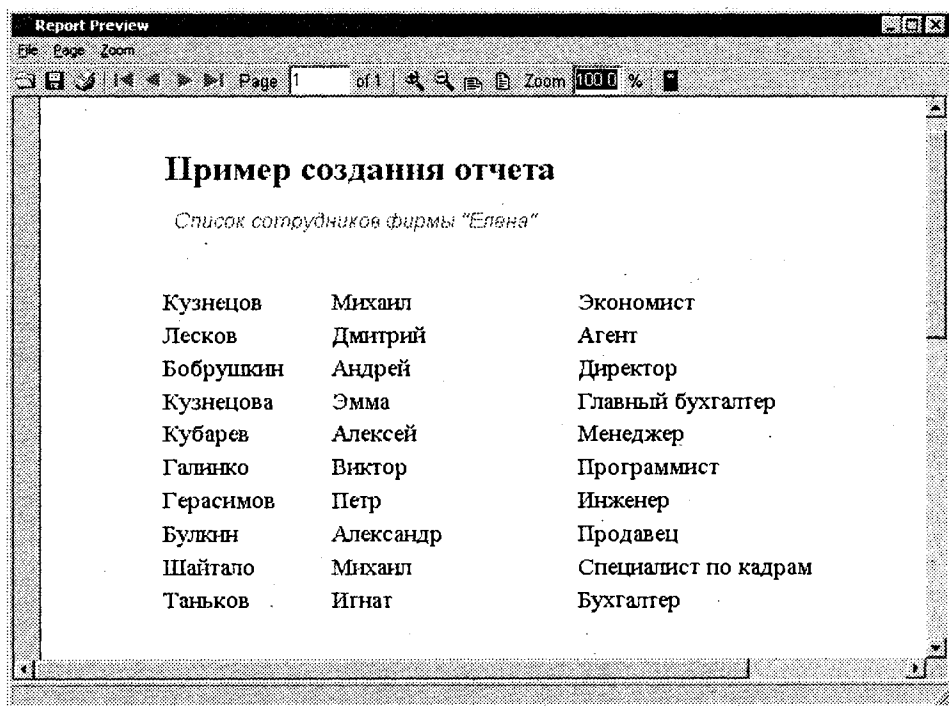


Рис. 12.9. Предварительный просмотр отчета

11. Созданный отчет сохраните в файле проекта, имя которого необходимо задать свойству ProjectFile компонента TRvProject.

Получившийся отчет содержит в себе несколько компонентов. На странице Event Editor можно тут же задать реакцию на события, связанные с этими компонентами, и откомпилировать созданные обработчики на языке Delphi.

Использование отчетов в Delphi

Управлять созданием и использованием отчета можно программно. Однако за исключением сложных приложений в этом нет необходимости — обычно вполне достаточно предварительного просмотра отчета и вывода его на печать.

Следующий фрагмент демонстрирует возможность печати отчета. Для этого мы добавили на форму кнопку и задали реакцию на щелчок на ней:

```
procedure TForm9.Button1Click(Sender: TObject);
begin
  Form9.RvProject1.Open;
  Form9.RvProject1.ExecuteReport('Report1');
  Form9.RvProject1.Close;
end;
```

Сначала мы открыли сохраненный ранее отчет. Затем с помощью метода `ExecuteReport` инициировали печать или предварительный просмотр отчета `Report1` (в большинстве случаев проект будет содержать несколько видов отчетов). Результат выполнения метода аналогичен представленному на рис. 12.9.

Завершив работу с отчетом, его нужно закрыть. Платформа .NET, хотя и очищает память от мусора, но делает это недостаточно оперативно.

Отчет можно печатать и сохранять также с помощью других компонентов:

- ☐ `TRvNDRWriter` — сохраняет отчет в двоичной форме (NDR-поток);
- ☐ `TRvRenderPreview`, `TRvRenderPrinter` — позволяют осуществлять предварительный просмотр и печать отчета (предварительно подготовленного компонентом `TRvNDRWriter`);
- ☐ `TRvRenderPDF`, `TRvRenderHTML`, `TRvRenderRTF`, `TrvRenderText` — сохранение отчета, предварительно подготовленного компонентом `TRvNDRWriter`, соответственно в формате PDF, HTML, RTF или TXT.

Часть IV

Компоновка приложения и управление проектом

ГЛАВА 13 Система меню и панель инструментов приложения

Меню является одним из наиболее важных элементов программного интерфейса. С его помощью обеспечивается доступ пользователя ко всем функциям приложения. Именно поэтому такой элемент интерфейса нашел широкое применение еще в программах для DOS, а в программах, функционирующих в среде Windows, он стал обязательным.

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 наступлением эпохи Windows ситуация изменилась, и в настоящее время панели инструментов присутствуют практически во всех современных приложениях.

Планирование приложения

Разработка любого приложения должна начинаться с его планирования — определения требований, предъявляемых к приложению.

Для приложения, предназначенного для работы с базами данных, можно условно выделить три основных этапа его создания.

1. Разработка структуры базы данных, то есть состава и структуры таблиц, из которых состоит база данных, а также отношений между ними.
2. Определение функций, выполняемых приложением.
3. Разработка интерфейса пользователя.

Перечисленные этапы не обязательно должны выполняться в указанной последовательности. Однако обычно разработку приложения рекомендуется начинать именно с разработки структуры базы данных, так как значительная часть про-

граммного кода жестко привязана к ее структуре. Поэтому модификация структуры базы данных на завершающих стадиях разработки приложения может привести к возникновению трудно обнаруживаемых ошибок.

При разработке функций приложения определяются методы обработки данных, обеспечивается контроль вводимых пользователем значений, а также контроль за удалением и модификацией содержащихся в таблицах данных. Кроме того, при разработке приложений, работающих с локальными базами данных, может потребоваться реализация функций разграничения доступа к информации, содержащейся в базе данных.

Разработка интерфейса пользователя обычно сводится к разработке форм, окон диалога, 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 очень красивым, но непривычным и малопонятным для «среднего» пользователя интерфейсом.

Меню окончательно формируется на последней стадии разработки приложения, когда все функции программы определены и создан окончательный вариант используемой структуры данных. Меню, как правило, содержит полный набор команд, обеспечивающих доступ пользователя ко всем 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 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.

Панели инструментов содержат кнопки, обеспечивающие быстрый доступ к командам меню (конечно, панели инструментов могут содержать и «инструменты» в прямом смысле этого слова, например инструменты для рисования в графических редакторах). Как правило, панели инструментов не обеспечивают доступа ко всем функциям приложения, то есть содержат сокращенный набор команд.

Современные средства программирования 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. Кроме того, в панелях инструментов могут быть полезными всплывающие подсказки о назначении кнопок, так как размещаемые на них значки далеко не всегда вызывают правильные ассоциации с соответствующими командами.

Создание главного меню

Строка меню располагается в верхней части главной формы приложения. Доступ к командам этого меню 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 VCL Delphi имеется специальный компонент TMainMenu, расположенный на странице Standard палитры компонентов.

Компонент TMainMenu обычно относят к невидимым, хотя главное меню отображается во время разработки приложения. Основные свойства класса TMainMenu приведены в табл. 13.1.

Таблица 13.1. 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 в качестве значков, отображаемых слева от соответствующих им команд меню
OwnerDraw	Boolean	Определяет, как будет происходить прорисовка меню. Если имеет значение false, прорисовка выполняется автоматически, если true — для отображения меню необходимо программировать обработчик события OnDrawItem. Обычно это свойство 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 пункта меню (объекта TMenuItem).

```
procedure Unmerge(Menu: TMainMenu);
```

Исключает из меню формы пункты, добавленные при слиянии двух меню.

ПРИМЕЧАНИЕ

Если свойство `AutoMerge` дочерней формы равно `true`, то методы `Merge` и `Unmerge` вызываются автоматически при открытии и закрытии дочерней формы.

Класс TMenuItem

Каждый пункт меню является экземпляром класса `TMenuItem`. Причем объект `TMenuItem` может либо быть командой, либо подменю, открывающим доступ к пунктам следующего уровня вложенности. Количество уровней вложенности не ограничено. Основные свойства класса `TMenuItem` приведены в табл. 13.2.

Таблица 13.2. Основные свойства класса `TMenuItem`

Свойство	Тип	Описание
Action	TBasicAction	Определяет действие, связанное с данной командой меню
Bitmap	TBitmap	Определяет значок, показываемый слева от команды меню
Break	TMenuBreak = (mbNone, mbBreak, mbBarBreak)	Используется для отображения команд меню в несколько столбцов: элемент меню отображается в текущей колонке (mbNone); начиная с текущего, пункты меню отображаются в следующей колонке (mbBreak); начиная с текущего, пункты меню отображаются в следующей колонке, причем между колонками располагается разделитель (mbBarBreak)
Caption	String	Текст команды
Checked	Boolean	Используется для команд меню, функционирующих подобно флажкам. Если данное свойство имеет значение <code>true</code> , то команда меню помечается (флажок устанавливается)
Default	Boolean	Используется для команд подменю. Если значение свойства <code>Default</code> задано равным <code>true</code> , то данная команда будет выполняться при двойном щелчке на имени подменю. Из всех команд, входящих в подменю, только одна может иметь значение этого свойства равным <code>true</code>
Enabled	Boolean	Определяет, доступна команда меню (<code>true</code>) или нет (<code>false</code>). Недоступные команды отображаются серым цветом
GroupIndex	Byte	Используется при объединении двух меню
ImageIndex	TImageIndex	Указывает на изображение в коллекции, заданной в свойстве <code>Images</code> объекта <code>TMainMenu</code> , которое будет отображаться слева от команды. Данное свойство имеет более высокий приоритет, чем свойство <code>Bitmap</code>
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 переключателям. Для группировки пунктов меню используется свойство <code>GroupIndex</code>
ShortCut	TShortCut	Определяет клавиатурные сокращения для команд 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 пункты меню и освобождает занимаемую ими память.

`procedure Click;`

Вызывает событие `OnClick` для данного пункта меню.

`function Find(ACaption: string): TMenuItem;`

Возвращает указатель на пункт меню, соответствующий указанному в `ACaption` тексту команды. Если такой пункт не найден, возвращает `nil`.

`function IndexOf(Item: TMenuItem): Integer;`

Возвращает порядковый номер пункта меню.

`procedure Insert(Index: Integer; Item: TMenuItem);`

Добавляет в меню пункт `Item`, располагая его в позиции `Index`.

`function InsertNewLineAfter(AItem: TMenuItem): Integer;`

Добавляет разделитель после указанного пункта меню.

`function InsertNewLineBefore(AItem: TMenuItem): Integer;`

Добавляет разделитель перед указанным пунктом меню.

`function IsLine: Boolean;`

Определяет, является пункт меню разделителем или нет.

`procedure Remove(Item: TMenuItem);`

Удаляет указанный пункт 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` равно `true`.

Работа с редактором меню

Для создания меню в Delphi используется специальный редактор (рис. 13.1), 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.

Создание и удаление команд меню

Для добавления в меню нового пункта (команды или подменю) выберите с помощью мыши (или клавиш перемещения курсора) свободный элемент (такие элементы всегда есть внизу каждого выпадающего меню и в правой части строки меню) и введите текст для пункта в поле ввода свойства `Caption` в инспекторе объектов.

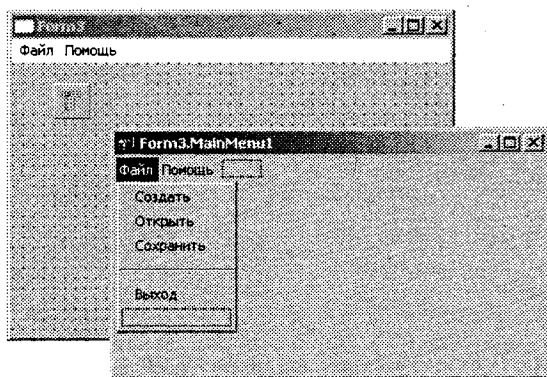


Рис. 13.1. Окно редактора меню

Для вставки новых пунктов меню между уже существующими используйте команду **Insert** контекстного меню редактора или клавишу **Ins**. Созданный при этом новый пункт будет располагаться выше (или левее) того пункта, который был выделен перед выполнением команды **Insert**.

Чтобы удалить пункт меню, можно воспользоваться либо клавишей **Del**, либо командой **Delete** контекстного меню.

При написании текста пункта 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 клавиш 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 к команде меню значка, который отображается слева от команды (рис. 13.2).

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 значков по умолчанию устанавливается в каталог **/Images/Default**.

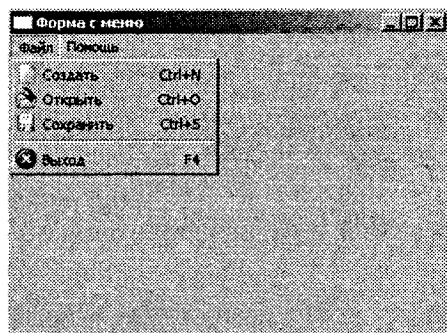


Рис. 13.2. Пример меню со значками

Назначение командам меню оперативных клавиш

Для задания оперативных клавиш используется свойство `ShortCut` класса `TMenuItem`. В инспекторе объектов поле ввода данного свойства содержит раскрывающийся 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 обозначения: `Alt`, `Ctrl`, `Shift`, `BkSp`, `Ins`, `Del`.

Комбинация оперативных клавиш, назначенных команде меню, отображается справа от текста команды (см. рис. 13.2).

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. 13.2, разделитель отделяет команду `Выход`).

Разделители можно создавать в редакторе меню точно так же, как и обычные команды. Для этого в свойстве `Caption` пункта меню следует просто ввести символ дефиса (-).

Создание подменю

Чтобы создать подменю, выполните следующие действия.

1. Наведите указатель мыши на команду меню.
2. Щелкните правой кнопкой мыши.
3. Выберите команду `Create SubMenu` в контекстном 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 сопровождается выполнением определенных действий. Для задания этих действий можно использовать два способа:

- ☐ задать обработчик события `OnClick` пункта меню;
- ☐ задать для пункта 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;
- ☐ просто выбрать эту команду в меню формы во время разработки приложения.

В обоих случаях IDE Delphi автоматически активизирует окно редактора кода и генерирует заголовок 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 OpenFileDialog.Execute  
    then begin  
        ....  
    end;  
end;
```

Использование свойства `Action`

Для использования свойства `Action` необходим 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. 13.3), который открывается двойным щелчком мыши на компоненте `TActionList`, помещенном на форму.

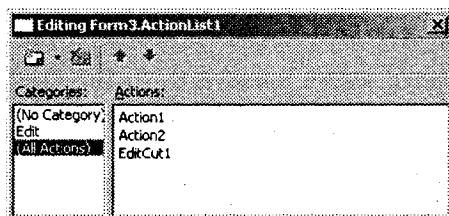


Рис. 13.3. Окно редактора действий

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**, включая **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** пункта меню, то соответствующая команда будет отображаться согласно параметрам, указанным для действия (используются заданные для действия текст, 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 применять 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 — задействуется тот же редактор меню, а свойства и методы класса **TPopupMenu** аналогичны рассмотренным ранее свойствам и методам класса **TMenuItem**.

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 быстрый доступ к командам.

Хотя панели инструментов могут создаваться несколькими способами, наиболее простой и естественный из них — использование специального компонента **TToolBar**, который по умолчанию находится на странице Win32 палитры компонентов.

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. 13.3.

Таблица 13.3. Основные свойства класса TToolBar

Свойство	Тип	Описание
ButtonWidth	Integer	Ширина кнопок панели инструментов
ButtonHeight	Integer	Высота кнопок панели инструментов
Images	TCustomImageList	Ссылка на список изображений, которые будут отображаться на кнопках, находящихся в обычном состоянии
DisabledImages	TCustomImageList	Ссылка на список изображений, отображаемых на недоступных кнопках
Flat	Boolean	Внешний вид кнопок: обычные (false) или плоские (true)
HotImages	TCustomImageList	Коллекция изображений, отображаемых на кнопках при наведении 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 перетаскивании панели мышью: 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. 13.4.

Таблица 13.4. 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 равно true)
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 изображение из списка Images, отображаемое 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 выводятся, если false — нет
Style	TToolButtonStyle = (tbsButton, tbsCheck, tbsDropDown, tbsSeparator, tbsDivider);	Стиль кнопки: обычная кнопка (tbsButton); кнопка с двумя состояниями, являющаяся аналогом флажка (tbsCheck); кнопка с выпадающим меню (tbsDropDown); разделитель (tbsSeparator или tbsDivider)

Чтобы поместить кнопку на панель 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 перетаскивать мышью.

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 щелчка на кнопке.

События, определенные для класса TToolButton, генерируются при щелчке на кнопке (OnClick) или при перетаскивании кнопки и аналогичны событиям компонента TToolBar.

Так же как и команды меню, кнопки на панели инструментов могут использоваться в качестве аналогов флажков (TCheckBox) и переключателей (TRadioButton). В первом случае достаточно присвоить свойству Style кнопки значение tbsCheck. После этого кнопка сможет 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; при этом свойству Style всех кнопок группы следует присвоить 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 уже «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 (хотя ширина разделителя может отличаться от ширины обычной кнопки).

Различие между стилями tbsSeparator и tbsDivider заключается в следующем: в первом случае кнопки разделяются просто некоторым интервалом, на котором никакие дополнительные элементы не отображаются (первый разделитель на рис. 13.4), во втором случае кнопки разделяются вертикальной чертой (второй разделитель на рис. 13.4).

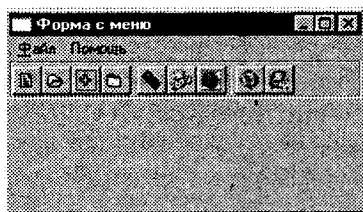


Рис. 13.4. Форма с панелью инструментов

Обработка щелчка на кнопке

Обработка щелчка на кнопке панели инструментов может быть реализована 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 щелчок на кнопке инициирует действие, связанное с этим объектом. Причем неважно, каким образом задана реакция на щелчок — 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, палитра компонентов и т. п.

В VCL Delphi имеются два компонента, выполняющие функции контейнеров для панелей инструментов: `TCoolBar` и `TControlBar`. Первый из них является разработкой фирмы Microsoft и содержится в библиотеке `comctl32.dll`. Второй разработан фирмой Inprise и входит в состав VCL. С точки зрения функциональных возможностей между двумя этими компонентами нет существенных различий.

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 требуется специальный редактор, который открывается двойным щелчком мыши на компоненте `TCoolBar`, помещенном на форму, или при щелчке на кнопке с многоточием после свойства `Bands` инспектора объектов. 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 перетаскивать панели инструментов из контейнера на форму и наоборот. На рис. 13.5 показана форма с двумя панелями инструментов: одна из них расположена в контейнере `TCoolBar`, вторая — «плавающая».

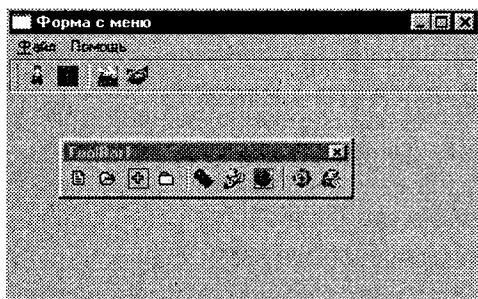


Рис. 13.5. Пример «плавающей» панели инструментов

ПРИМЕЧАНИЕ

Путем настройки свойств DockSite и DragMode можно реализовать механизм перетаскивания (drag-and-dock), не написав ни одной строки программного кода. Однако следует иметь в виду, что на практике часто возникают ситуации, когда требуется организовывать механизм перетаскивания «вручную», то есть задавая соответствующий программный код. К сожалению, ограниченный объем книги не позволяет рассмотреть этот вопрос более подробно.

Компонент TControlBar

Использование компонента TControlBar не требует создания дополнительных объектов. Достаточно просто поместить панель инструментов (или любой элемент управления) в контейнер TControlBar. В остальном работа с этим компонентом практически ничем не отличается от работы с контейнером TCoolBar.

ГЛАВА 14 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 ею 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 приложения Windows Forms для платформы .NET;
- ☐ разрабатывать динамически связываемые библиотеки (библиотеки DLL);
- ☐ создавать элементы ActiveX;
- ☐ создавать VCL-компоненты для Delphi;
- ☐ создавать приложения для Интернета.

При создании приложений VCL разработка начинается с создания 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 одним из следующих способов:

- ☐ выбором команды File ► New Application в главном меню;
- ☐ выбором команды File ► New и последующим двойным щелчком на объекте Application в открывшемся окне хранилища объектов.

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. Помимо этих трех основных файлов проект содержит еще три файла, в которых находится служебная информация:

- ☐ файлы с расширениями `dof` и `cfg` содержат сведения о параметрах проекта, заданных в окне диалога **Project Options**;
- ☐ файл с расширением `res` — это файл ресурсов проекта.

Модуль формы проекта

При создании нового приложения в редактор кода автоматически загружается файл с расширением `pas`, содержащий код модуля формы (листинг 14.1). По структуре этот файл является обычным модулем на языке **Object Pascal**.

- ☐ Объявляются основные стандартные модули Delphi, необходимые для функционирования компонентов, входящих в приложение. Затем, если на форму помещается компонент, для которого требуется модуль, не указанный предварительно в разделе `uses` модуля формы, объявление этого модуля добавляется автоматически.
- ☐ Объявляется новый класс формы `TForm1`, являющийся потомком стандартного класса `TForm`. В данном классе будут инкапсулированы все компоненты, размещаемые на форме.
- ☐ Объявляется переменная `Form1` — экземпляр класса `TForm1`. Если затем с помощью инспектора объектов изменить имя формы, то имя этой переменной также будет автоматически изменено.
- ☐ Директивой `{ $R *.DFM }` подключается файл ресурсов формы.

Листинг 14.1. Код модуля формы

```
unit Unit1;

interface

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

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

var
```

Листинг 14.1 (продолжение)

```
Form1: TForm1;  
  
Implementation  
  
{ $R *.DFM }  
  
end.
```

ПРИМЕЧАНИЕ

Файл модуля проекта фактически является шаблоном, в который затем добавляются 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 14.2). 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 IDE Delphi.

Листинг 14.2. 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.

☐ **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 далее):**

- **Initialize** — инициализация приложения;
- **CreateForm(TForm1, Form1)** — создание формы Form1;
- **Run** — запуск приложения.

Файл описания формы проекта

Файл формы (DFM-файл) 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 файла описания формы, на которой размещены кнопка (компонент **TButton**) и поле ввода (компонент **TEdit**), приведен в листинге 14.3. Сама форма показана на рис. 14.1.

Листинг 14.3. Файл описания формы

```
object Form1: TForm1
Left = 295
Top = 164
Width = 304
Height = 190
Caption = 'Form1'
Color = clBtnFace
Font.Charset = DEFAULT_CHARSET
Font.Color = clWindowText
Font.Height = -11
Font.Name = 'MS Sans Serif'
Font.Style = []
OldCreateOrder = False
PixelsPerInch = 96
TextHeight = 13
object Button1: TButton
Left = 40
Top = 96
Width = 121
Height = 25
Caption = 'Button1'
TabOrder = 0
end
object Edit1: TEdit
Left = 40
Top = 48
Width = 121
Height = 21
TabOrder = 1
Text = 'Edit1'
end
end
```

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, редактируя текст описания формы, — достаточно 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 изменять файл описания формы вручную обычно не требуется.

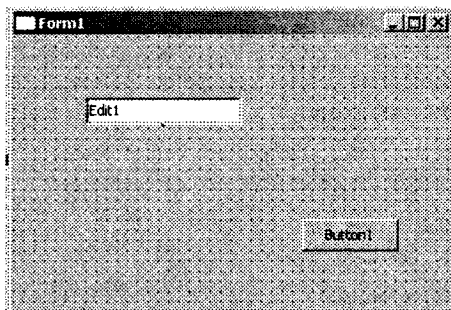


Рис. 14.1. Пример формы проекта

Добавление в проект 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 создании нового проекта приложения 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** главного меню (или команду **File ► New** с последующим выбором объекта **Form** в окне хранилища объектов). Главной формой при этом остается форма, созданная 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** главного меню, а затем выберите объект **Unit** в открывшемся окне хранилища объектов.

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 включается в проект и 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**, содержащее список всех форм и модулей, принадлежавших проекту (рис. 14.2).

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.

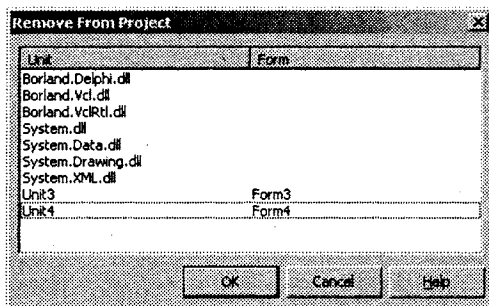


Рис. 14.2. Окно диалога Remove From Project предназначено для удаления модуля из проекта

Для подключения вновь созданных модулей (неважно, связанных с формами или нет) используйте команду File ► Use Unit главного меню. При ее выборе открывается окно диалога Use Unit, содержащее список всех модулей проекта (рис. 14.3), не объявленных в разделе uses текущего файла в редакторе кода. Выберите в списке те модули, которые вы хотите подключить, и щелкните на кнопке OK.

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.

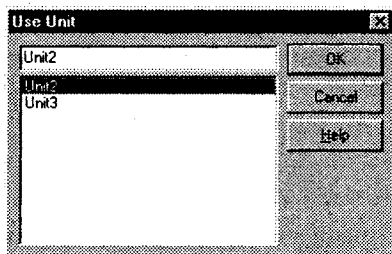


Рис. 14.3. Окно диалога Use Unit используется для подключения модулей

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 доступна во всех модулях приложения. Основные свойства класса TApplication приведены в табл. 14.1.

Таблица 14.1. Основные свойства класса TApplication

Свойство	Тип	Описание
Active	Boolean	Если значение свойства равно true, то приложение активно
CurrentHelpFile	string	Имя файла контекстной справки
HelpFile	string	Имя файла справки приложения
ExeName	string	Имя исполняемого файла приложения
Handle	HWD	Дескриптор главного окна приложения
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.

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;
- ☐ OnDeactivite — вызывается при деактивизации приложения;
- ☐ OnMessage — вызывается при получении сообщения от операционной системы;
- ☐ OnShortCut — вызывается при нажатии любой клавиши.

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, может нерационально использовать память — хранить информацию, не нужную 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. 14.4).

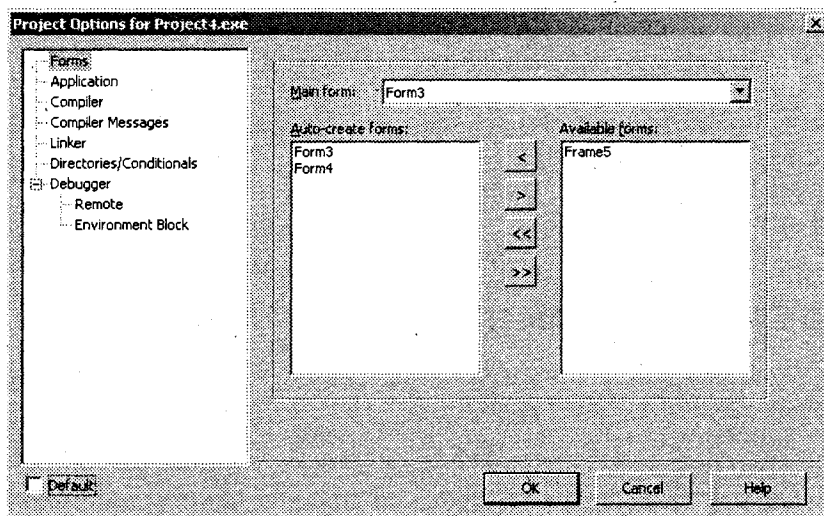


Рис. 14.4. Вкладка Forms окна диалога Project Options предназначена для управления формами

На вкладке Forms имеются следующие элементы управления:

- ☐ в списке Auto-create forms перечислены формы, которые создаются автоматически;
- ☐ в списке Available forms перечислены формы, содержащиеся в проекте, но не создаваемые при запуске приложения;
- ☐ кнопки > и < служат для перемещения выделенной формы из одного списка в другой;
- ☐ кнопки >> и << служат для перемещения всех форм, находящихся в одном списке, в другой список;
- ☐ раскрывающийся список Main form позволяет выбрать из всех форм проекта главную.

ПРИМЕЧАНИЕ

Главная форма обязательно должна создаваться автоматически. Поэтому если выбрать в списке Main form форму, находящуюся в списке Available forms, она автоматически переместится в список Auto-create forms.

Управлять процессом создания форм можно также путем редактирования кода главного файла проекта. Для всех автоматически создаваемых форм перед выпол-

нением 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 метода `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, начиная с версии 4, существует возможность объединения нескольких проектов в одну группу. Создание группы проектов полезно, например, в том случае, если приложение использует динамические 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

Создание новой группы проектов выполняется с помощью команд `Add New Project` и `Add New Project Group` меню `Project` или контекстного меню менеджера проектов.

Группу проектов можно также создать на основе уже существующего проекта. Для 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 группы проектов генерируется файл ее описания, имеющий расширение `brg` 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. 14.5 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.

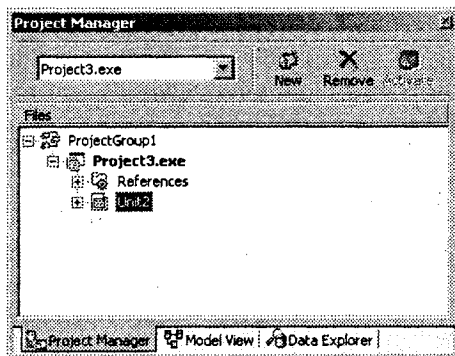


Рис. 14.5. Окно менеджера проектов

Управление проектами, входящими в группу проектов, производится либо с помощью кнопок панели инструментов, либо с помощью команд контекстного меню. Менеджер проектов позволяет добавлять новые проекты в группу, удалять проекты из группы, а также добавлять и удалять модули, входящие в каждый отдельный проект.

Из всех проектов, содержащихся в группе, только один может быть активным. Именно активный проект запускается командой **Run** ► **Run**. Выбор активного проекта производится в окне менеджера проектов одним из следующих способов:

- ☐ щелчком на кнопке **Activate**;
- ☐ выбором проекта в раскрывающемся списке;
- ☐ двойным щелчком на имени проекта.

Окно настройки параметров проекта

Для настройки параметров проекта служит окно диалога **Project Options**, открываемое командой **Project** ► **Options**. Оно содержит семь вкладок: **Forms**, **Application**, **Compiler**, **Linker**, **Directories/Conditionals**, **Version Info** и **Packages**. Каждая из перечисленных вкладок позволяет сделать ряд настроек, влияющих на проект:

- ☐ с вкладкой **Forms** мы уже познакомились (см. раздел «Управление формами проекта»);
- ☐ на вкладке **Directories/Conditionals** задаются пути размещения откомпилированных файлов;

- ☐ вкладка **Version Info** дает возможность добавлять к проекту информацию о версии приложения;
 - ☐ вкладка **Packages** используется для настройки пакетов (см. главу 8).
- Назначение остальных вкладок описывается далее.

Вкладка Application

Вкладка **Application** (рис. 14.6) позволяет сделать 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.
- ☐ **Icon** — 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**.

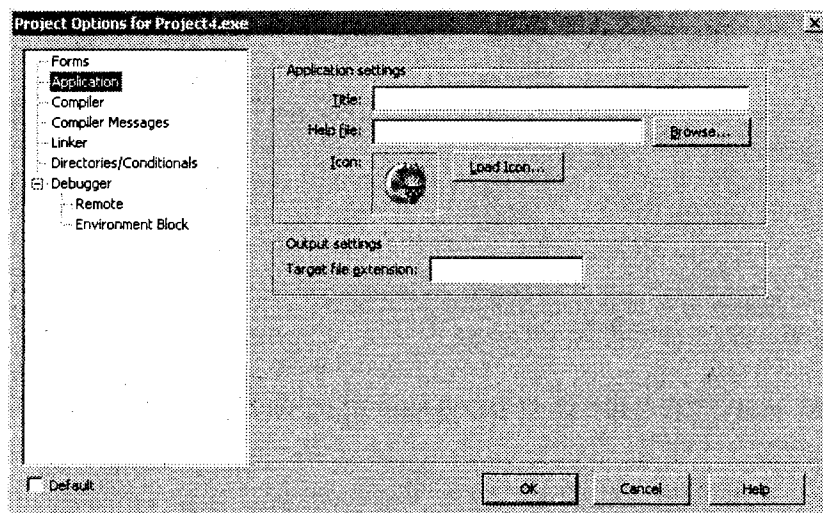


Рис. 14.6. Вкладка **Application** окна диалога **Project Options**

Вкладка Compiler

Элементы управления вкладки **Compiler** (рис. 14.7) позволяют оптимизировать работу компилятора (задаваемые параметры могут быть также изменены с помощью директив компилятора).

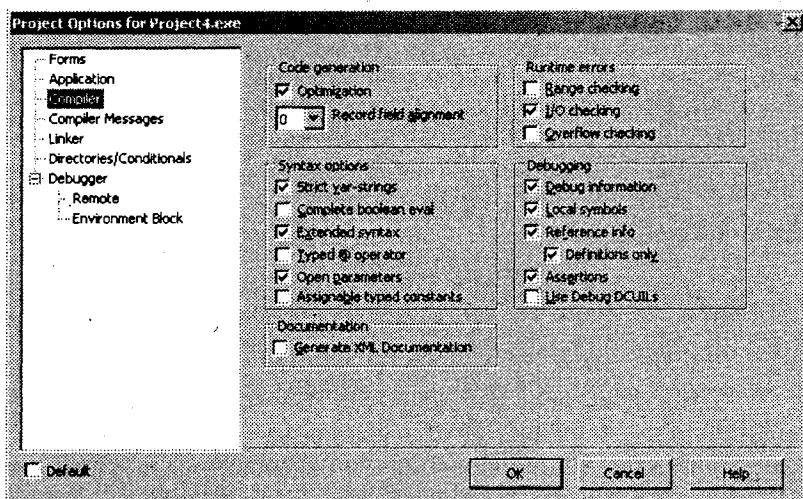


Рис. 14.7. Вкладка Compiler окна диалога Project Options

- ☐ Optimization — включение режима оптимизации генерируемого компилятором кода (директива `{S0}`).
- ☐ Aligned record fields — выравнивание данных по двойному слову (директива `{SA}`). Установка данного флажка приводит к повышению скорости выполнения программы при одновременном увеличении ее объема.
- ☐ Pentium-save FDIV — генерация кода, программно исправляющего ошибку вычисления, допускаемую первыми процессорами Pentium (директива `{SU}`).
- ☐ Range checking — проверка нахождения значения переменной в допустимом диапазоне (директива `{SR}`).
- ☐ I/O checking — проверка наличия ошибок при выполнении операций ввода-вывода (директива `{SI}`).
- ☐ Overflow checking — диагностика ошибок переполнения при выполнении целочисленных операций (директива `{SQ}`).
- ☐ Strict var-strings — проверка соответствия формальных и фактических параметров строкового типа (директива `{SV}`). Данный флажок имеет смысл устанавливать только в случае установки флажка Open parameters.
- ☐ Complete boolean eval — полное вычисление логического выражения, даже если результат известен заранее (директива `{SB}`).
- ☐ Extended syntax — разрешение вызова функции как процедуры, то есть без присваивания возвращаемого значения, которое игнорируется (директива `{SX}`).
- ☐ Typed @ operator — определение, какой указатель, типизированный или нет, возвращается оператором @ (директива `{ST}`).
- ☐ Open parameters — разрешение на использование в качестве параметров «открытых строк». То есть длина строки, передаваемой в качестве строкового параметра, может отличаться от длины, заданной при объявлении формального параметра (директива `{SP}`).

ПРИМЕЧАНИЕ

Директивы компилятора помещаются в текст программы. Они заключаются в фигурные скобки и состоят из символа \$ и следующей за ним буквы латинского алфавита (идентифицирующей назначаемый режим), за которой следует либо знак + (включает режим), либо знак – (выключает режим). Директивы компилятора имеют более высокий приоритет, чем установки IDE.

Вкладка Linker

Вкладка Linker служит для настройки параметров, влияющих на компоновку проекта (рис. 14.8):

- ☐ Generate console application — создание консольного приложения;
- ☐ Include remote debug symbols — этот флажок следует установить, если необходима удаленная отладка приложения;
- ☐ Generate .PDB debug info file — при компиляции формируется pdb-файл, в который записывается информация, в дальнейшем используемая при запуске приложения;
- ☐ Generate .DRCIL file — генерируется drcil-файл, в котором хранятся переменные строкового типа.
- ☐ На этой же вкладке указывается минимальный и максимальный размеры стека. В поле EXE Description можно указать комментарии к исполняемому файлу — до 255 символов.

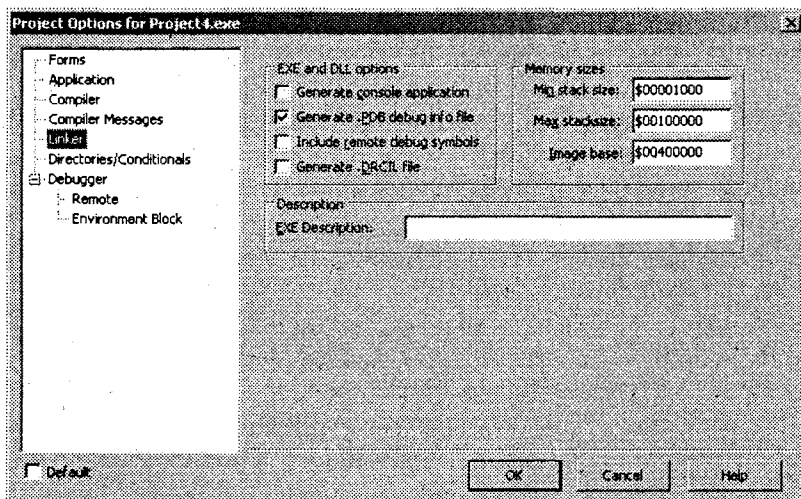


Рис. 14.8. Вкладка Linker окна диалога Project Options

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 отладчика IDE Delphi.

При попытке запустить неоткомпилированный проект компиляция будет выполняться 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.

- ☐ **Compile (Ctrl + F9)** — компилирует файлы, в которые были внесены изменения после выполнения последней компиляции. Затем выполняется компоновка приложения, в результате формируется либо исполняемая программа, либо динамически связываемая библиотека. Для группы проектов выполняется компиляция активного проекта.
- ☐ **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.
- ☐ **Add to Respository** — добавляет проект в хранилище ваших «удачных решений». Код, помещенный в хранилище, можно использовать при разработке других приложений.
- ☐ **Compile All Projects** — компиляция всех проектов, входящих в текущую группу, в том порядке, в котором они перечисляются в окне менеджера проектов. При выполнении данной команды компилируются только те файлы, которые изменялись со времени последней компиляции.
- ☐ **Build All Projects** — эта команда аналогична предыдущей, 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 используются для отладки приложения.

- ☐ **Run (F9)** — запускает текущий проект. Если проект не откомпилирован (или в проект вносились изменения после последней компиляции), то перед запуском автоматически проводится компиляция.
- ☐ **Step Over (F8)** и **Trace Into (F7)** — эти команды 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.
- ☐ **Trace to Next Source Line (Shift + F7)** — обеспечивает пошаговую отладку подпрограмм косвенного вызова.
- ☐ **Run to Cursor (F4)** — запускает программу и выполняет ее до строки, на которой расположен текстовый курсор в редакторе кода.

ГЛАВА 15 Коллективная разработка приложений

Создание крупных информационных систем требует согласованной работы целой группы программистов. Несколько лет назад проблемы организации взаимодействия отдельных разработчиков при подготовке крупных проектов были актуальными в основном для больших фирм-производителей программного обеспечения. Однако с появлением и развитием средств быстрой разработки приложений (RAD) 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 в рамках крупного проекта стали актуальными и для небольших рабочих групп. Это нашло свое отражение на рынке программного обеспечения. Включение в программные средства быстрой разработки приложений эффективных механизмов, обеспечивающих поддержку коллективной разработки, становится одним из факторов, повышающих конкурентоспособность данного программного продукта.

Системы контроля версий

Рассмотрим спектр задач, решаемых системами коллективной разработки приложений. Основной из них является обеспечение управляемости и контролируемости процессов разработки и сопровождения приложения. Для этого необходимо 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 (исполняемых файлов, библиотек DLL, элементов ActiveX, объектных модулей, документов), необходимых для компиляции и запуска приложения.

Обе указанные выше функции реализуются с помощью так называемых *систем контроля версий проектов* (Project Version Control Systems, PVCS). 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. Системы PVCS должны обеспечивать:

- ☐ идентификацию состояния как отдельных компонентов, так и проекта в целом;
- ☐ контроль за 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 объектов, помещенных в PVCS-хранилище, присваиваются идентификаторы версий самого объекта и проекта в целом. Это позволяет определить, какие именно файлы должны быть использованы для сборки заданной версии приложения.

Хранение файлов и контроль за их изменением

Хранилища объектов в системах PVCS могут организовываться на базе самых разных технологических решений, вплоть до применения специализированных баз данных. Допускается также использование в рамках одной системы 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 того, 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 контроля версий обязательно должна обеспечивать *блокировку*. Блокировка преследует две основные цели.

- ☐ *Обеспечение централизованного управления файлами проекта.* В этом случае задачей блокировки является устранение возможности случайной или наме-

ренной 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, включая применение паролей для снятия блокировки, шифрование и некоторые другие.

- *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

Мы рассмотрели основные функции систем PVCS. Теперь приведем последовательность операций, выполняемых при их использовании.

1. Ввод исходной информации о структуре проекта и его составляющих. Создание первой версии проекта в PVCS-хранилище.
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.
3. Предоставление отдельных составляющих проекта для изменения с учетом прав доступа и возможности блокировки разных версий объекта до момента помещения модифицированного объекта в хранилище.
4. Занесение в PVCS-хранилище измененных (или вновь созданных) составляющих проекта с присвоением номеров версий как самим составляющим, так и проекту в целом.
5. Предоставление всех составляющих проекта заданной версии для компиляции либо всего проекта, либо отдельной его составляющей.

Существует множество инструментов, предназначенных для контроля версий проектов. Наиболее популярны StarTeam, TeamSource, Microsoft Visual Source Safe QSC Team Conerence.

В качестве примера рассмотрим систему контроля версий TeamSource фирмы Borland, получившую широкое распространение благодаря тому, что она входила в поставку Delphi версий 5–7.

Программа TeamSource

Хотя программа 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 задействовать хранилище и контроллер версий системы контроля версий Merant за счет подключения специального расширения. Также имеется возможность создания собственного расширения для управления хранилищем версий, например, для использования базы данных в качестве такого хранилища. Поскольку программа TeamSource поставляется вместе с исходными текстами, написание расширений не представляет собой сверхсложной задачи.

Вызов TeamSource из главного окна среды разработки IDE Delphi выполняется командой Tools ► TeamSource (для Delphi 7 и 8 предусмотрена интеграция с системой StarTeam фирмы Borland).

Структура программы TeamSource

Функционирование программы TeamSource основано на использовании подключаемых модулей (plug-ins), разрабатываемых на основе расширения интерфейса API программы TeamSource. 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, запись комментариев к файлам и проектам, а также ряд других операций. Контроллеры располагаются в подключаемых модулях, 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 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 (bookmarks). Установленная закладка отмечает текущую версию всех составляющих проекта.

Использование закладок в значительной степени упрощает управление файлами при сборке проекта, а также при указании текущей версии проекта. При необходимости закладку можно снабдить комментариями.

Хранилище TeamSource

Как уже отмечалось, хранилище TeamSource организовано по файловому принципу. Для каждого проекта выделяется каталог, называемый корневым (root), в котором создается структура вложенных каталогов и файлов, соответствующая файлам и каталогам, включенным в описание проекта. Ниже описана структура файлов и каталогов, которая создается для каждого корневого каталога изначально.

❑ Archives — каталог, в котором содержатся версии файлов проекта. Файлы хранятся в архивированном виде в формате ZLib. В каталоге представлены все версии каждого из файлов проекта. Имена присваиваются файлам по следующему принципу: к имени исходного файла (включая расширение) добавляется расширение z (например, файл project.dpr получит имя project.dpr.z). Помимо 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, называемых файлами истории, 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 единственный 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 программы TeamSource производится отдельно от установки Delphi, возможна интеграция TeamSource и Delphi 8. В этом случае команды системы контроля версий становятся доступными в меню Team главного меню Delphi.

При первом запуске программы TeamSource открывается окно диалога Welcome to TeamSource, в котором требуется ввести ряд параметров, необходимых для идентификации пользователя (рис. 15.1):

❑ имя пользователя, указанное в поле Your user name, является сетевым именем компьютера, на котором установлена программа TeamSource, и в окне Welcome to TeamSource изменено быть не может;

☐ в поле Your full name вводится полное имя пользователя, под которым он работает над проектами в программе TeamSource;

☐ в поле Your email address указывается адрес электронной почты 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.

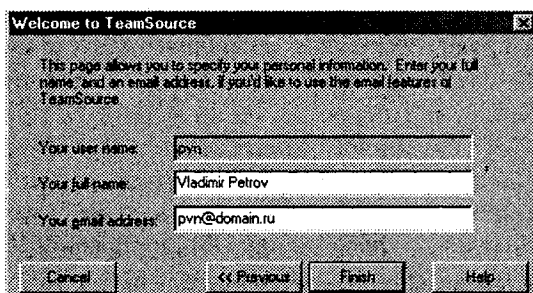


Рис. 15.1. Окно диалога Welcome to TeamSource

Настройка параметров программы

После заполнения полей окна диалога Welcome to TeamSource открывается главное окно программы TeamSource (рис. 15.2). Внешне оно очень похоже на главное окно программы Microsoft Outlook.

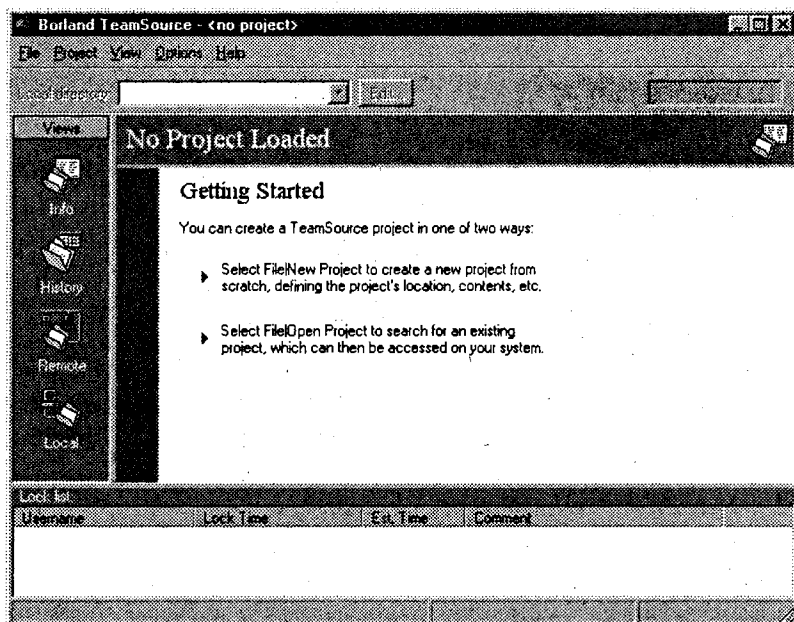
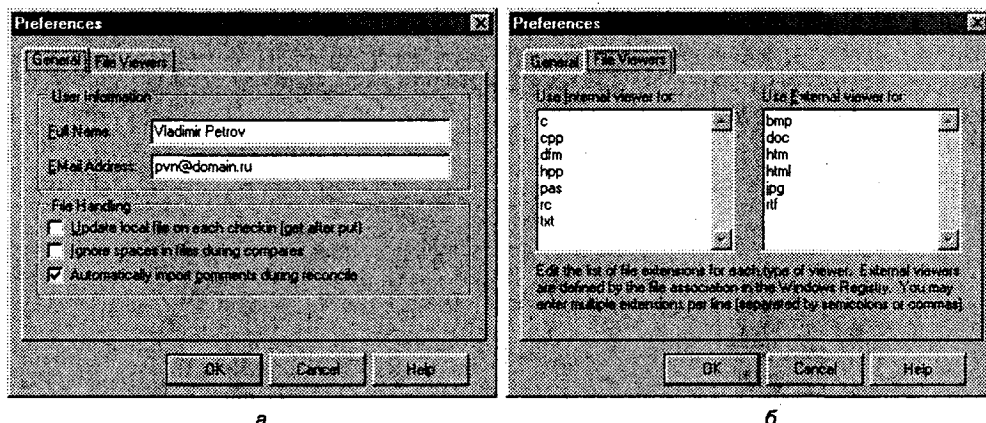


Рис. 15.2. Главное окно программы TeamSource

В строке меню TeamSource всего пять меню.

- ☐ Меню File включает стандартный набор команд:
 - New Project — создание нового проекта;
 - Open Project — открытие ранее созданного проекта;
 - Close Project — закрытие проекта;
 - Exit — выход из программы.
- ☐ Команды меню Project становятся доступными только в том случае, если открыт какой-либо проект. Поскольку сразу после запуска ни один проект не открыт, то все команды данного меню пользователю недоступны, и их мы рассмотрим позднее.
- ☐ 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 параметры программы TeamSource.
- ☐ Меню Help обеспечивает доступ к справочной системе.

Перед тем как создавать проект, познакомимся с основными параметрами программы, которые настраиваются с помощью команды Options ► Preferences. При выборе этой команды открывается окно диалога Preferences с двумя вкладками. На вкладке General задаются параметры пользователя программы TeamSource: имя пользователя и адрес электронной почты (которые задавались при первом запуске программы), а также параметры, характеризующие режим работы программы (рис. 15.3, а).



а

б

Рис. 15.3. Вкладки окна диалога Preferences

Группа флажков File Handling позволяет настроить режим работы программы.

- ☐ Update local file on each checkin (get after put) — при установке этого флажка обновление локальных копий файлов проекта будет производиться при каждой

записи в хранилище. Этот режим 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 операции Check-In, версию и т. п.

- ❑ Ignore spaces in files during compares — этот флажок позволяет игнорировать пробелы в начале и в конце строк при сравнении текстовых файлов. Благодаря этому можно избежать выделения подобных строк как измененных при сравнении версий.
- ❑ Automatically import comments during reconcile — этот флажок позволяет автоматически импортировать комментарии из окна Recommended changes при выполнении операции Check-In.

На вкладке File Viewers окна диалога Preferences указываются средства просмотра файлов различных типов (рис. 15.3, б). По умолчанию файлы с расширениями с, cpp, dfm, hpp, pas, rc и txt располагаются в списке Use Internal viewers for. Это говорит о том, что они открываются с использованием внутренних средств программы TeamSource. В список Use External viewers for помещаются файлы, содержимое которых отображается средствами внешних программ, ассоциированных с расширением файла.

ПРИМЕЧАНИЕ

При редактировании обоих списков на этой вкладке 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 создать новый проект или импортировать данные из уже существующего проекта (рис. 15.4). Поскольку мы создаем новый проект, то следует оставить установленный по умолчанию переключатель Create new project from scratch.

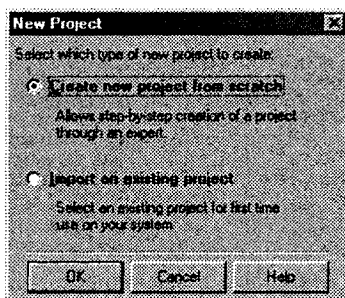


Рис. 15.4. Окно диалога New Project

После щелчка на кнопке OK в окне диалога New Project запускается мастер создания проекта. Создание проекта выполняется за семь шагов.

1. На первом шаге требуется задать (рис. 15.5):

- имя проекта (это имя будет затем использоваться во всех операциях TeamSource);
- имя файла проекта (без указания пути);
- контроллер версий (выбирается в списке, в котором отображаются все доступные контроллеры, подключенные к TeamSource).

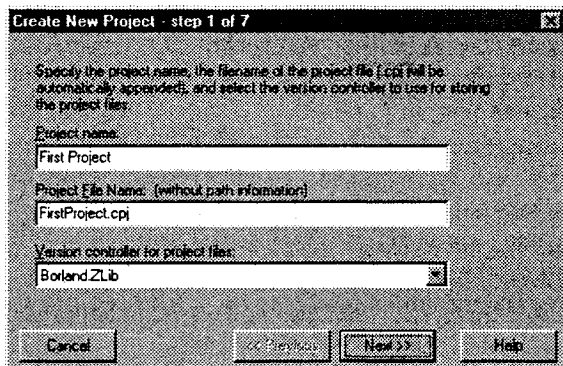


Рис. 15.5. Первый шаг создания проекта TeamSource

ПРИМЕЧАНИЕ

Указанное имя проекта в дальнейшем можно изменить только редактированием файла проекта.

2. На втором шаге указывается каталог, в котором будут храниться файлы проекта (рис. 15.6).

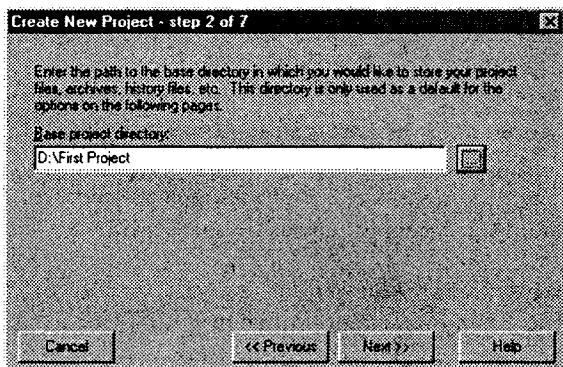


Рис. 15.6. Второй шаг создания проекта в TeamSource

ПРИМЕЧАНИЕ

Каталог проекта может располагаться не только на локальном, но и на удаленном компьютере.

3. На третьем шаге (рис. 15.7) задаются каталоги для хранилища версий (Archives), файлов истории (History) и блокировок (Locks). 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.

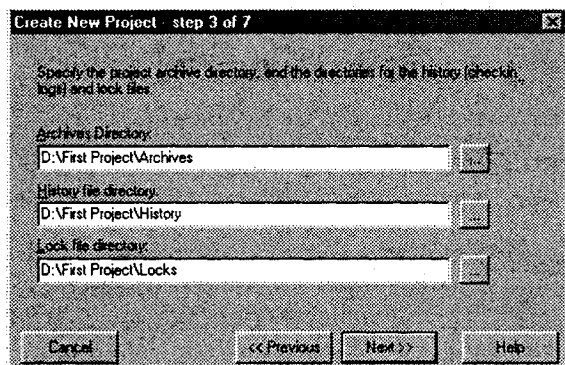


Рис. 15.7. Третий шаг создания проекта TeamSource

4. На следующем шаге задаются параметры создания резервной копии хранилища версий (рис. 15.8).

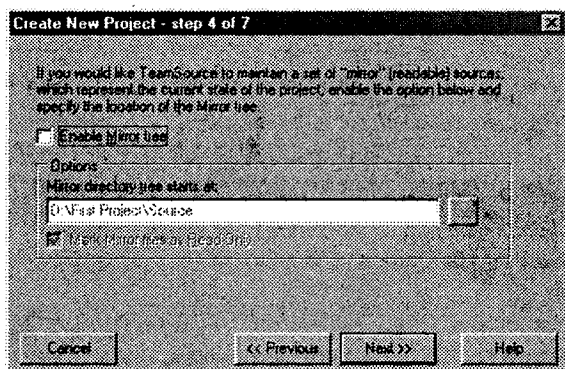


Рис. 15.8. Четвертый шаг создания проекта TeamSource

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 «только для чтения».

5. На пятом шаге задаются имена и местоположение на диске (или в сети) файлов истории и журнала (рис. 15.9). Кроме того, указывается SMTP-сервер, который будет использоваться для рассылки сообщений подсистемой оповещения.

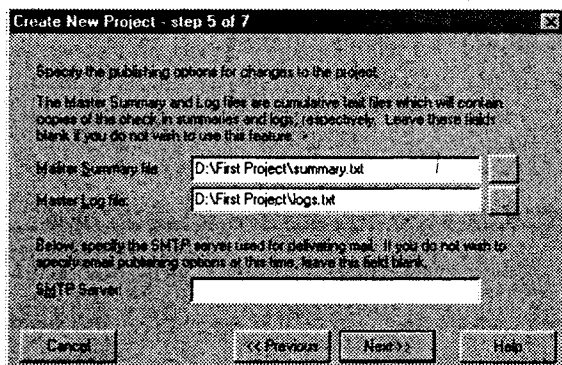


Рис. 15.9. Пятый шаг создания проекта TeamSource

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. 15.10). В противном случае (если SMTP-сервер не задан) будет произведен переход сразу к седьмому шагу.

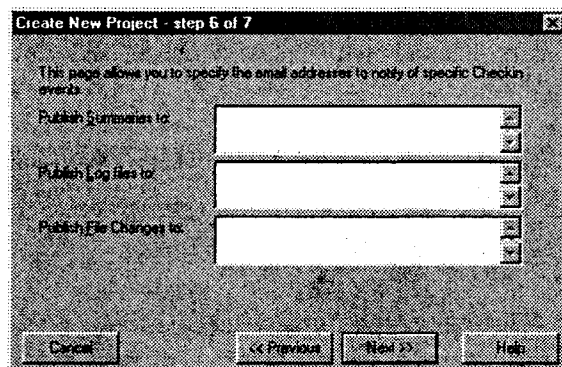


Рис. 15.10. Шестой шаг создания проекта TeamSource

Указанные здесь адреса электронной почты определяют, какую информацию будет получать каждый из участников проекта при изменениях в хранилище версий.

7. На седьмом шаге выводится окно, содержащее всю информацию о проекте, введенную на предыдущих шагах (рис. 15.11). При обнаружении ошибки можно вернуться назад, щелкнув на кнопке Previous, и внести необходимые коррективы.

После щелчка на кнопке Finish TeamSource запросит данные о локальных каталогах проекта и о типах файлов, обрабатываемых контроллером версий. Данная информация вводится с помощью специального мастера Content Wizard (рис. 15.12). Окно диалога Local Directory Wizard позволяет указать список типов файлов, обрабатываемых контроллером версий для каждого из локальных каталогов. Можно также удалить любой из каталогов, кроме самого верхнего.

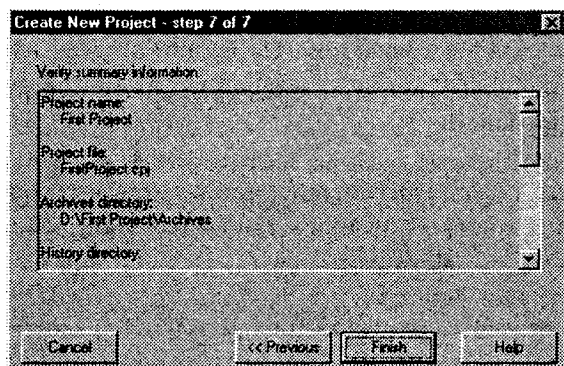


Рис. 15.11. Заключительный шаг создания проекта TeamSource

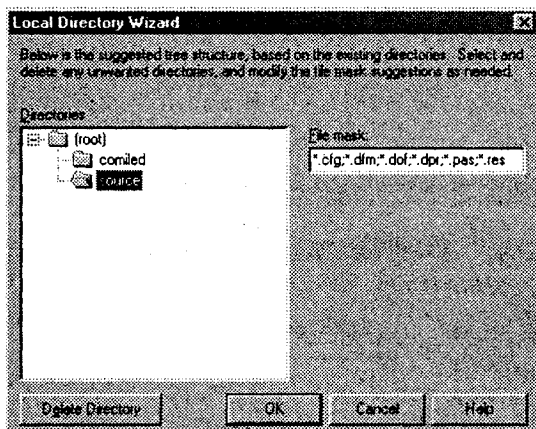


Рис. 15.12. Окно мастера настройки структуры локальных каталогов Local Directory Wizard

ПРИМЕЧАНИЕ

Вызов мастера Content Wizard может быть осуществлен уже после создания проекта с помощью команды Project ► Content Wizard. Доступ к данной команде возможен только при блокировке файлов проекта. Как производится установка и снятие блокировки, будет рассмотрено несколько позднее.

Настройка параметров проекта

Параметры проекта настраиваются с помощью окна диалога Project Options, открываемого командой Project ► Options. Данная команда доступна в 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 установить блокировку (о том, как это сделать, будет рассказано далее). После блокировки окно диалога Project Options становится доступным для настройки.

Окно диалога Project Options содержит четыре вкладки. Ниже перечислены элементы управления вкладки General (рис. 15.13).

- ☐ В поле Project name отображается имя текущего проекта. Данное поле недоступно для редактирования.
- ☐ В поле Version info file name указывается имя файла, содержащего информацию о версии проекта. Этот файл создается в корневом каталоге локального проекта автоматически при объединении в приложение всех составляющих проекта заданной версии (операция Pull).
- ☐ При установке флажка Detect new local directories программа TeamSource будет проверять, не содержит ли корневой каталог локального проекта новых вложенных каталогов, и в случае их обнаружения автоматически запускать мастер Content Wizard.
- ☐ При установке флажка Require summary comments программа 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 потребует ввода комментариев к каждому файлу.
- ☐ Текстовое поле Build numbers предназначено для задания исходного номера версии проекта.

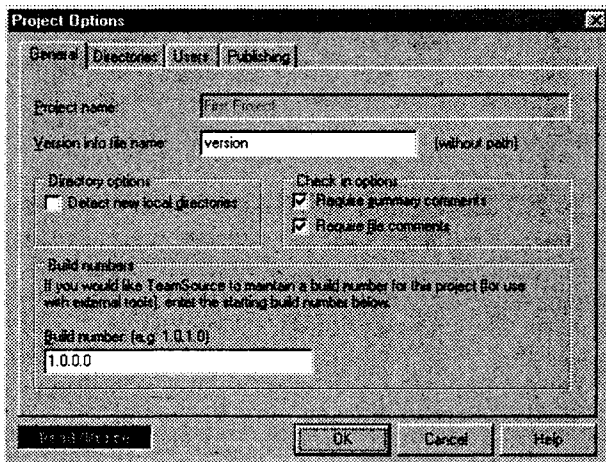


Рис. 15.13. Вкладка General окна диалога настройки параметров проекта

Вкладка Directories используется для задания путей к файлам проекта (рис. 15.14).

- ☐ В текстовом поле Archives directory выводится информация о местоположении файлов хранилища версий. Данное поле недоступно для редактирования.
- ☐ Поле History directory позволяет изменить каталог, содержащий файлы истории.
- ☐ В 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.

- ❑ Поле Mirror directory доступно для редактирования при установленном 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».

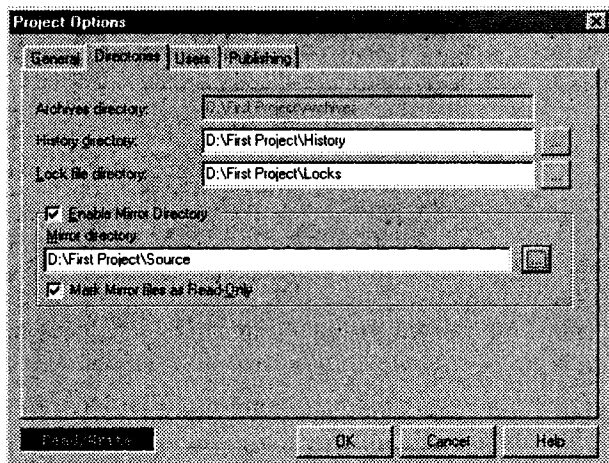


Рис. 15.14. Вкладка Directories окна диалога настройки параметров проекта

На вкладке Users указываются имена участников проекта и права их доступа к проекту (рис. 15.15). Список Authorized users содержит имя пользователя (столбец Username) и информацию о его правах доступа (столбец Access).

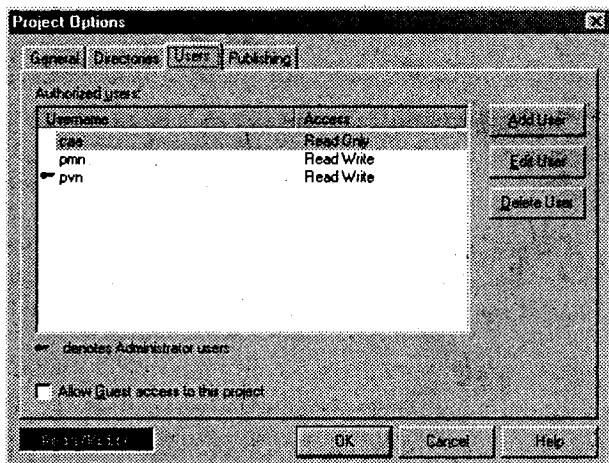


Рис. 15.15. Вкладка Users окна диалога настройки параметров проекта

Для добавления нового участника проекта следует щелкнуть на кнопке Add. При этом открывается окно диалога User Information (рис. 15.16). В поле Username этого

окна вводится имя нового участника проекта, а группа переключателей Access Rights позволяет задать права доступа к проекту:

- ☐ Read-Only — только чтение файлов проекта;
- ☐ Read-Write — возможность модификации файлов проекта.

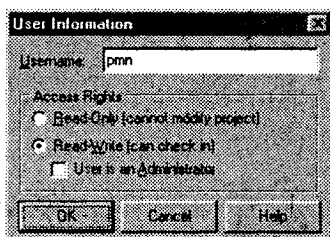


Рис. 15.16. Окно диалога User Information

Флажок User is an Administrator служит для наделения пользователя 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 «только для чтения».

На вкладке Publishing указываются адреса электронной почты, по которым производится рассылка различного рода информации (рис. 15.17):

- ☐ сводных отчетов об операциях с хранилищем версий;
- ☐ журналов операций с хранилищем версий;
- ☐ подробных отчетов об изменениях в отдельных файлах при выполнении операций Check-In.

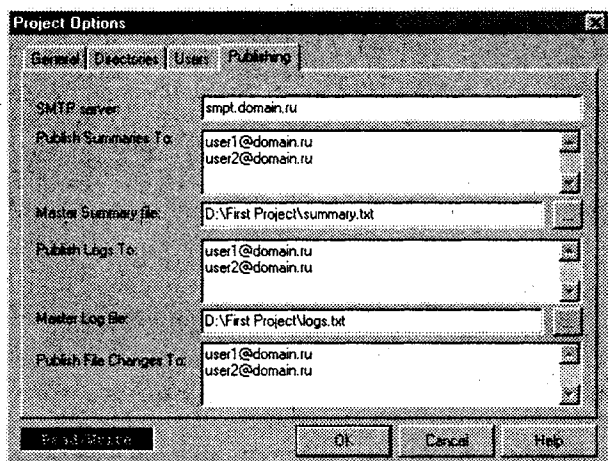


Рис. 15.17. Вкладка Publishing окна диалога настройки параметров проекта

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 Project ► Controllers. Однако обращаться к данному окну имеет смысл только в том случае, если используются расширения TeamSource, так как контроллер версий, входящий в поставку TeamSource, не имеет настраиваемых в окне Controller Options параметров.

Работа с проектом

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 котором в окне TeamSource отображается информация о текущем проекте (этот режим нами фактически уже рассмотрен, так как его возможности 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. 15.18).

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. 15.18, в локальном проекте обнаружены следующие изменения:

- ☐ создан новый файл Calculation.pas, отсутствующий в хранилище версии — рекомендуется занести его в хранилище, выполнив операцию Check-In;
- ☐ обнаружены изменения в файлах SQL.exe, SQL_main.dcu, SQL.dpr, SQL.cfg, SQL.dof, SQL_main.dfm, SQL_main.pas, которые имеют более новую версию или более позднее время создания, чем их аналоги в хранилище, — рекомендуется занести их в хранилище, выполнив операцию Check-In;
- ☐ файл SQLBDE.res удален из локальной копии проекта — рекомендуется 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. 15.18, 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, поэтому рекомендуется привести время создания файла в соответствие с версией в хранилище, выполнив операцию Touch;
- ☐ файл SQLBDE.dpr изменен, однако время его создания более раннее, чем у версии, находящейся в хранилище, то есть имеет место конфликт версий (такого рода конфликты не могут быть разрешены автоматически, поэтому рекомендуется исправить файл вручную, о чем говорит сообщение Correct by hand).

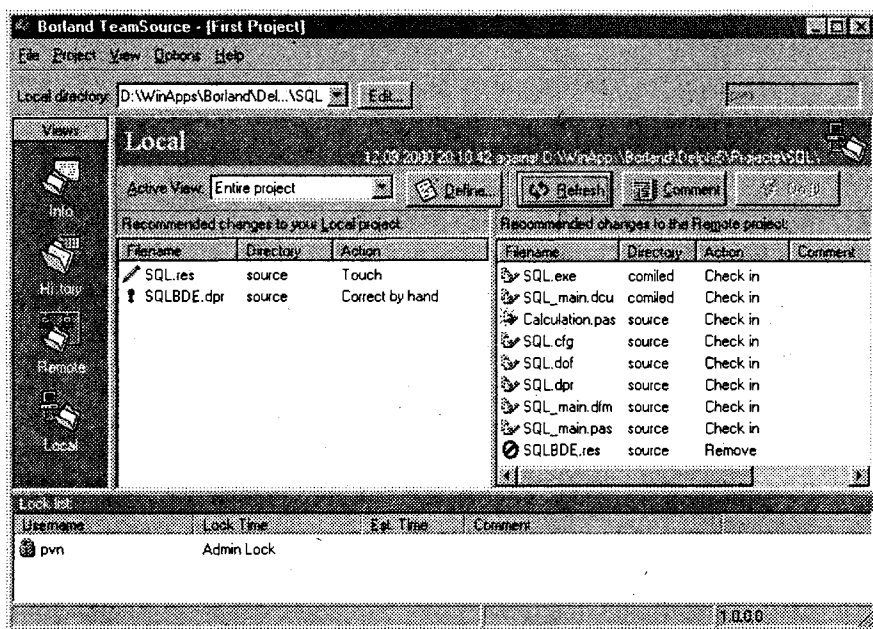


Рис. 15.18. Главное окно TeamSource в режиме Local

Для выполнения рекомендованных действий над файлом выделите его в списке и щелкните на кнопке Do It!.

Если после выделения одного или нескольких файлов щелкнуть на выделении правой кнопкой мыши, 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:

- ☐ первой всегда является 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, в том числе 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.

В дополнение к этим командам контекстное меню панели Recommended changes to your Local project содержит команду Change File Status, позволяющую 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. 15.19 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 минус (-). Новый вариант выделяется желтым цветом и помечается символом плюс (+). В нижней части окна Comparing выводится комментарий к измененному файлу.

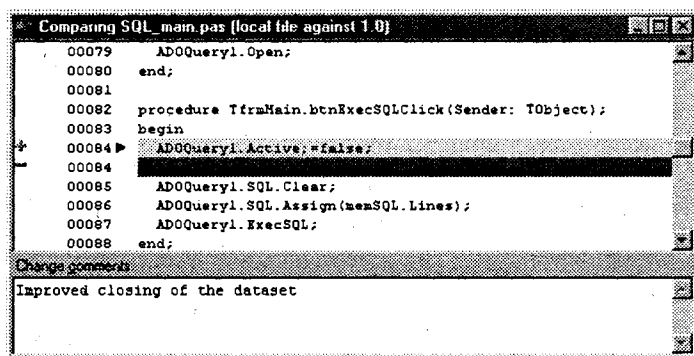


Рис. 15.19. Окно диалога Comparing

Работа с хранилищем версий

В режиме Remote в главном окне TeamSource отображается состав хранилища версий (рис. 15.20). 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.

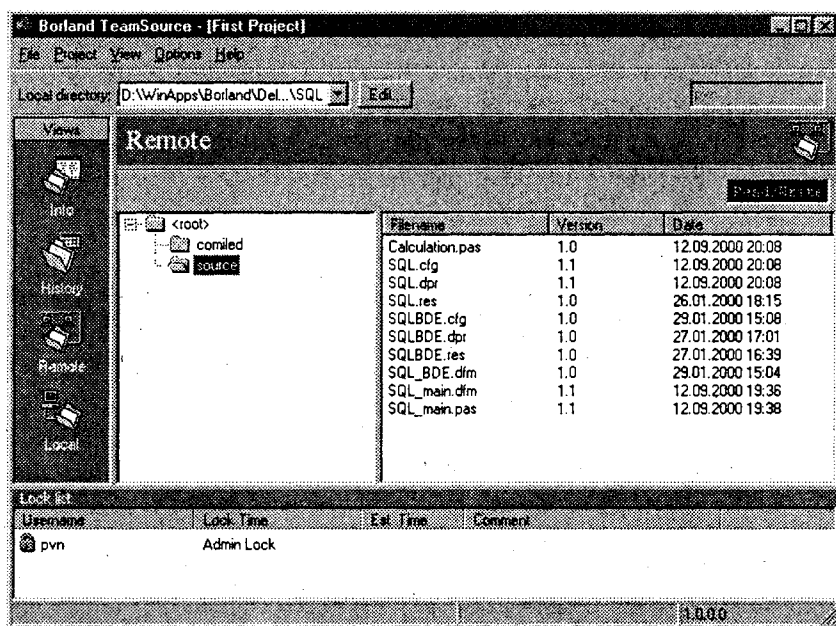


Рис. 15.20. Окно программы TeamSource в режиме Remote

При снятой блокировке проекта можно лишь просматривать содержимое хранилища версий в режиме «только для чтения». Поэтому в том случае, когда требуется внести изменения в хранилище (например, удалить лишние файлы), следует установить блокировку.

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. 15.21). Если перед выбором этой команды было выделено несколько файлов, то каждый из них открывается в собственном окне.
- ☐ View Any Revision — просмотр любой версии выделенного файла. При выборе этой команды открывается окно диалога *Select A Revision*, в котором предлагается выбрать версию файла среди всех, имеющих в хранилище (рис. 15.22). Выбранная версия файла отображается в окне просмотра, полностью аналогичном показанному на рис. 15.21.
- ☐ Save Revision As — сохранение выделенной версии файла поверх любой из версий, имеющих в хранилище. При выборе этой команды также открывается окно диалога *Select A Revision*, в котором указывается версия, поверх которой будет производиться запись.

- ☐ 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, в котором следует выбрать две сравниваемые версии (рис. 15.23). Результаты сравнения выводятся в окне, подобном показанному на рис. 15.19.

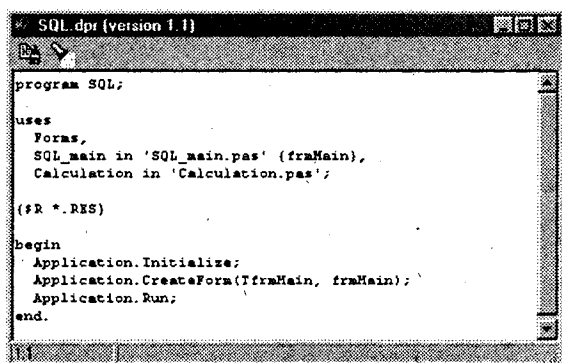


Рис. 15.21. Окно просмотра текстового файла проекта

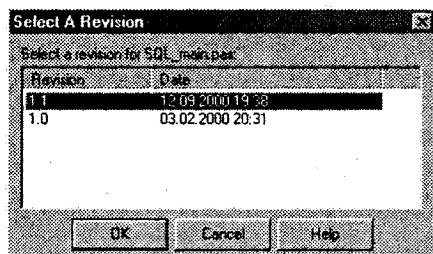


Рис. 15.22. Окно диалога Select A Revision

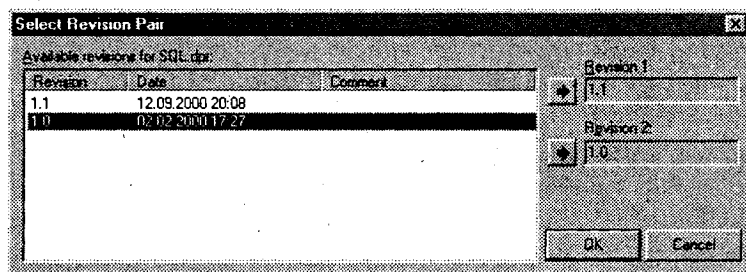


Рис. 15.23. Окно диалога Select Revision Pair

- ☐ Set Revision Number — присвоить выделенному файлу любой номер версии из тех, что имеются в хранилище. Для выбора версии открывается окно диалога Select A Revision (см. рис. 15.22).

- ☐ **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 **Select A Revision** (см. рис. 15.22), 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);
- ☐ **Properties** — изменить свойства выделенного каталога.

При выборе последней команды открывается окно диалога **Directory Properties**, в котором можно изменить параметры выделенного каталога хранилища версий и режим обработки соответствующего ему локального каталога (рис. 15.24).

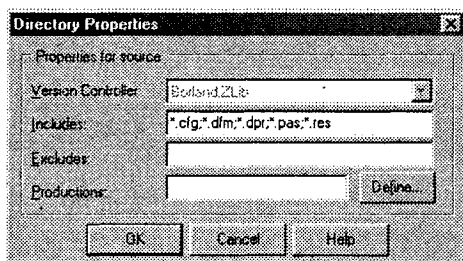


Рис. 15.24. Окно диалога **Directory Properties**

Ниже перечислены поля окна диалога **Directory Properties**.

- ☐ В поле **Version controller** отображается имя используемого контроллера версий. Это поле недоступно для редактирования.
- ☐ Поле **Includes** представляет собой список шаблонов файлов, заносимых в хранилище версий из локального каталога. Например, для Delphi типичными шаблонами для файлов с исходными текстами являются ***.dpr**, ***.pas**, ***.dfm**.
- ☐ Поле **Excludes** — это 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.
- ☐ С помощью поля **Productions** и кнопки **Define** можно 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, открывающемся при щелчке на кнопке **Define**.

Работа в режиме History

В режиме History отображаются все изменения, внесенные в проект, с момента его создания. Главное окно программы TeamSource в данном режиме делится на две панели (рис. 15.25). В левой панели отображаются дата и время внесения изменения в хранилище версий, а также имя пользователя, внесшего эти изменения. В правой панели указывается, какие файлы проекта были изменены.

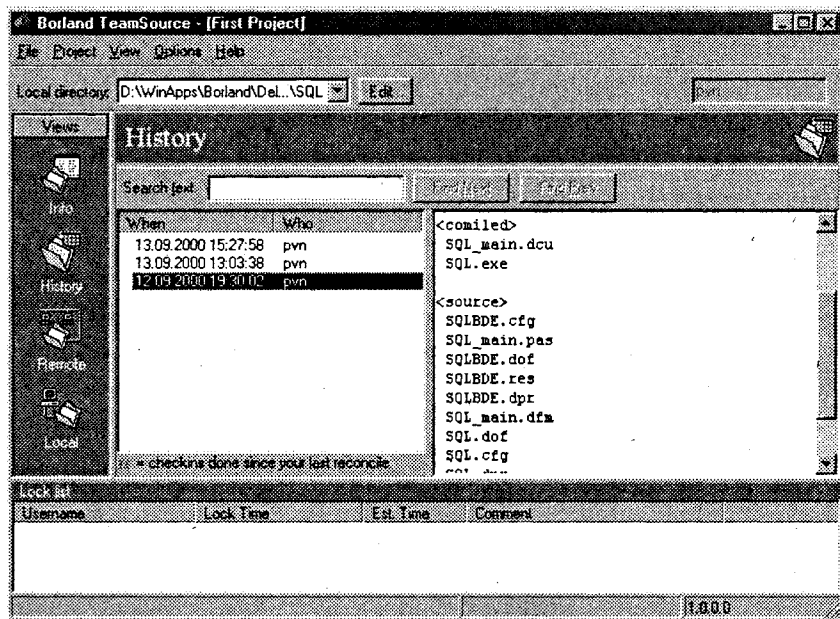


Рис. 15.25. Окно программы TeamSource в режиме History

Блокировка проекта

При работе с проектом многие операции выполняются только в случае блокировки. Уже отмечалось, что настройку параметров проекта и модификацию хранилища версий можно выполнять только при блокировке проекта.

Блокировка проекта может быть выполнена любым пользователем, обладающим соответствующими правами доступа к проекту. Для блокировки используется команда **Project ► Request Lock** главного меню TeamSource. При выборе данной команды открывается окно диалога **Lock Information**, в котором задаются параметры блокировки (рис. 15.26):

- ☐ в поле **Lock Comment** вводится комментарий;
- ☐ счетчик **Estimated time you will need the lock** позволяет задать время блокировки в минутах;
- ☐ флажок **Lock as Administrator Lock** говорит о том, что блокировка устанавливается администратором (в этом случае блокировка не ограничена по времени).

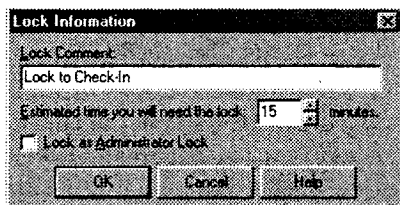


Рис. 15.26. Окно диалога Lock Information

После установления 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 версий проекта. Обычно эта пометка используется при сборке проекта (операция Pull).

Управление закладками осуществляется с помощью окна диалога Bookmarks (рис. 15.27), открывающегося после выбора команды Project ► Bookmarks главного меню TeamSource.

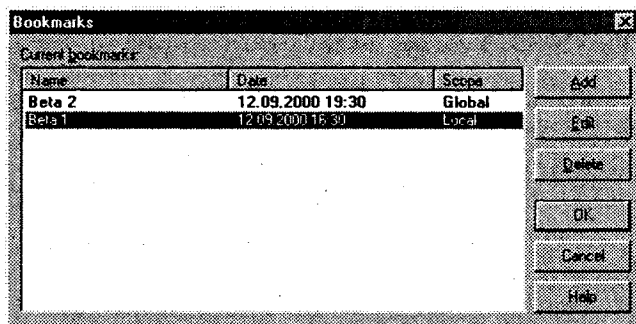


Рис. 15.27. Окно управления закладками проекта

Для добавления новой закладки следует щелкнуть на кнопке Add. Редактирование ранее созданной закладки выполняется щелчком на кнопке Edit. После выполнения любого из этих действий открывается окно диалога Bookmark Properties, в котором задаются (или изменяются) свойства закладки (рис. 15.28):

- ☐ название закладки — текстовое поле Name;
- ☐ дата и время версии проекта, для которой создается закладка — список Date;

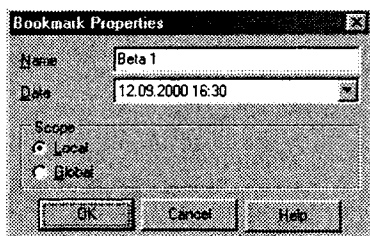


Рис. 15.28. Окно диалога Bookmark Properties

- ☐ тип закладки (глобальная или локальная) — переключатели группы Scope (Local или Global).

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 компиляции. Это могут быть как файлы, соответствующие текущей версии проекта, так и файлы любой из более ранних версий, 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. 15.29).

- ☐ В списке Bookmarks выбирается закладка, соответствующая версии запрашиваемых файлов. Если производится сборка текущей версии, то в данном списке следует оставить предлагаемый по умолчанию вариант None.
- ☐ При установке флажка Fast Pull будут копироваться только те файлы, которые отсутствуют в локальных копиях (или локальные версии которых старше запрашиваемой).

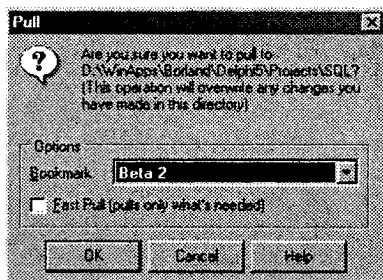


Рис. 15.29. Окно диалога Pull

ГЛАВА 16 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, 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 и открывающийся при выборе соответствующей команды меню или нажатии клавиши F1. Однако в большинстве случаев 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 *контекстная справка*, позволяющая быстро получать информацию о любом окне программы, 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 на каком-либо элементе 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 пользователя 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 работе приложения 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;
- ☐ контекстную справку, вызываемую нажатием клавиши F1 (в большинстве случаев содержимое контекстной 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) создаются средствами Delphi.

Вопросы разработки файлов справки и интеграции их в приложения Delphi будут рассмотрены далее, 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 данные компоненты обладают рядом необходимых для этого свойств. 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. 16.1.

Текст подсказки, задаваемый в свойстве Hint, может состоять из двух частей, которые разделяются между собой символом вертикальной черты (|). Например, текст подсказки может иметь следующий вид:

Печать | Печать текущего документа

Таблица 16.1. 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

В этом случае во всплывающей подсказке отображается только первая часть текста. Вторая часть подсказки (обычно развернутая) требуется 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. 16.1. Кроме того, рядом важных свойств, определяющих параметры отображения подсказки, обладает невизуальный объект Application (табл. 16.2).

Таблица 16.2. Свойства объекта Application

Свойство	Тип	Описание
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:

```
procedure (var HintStr: string; var CanShow: Boolean;  
var HintInfo: THintInfo)
```

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 обработчике 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 (true) или нескольких панелей (false), на которые выводится какая-либо информация.

В том случае, когда значение свойства SimplePanel равно true, в строке состояния можно выводить только текстовые сообщения. Для этого в строке 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. 16.1). Для его открытия можно воспользоваться одним из следующих способов:

- ❑ выполнить двойной щелчок на компоненте TStatusBar, размещенном на форме;
- ❑ выбрать команду Panels Editor в контекстном меню компонента TStatusBar;
- ❑ щелкнуть на кнопке с многоточием в поле ввода свойства Panels компонента TStatusBar инспектора объектов.

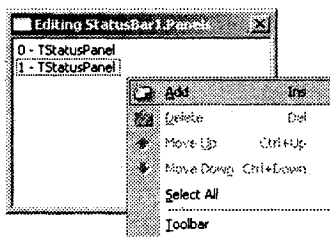


Рис. 16.1. Окно редактора панелей и контекстное меню компонента TStatusBar

Для добавления, удаления и изменения местоположения панелей используются команды контекстного меню, которые не требуют пояснений. Каждая панель пред-

ставляет 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` — информация выводится в любой форме.

При использовании стиля панели `psOwnerDraw` разработчик программы должен сам позаботиться о прорисовке прямоугольной области, занимаемой панелью строки состояния. Для этого следует задействовать 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. Обновленная версия данной утилиты, называемая WinHelp 4, состоит из одного файла с именем `winhelp.exe`.

В настоящее время корпорация 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 широко используются, и практически все средства разработки приложений обеспечивают интеграцию программ с файлами справки WinHelp.

Основные элементы справочной системы WinHelp 4

Файл справки в формате WinHelp состоит из нескольких элементов, которые создаются в ходе 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. Этот фрагмент 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, как правило, имеет заголовок, отображаемый в верхней части окна просмотра, строковый идентификатор, числовой идентификатор, набор ключевых слов, по которым можно найти раздел, а также перекрестные ссылки на другие разделы.

Перекрестные ссылки

Отдельные темы справочной системы связываются между собой при помощи ссылок. Для пользователя ссылки представляются в виде кнопок или выделенного цветом и подчеркиванием текста. При разработке справочной системы можно создать ссылку на другую тему или на временное окно, которое 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 spf, имеющий такое же имя, как и файл справки.

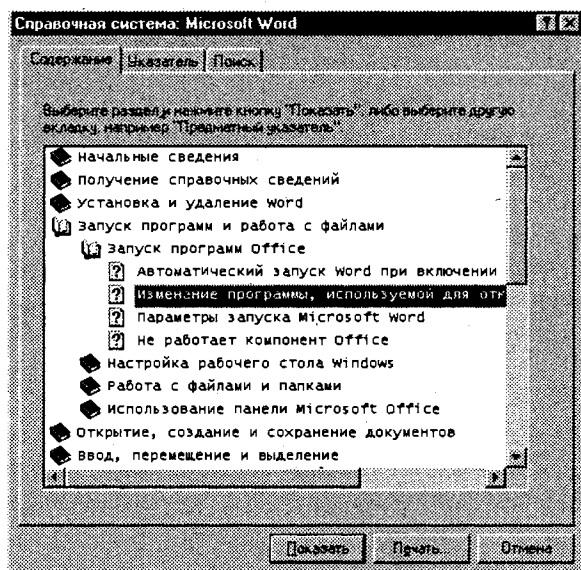


Рис. 16.2. Окно справочной системы

При запуске файла справки в системе WinHelp 4 открывается специальное окно с тремя вкладками (рис. 16.2).

- ❑ На вкладке **Содержание** 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.
- ❑ Вкладка **Указатель** — это предметный указатель справочной системы, обеспечивающий поиск по ключевым словам. Предметный указатель составляется разработчиком справочной системы путем связывания ключевых слов с идентификаторами соответствующих тем.
- ❑ С помощью вкладки **Поиск** 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.

- ❑ Текст выбранной темы отображается в основном окне.
- ❑ Вторичное окно позволяет вывести дополнительную информацию, не покидая текущий раздел справочной системы, 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 служит для вывода пояснений к терминам. Такие окна также часто используются при организации контекстной справки приложения для краткого описания элемента на экране.

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 перехода, вызова всплывающего меню или макрокоманды.

В WinHelp 4 определено более 30 макрокоманд. Основные из них приведены в табл. 16.3.

Таблица 16.3. Основные макрокоманды WinHelp 4

Макрокоманда	Описание
KLink	Переход к теме, соответствующей обычным ключевым словам. Если таких тем несколько, открывается окно найденных разделов, и пользователь может выбрать тему для просмотра
ALink	Переход к теме, соответствующим скрытым ключевым словам. Если таких тем несколько, то открывается окно найденных разделов, и пользователь может выбрать тему для просмотра
Find	Открытие окна справочной системы на вкладке Поиск
ExecFile	Запуск указанной программы
ShellExecute	Запуск указанной программы, печать файла или открытие файла с помощью связанной с ним программы
ShortCut	Запуск указанной программы, если она не запущена, или ее активизация с передачей ей сообщения WM_COMMAND, если программа запущена
ControlPanel	Запуск приложения панели управления
MPrintID	Печать темы
BrowseButtons	Добавление к окну справки кнопки просмотра вперед (>>) и назад (<<)
CreateButton	Создание в окне справки кнопки с указанным именем, связанной с указанным макросом
InsertMenu	Добавление нового пункта к главному меню окна справки
AppendMenu	Добавление нового пункта к указанному подменю

Создание файла справки

Существует довольно много как коммерческих, так и бесплатных средств разработки файлов справки. Мы рассмотрим процесс создания файла справки с помощью программы Microsoft Help Workshop.

Для создания тем справок, помимо программы Microsoft Help Workshop, необходим текстовый редактор, способный сохранять тексты в формате RTF. Лучше всего, вероятно, использовать редактор MS Word, 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. 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, определяющих 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 ► Сноска в глав-

ном меню. При этом откроется окно диалога, в котором задаются параметры сноски (рис. 16.3). Изменять в этом окне следует только символ сноски.

ВНИМАНИЕ

Все атрибуты темы следует помещать в самом ее начале, перед текстом темы.

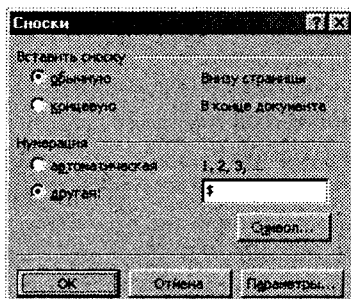


Рис. 16.3. Окно диалога задания параметров сноски

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.

- ☐ Сноска # задает идентификатор темы, которым является уникальная текстовая строка, используемая в дальнейшем для ссылки на раздел. Это единственный обязательный атрибут. Строка может задаваться как латинскими, так и русскими буквами и состоять из нескольких слов.
- ☐ Сноска \$ задает заголовок темы.
- ☐ Сноска К (прописная латинская буква) задает ключевые слова, по которым производится поиск в предметном указателе справки (вкладка *Указатель* окна справки). Текст сноски может состоять из нескольких слов, разделенных пробелами. С помощью одной сноски можно задать 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.
- ☐ Сноска А (прописная латинская буква) задает ключевые слова, которые не включаются в предметный указатель, а используются только для переходов с помощью макроса ALink.
- ☐ Сноска + позволяет задать номер, определяющий положение темы в последовательности просмотра, производимого с помощью кнопок >> и <<. Значения этого атрибута можно не указывать, в этом случае номера будут устанавливаться автоматически в соответствии с последовательностью тем в RTF-файле.

- ☐ Сноска ! позволяет указать один или несколько макросов, запускаемых перед отображением темы.
- ☐ Сноска * служит для задания директивы компилятора и позволяет включать или не включать тему в справочную систему в зависимости от параметров компиляции.
- ☐ Сноска > определяет идентификатор окна, в котором будет отображаться тема.

Текст темы

Текст темы должен следовать после атрибутов. При написании текста доступны все возможности редактора MS Word — внедрять в текст формулы, рисунки из графических файлов или буфера обмена, использовать различные варианты форматирования текста и разные шрифты.

ПРИМЕЧАНИЕ

Векторные картинки, нарисованные непосредственно в редакторе Word, в текст справки не включаются.

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 описанную ниже процедуру.

1. Выделите абзац, текст которого не должен прокручиваться.
2. Выберите в главном меню MS Word команду **Формат** ► **Абзац**. Откроется окно диалога **Абзац**.
3. Перейдите на вкладку **Положение на странице** и установите флажок **не отрывать от следующего** (рис. 16.4).

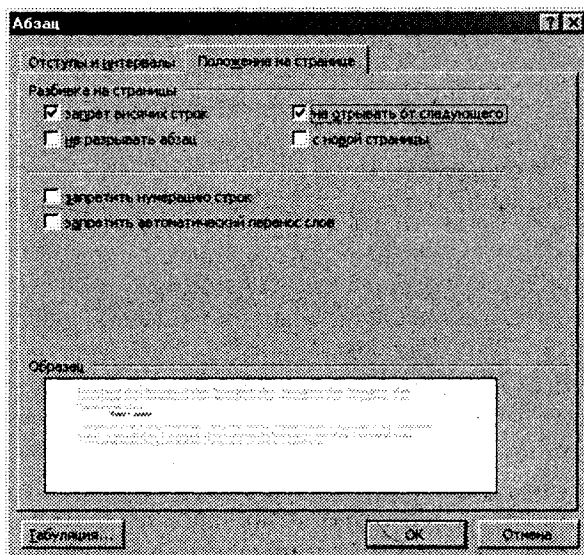


Рис. 16.4. Задание области текста, постоянно присутствующей на экране

Перекрестные ссылки

В текстах тем могут быть перекрестные ссылки, позволяющие переходить к другим темам. 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.

Сначала создадим ссылку, обеспечивающую *непосредственный переход* к нужной теме.

1. Выделите слово или сочетание слов, которое соответствует ссылке, и выберите команду **Формат** ▶ **Шрифт**.
2. В списке Подчеркивание открывшегося окна диалога Шрифт выберите вариант двойного подчеркивания (рис. 16.5).

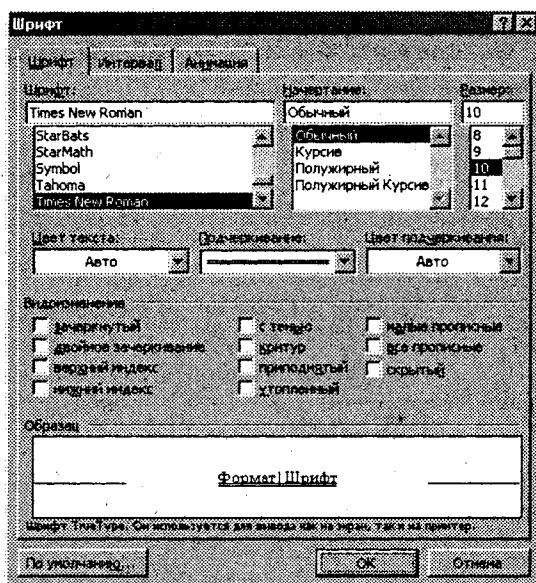


Рис. 16.5. Установка двойного подчеркивания

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**. Вновь откроется окно диалога **Шрифт**.

Установите в группе **Видоизменение флажков** **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 командой **Сервис** ► **Параметры**.

Для создания переходов по ключевым словам следует использовать макрос **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 сносок **K**, а **ALink** — по ключевым словам, заданным с помощью сносок **A**. Далее мы рассмотрим только макрос **Klink**, поскольку синтаксис макроса **ALink** аналогичен:

```
!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 для перехода.
- ☐ **Тип** определяет действия макроса при обнаружении одного или нескольких ключевых слов. Этот параметр может принимать одно или несколько (разделенных пробелами) значений, которые могут быть как символьными, так и числовыми:
 - **JUMP**, или **1**, — если найдена только одна тема, то на нее сразу производится переход;
 - **TITLE**, или **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 с этой темой;
 - **TEST**, или **4**, — возвращает значение, показывающее, нашлось хотя бы одно соответствие заданным ключевым словам или нет.
- ☐ **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 окно, заданное 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 щелчках на которых будет открываться окно с небольшим комментарием. Такое всплывающее окно не имеет области заголовка и полос прокрутки, однако текст в этом окне может содержать ссылки любого вида. Создадим ссылку, открывающую всплывающее окно.

1. Подготовьте тему с текстом, который необходимо вывести во всплывающем окне. Создание такой темы не имеет никаких особенностей.
2. В тексте темы, в которой вы организуете ссылку, открывающую всплывающее окно, выделите текст (он будет играть роль ссылки) и задайте для него одинарное подчеркивание.
3. Сразу за подчеркнутым текстом (без пробелов) укажите идентификатор темы, который должен быть показан во всплывающем окне.
4. Выделите идентификатор темы и с помощью окна диалога **Шрифт** сделайте его скрытым.

Тему справки, на которую указывает ссылка, можно также вывести во *вторичном* окне — окне, которое открывается поверх основного. Ссылка, выводящая тему в таком окне, 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 вторичном окне.
3. После идентификатора темы введите символ больше (>), а сразу за ним (без пробела) — идентификатор окна, в котором должна выводиться тема.
4. Выделите целиком идентификатор темы, символ больше (>) и идентификатор окна и в окне диалога **Шрифт** установите флажок **Скрытый**.

ПРИМЕЧАНИЕ

Вторичное окно, идентификатор которого указывается в описанной ссылке, должно быть создано в файле проекта справки. Более подробно вопрос создания окон будет рассмотрен далее.

Кнопки

В текст тем справочного файла можно включать кнопки, при щелчках на которых 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 щелчке на кнопке (макросы отделяются друг от друга двоеточием).

Изображения

В текст темы можно внедрять изображения, хранящиеся в файлах с расширениями bmp, dib, wmf, shg, mrb. 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 также просто поместить в текст, используя возможности редактора MS Word. Изображения, размещенные в тексте RTF-файла, могут быть ссылками точно так же, как и текст.

Подготовка файла справки

Чтобы подготовить файл справки в формате HLP с помощью программы Microsoft Help Workshop, необходимо выполнить описанную ниже процедуру.

1. Создать новый файл проекта справки.
2. Выполнить настройку проекта (указать RTF-файл с текстом тем, создать необходимые окна и т. п.).
3. Выполнить компиляцию.

Рассмотрим каждый из перечисленных этапов подготовки файла справки.

Создание файла проекта справки

Файл проекта справки является обычным текстовым файлом формата ASCII и может быть создан с помощью любого текстового редактора. Однако такой подход требует детального изучения синтаксиса этого файла. Программа Help Workshop в значительной степени облегчает разработку файла проекта справки, позволяя выполнить необходимую настройку в интерактивном режиме.

1. Откройте программу Microsoft Help Workshop, запустив файл hsw.exe или hcrtf.exe (не важно какой).
2. Выберите команду File ► New в главном меню открывшегося окна Help Workshop, затем в списке открывшегося окна диалога New выберите пункт 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 файл проекта справки, текст которого появится в окне Microsoft Help Workshop (рис. 16.6).

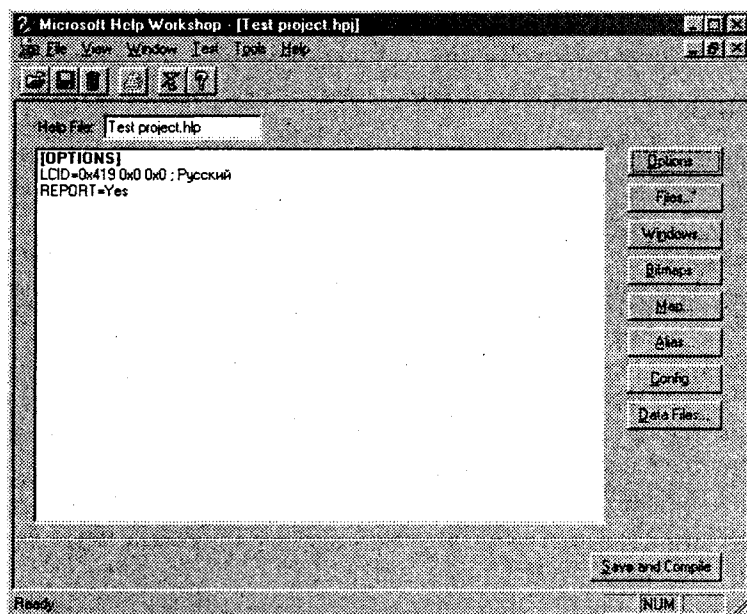


Рис. 16.6. Окно Microsoft Help Workshop с созданным файлом проекта справки

Настройка файла проекта справки

Следующим шагом по созданию файла справки является настройка проекта справки. Главное здесь — указать имя файла (файлов), содержащих текст тем, и создать окна, используемые при просмотре справки. Кроме того, следует задать ряд дополнительных параметров, необходимых для организации взаимодействия файла справки с приложением. Вся настройка выполняется в окнах диалогов, открывающихся при щелчках на кнопках, расположенных в окне Microsoft Help Workshop справа от текста файла проекта.

Окно диалогов Topic Files открывается при щелчке на кнопке Files и служит для указания файлов, содержащих тексты тем (рис. 16.7). Для добавления файла к проекту щелкните на кнопке Add и выберите в открывшемся окне диалогов открытия файла RTF-файл с текстами тем. Если таких файлов несколько, то повторите эту операцию необходимое количество раз. Ошибочно указанный файл 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 диалогов Windows Properties открывается при щелчке на кнопке Windows и используется для описания окон, которые применяются для отображения тем справки. Это окно содержит пять вкладок, на которых указываются различные свойства создаваемого окна.

- ☐ На вкладке General задаются основные свойства окна (рис. 16.8, а):
 - в поле Title bar text указывается текст заголовка окна;
 - при установке флажка Auto-size height высота окна выбирается автоматически;

- флажок **Keep Help window on top** обеспечивает размещение окна всегда поверх всех остальных.
- Вкладка **Position** позволяет задать размер и расположение окна на экране (рис. 16.8, б).
- Вкладка **Buttons** используется для размещения кнопок на панели инструментов окна (рис. 16.8, в).
- Вкладка **Color**, показанная на рис. 16.8, г, служит для задания цвета фона окна как для основной области (**Topic area color**), так и для непрокручиваемой области текста (**Nonscrolling area color**).
- На вкладке **Macros** можно ввести текст макроса, который будет выполняться при открытии окна.

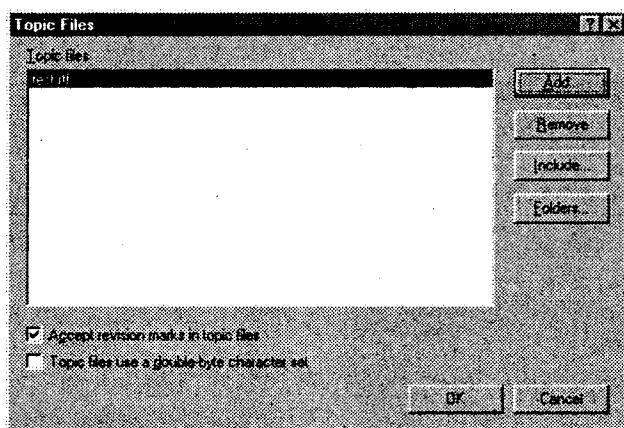


Рис. 16.7. Окно диалога Topic Files

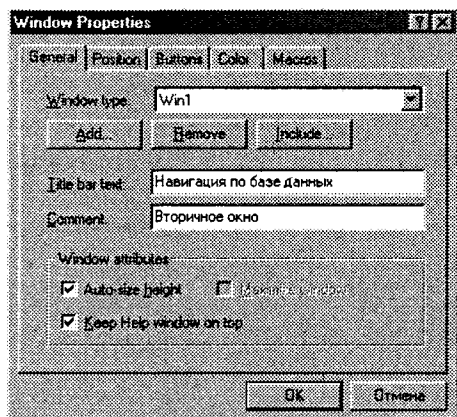
Окно диалога **Bitmap Folders** открывается при щелчке на кнопке **Bitmaps** и позволяет указать каталоги, в которых следует искать графические файлы, включаемые в тексты тем с помощью команд **bmc**, **bml** или **bmr**.

Окно диалога **Map** открывается при щелчке на одноименной кнопке (рис. 16.9). Оно предназначено для создания карты соответствий символьных идентификаторов тем справки целочисленным номерам, которые необходимы для создания контекстной справки приложения.

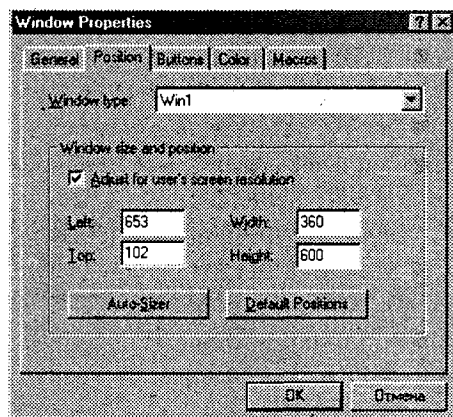
Окно **Topic ID Alias** открывается при щелчке на кнопке **Alias** и используется для задания псевдонимов идентификаторов тем. Псевдонимы позволяют указать на идентичность двух идентификаторов, что может понадобиться, например, при объединении двух (или нескольких) тем в одну.

Окно **Configuration Macros**, открывающееся при щелчке на кнопке **Config**, предназначено для задания макросов, которые будут выполняться при каждом обращении к справке.

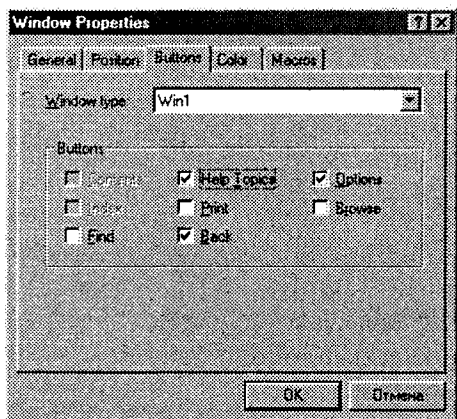
Окно диалога **Options**, с помощью которого задается ряд параметров файла справки, открывается при щелчке на кнопке **Options**. Это окно содержит восемь вкладок.



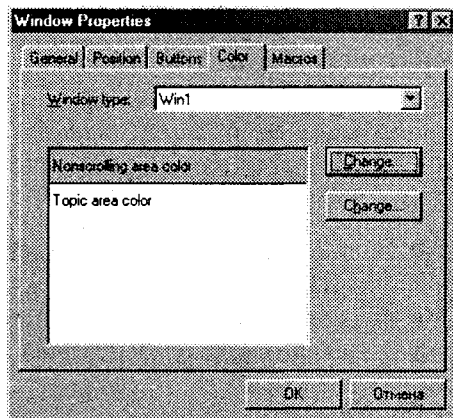
а



б



в



г

Рис. 16.8. Вкладки окна диалога Windows Properties

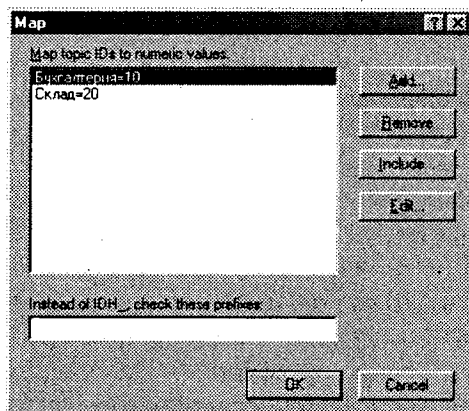
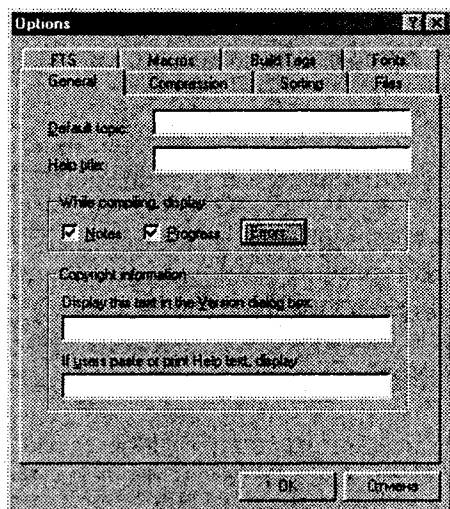


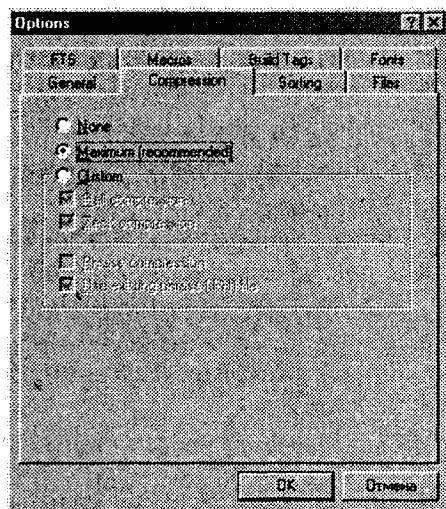
Рис. 16.9. Окно диалога Map

- ☐ На вкладке **General** задаются основные параметры файла справки (рис. 16.10, а):
 - в поле **Default topic** указывается тема, к которой справочная система будет обращаться по умолчанию;
 - в поле **Help title** вводится заголовок окна справочной системы;
 - в поле **Display this text in the Version dialog box** задается текст, отображаемый в окне диалога, открываемом при выборе команды **Справка ► Версия** главного меню программы WinHelp при просмотре справочного файла;
 - в поле **If user paste or print Help text, display** задается текст, присоединяемый к тексту темы при его печати или копировании в буфер обмена;
 - с помощью элементов управления, расположенных в группе **While compiling, display**, указывается, какая информация должна выводиться в процессе компиляции.
- ☐ На вкладке **Compression** устанавливаются параметры сжатия файла справки (рис. 16.10, б). 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** позволяет задать (рис. 16.10, в):
 - в поле **Help File** — имя результирующего HLP-файла;
 - в поле **Log File** — имя LOG-файла, в который будут заноситься все сообщения компилятора;
 - в списке **Rich Text Format Files** — имя RTF-файла с текстами тем;
 - в поле **Contents file** — имя файла содержания справки;
 - в поле **TMP folder** — каталог для хранения временных файлов;
 - в поле **Substitute path prefix** — путь для поиска RTF-файлов и графических файлов, включаемых в справочный файл.
- ☐ На вкладке **FTS** можно настроить параметры создания полнотекстового индексного файла, используемого при поиске по всему тексту (рис. 16.10, г). Обычно такой файл нужен.
- ☐ Вкладка **Font** позволяет изменить шрифты, используемые для отображении 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 в виде знака * будут включаться или не включаться в файл справки.

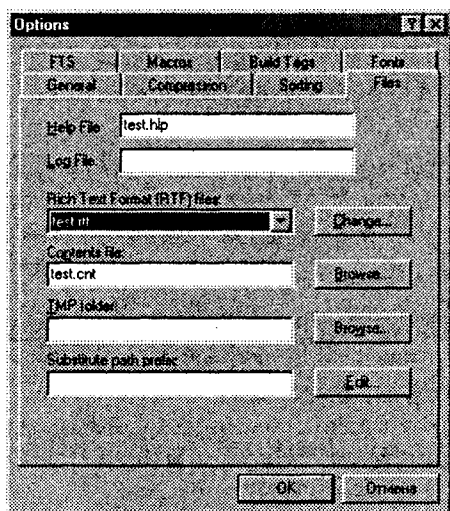
Последнее, что необходимо выполнить перед компиляцией файла проекта справки, — создать файл содержания и связать его с файлом проекта. Файл содержания представляет собой текстовый файл (с расширением **cnt**), который можно создать с помощью **Microsoft Help Workshop**.



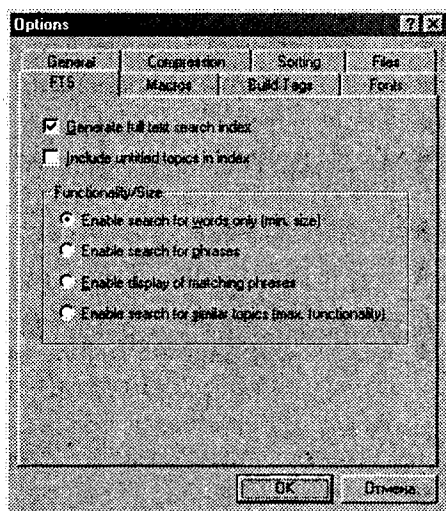
a



б



в



г

Рис. 16.10. Вкладки окна диалога Options

Для создания нового файла содержания выполните следующее.

1. Выберите команду **File ► New** в главном меню программы Help Workshop и в списке открывшегося окна диалога New выберите пункт 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 (в котором находится курсор), а вторая — ниже. При щелчке на любой из этих кнопок открывается окно диалога Edit Contents Tab Entry (рис. 16.11).

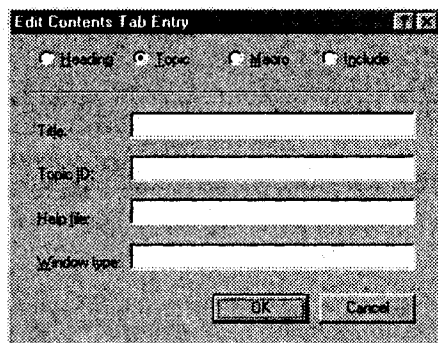


Рис. 16.11. Окно диалога Edit Contents Tab Entry

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 Title. Все остальные поля при установке этого переключателя становятся недоступными.
- ☐ **Topic** — создается ссылка на тему файла справки:
 - наименование ссылки задается в поле Title;
 - идентификатор темы, которая будет открыта при щелчке на ссылке, вводится в поле Topic ID;
 - имя файла справки, в котором расположена данная тема, указывается в поле Help file (если файл справки один, то заполнять данное поле необязательно);
 - идентификатор окна, в котором будет отображаться тема, вводится в поле Window type (если тема должна отображаться в окне, заданном по умолчанию, то в этом поле ничего указывать не надо).
- ☐ **Macro** — создается ссылка на макрос. Наименование ссылки задается в поле Title, макрос — в поле Macro.
- ☐ **Include** — включает внешний файл содержания.

Каждый вновь введенный пункт содержания сразу отображается в окне Microsoft Help Workshop (рис. 16.12).

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 рассмотренное ранее окно диалога Options (см. рис. 16.10).

Компиляция файла проекта справки

После завершения настройки файла проекта следует произвести компиляцию. В результате компиляции будет создан файл справки в формате HLP. Для компиляции щелкните на кнопке Save and Compile, расположенной в правом нижнем углу окна Help Workshop, либо сохраните файл проекта и выберите команду File ► Compile в главном меню. После компиляции в окне Help Workshop будет выведена информация о результатах компиляции (рис. 16.13).

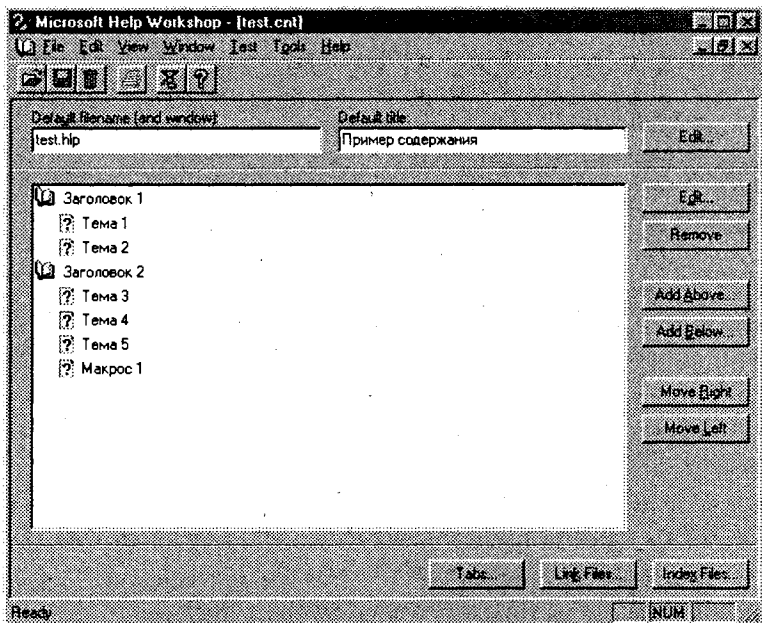


Рис. 16.12. Окно редактора содержания Microsoft Help Workshop

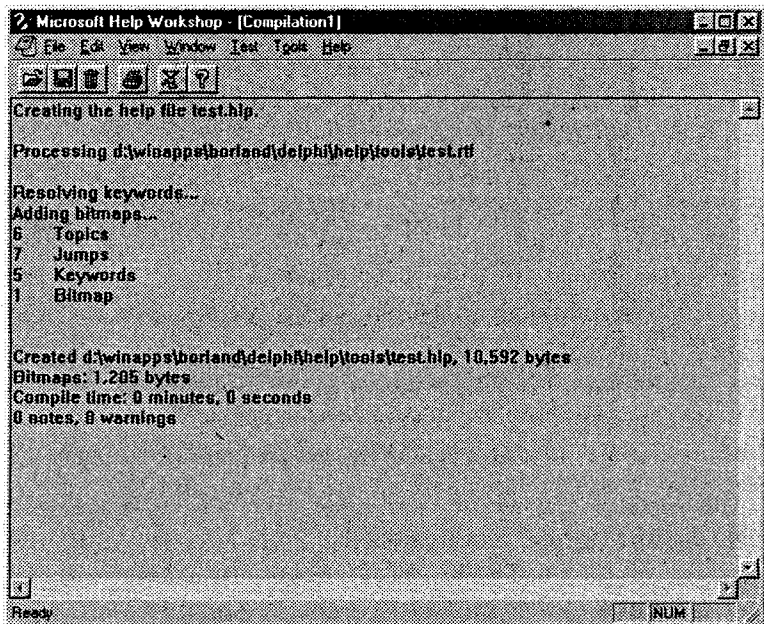


Рис. 16.13. Окно программы Help Workshop с отчетом о результатах компиляции

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** в главном меню. При этом откроется окно диалога **View Help File**, с помощью которого можно проверить работу файла справки в разных режимах (рис. 16.14).

С помощью группы переключателей **Open Help file as if it were** задается способ открытия файла справки:

- ☐ **Invoked by a program** — файл справки открывается так же, как при вызове из приложения;
- ☐ **A pop-up** — тема отображается во всплывающем окне;
- ☐ **A double-clicked file icon** — файл справки открывается так же, как при двойном щелчке на его значке.

В списке **Mapped Topic IDs** можно выбрать идентификатор темы, которая будет выводиться первой при обращении к справке. Этот список позволяет моделировать работу контекстной справки.

После завершения настройки в окне диалога **View Help File** щелкните на кнопке **View Help** для запуска программы WinHelp.

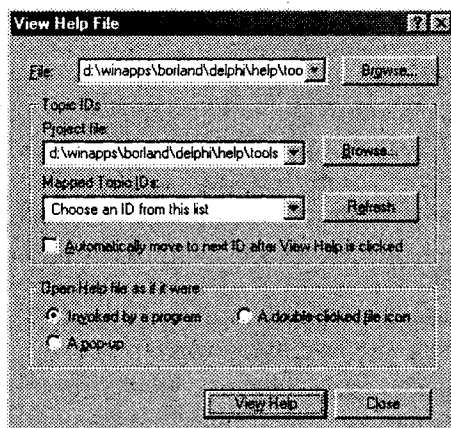


Рис. 16.14. Окно тестирования файла справки

ПРИМЕЧАНИЕ

В список **Mapped Topic IDs** заносятся только те темы, идентификаторам которых поставлены в соответствие числовые значения в окне диалога **Map**.

Создание файла справки в формате HTML Help

Рост интереса к Интернету вызвал бурное развитие средств разработки для Web. Практически все современные средства разработки включают инструментарий для

работы с Web. Поэтому нет ничего удивительного в том, что появился новый вид справочных систем, основанный на использовании языка HTML.

Средством навигации в таких справочных системах является браузер Internet Explorer. В настоящее время наиболее популярными программами для создания справочных систем приложений являются программы HTML Help Workshop и Microsoft Document Explorer. Они, как правило, включаются во все основные продукты Microsoft.

Система HTML Help Workshop, несомненно, имеет ряд преимуществ перед традиционной системой WinHelp Workshop, среди которых главными являются следующие:

- ❑ язык разметки гипертекста (HyperText Markup Language, HTML) обеспечивает широкие возможности навигации в больших массивах информации;
- ❑ система HTML Help более эффективна при использовании для публикации крупных документов.

Среди недостатков HTML Help можно отметить такие:

- ❑ для использования системы HTML Help требуется установить довольно много программных средств, включая программу просмотра файла справки (которая состоит из нескольких файлов) и DCOM95 версии не ниже 1.2;
- ❑ средства просмотра справки в формате HTML Help требуют гораздо больших ресурсов компьютера, вследствие этого вывод текста справки на не очень мощных компьютерах выполняется крайне медленно;
- ❑ система HTML Help не поддерживает вторичные окна, что в ряде случаев является довольно серьезным неудобством.

Кроме того, необходимо иметь в виду, что к скомпилированным файлам HTML Help можно обращаться только в 32-разрядной среде Windows. Ввиду наблюдаемого в последнее время роста интереса к операционным системам семейства Unix данный недостаток может оказаться очень существенным.

Далеко не все производители программных средств поддерживают систему HTML Help. В частности, Delphi 8 не имеет встроенных средств для работы со справочной системой данного типа. Несмотря на это, в приложениях, разработанных в Delphi, все же лучше использовать справочную систему HTML Help.

Развитием методов разработки справочных систем приложений является размещение справочной системы на сервере разработчиков систем проектирования. Такой подход позволяет производить постоянную корректировку справочной информации. Доступ же к справке через Интернет для современных пользователей проблем не представляет.

Основные элементы справочной системы HTML Help

Несмотря на все достоинства, язык HTML не очень пригоден для создания контекстных 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 мало пригодны для просмотра файлов справки;
- ❑ нет возможности создавать различные типы окон для вывода текста справки.

Для устранения перечисленных недостатков корпорация 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, включая оглавление, предметный указатель и средства полнотекстового поиска;
- ❑ настраиваемым интерфейсом, напоминающим интерфейс WinHelp и снабженным настраиваемыми окнами и панелями инструментов;
- ❑ контекстным интерфейсом API для организации взаимодействия с прикладными программами.

Все эти дополнения делают систему HTML Help очень похожей на WinHelp.

Создание файла справки в формате HTML

Для разработки файлов справки существует несколько инструментов. Мы будем рассматривать создание справочного файла с помощью бесплатного инструментального пакета HTML Help Workshop корпорации Microsoft. Его можно найти на сервере Microsoft.

Процесс разработки файла справки с помощью Microsoft HTML Help Workshop в целом напоминает разработку hlp-файла, о которой мы уже рассказывали.

1. Создание исходных файлов справочной системы.
2. Компиляция файла проекта, после выполнения которой все файлы справочной системы объединяются в один файл справки.
3. Тестирование и отладка справочной системы.

Рассмотрим каждый из перечисленных этапов.

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;
- ❑ 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 задействовать все возможности языка HTML. Помимо 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 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, в нем описаны разнообразные варианты настройки интерфейса и поведения справочной системы.

Для создания файла проекта выполните описанную ниже процедуру.

1. Запустите программу HTML Help Workshop и создайте новый проект справочной системы, выбрав команду File ► New.
2. В списке открывшегося окна диалога New выберите пункт Project и щелкните на кнопке OK (рис. 16.15). Появляется окно диалога New Project. С этого момента к созданию справочной системы подключается мастер проекта.

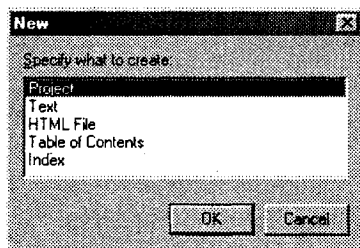
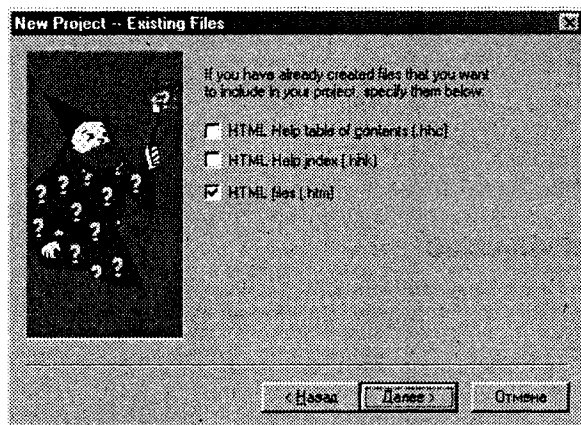


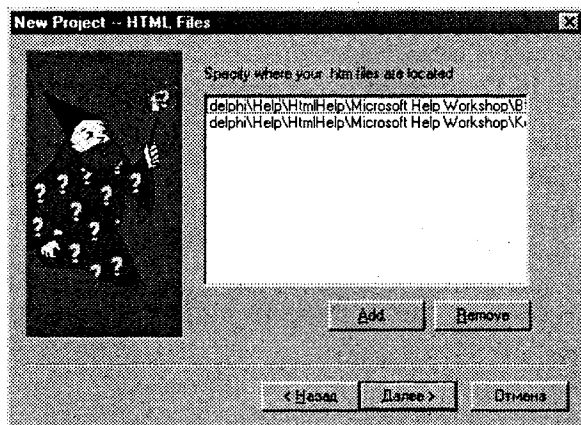
Рис. 16.15. Окно диалога New

3. 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.
4. Выберите в открывшемся окне New Project — Existing Files форматы файлов тем, которые должны быть включены в состав справочной системы (рис. 16.16, а). Если вы создали темы справочной системы в виде HTML-файлов, то вам потребуется установить флажок HTML files (.htm) и щелкнуть на кнопке Далее.
5. В открывшемся окне New Project — HTML Files, используя кнопки Add и Remove, включите в проект ранее созданные HTML-файлы с темами (рис. 16.16, б). За-

тем, щелкнув на кнопке **Далее**, перейдите в окно диалога **New Project — Finish** и щелкните на кнопке **Готово**.



а



б

Рис. 16.16. Подключение файлов тем к проекту справки

После выполнения перечисленных действий появляется окно **HTML Help Workshop** (рис. 16.17). Рассмотрим основные элементы управления этого окна.

Окно программы **HTML Help Workshop** 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 знать язык HTML.

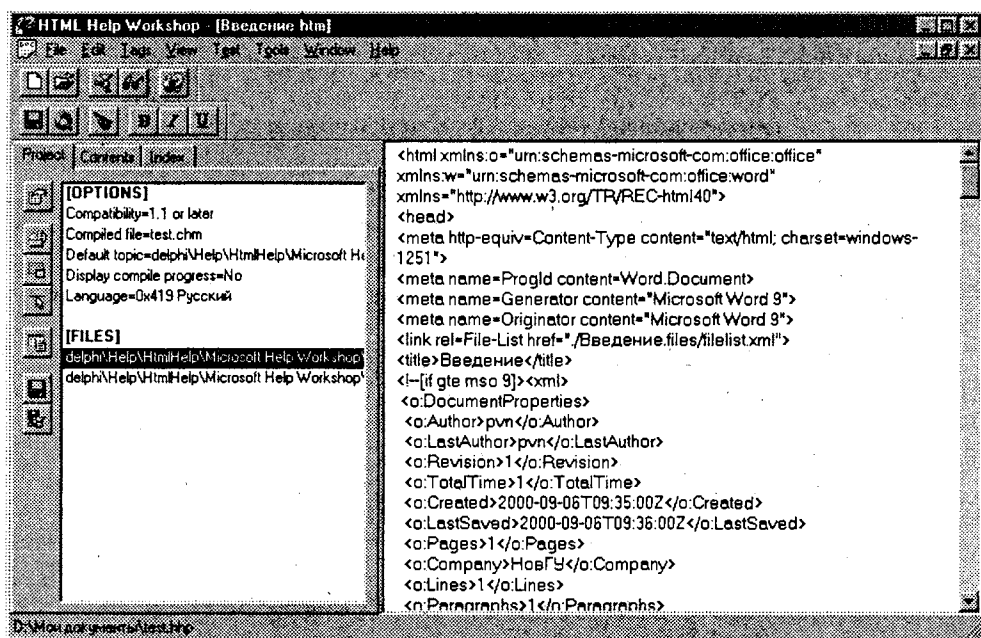


Рис. 16.17. Главное окно программы HTML Help Workshop

HTML Help Workshop предоставляет возможность просмотра содержимого файлов с темами в веб-браузере (рис. 16.18). 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. 16.19), которое открывается при двойном щелчке на разделе [Options] или при щелчке на кнопке Change Project Options панели инструментов, расположенной в левой части окна HTML Help Workshop.

Окно диалога Options содержит четыре вкладки: General, Files, Compiler и Merge Files.

- ☐ Вкладка General предназначена для настройки следующих параметров:
 - Title — заголовок окна справочной системы;
 - Default file — файл темы, появляющейся при открытии справочной системы;
 - Default window — окно, открываемое при открытии справочной системы;
 - Language — язык текста справки;
 - Font — шрифт текста справки.
- ☐ Вкладка Files позволяет разместить файлы справочной системы (Compiled file), указателей (Index file) и содержания (Contents file).
- ☐ На вкладке Compiler задаются параметры компиляции справочной системы.
- ☐ На вкладке Merge Files можно указать откомпилированные файлы справки, которые должны объединяться в процессе работы приложения.

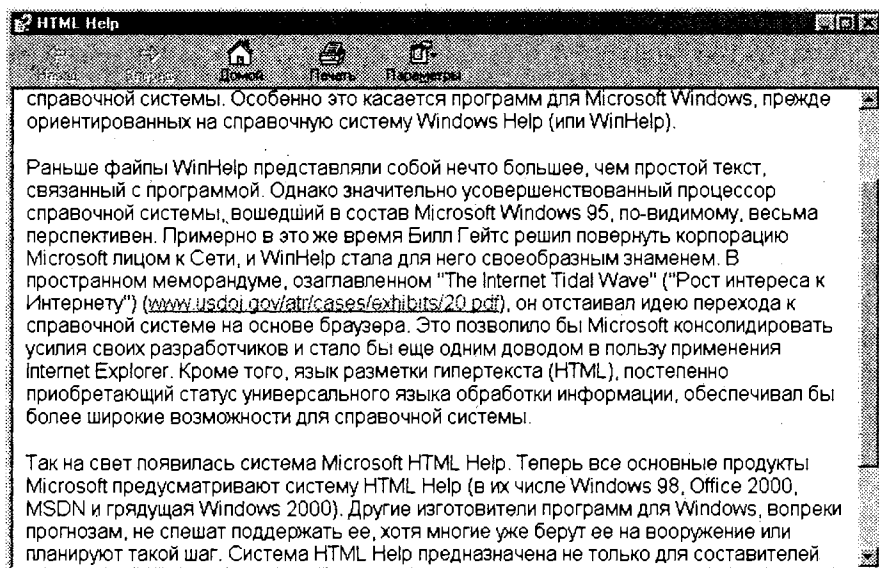


Рис. 16.18. Просмотр текста темы

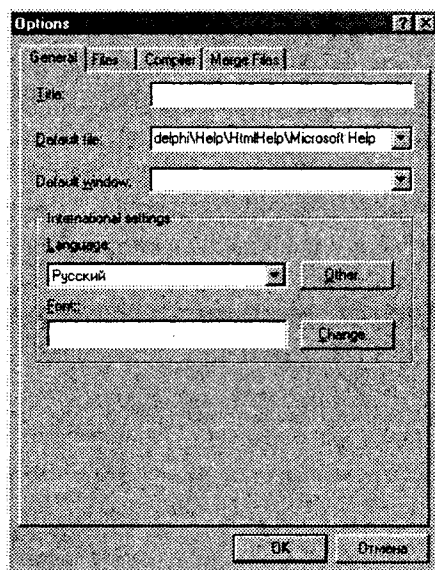


Рис. 16.19. Окно диалога Options

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 варианты настройки, определяющие внешний вид окна 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 задания 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 (отображать панель навигации при открытии окна или нет, 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 задействовать в приложениях, необходимо связать файлы тем с численными идентификаторами. HTML Help Workshop поддерживает двухступенчатое связывание: сначала каждой теме присваивается псевдоним, а затем псевдонимы связываются с численными значениями. (Псевдоним в системе HTML Help фактически является 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.

1. Откройте окно диалога **HtmlHelp API information**.
2. Перейдите на вкладку **Alias** и щелкните на кнопке **Add** (рис. 16.20).

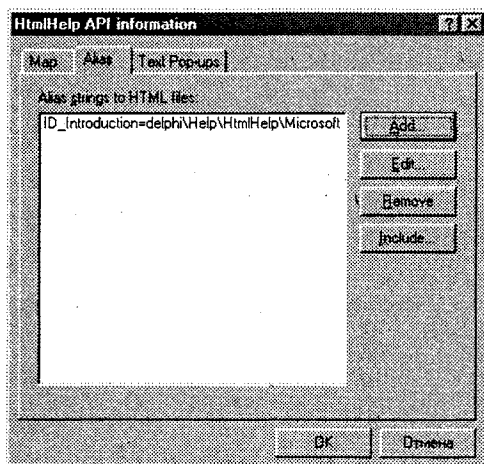


Рис. 16.20. Вкладка **Alias** окна диалога **HtmlHelp API information**

3. Выберите в раскрывающемся списке Use it to refer to this HTML file открывшегося окна диалога Alias имя файла темы и введите в расположенное над ним поле псевдоним, который будет связан с этим файлом (рис. 16.21).

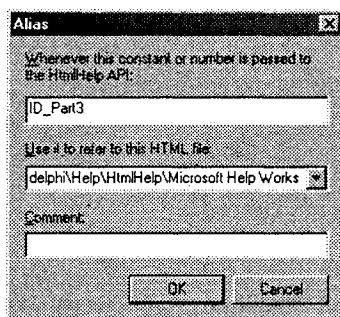


Рис. 16.21. Задание псевдонима для файла темы

4. Повторите шаг 3 необходимое количество раз в соответствии с имеющимся количеством файлов справочной системы.
5. Сохраните файл проекта справки.

Обратите внимание, что после создания псевдонимов в левой части окна HTML Help Workshop появляется 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 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 появится 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 отображения тем во всплывающих окнах необходимо создать два дополнительных файла: файл с текстами тем и файл связи.

Файл с текстами тем создается в любом 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, — это 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 вкладки `Text Pop-ups` окна диалога `HtmlHelp API Information`: 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 файла справки. Для включения содержания в справочную систему необходимо создать отдельный файл содержания.

1. В окне программы `HTML Help Workshop` 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. 16.22).

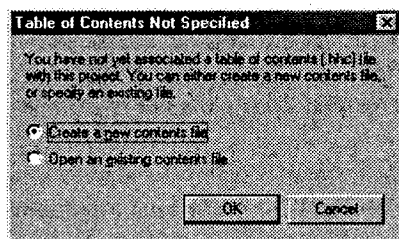


Рис. 16.22. Окно диалога `Table of Contents Not Specified`

2. Если файл содержания справочной системы не был создан вами ранее, то оставьте установленным предлагаемый по умолчанию переключатель `Create a new contents file` и щелкните на кнопке `OK`. Появится стандартное окно диалога сохранения файла.
3. Укажите каталог и имя создаваемого файла содержания и щелкните на кнопке `Сохранить`. В результате будет создан пустой файл содержания, не содержащий никакой информации.

4. Для добавления заголовка или строки ссылки на тему справочной системы щелкните соответственно на кнопке **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. 16.23).

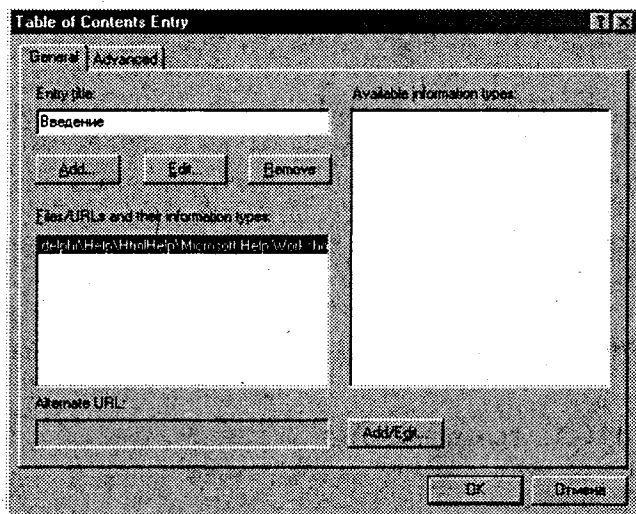


Рис. 16.23. Окно диалога Table of Contents Entry

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 многострочно текстовом поле. Щелкните на кнопке **OK**. В результате в разделе **Contents** появится строка содержания с заданным наименованием.
6. Повторите эту процедуру для всех страниц данного раздела и всех разделов проекта. При необходимости с помощью кнопок со стрелками можно изменять положение элементов содержания в иерархической структуре.

Создание индексных файлов

Помимо содержания система **HTML Help** предусматривает еще два основных инструмента 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.

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. 16.22).
 2. Примите предлагаемый по умолчанию вариант Create a new index file и щелкните на кнопке OK. Появится стандартное окно диалога сохранения файлов, 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, отредактировать ранее созданный ключ или удалить ключ.
3. Для добавления нового ключа щелкните на кнопке Insert a keyword, в результате чего откроется окно диалога Index Entry (рис. 16.24).

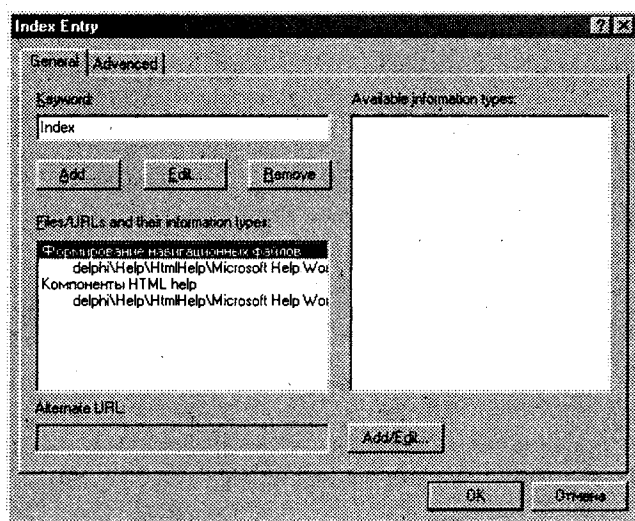


Рис. 16.24. Окно диалога Index Entry

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.

ПРИМЕЧАНИЕ

Для обеспечения полнотекстового поиска достаточно перейти на вкладку Compiler в окне диалога Options и установить флажок Compile full-text search information. Тогда компилятор сформирует поисковую базу данных и сохранит ее в CHM-файле.

Компиляция и тестирование файла справки

Окончательным этапом подготовки проекта справочной системы является его компиляция. Перед компиляцией необходимо сохранить все файлы проекта, для чего следует перейти на вкладку Project и щелкнуть на кнопке Save project, contents and index files панели инструментов этой вкладки.

Для компиляции созданного проекта щелкните на кнопке Compile HTML file на панели инструментов HTML Help Workshop. Сообщения компилятора будут появляться 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. 16.25.

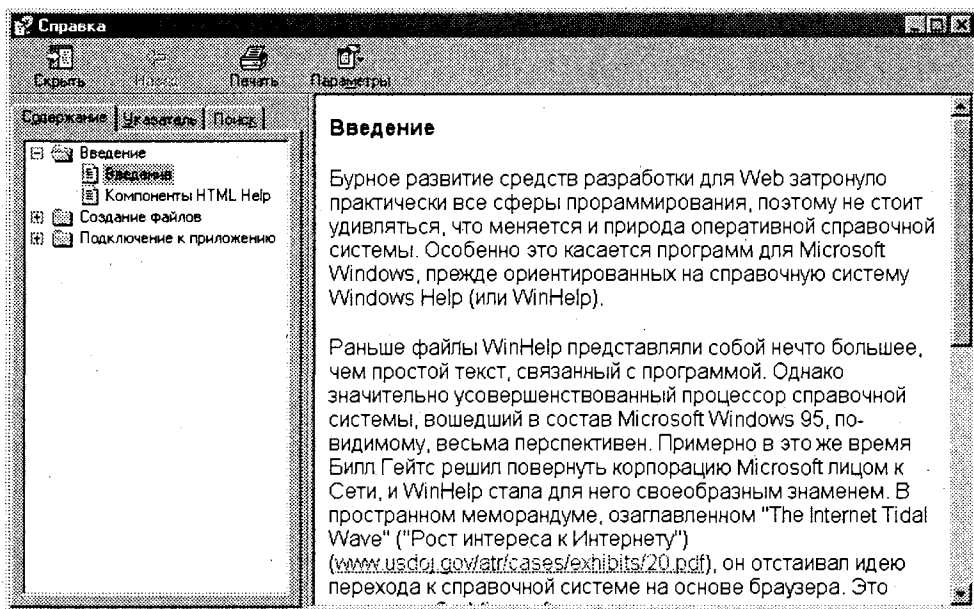


Рис. 16.25. Окно справочной системы HTML Help

Использование справочной системы в приложениях

Итак, мы рассмотрели процедуры создания справочных файлов в форматах WinHelp и HTML Help. Однако для создания полноценной справочной системы этого недостаточно — необходимо еще интегрировать файлы справки с приложением.

Метод интеграции справочных файлов с приложением зависит от формата используемых файлов.

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.

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`, открывающееся после выбора команды `Project ► Options` в главном меню Delphi. В этом окне перейдите на вкладку `Application` и укажите имя файла справки в поле `Help file` (рис. 16.26). 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 введя эту строку.

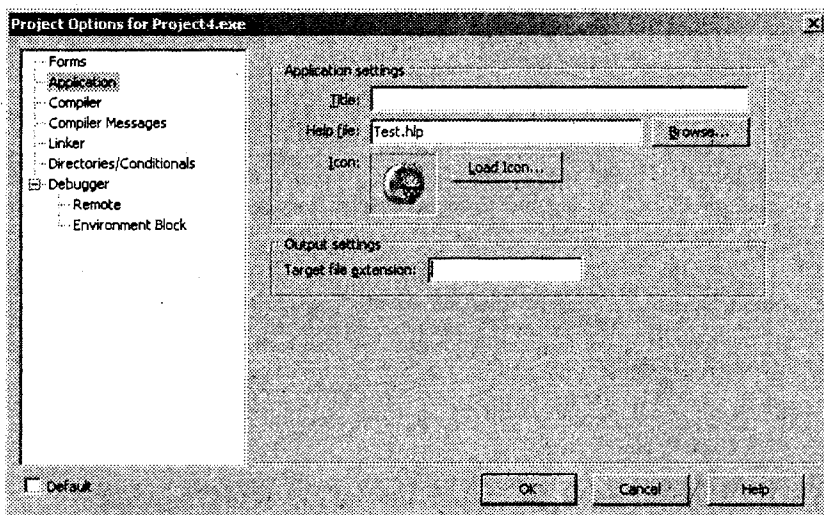


Рис. 16.26. Подключение файла справки к приложению

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 действий мы получим работоспособную контекстную 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 посредством 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, используемые для вызова справки.

```
function HelpCommand (Command: Word; Data: Longint): Boolean
```

Вызывает команду WinHelp API, 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 WinHelp API. 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 `HELP_CONTEXT WinHelp API`, 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:

```
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 этот фрагмент кода ведет 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`, то есть 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 перемен-

ной-флага в этот момент равно `false`, то выполняться будут операторы секции `else` оператора `if`, то есть переменной `F` будет присвоено значение `true`, что при повторном обращении к справке обеспечит вывод темы во всплывающем окне.

Подключение к приложению справочных файлов формата 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 не обязательно делать самим, в Интернете можно найти уже готовые решения. Например, библиотеку модулей Delphi с заголовками функций API HTML Help, а также с примерами и подробным описанием можно бесплатно загрузить с сервера [ftp://delphi-jedi.org/api/HtmlHelp.zip](http://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 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, выводимому в разделе [FILES] файла проекта справки.

Команды HtmlHelp

Для поддержания контекстной 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::<нуть>\TOPIc.htm '),
    HH_DISPLAY_TOPIC, 0);
HtmlHelp(0, PChar('HelpFile.chm'), HH_DISPLAY_TOPIC,
    DWORD(PChar('<нуть>\TOPIc.htm')));
```

В первом случае в окне справки будет показана тема, заданная 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 полный путь к теме 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 искомое ключевое слово. 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('<нуть>\Topic.htm')));
HtmlHelp(0, PChar('HelpFile.chm::<нуть>\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 для тем, отображаемых во всплывающих окнах (см. выше). Формат вызова функции `HtmlHelp` для данной команды следующий:

```
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` задается 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 тем для всплывающих окон».

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 приложение

Итак, мы рассмотрели основные возможности функции, позволяющей вызывать программу 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 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 наш файл не является `hlp`-файлом, 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 должны 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

Технология СОМ

ГЛАВА 17 Доступ к данным из приложений Microsoft Office

Наиболее популярным пакетом офисных программ является MS Office, занимающий в последние годы громадную долю рынка офисных приложений. На сегодняшний день ни в России, ни за рубежом нет достойной альтернативы MS Office, несмотря на все его многочисленные недостатки. В ряде случаев приложения MS Office удобно использовать для документирования информации, содержащейся в базах данных.

Благодаря тому, что приложения MS Office поддерживают обмен данными с использованием технологии OLE Automation, можно легко организовать передачу данных в приложения MS Office из любых других программ, написанных на любом алгоритмическом языке, имеющем средства поддержки OLE Automation.

Delphi 8 не поддерживает технологию OLE Automation, поэтому изложение материала в этой и следующей главах будет относиться к Delphi 7. К тому же в стандартную поставку Delphi 8 не входит компонент для взаимодействия с MS Office.

Основные понятия технологии автоматизации

Кратко рассмотрим основные понятия, необходимые для понимания основ технологии OLE Automation, где OLE означает связывание и внедрение объектов (Object Linking and Embedding), а automation — автоматизацию. Более подробно об этой технологии будет рассказано в следующих главах.

Технология OLE Automation — это технология обмена объектами между приложениями Windows. Взаимодействие между объектами осуществляется благодаря использованию *интерфейсов*. Интерфейс является одним из ключевых понятий технологии COM, развитием которой и является технология автоматизации. Базовым интерфейсом объектов COM является интерфейс IUnknown. Объекты автоматизации взаимодействуют с помощью так называемых *интерфейсов диспетчеризации*, построенных на основе базового интерфейса IDispatch.

ПРИМЕЧАНИЕ

На момент написания данной книги платформа .NET была не слишком распространена, поэтому в этой и следующей главах описаны основы технологии взаимодействия между программами, существовавшей ранее — технологии COM. Технология COM еще долгое время будет использоваться для создания информационных систем, несмотря на активно вытесняющую ее технологию .NET. Кроме того, знакомство с технологией COM полезно ввиду того, что многие применяемые в .NET идеи являются развитием COM.

Приложение может выступать в качестве:

- ☐ сервера автоматизации;
- ☐ клиента автоматизации;
- ☐ клиента и сервера автоматизации одновременно.

Клиентом автоматизации является приложение, которое может использовать автоматизированные объекты, созданные другими приложениями.

Сервером автоматизации служит приложение, которое позволяет использовать свои объекты клиентам автоматизации.

Автоматизация позволяет расширить функциональные возможности одного приложения (клиента) за счет использования объектов другого приложения (сервера). Одно из дополнительных преимуществ автоматизации состоит в возможности работы с приложением, являющимся сервером автоматизации, без создания видимого экземпляра этого приложения. В приложениях MS Office автоматизация реализуется за счет использования языка VBA (Visual Basic for Applications).

Любой объект, допускающий автоматизацию, можно автоматизировать и использовать в других приложениях.

Структура пакета Microsoft Office

Комплект программных продуктов Microsoft Office включает несколько приложений, образующих единую среду для обработки самой различной информации. В состав стандартной поставки Microsoft Office входят:

- ☐ MS Word — текстовый редактор;
- ☐ MS Excel — электронные таблицы;
- ☐ PowerPoint — средство для подготовки и демонстрации презентаций;
- ☐ MS Outlook — инструмент для организации и планирования работы, который включает почтовую программу, базу данных по контактам, календарь и т. п.

Кроме того, в состав профессиональной версии Microsoft Office дополнительно входит средство управления базами данных MS Access.

Microsoft Office содержит набор инструментов, общих для всех приложений, таких как инструменты проверки правописания и грамматики, графический редактор, редактор организационных диаграмм, редактор математических формул и т. п. Кроме того, приложения Microsoft Office легко могут обмениваться информацией

друг с другом. Например, табличные данные можно обработать в Excel, построить на их основе графики, а затем внедрить их в текст документа Word.

Данные для подготовки какого-либо документа могут передаваться в приложения MS Office с использованием драйверов ODBC. Благодаря этому имеется возможность внедрения в документы информации, хранящейся в базах данных.

Приложения MS Office являются полными серверами автоматизации. Это означает, что приложение может выполняться и как сервер автоматизации, и как обычное приложение. При работе в качестве сервера приложение предоставляет специальные объекты, методы и свойства, позволяющие управлять этим сервером другому приложению (не обязательно из пакета MS Office). Например, Microsoft Word можно запустить в виде самостоятельного приложения, создать документ, проверить орфографию и сохранить его в виде файла. Эти же действия можно выполнить из другого приложения, которое вызовет Word в качестве сервера автоматизации и будет программно управлять им. Таким образом, все основные приложения MS Office могут управляться с помощью внешних программ с использованием технологии OLE Automation.

Благодаря технологии OLE Automation можно передать любые данные информационной системы в приложения Microsoft Office. Эта возможность может быть очень полезной в тех случаях, когда документация предприятия ведется средствами MS Office, а данные должны извлекаться из базы данных. В таких случаях клиентская программа информационной системы может производить необходимую выборку из базы данных и передавать ее результаты в приложение MS Office.

Может возникнуть вопрос: а для чего вообще нужны какие-то «третьи» программы, если MS Office поддерживает взаимодействие с внешними базами данных? Действительно, в ряде случаев можно создавать клиентскую часть информационных систем на базе офисных приложений Microsoft (в первую очередь на базе Access). Однако такое решение приемлемо лишь для локальных баз данных и небольших информационных систем. Причин этому несколько, приведем лишь главные из них.

- ❑ Пакет MS Office хорошо взаимодействует далеко не со всеми базами данных. Полностью поддерживаются лишь ряд форматов таблиц локальных баз данных (dBase, FoxPro, Paradox) и MS SQL Server. Кроме того, следует учитывать, что технология ODBC является одним из самых медленных способов взаимодействия с базами данных (а при работе с «неродными» базами данных обмен информацией возможен только посредством ODBC).
- ❑ Пакет MS Office требует довольно много дискового пространства и требователен к ресурсам компьютера (в первую очередь — к объему оперативной памяти).
- ❑ Пакет MS Office может работать только на платформе Windows 95/98/NT/2000/XP.

В случае создания специального клиентского приложения большинство этих недостатков можно преодолеть. Поясним это на примере использования Delphi для разработки клиентской программы.

- ❑ В Delphi имеется встроенная поддержка работы со многими распространенными форматами баз данных через драйверы BDE, что гораздо быстрее, чем через ODBC (хотя технология ODBC также поддерживается). Кроме того, существует большое количество библиотек компонентов (как коммерческих, так и свободно распространяемых) для доступа к различным базам данных напрямую, без помощи BDE.
- ❑ Обычно клиентские программы, разработанные в Delphi, занимают сравнительно мало места на диске и нетребовательны к ресурсам. Хотя, конечно, здесь трудно делать обобщения, так как многое зависит от набора функций, реализуемых программой, и квалификации программиста.
- ❑ При разработке собственной программы всегда имеется возможность исправления ошибок.
- ❑ Система визуального программирования Kylix фирмы Borland является аналогом Delphi для операционной системы Linux. Обеспечивается совместимость Delphi и Kylix на уровне исходных кодов программ (правда только при условии использования новой библиотеки компонентов CLX — библиотека VCL в Kylix не поддерживается).

Таким образом, во многих случаях имеет смысл разрабатывать собственные клиентские приложения для работы с информационной системой, даже если документация ведется средствами MS Office.

Обмен данными с приложениями MS Office может выполняться с помощью одной из двух технологий:

- ❑ DDE (Dynamical Data Exchange — динамический обмен данными) — эта технология, которая использовалась еще в Windows 3.X, в настоящее время считается устаревшей;
- ❑ OLE Automation — эта технология позволяет управлять приложениями-серверами путем позднего или раннего связывания.

Методы взаимодействия с сервером автоматизации

Как уже отмечалось, до появления технологии .NET технологией взаимодействия между различными приложениями являлась технология OLE Automation. Работа с сервером автоматизации мало отличается от работы с обычным объектом Delphi, при этом различают два метода доступа к серверу автоматизации: позднее связывание и раннее связывание. Остановимся на назначении и функционировании каждого из них более подробно.

Доступ к объекту автоматизации путем позднего связывания

При *позднем связывании* компилятору Delphi не требуется никакой информации об используемом объекте. Этот факт следует отнести к недостаткам позднего свя-

зывания, так как компилятор не может проверить синтаксис кода, применяемого для доступа к серверу автоматизации.

Для доступа к серверу автоматизации используется следующая функция Delphi:

```
function CreateOleObject(const ClassName: string): IDispatch;
```

С помощью параметра `ClassName` передается строковый идентификатор программы, связанный с объектом автоматизации. Функция `CreateOleObject` возвращает указатель на интерфейс `IDispatch`, который используется для связи с объектом автоматизации.

ПРИМЕЧАНИЕ

Функция `CreateOleObject` находится в модуле `ComObj`, поэтому при ее использовании необходимо включить данный модуль в раздел `uses`.

Переменная, которая будет являться указателем на интерфейс (то есть которой присваивается значение, возвращаемое функцией `CreateOleObject`), должна иметь тип `OleVariant`, специально предназначенный для работы с объектами автоматизации.

После получения ссылки на объект автоматизации можно вызывать любые методы данного объекта.

Пример позднего связывания

Рассмотрим простой пример передачи текста в MS Word с использованием функции `CreateOleObject`. Создайте в Delphi новое приложение, разместите на форме кнопку и задайте для нее следующий метод-обработчик события `OnClick`:

```
procedure TForm1.Button1Click(Sender: TObject);
var W : OleVariant;
begin
  W:=CreateOleObject('Word.Basic');
  W.AppShow;
  W.FileNew;
  W.Insert('Пример создания нового документа'+#13);
  W.Insert('и передачи в него текста'+#13);
end;
```

Если теперь откомпилировать и запустить приложение, то после щелчка на кнопке, с которой связан приведенный выше обработчик события, откроется редактор MS Word, в нем будет создан новый документ с двумя строками, указанными в качестве параметра метода `Insert`.

Прокомментируем приведенный фрагмент кода. В первой строке мы вызвали функцию `CreateOleObject` и получили указатель на интерфейс объекта автоматизации `Basic` сервера автоматизации `Word`. После этого мы имеем возможность обращаться к методам и свойствам сервера `Word`, поддерживаемых интерфейсом объекта `Basic` так же, как и к методам обычных объектов Delphi. При вызове функции `CreateOleObject` производится запуск приложения MS Word. Однако окно редактора не будет отображаться на экране компьютера. Чтобы пользователь мог увидеть его, вызывается метод `AppShow`. Следующие строки содержат команды создания

нового документа (вызывается метод `FileNew`) и вставки строк текста в документ (метод `Insert`).

ВНИМАНИЕ

Обратите внимание, что компилятор Delphi ничего не «знает» о методах и свойствах объекта `Word`, и если попытаться вызвать несуществующий метод, программа все равно будет откомпилирована. Ошибка проявится только во время выполнения программы. Это является очень серьезным недостатком позднего связывания, так как не позволяет выявлять ошибки в вызовах методов объекта автоматизации на стадии компиляции.

ПРИМЕЧАНИЕ

В рассмотренном примере мы использовали интерфейс, управляющий приложением MS Word с помощью команд языка `WordBasic`, который применялся для написания макросов в предыдущих версиях Word (до 6-й версии включительно). Сейчас макросы Word и других приложений MS Office пишутся на языке VBA — разновидности языка `Visual Basic`. Однако сохранилась поддержка и языка `WordBasic`. Мы задействовали язык `WordBasic` вследствие его простоты и наглядности. На языке VBA фрагмент кода, выполняющий те же самые действия, был бы более громоздким и менее понятным. Для использования VBA в качестве параметра функции `CreateOleObject` следует указать строку `Word.Application`.

Доступ к объекту автоматизации путем раннего связывания

При раннем связывании компилятор должен получить информацию о методах и свойствах, которыми обладает объект автоматизации. Информация такого рода содержится в библиотеках типов, поставляемых вместе с сервером автоматизации. Это могут быть файлы в одном из следующих форматов:

- ☐ `Type Library (.tlb)`;
- ☐ `Object Type Library (.olb)`;
- ☐ в файлах элементов управления `ActiveX (.ocx)`;
- ☐ в файлах ресурсов, скомпонованных с исполняемыми файлами (`.exe`) или динамическими библиотеками (`.dll`).

В библиотеках типов обычно содержится следующая информация:

- ☐ информация о перечислениях, используемых сервером автоматизации;
- ☐ описания обычных интерфейсов, интерфейсов диспетчеризации и классов составных объектов;
- ☐ ссылки на описания типов в других библиотеках.

Чтобы импортировать информацию о типах в проект, используется команда `Project ► Import Type Library` главного меню IDE Delphi. При выборе этой команды открывается окно диалога `Import Type Library` (рис. 17.1), в котором содержится список всех серверов автоматизации и элементов `ActiveX`, зарегистрированных в системе. Классы, описанные в библиотеке, приводятся в списке `Class names`.

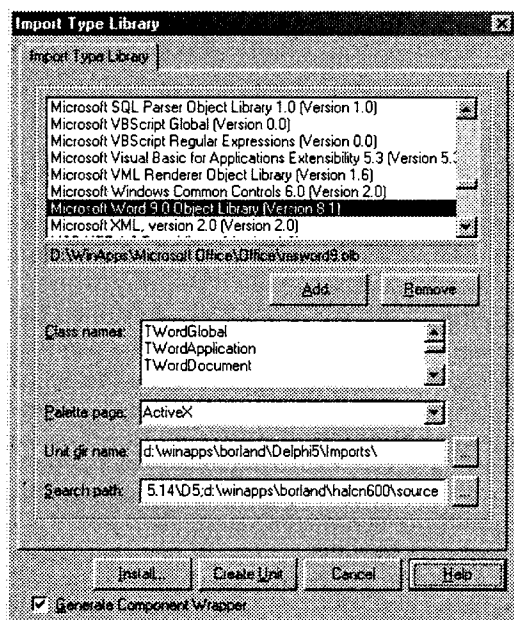


Рис. 17.1. Окно диалога импорта библиотеки типов

Для импорта библиотеки типов в проект следует выбрать нужный сервер и щелкнуть на кнопке **Install** или **Create Unit**. В обоих случаях Delphi сгенерирует файл, содержащий описание типов выбранной библиотеки в синтаксисе языка Object Pascal, и добавит его к текущему проекту. Отличие между ними состоит в том, что при щелчке на кнопке **Install** Delphi дополнительно к созданию модулей предложит включить их в пакет (новый или уже существующий), который затем будет откомпилирован и установлен. Использование пакетов удобно в том случае, если выбранная библиотека типов будет затем задействована во многих проектах или выполняется установка визуальных элементов ActiveX.

Многие методы объектов автоматизации могут содержать необязательные формальные параметры, то есть параметры, значения которых заданы по умолчанию и которые можно не указывать при вызове метода. Однако синтаксис языка Object Pascal требует, чтобы все формальные параметры, перечисленные в заголовке процедуры или функции, обязательно указывались при вызове. Для решения этой проблемы в модуле **System** Delphi 7 задана специальная константа **EmptyParam**. Данную константу следует указывать в качестве необязательного параметра при вызове методов объектов автоматизации, если вы хотите использовать значение этого параметра, заданное по умолчанию.

ПРИМЕЧАНИЕ

Обратите внимание, что в библиотеке типов содержится только описание методов объекта, но не их реализация. Библиотека типов примерно соответствует модулю Object Pascal, содержащему только раздел **interface**.

В Delphi версии 7 при работе с приложениями Microsoft Office нет необходимости импортировать библиотеки типов, так как соответствующие модули уже входят в поставку данной версии. В более ранних версиях (до пятой) Delphi библиотеку типов следует импортировать в проект.

Пример раннего связывания

В поставку Delphi 7 включен целый ряд компонентов, значительно облегчающих взаимодействие с приложениями MS Office. Данные компоненты размещены на странице **Servers** палитры компонентов. Однако вначале мы познакомимся с более общим случаем — передачей текста в MS Word без использования специальных компонентов. Для этой цели рассмотрим пример, аналогичный приведенному для позднего связывания: из программы, написанной в Delphi, вызовом MS Word, создадим новый документ и передадим в него текст.

1. Создайте новое приложение.
2. Если вы используете Delphi 7, то просто включите в раздел **uses** модуля формы модуль **Word2000**, **Word97** или **WordXP** (в зависимости от версии MS Office, установленной на вашем компьютере).

СОВЕТ

Для более ранних версий Delphi (до пятой) следует выбрать в меню команды **Project** ► **Import Type Library**, выделить в списке зарегистрированных серверов автоматизации **Microsoft Word Object Library** и щелкнуть на кнопке **Create Unit**. После этого будет создан и добавлен к текущему проекту модуль **Word_TLB**.

3. Поместите на форму кнопку и задайте для нее следующий обработчик события **OnClick**:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  W : _Application;
  D : OleVariant;
begin
  W:=CoWordApplication.Create;
  W.Visible:=true;
  W.Documents.Add(EmptyParam, EmptyParam,
    EmptyParam, EmptyParam);
  W.Selection.InsertAfter('Пример создания нового документа'+#13);
  W.Selection.InsertAfter('и передачи в него текста'+#13);
  D:=wdCollapseEnd;
  W.Selection.Collapse(D);
end;
```

Последовательность действий, выполняемых в данном фрагменте кода, примерно такая же, как и в предыдущем примере. Однако для работы с сервером автоматизации Word мы в данном случае используем интерфейс **_Application**. Поскольку имеет место раннее связывание, на этапе компиляции нам доступна информация о методах и свойствах различных интерфейсов, и мы можем объявить переменную нужного типа.

Поясним назначение некоторых фрагментов кода.

- ❑ Связь с сервером автоматизации устанавливается с помощью метода `Create` класса `CoWordApplication`, описание которого также содержится в библиотеке типов. При вызове данного метода запускается приложение MS Word, и переменная `W` получает указатель на интерфейс `_Application` сервера Word.
- ❑ Чтобы окно MS Word было видимым, свойству `Visible` интерфейса `_Application` следует присвоить значение `true`.
- ❑ Для создания нового документа используется метод `Add` объекта `Documents` — коллекции всех документов, открытых в приложении MS Word.
- ❑ Добавление строк текста к созданному документу выполняется с помощью метода `InsertAfter` объекта `Selection`, который указывает на выделенный фрагмент текста или, при отсутствии выделения, на текущую позицию курсора. Текст, добавленный таким образом, будет выделен. Для снятия выделения используется метод `Collapse` объекта `Selection`.

ПРИМЕЧАНИЕ

Методы и свойства интерфейса `_Application` почти полностью соответствуют функциям языка VBA, который используется в приложениях MS Office для создания макросов. Более подробно объекты VBA будут рассмотрены позднее.

В рассмотренном примере выполняется раннее связывание с сервером автоматизации, и корректность вызова методов сервера проверяется на этапе компиляции. Такой подход является более прогрессивным по сравнению с методом позднего связывания, так как позволяет выявлять ошибки во время компиляции. Однако при этом требуется импортировать библиотеки типов, которые не всегда поставляются с сервером автоматизации.

Компоненты Delphi для взаимодействия с серверами автоматизации MS Office

В поставку Delphi 7 включен ряд компонентов, специально предназначенных для организации взаимодействия с приложениями MS Office. Данные компоненты располагаются на странице `Servers` палитры компонентов IDE Delphi. Компоненты доступа к серверам автоматизации могут использоваться и для доступа к вложенным объектам приложений MS Office (таким как `Document`, `ExcelWorkbook` и т. п.).

Все компоненты доступа к серверам автоматизации являются потомками базового класса COM-серверов `TOLEServer`. В этом классе объявлен ряд абстрактных методов и свойств, позволяющих устанавливать связь с сервером автоматизации. Потомки класса `TOLEServer` создаются при импорте библиотек типов.

Рассмотрим опубликованные свойства компонентов серверов COM, устанавливающих контакт с интерфейсом `Application` (компоненты `WordApplication`, `ExcelApplication`, `PowerPointApplication`, `OutlookApplication`). Наряду с традиционными для всех компонентов свойствами `Name` и `Tag` они содержат четыре дополнительных.

- ❑ `AutoConnect` : `Boolean` — определяет, загружать (`true`) или нет (`false`) сервер при запуске приложения.

- ❑ **AutoQuit** : Boolean — определяет, разрывать (true) или нет (false) связь с сервером при завершении работы приложения.
- ❑ **ConnectKind** : TConnectKind — определяет способ установления соединения с сервером. Это свойство может принимать следующие значения:
 - **ckRunningOrNew** — если приложение-сервер уже выполняется, то производится подключение к имеющемуся экземпляру сервера, в противном случае производится запуск нового экземпляра сервера;
 - **ckNewInstance** — всегда создается новый экземпляр сервера;
 - **ckRunningInstance** — подключение только к уже запущенному серверу;
 - **ckRemote** — подключение к удаленному серверу;
 - **ckAttachToInterface** — подключение не выполняется.

В последнем случае (если свойство **ConnectKind** равно **ckAttachToInterface**) соединение с сервером производится с помощью метода **ConnectTo**, и значение свойства **AutoConnect** не может устанавливаться в true. Обычно значение **ckAttachToInterface** свойства **ConnectKind** неприменимо для серверов **Application**.

- ❑ **RemoteMachineName** — сетевое имя компьютера, на котором расположен сервер. Указывается при установке свойства **ConnectKind** равным **ckRemote**.

Значительная часть остальных серверов, представленных на странице **Servers** палитры компонентов и предназначенных для работы с вложенными объектами приложений MS Office, обладают точно такими же опубликованными свойствами, за исключением свойства **AutoQuit**, которое у них отсутствует.

При использовании компонентов-серверов COM для доступа к приложениям MS Office достаточно поместить на форму соответствующий компонент и затем вызывать требуемые методы сервера автоматизации, так же как методы обычного объекта Delphi. Например, для того чтобы выполнить подключение к серверу Word, создать в нем новый документ и передать в него текст, можно использовать следующий фрагмент кода:

```
...
WordApplication1.Connect:
WordApplication1.Visible:=true;
WordApplication1.Documents.Add(EmptyParam.EmptyParam,
  EmptyParam.EmptyParam);
WordApplication1.Selection.InsertAfter('Пример
  создания нового документа'+#13);
WordApplication1.Selection.InsertAfter('и передачи
  в него текста'+#13);
...
```

Здесь метод **Connect** устанавливает связь с сервером Word. В зависимости от значения свойства **ConnectionKind** выполняется подключение к запущенному приложению MS Word либо создается новый экземпляр сервера.

ПРИМЕЧАНИЕ

При установке свойства **AutoConnect** в значение true метод **Connect** вызывать не надо — связь с сервером устанавливается сразу при запуске приложения.

Весь остальной текст приведенного фрагмента кода аналогичен предыдущему и не требует пояснений.

СОВЕТ

Следует отметить, что компоненты-серверы WordApplication, ExcelApplication, PowerPointApplication, OutlookApplication устанавливают связь с сервером автоматизации через интерфейс Application. Поэтому их методы и свойства фактически являются методами и свойствами сервера, используемого при написании макросов на языке VBA в приложениях MS Office. В справочной системе Delphi практически не содержится информации о компонентах-серверах. Поэтому, если при работе с ними возникают затруднения, следует обращаться к справке по VBA, поставляемой с MS Office.

Из методов компонентов для связи с объектами автоматизации выделим следующие:

- ☐ Connect — устанавливает связь с сервером MS Word (данный метод следует использовать, если значение свойства AutoConnect компонента WordApplication равно false);
- ☐ ConnectTo — выполняет подключение к уже существующему объекту (в качестве аргумента этого метода указывается объект, с которым связывается компонент);
- ☐ Disconnect — разрывает соединение с сервером.

Помимо свойств и методов, компоненты-серверы могут обрабатывать ряд событий сервера автоматизации. Количество и виды обрабатываемых событий зависят от конкретного компонента.

Взаимодействие с приложениями MS Office

Как уже отмечалось, VCL-компоненты, предназначенные для взаимодействия с приложениями MS Office, практически не документированы в справочной системе Delphi. Попытаемся частично восполнить этот недостаток и приведем краткое описание структуры объектов основных приложений MS Office. Так как в версии 7 Delphi имеется ряд компонентов, предназначенных для программирования взаимодействия с приложениями MS Office, то в приводимых примерах будут использованы именно эти компоненты (напомним, что данные компоненты размещены на странице Servers палитры компонентов IDE Delphi).

Взаимодействие с MS Word

Приложение MS Word имеет довольно сложную структуру объектов, поэтому мы не будем рассматривать все объекты, а ограничимся только теми, которые необходимы для выполнения основных операций с приложением. Для получения дополнительной информации следует обращаться к справочной системе по VBA, входящей в поставку MS Office.

Основным объектом, представляющим собственно приложение MS Word, является объект Application. Он имеет довольно сложную иерархическую структуру,

состоящую из большого количества встроенных объектов (рис. 17.2). В Delphi объекту Application MS Word соответствует класс TWordApplication.

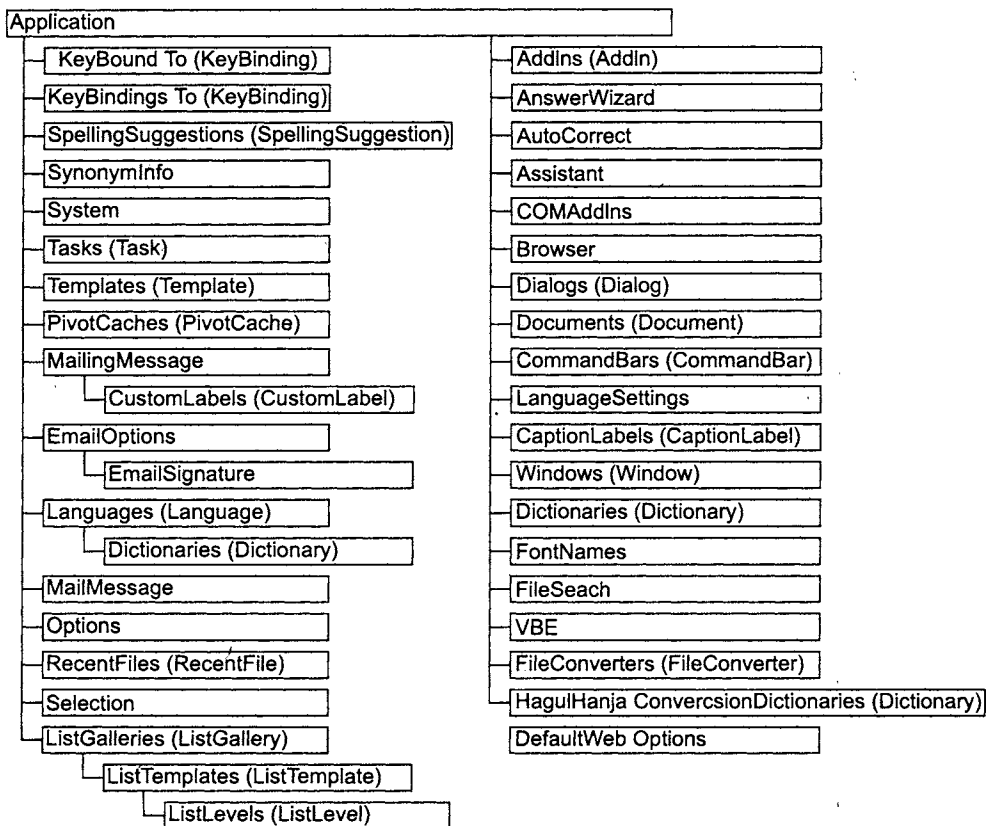


Рис. 17.2. Структура объекта Application приложения MS Word

Основные методы и свойства объекта Application

Объект Application содержит лишь небольшое число методов и свойств, к которым возникает необходимость обращаться при управлении приложением MS Word извне. Тем не менее, некоторые из свойств очень важны, поэтому мы их рассмотрим подробнее.

- ❑ **Visible** : Boolean — управляет видимостью окна приложения MS Word. Если значение этого свойства равно false, то пользователь не будет видеть даже запущенное приложение MS Word.
- ❑ **ActiveDocument** : Document — ссылка на объект Document, который в текущий момент является активным.
- ❑ **Options** : Options — объект, содержащий описание параметров MS Word.
- ❑ **Documents** : Documents — коллекция объектов Document, содержащая все открытые в данный момент документы.

- ☐ Selection : Selection — указывает на выделенный фрагмент текста или на текущую позицию курсора.

ПРИМЕЧАНИЕ

Не следует путать опубликованные свойства класса TWordApplication со свойствами объекта Application приложения MS Word. Первые являются обыкновенными свойствами, имеющими соответствующие им поля и методы для записи и чтения. Вторые — фактически абстрактные свойства, заголовки которых импортируются из библиотеки типов сервера автоматизации MS Word.

Из методов объекта Application выделим один:

Quit(SaveChanges, Format):

Этот метод закрывает приложение MS Word. Параметр SaveChanges указывает, следует ли сохранять изменения в открытых документах. Его возможные значения:

- ☐ wdDoNotSaveChanges — не сохранять изменения;
- ☐ wdPromptToSaveChanges — запросить у пользователя, сохранять ли изменения;
- ☐ wdSaveChanges — сохранить изменения.

Параметр Format позволяет задать формат, в котором будут сохранены документы:

- ☐ wdOriginalDocumentFormat — сохранить в исходном формате;
- ☐ wdPromptUser — запросить формат у пользователя;
- ☐ wdWordDocument — сохранить в формате документа Word.

При организации работы с документами MS Word из стороннего приложения очень важной является коллекция Documents, входящая в состав объекта Application и включающая в себя множество объектов Document.

Коллекция Documents

Коллекция Documents содержит все открытые в текущий момент документы (объекты Document). Коллекция Documents обладает всего четырьмя свойствами, из которых наиболее важным является свойство Count, имеющее тип Integer и содержащее информацию о количестве открытых документов.

Можно выделить ряд методов коллекции Documents выполняющих наиболее важные функции, такие как создание, открытие, закрытие и сохранение документов.

Add(Template: OleVariant; NewTemplate: OleVariant;
DocumentType: OleVariant; Visible: OleVariant):

Создает новый документ. Все параметры данного метода являются необязательными и имеют следующий смысл:

- Template — имя шаблона, на основе которого будет создан новый документ, если данный параметр не указан (точнее, если вместо него указано значение EmptyParam), то документ создается на основе шаблона normal.dot;
- NewTemplate — указывает, создается обычный документ (false) или шаблон (true), по умолчанию имеет значение false;

- **DocumentType** — тип создаваемого документа, возможные значения: **wdNewBlankDocument**, **wdNewEmailMessage**, **wdNewFrameset**, **wdNewWebPage** (по умолчанию имеет значение **wdNewBlankDocument**);
- **Visible** — определяет, будет отображаться окно с созданным документом (**true**) или нет (**false**), значение по умолчанию — **true**.

ПРИМЕЧАНИЕ

При передаче логических параметров можно использовать вместо **false** значение 0, а вместо **true** — любое целое положительное число.

```
Open(FileName: OleVariant; ConfirmConversions: OleVariant;  
  ReadOnly: OleVariant; AddToRecentFiles: OleVariant;  
  PasswordDocument: OleVariant; PasswordTemplate: OleVariant;  
  Revert: OleVariant; WritePasswordDocument: OleVariant;  
  WritePasswordTemplate: OleVariant; Format: OleVariant;  
  Encoding: OleVariant; Visible: OleVariant);
```

Открывает существующий (созданный ранее) документ. Из всех параметров метода **Open** обязательным является только первый — **FileName**, который задает имя открываемого файла. Остальные параметры имеют следующий смысл:

- **ConfirmConversions** — определяет, отображать (**true**) или нет (**false**) окно диалога преобразования файла, если файл не является документом MS Word;
- **ReadOnly** — задание данного параметра равным **true** означает, что документ будет открыт в режиме «только для чтения»;
- **AddToRecentFiles** — определяет, заносить (**true**) или нет (**false**) открываемый документ в список недавно использованных файлов, расположенный в нижней части меню **Файл**;
- **PasswordDocument** — пароль для открытия документа;
- **PasswordTemplate** — пароль для открытия шаблона;
- **Revert** — при попытке повторного открытия уже открытого документа данный параметр определяет, игнорировать ли все изменения, внесенные в документ, и открыть файл заново (**true**) или продолжить работу с уже открытым документом (**false**);
- **WritePasswordDocument** — пароль для сохранения внесенных в документ изменений;
- **WritePasswordTemplate** — пароль для сохранения внесенных в шаблон изменений;
- **Format** — фильтр, используемый при открытии документа (может задаваться с помощью одной из констант, приведенных в табл. 17.1), по умолчанию имеет значение **wdOpenFormatAuto**;
- **Encoding** — кодовая страница, которая используется в MS Word при отображении документа (по умолчанию применяется системная кодовая таблица);
- **Visible** — определяет видимость окна, в котором открывается документ.

Save(NoPrompt: OleVariant; OriginalFormat: OleVariant);

Выполняет сохранение всех открытых документов. Для документов, сохраняемых первый раз, открывается окно диалога Сохранить как. Оба параметра этого метода являются необязательными и имеют следующее назначение:

- NoPrompt — определяет, выдавать (false) или нет (true) запрос на сохранение изменений в документе;
- OriginalFormat — задает формат сохраняемого документа (wdOriginalDocumentFormat — сохранить документ в текущем формате, wdPromptUser — запросить формат у пользователя, wdWordDocument — сохранить в формате MS Word).

Close(SaveChanges: OleVariant; OriginalFormat: OleVariant;
RouteDocument: OleVariant);

Закрывает все открытые документы. Все параметры данного метода необязательны и имеют следующий смысл:

- SaveChanges — определяет действия, выполняемые при закрытии документа (wdDoNotSaveChanges — не сохранять изменений в документе, wdPromptToSaveChanges — запросить у пользователя, сохранять изменения или нет, wdSaveChanges — сохранять изменения в документе);
- OriginalFormat — задает формат документа при сохранении (полностью аналогичен одноименному параметру метода Save);
- RouteDocument — определяет, может данный документ быть отослан следующему получателю (true) или нет (false).

Item(Index: OleVariant);

Возвращает ссылку на объект Document. С помощью параметра Index указывается либо порядковый номер документа, либо имя документа. Это возможно благодаря тому, что параметр передается в виде переменной типа OleVariant (переменные вариантного типа совместимы практически со всеми типами данных). Нумерация документов начинается с 1. Имя документа обязательно должно включать расширение.

Таблица 17.1. Константы, определяющие формат открываемого файла

Константа	Формат файла
wdOpenFormatAllWord	Документ MS Word
wdOpenFormatAuto	Формат определяется автоматически
wdOpenFormatDocument	Документ MS Word 2000
wdOpenFormatEncodedText	Текстовый файл
wdOpenFormatRTF	Текст в формате RTF
wdOpenFormatTemplate	Шаблон MS Word
wdOpenFormatText	Текст в формате ASCII
wdOpenFormatUnicodeText	Текст в формате Unicode
wdOpenFormatWebPages	HTML-документ

Основные методы и свойства объекта Document

Каждый из элементов коллекции Documents представляет собой ссылку на объект Document, являющийся документом, открытым в приложении MS Word. Объект Document имеет довольно сложную структуру и содержит большое количество встроенных объектов. Для упрощения программирования работы с объектами Document в VCL Delphi имеется специальный класс TWordDocument.

Многие свойства объекта Document, в свою очередь, являются объектами, имеющими достаточно сложную и разветвленную структуру. Мы ограничимся рассмотрением только основных объектов и методов, которые используются для выполнения типовых операций, включая создание, открытие, закрытие и сохранение документов. Кроме того, важно знать свойства объекта Document, позволяющие получить доступ к абзацам, стилям, таблицам и окнам документа.

Основные свойства объекта Document приведены в табл. 17.2.

Таблица 17.2. Основные свойства объекта Document

Свойство	Тип	Описание
AttachedTemplate	Template	Объект, указывающий на шаблон документа
AutoHyphenation	Boolean	Включает (true) или выключает (false) режим автоматической расстановки переносов
Content	Range	Объект, содержащий весь текст документа
FullName	WideString	Имя документа и путь к нему
GrammarChecked	Boolean	Определяет, была выполнена проверка грамматики в документе (true) или нет (false)
GrammaticalErrors	ProofreadingErrors	Коллекция объектов Range, каждый из которых содержит предложение с грамматическими ошибками, найденными в документе. Каждое предложение может содержать несколько ошибок. При отсутствии ошибок принимает значение 0
Name	WideString	Имя файла документа
PageSetup	PageSetup	Объект, содержащий параметры страниц документа (поля, размер бумаги и т. п.)
Paragraphs	Paragraphs	Коллекция объектов Paragraph, каждый из которых является абзацем в документе. Содержит все абзацы документа
ReadOnly	Boolean	Включает (true) или выключает (false) режим «только для чтения»
Saved	Boolean	Определяет, были (false) или нет (true) внесены изменения в документ с момента последнего сохранения
SaveFormat	Integer	Формат документа
ShowSpellingErrors	Boolean	Включает (true) или выключает (false) режим подчеркивания слов с орфографическими ошибками

продолжение ➤

Таблица 17.2 (продолжение)

Свойство	Тип	Описание
ShowGrammaticalErrors	Boolean	Включает (true) или выключает (false) режим подчеркивания предложений с грамматическими ошибками
Styles	Styles	Коллекция объектов Style, содержащая все стили документа
Tables	Tables	Коллекция объектов Table, содержащая все таблицы документа
TablesOfContents	TablesOfContents	Коллекция объектов TablesOfContent, содержащая все оглавления документа
Type_	TOLEEnum	Тип документа: обычный документ (wdTypeDocument) или шаблон (wdTypeTemplate)
Words	Words	Коллекция объектов Word, содержащая все слова документа

Методы объекта Document позволяют выполнять различные действия с документами.

CheckSpelling:

Выполняет проверку орфографии в документе. При обнаружении ошибки содержащее документ окно MS Word становится активным, кроме того, открывается окно диалога Правописание.

Close(SaveChanges, OriginalFormat, RouteDocument):

Закрывает документ. Данный метод полностью аналогичен методу Close коллекции Documents.

Range(Start: OleVariant; End: OleVariant):

Возвращает объект Range, содержащий фрагмент текста документа, начиная с символа, имеющего порядковый номер Start и заканчивая символом с порядковым номером End. Невидимые символы также учитываются при создании объекта Range.

Save(NoPrompt, OriginalFormat):

Аналогичен методу Save коллекции Documents.

Activate:

Делает документ активным.

Undo(Times: OleVariant):

Отменяет последние выполненные действия. Возвращает значение true, если действия были успешно отменены. Количество отменяемых действий определяется параметром Times.

При работе с внутренними объектами OLE-серверов можно использовать переменные типа OleVariant. Например, следующий фрагмент кода добавляет после 100-го символа документа текст «new text», причем работа с объектом Range производится с помощью переменной вариантного типа:

```
...
var
R, S, E : OleVariant;
```

```
...
S:=0;
E:=100;
R:=WordDocument1.Range(S,E);
R.InsertAfter('new text');
...
```

Обратите внимание, что в данном случае компилятор не может проверить корректность вызова методов объекта R. Иными словами, такой подход подразумевает позднее связывание с объектом типа Range. Поэтому в обычных случаях следует использовать не переменные OleVariant, а переменные типа объекта, с которым выполняются действия:

```
...
var
S, E : OleVariant;
R : Range;
...
S:=0;
E:=100;
R:=WordDocument1.Range(S,E);
R.InsertAfter('new text');
...
```

Можно также вызывать методы объекта без объявления соответствующей переменной:

```
...
WordDocument1.Range(S,E).InsertAfter('new text');
...
```

В этом случае метод Range относится к объекту Document, а метод InsertAfter — к объекту Range.

Ввод и форматирование текста в MS Word

С помощью методов объектов Application и Document можно выполнить ряд важных действий: создать новый документ, открыть существующий документ, выполнить проверку грамматики и орфографии в документе и т. п. Однако методы объектов Application и Document не позволяют реализовать передачу текста в документ MS Word, а также задать параметры форматирования текста. Для этого следует использовать методы объектов, являющихся свойствами более крупных объектов — Application или Document. Основными объектами, с помощью которых осуществляется ввод и форматирование текста, являются объекты Paragraph, Selection и Range.

Коллекция Paragraphs и объект Paragraph

Объект Paragraph является элементом коллекции Paragraphs (которая, в свою очередь, является одним из свойств объекта Document). Коллекция Paragraphs содержит все абзацы документа. Методы коллекции Paragraphs позволяют выполнять над абзацами ряд действий, главным из которых является метод Add, добавляющий новый абзац к документу. Данный метод имеет следующий синтаксис:

```
function Add(var Range: OleVariant): Paragraph
```

Параметр `Range` является необязательным и задает фрагмент текста документа, перед которым будет располагаться добавляемый абзац. Если при вызове метода `Add` в качестве параметра передать `EmptyParam`, то добавляемый абзац окажется в конце документа.

Элементами коллекции `Paragraphs` являются объекты `Paragraph` — абзацы документа. В Delphi для доступа к элементам коллекции `Paragraphs` следует использовать метод `Item`:

```
function Item(Index: integer): Paragraph
```

Данный метод возвращает абзац документа, имеющий порядковый номер `Index`. Например, для доступа к первому абзацу активного документа можно использовать следующий фрагмент кода:

```
...
var
  P : Paragraph;
begin
  ...
  WordDocument1.ConnectTo(WordApplication1.ActiveDocument);
  P:=WordDocument1.Paragraphs.Item(1);
  ...
end;
...
```

Из свойств объекта `Paragraph` наибольший интерес представляют те, которые позволяют задавать параметры форматирования текста. Основные из них приведены в табл. 17.3.

Хотя для объекта `Paragraph` определен ряд методов, особого интереса они не представляют, поэтому мы их здесь рассматривать не будем. Для получения информации о них обращайтесь к справочной системе по VBA для MS Word.

Таблица 17.3. Основные свойства объекта `Paragraph`, определяющие параметры форматирования текста абзаца

Свойство	Тип	Описание
Alignment	TOfEnum	Способ выравнивания текста абзаца: выравнивание по левому краю (<code>wdAlignParagraphLeft</code>), выравнивание по центру (<code>wdAlignParagraphCenter</code>), выравнивание по правому краю (<code>wdAlignParagraphRight</code>), выравнивание по ширине (<code>wdAlignParagraphJustify</code>)
FirstLineIndent	Single	Отступ первой строки в пунктах. Может принимать как положительные (устанавливается отступ), так и отрицательные (устанавливается выступ) значения
Hyphenation	Boolean	Определяет, выполнять (<code>true</code>) или нет (<code>false</code>) расстановку переносов для данного абзаца
LeftIndent	Single	Величина левого отступа абзаца в пунктах
RightIndent	Single	Величина правого отступа абзаца в пунктах
LineSpacingRule	TOfEnum	Величина межстрочного интервал для абзаца: полуторный (<code>wdLineSpace1pt5</code>), двойной (<code>wdLineSpaceDouble</code>), минимум (<code>wdLineSpaceAtLeast</code>), точно (<code>wdLineSpaceExactly</code>), множитель (<code>wdLineSpaceMultiple</code>), одинарный (<code>wdLineSpaceSingle</code>)

Свойство	Тип	Описание
LineSpacing	Single	Величина межстрочного интервала для абзаца (в пунктах). Значение этого свойства используется в том случае, если свойству LineSpacingRule задано одно из следующих значений: wdLineSpaceAtLeast (межстрочный интервал может быть больше или равен значению, определенному свойством LineSpacing), wdLineSpaceExactly (величина межстрочного интервала всегда точно соответствует значению, заданному в свойстве LineSpacing и не зависит от размера шрифта абзаца), wdLineSpaceMultiple (значение межстрочного интервала в LineSpacing раз больше чем при выборе одинарного интервала, который зависит от размера шрифта абзаца)
SpaceAfter	Single	Величина отступа перед абзацем (в пунктах)
SpaceBefore	Single	Величина отступа после абзаца (в пунктах)

Объекты Range и Selection

Важнейшими объектами, предназначенными для работы с текстом в MS Word, являются объекты Range и Selection. Именно они содержат методы, которые позволяют передавать текст в документ MS Word. Эти объекты имеют очень много общего. В частности, методы работы с текстом и свойства, определяющие формат текста, у них идентичны. Основное различие между ними заключается в следующем:

- ❑ объект Range содержит некоторый непрерывный фрагмент текста документа, и для доступа к этому объекту используется рассмотренный ранее метод Range объекта Document;
- ❑ объект Selection позволяет работать с выделенным текстом, и для доступа к этому объекту используется одноименное свойство объекта Application (при отсутствии выделения это свойство указывает на текущее положение курсора в активном документе).

ПРИМЕЧАНИЕ

Следует отметить, что объекты Range и Selection имеют сложную структуру, которая во многом подобна структуре объекта Document. Многие свойства и методы объекта Document присущи также объектам Range и Selection.

Наиболее интересными свойствами объекта Range являются те, что определяют параметры шрифта, которым отображается фрагмент текста документа, относящегося к данному объекту. Информация об этих свойствах приведена в табл. 17.4.

Объект Selection не имеет свойств для управления стилями Bold, Italic и Underline, однако он обладает свойством Range, которое является объектом типа Range, содержащим выделенный текст. Таким образом, для применения полужирного стиля шрифта к выделенному тексту можно использовать следующую строку:

```
WordApplication1.Selection.Range.Bold=1;
```

Объект Selection также обладает свойством Font, полностью аналогичным одноименному свойству объекта Range.

Таблица 17.4. Свойства объекта Range, определяющие стиль шрифта

Свойство	Тип	Описание
Bold	Integer	Определяет, отображается текст полужирным шрифтом (1) или нет (0). Кроме того, это свойство может принимать значение wdUndefined, если во фрагменте текста используются разные стили. При задании этого свойства можно также использовать константу wdToggle, чтобы изменить текущее значение стиля на противоположное
Italic	Integer	Полностью аналогично предыдущему свойству за исключением того, что предназначено для управления курсивным начертанием текста
Underline	ToleEnum	Предназначено для управления подчеркиванием текста
Font	_Font	Тип шрифта, который используется для форматирования текста. В палитре компонентов Delphi имеется специальный компонент WordFont, предназначенный для управления параметрами шрифта

Объекты Range и Selection имеют еще два интересных свойства, общих для обоих объектов: Start и End (оба свойства имеют тип Integer). Первое определяет начальный символ диапазона или выделения, второе — конечный символ.

Из всего множества методов объектов Range и Selection рассмотрим только те, которые предназначены для работы с буфером обмена и для передачи текста.

Прежде всего рассмотрим метод TypeText, который определен только для объекта Selection. С помощью данного метода можно производить вставку текста в выделение. Синтаксис данного метода следующий:

```
procedure TypeText(const Text: WideString);
```

Результат выполнения метода TypeText зависит от свойства ReplaceSelection объекта Options: если оно равно true, то весь выделенный фрагмент текста заменяется строкой, переданной в качестве параметра Text; если оно имеет значение false, то новый текст вставляется перед выделением (по умолчанию значение свойства ReplaceSelection равно true).

Текст, введенный с помощью метода TypeText, не включается в выделение. Например, при выполнении следующего фрагмента кода будет создан новый документ, и в его начало добавятся две строки, которые не выделяются:

```
WordApplication1.Connect;
WordApplication1.Documents.Add(EmptyParam,
    EmptyParam, EmptyParam, EmptyParam);
WordApplication1.Visible:=true;
WordApplication1.Selection.TypeText('Пример использования'+#13);
WordApplication1.Selection.TypeText('метода TypeText'+#13);
```

Помимо метода TypeText, объект Selection имеет еще два метода для ввода текста в документ: InsertBefore и InsertAfter. Синтаксис вызова этих методов точно такой же, как и для метода TypeText — в качестве аргумента передается строка типа WideString, которая добавляется к документу. Метод InsertBefore добавляет строку *перед* выделением, метод InsertAfter — *после* выделения. В отличие от метода TypeText при вызове методов InsertBefore и InsertAfter добавляемый текст включается в выделение. При выполнении следующего фрагмента кода добавленный текст будет выделен:

```
WordApplication1.Connect;  
WordApplication1.Documents.Add(EmptyParam,  
    EmptyParam, EmptyParam, EmptyParam);  
WordApplication1.Visible:=true;  
WordApplication1.Selection.InsertAfter('Пример использования'+#13);  
WordApplication1.Selection.InsertAfter('метода InsertAfter'+#13);
```

Благодаря тому что текст, вводимый с помощью методов `InsertBefore` и `InsertAfter`, выделяется, можно легко задать параметры форматирования вводимого текста отличными от принятых по умолчанию. Для этого следует просто изменить значение рассмотренных выше свойств, задающих форматирование текста.

Методы `InsertBefore` и `InsertAfter` определены и для объекта `Range`. Работают они точно так же, как и с объектом `Selection`: добавляемый текст вставляется либо в начало, либо в конец диапазона и становится частью диапазона.

Объекты `Selection` и `Range` имеют ряд идентичных методов, предназначенных для работы с буфером обмена:

- ☐ `Paste` — копирует содержимое буфера в объект `Selection` или `Range` (при вызове данного метода содержимое диапазона или выделения заменяется содержимым буфера обмена);
- ☐ `Copy` — копирует текст из объекта `Selection` или `Range` в буфер обмена.

Метод `Collapse` выполняет «сжатие» диапазона или выделения «в точку» к начальной или конечной позиции входящих в них символов. После этого значения свойств `Start` и `End` становятся равными. Синтаксис метода `Collapse`:

```
procedure Collapse(var Direction: OleVariant);
```

Необязательный параметр `Direction` определяет направление, в котором «сжимается» диапазон или выделение. Его значение задается с помощью одной из следующих констант:

- ☐ `wdCollapseEnd` — «сжатие» к концу;
- ☐ `wdCollapseStart` — «сжатие» к началу (используется по умолчанию).

Работа с таблицами в MS Word

Мы рассмотрели основные объекты, позволяющие выполнять ввод и форматирование текста. Однако помимо текста часто требуется внедрять в документы MS Word информацию, представленную в табличной форме. Для работы с таблицами используются специальные объекты — коллекция `Tables` из объектов `Table`, каждый из которых представляет таблицу, содержащуюся в документе.

Объект Tables

Коллекция `Tables` является свойством документа, выделения или диапазона. При этом содержимое данной коллекции характеризует таблицы, входящие в состав соответствующего объекта более высокого уровня иерархии. Хотя объект `Tables` обладает сравнительно небольшим количеством свойств и методов, некоторые из них очень важны.

Среди свойств коллекции `Tables` выделим одно: `Count`, которое имеет тип `Integer` и содержит число объектов `Table` (количество таблиц), входящих в состав коллекции.

Объект `Tables` имеет всего два метода: `Add` и `Item`.

```
function Add(const Range: Range; NumRows: Integer;
  NumColumns: Integer; var DefaultTableBehavior: OleVariant;
  var AutoFitBehavior: OleVariant);
```

Добавляет новую таблицу. Параметры метода имеют следующий смысл:

- `Range` — диапазон, в котором создается таблица (если перед вызовом метода `Add` к диапазону, задаваемому этим параметром, не применен метод `Collapse`, то текст, содержащийся в нем, будет удален);
- `NumRows` — количество строк в создаваемой таблице;
- `NumColumns` — количество столбцов в создаваемой таблице;
- `DefaultTableBehavior` — этот необязательный параметр определяет, будут ли автоматически изменяться размеры ячеек таблицы при вводе текста (`wdWord8TableBehavior` — размеры ячеек не будут изменяться, `wdWord9TableBehavior` — размеры ячеек будут подбираться автоматически);
- `AutoFitBehavior` — определяет правила для автоматического подбора размеров ячеек.

```
function Item(index: Integer): Table;
```

Возвращает элемент коллекции с порядковым номером `index`.

Рассмотрим пример создания таблицы программным способом. Пусть это будет таблица, содержащая три строки и четыре столбца и расположенная в начале нового документа. Для этого потребуется всего несколько строк кода:

```
// Подключение к серверу Word
WordApplication1.Connect;
// Создание нового документа
WordApplication1.Documents.Add(EmptyParam,
  EmptyParam, EmptyParam, EmptyParam);
WordApplication1.Visible:=true;
// Ввод заголовка таблицы
WordApplication1.Selection.TypeText('Пример создания таблицы'+#13);
// Создание таблицы
WordApplication1.Selection.Tables.Add(
  WordApplication1.Selection.Range, 3, 4,
  EmptyParam, EmptyParam);
```

В данном примере мы внедряли таблицу во вновь созданный документ, поэтому в качестве параметра `Range` использовалось свойство `Range` объекта `Selection` (для нового документа это просто текущее положение курсора). Если требуется добавить таблицу в уже существующий документ, содержащий какой-то текст, то для указания местоположения таблицы можно использовать метод `Range` объекта `Document`. Например, для того чтобы добавить таблицу в конец документа `document.doc`, можно использовать следующий фрагмент кода:

```
var
  D : OleVariant;
  R : Range;
begin
  // Подключаемся к серверу Word
  WordApplication1.Connect;
  // Выбираем документ document.doc
```



```
D:='document.doc';
WordApplication1.Documents.Item(D);
WordApplication1.Visible:=true;
// Переходим в конец документа
D:=wdCollapseEnd;
R:=WordApplication1.ActiveDocument.Content;
R.Collapse(D);
// Добавляем таблицу
R.Tables.Add(R,3,4,EmptyParam,EmptyParam);
end;
```

Объект Table

Итак, мы выяснили, каким образом можно добавить новую таблицу в документ. Однако после вставки таблицы ее следует заполнить какой-нибудь информацией. Для этого следует использовать методы и свойства объекта Table. Основные свойства этого объекта приведены в табл. 17.5.

Таблица 17.5. Основные свойства объекта Table

Свойство	Тип	Описание
Columns	Columns	Коллекция объектов Column, каждый из которых представляет собой столбец таблицы
Rows	Rows	Коллекция объектов Row, каждый из которых представляет собой строку таблицы
Borders	Borders	Коллекция объектов Border, предназначенных для управления линиями сетки таблицы

Из всех методов объекта Table рассмотрим только один, который возвращает объект Cell, представляющий собой ячейку таблицы:

```
function Cell(Row, Column: integer): Cell;
```

Важнейшим свойством объекта Cell является свойство Range, представляющее собой объект Range, содержащий текст ячейки таблицы. Из методов объекта Cell следует выделить метод Select, выделяющий содержимое ячейки.

Таким образом, при заполнении таблицы можно применить любой из двух способов:

- ❑ использовать методы InsertBefore или InsertAfter объекта Range, являющегося свойством объекта Cell;
- ❑ вызвать метод Select объекта Cell и затем использовать для ввода текста методы объекта Selection.

В заключение рассмотрим пример экспорта в Word таблицы базы данных с использованием технологии OLE Automation. Будем передавать данные из таблицы Сотрудники базы данных sales.mdb. Пусть данные из этой таблицы после запуска приложения отображаются с помощью компонента DBGrid, а экспорт в Word производится при щелчке на кнопке. В этом случае нам понадобятся следующие компоненты:

- ❑ кнопка (Button);
- ❑ набор данных ADO (ADOTable);

- ❑ источник данных (DataSource);
- ❑ компонент для визуализации данных (DBGrid);
- ❑ компонент для связи с сервером автоматизации MS Word (WordApplication).

Настройка соединения компонента ADOTable с таблицами MS Access и подключение набора данных к компонентам визуализации данных подробно описаны в предыдущих главах, поэтому не будем на этом останавливаться. Полный код модуля приложения с комментариями приведен в листинге 17.1.

Листинг 17.1. Экспорт в Word таблицы базы данных

```
unit Office_u;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, Word2000, OleServer,
  StdCtrls, Grids, DBGrids, Db, ADODB;

type
  TForm1 = class(TForm)
    Button1: TButton;
    WordApplication1: TWordApplication;
    WordDocument1: TWordDocument;
    ADOTable1: TADOTable;
    DataSource1: TDataSource;
    DBGrid1: TDBGrid;
    procedure Button1Click(Sender: TObject);
    procedure FormShow(Sender: TObject);
    procedure FormClose(Sender: TObject; var Action:
      TCloseAction);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.DFM}

procedure TForm1.Button1Click(Sender: TObject);
var
  R : Range;
  T : Table;
  numRec,numField,i,j : integer;
  C : Cell;
begin
  // Получаем количество записей в таблице
  numRec:=ADOTable1.RecordCount;
  // Получаем количество полей таблицы
  numField:=ADOTable1.FieldCount;
  WordApplication1.Connect;
```

```
WordApplication1.Documents.Add(EmptyParam,
    EmptyParam,EmptyParam,EmptyParam);
WordApplication1.Visible:=true;
// Передаем в документ строку текста, которая
// будет являться заголовком таблицы
WordApplication1.Selection.TypeText('
    Таблица 1. Список сотрудников');
R:=WordApplication1.Selection.Range;
R.Tables.Add(r,numRec+1,numField,EmptyParam,
    EmptyParam);
T:=WordApplication1.ActiveDocument.Tables.Item(1);
// Создаем "шапку" таблицы – передаем в первую строку
// таблицы имена полей таблицы базы данных
for j:=0 to numField-1 do begin
    C:=T.Cell(1,j+1);
    C.Range.InsertAfter(ADOTable1.Fields[j].FieldName);
end;
// Передаем в таблицу Word данные из таблицы базы данных
for i:=1 to numRec do
    for j:=0 to numField-1 do begin
        C:=T.Cell(i+1,j+1);
        ADOTable1.RecNo:=i;
        C.Range.InsertAfter(ADOTable1.Fields[j].AsString);
    end;
end;

procedure TForm1.FormShow(Sender: TObject);
begin
    // Открываем набор данных при запуске приложения
    ADOTable1.Open;
end;

procedure TForm1.FormClose(Sender: TObject;
    var Action: TCloseAction);
begin
    // Закрываем набор данных при закрытии приложения
    ADOTable1.Close;
end;

end.
```

Взаимодействие с MS Excel

Приложение MS Excel, так же как и MS Word, состоит из большого количества объектов (рис. 17.3). В целом структура объектов MS Excel похожа на структуру объектов MS Word. Основным объектом, представляющим собой само приложение MS Excel, является объект **Application**. Данный объект находится на вершине иерархии объектов MS Excel. Все остальные объекты, встроенные в него, являются его свойствами. Причем многие объекты-свойства в свою очередь состоят из большого количества объектов и имеют довольно сложную структуру. Мы ограничимся рассмотрением только основных свойств, необходимых для передачи данных в приложение MS Excel и влияющих на способ представления этих данных.

- ❑ **Workbooks** — коллекция объектов **Workbook**, которая содержит все открытые рабочие книги MS Excel. Рабочую книгу MS Excel можно рассматривать как аналог документа MS Word.

- ❑ Sheets — коллекция объектов Sheet, каждый из которых представляет лист активной рабочей книги.
- ❑ Cells — объект Range, представляющий все ячейки на активном рабочем листе. В том случае, если активный объект не является рабочим листом, это свойство неприменимо.
- ❑ Charts — коллекция объектов Charts, каждый из которых представляет диаграмму активной рабочей книги.

В VCL Delphi имеется набор компонентов, предназначенных для работы со всеми основными объектами MS Excel:

- ❑ TExcelApplication — объект Application;
- ❑ TExcelWorkbook — объект Workbook;
- ❑ TExcelWorksheet — объект Sheet;
- ❑ TExcelChart — объект Chart.

Данные компоненты, так же как и компоненты для работы с MS Word, расположены на странице Servers палитры компонентов и представляют собой просто оболочки для объектов автоматизации MS Excel.

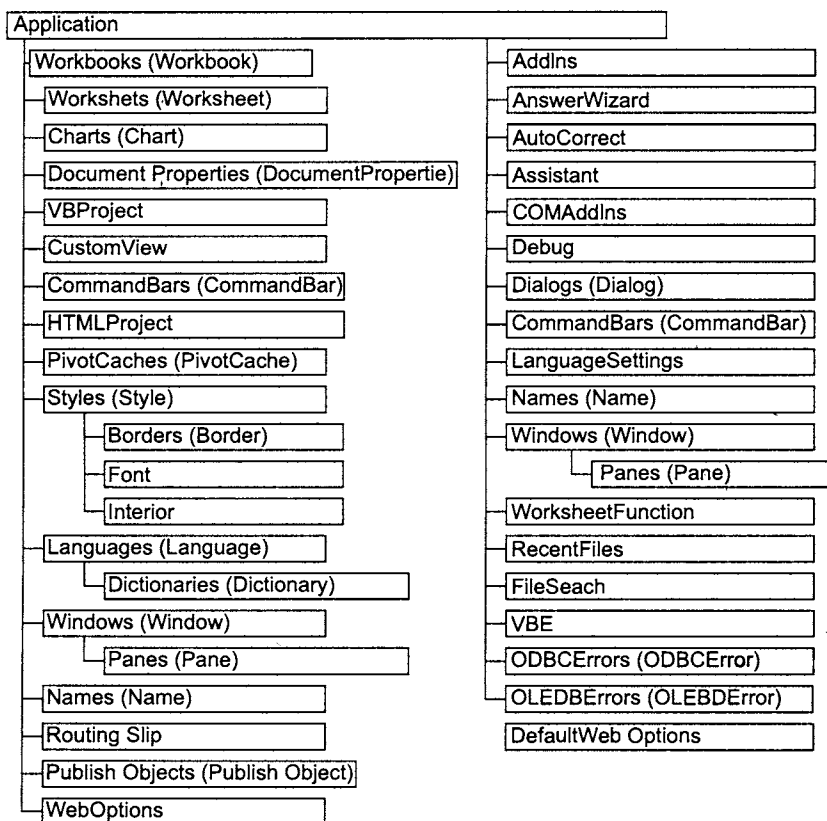


Рис. 17.3. Иерархия объектов MS Excel

Рабочие книги MS Excel

Как уже отмечалось, все рабочие книги MS Excel, открытые в данный момент, представлены в виде элементов коллекции *Workbooks*. Для создания новой рабочей книги, открытия существующей или закрытия всех рабочих книг следует использовать методы и свойства коллекции *Workbooks* объекта *Application*.

Основные методы и свойства коллекции *Workbooks*

Перечислим название и назначение основных методов и свойств коллекции *Workbooks*.

`function Add(Template: OleVariant; Icid: integer): _Workbook;`

Создает новую рабочую книгу. Параметр *Template* является необязательным и определяет шаблон, по которому будет создана рабочая книга. Если этот параметр имеет значение строки, определяющей имя файла существующей рабочей книги (включая путь), то создается рабочая книга с использованием указанного файла в качестве шаблона. Данный параметр также может задаваться с помощью константы *xlWBATExcel4IntlMacroSheet*, *xlWBATExcel4MacroSheet*, *xlWBATWorksheet* или *xlWBATChart*. В этом случае создаваемая книга будет содержать один лист соответствующего типа. Если параметр *Template* не задавать (точнее, задать для него значение *EmptyParam*), то создается рабочая книга с количеством листов, определяемым значением свойства *SheetsInNewWorkbook*. Параметр *Icid* задает идентификатор создаваемой рабочей книги.

`function Open(FileName: WideString; UpdateLinks: OleVariant;
ReadOnly: OleVariant; Format: OleVariant;
Password: OleVariant; WriteResPassword: OleVariant;
IgnoreReadOnlyRecommended: OleVariant;
Origin: OleVariant; Delimiter: OleVariant;
Editable: OleVariant; Notify: OleVariant;
Converter: OleVariant; AddToMRU: OleVariant;
Icid: integer): _Workbook;`

Открывает ранее созданную рабочую книгу. Все параметры, кроме первого, являются необязательными. Основные параметры имеют следующий смысл:

- **FileName** — определяет имя файла открываемой рабочей книги;
- **UpdateLinks** — определяет способ обновления связей в открываемом файле (0 — никакие связи не обновляются; 1 — обновляются внешние ссылки, но не обновляются удаленные ссылки; 2 — обновляются удаленные ссылки, но не обновляются внешние ссылки; 3 — обновляются оба типа ссылок), если данный аргумент отсутствует, то способ обновления связей запрашивается у пользователя;
- **ReadOnly** — определяет, допускается внесение изменений в открываемую рабочую книгу (*false*) или нет (*true*);
- **Format** — определяет вид символов-разделителей (1 — символы табуляции; 2 — запятые; 3 — пробелы; 4 — точки с запятой; 5 — разделители отсутствуют; 6 — вид разделителя определяется пользователем с помощью параметра *Delimiter*);
- **Password** — строка пароля для открытия защищенной рабочей книги (если при открытии защищенной книги данный аргумент отсутствует, то пароль запрашивается у пользователя);

- **WriteResPassword** — строка пароля для записи в рабочую книгу (если при открытии защищенной книги данный аргумент отсутствует, то пароль на запись запрашивается у пользователя);
- **IgnoreReadOnlyRecommended** — позволяет (при значении `true`) устранить вывод сообщения с рекомендацией открытия книги только для чтения;
- **Origin** — указывает, где был создан открываемый файл, что необходимо для правильного распознавания страницы кодировки (допустимое значение `xlMacintosh`, `xlWindows` или `xlMSDOS`), если данный аргумент отсутствует, то используются текущие параметры операционной системы;
- **Delimiter** — определяет символ-разделитель (если открывается текстовый файл и параметр **Format** имеет значение `b`).

`procedure Close(Icid: integer);`

Закрывает рабочую книгу с идентификатором `Icid`.

`function Item(index: OleVariant): _Workbook;`

Возвращает рабочую книгу с порядковым номером `index` или с именем, заданным параметром `index`.

Из свойств коллекции **Workbooks** следует выделить только свойство **Count**, которое имеет тип **Integer** и содержит информацию о количестве открытых в данный момент рабочих книг.

Рассмотрим пример создания новой рабочей книги Excel из программы, разработанной в Delphi. Для этого нам понадобится лишь один дополнительный компонент — **TExcelApplication**. Новая рабочая книга создается двумя строками кода:

```
// Устанавливаем соединение с сервером
// автоматизации Excel
ExcelApplication1.Connect;
// Создаем новую рабочую книгу
ExcelApplication1.Workbooks.Add(EmptyParam, 0);
```

После выполнения приведенного фрагмента кода будет создана рабочая книга, но пользователь не сможет ее увидеть, так как по умолчанию при соединении с сервером окно MS Excel невидимо. Для включения видимости следует присвоить значение `true` свойству **Visible** объекта **ExcelApplication**:

```
ExcelApplication1.Visible[0]:=true;
```

Обратите внимание на то, что свойство **Visible** объекта **ExcelApplication**, в отличие от аналогичного свойства объекта **WordApplication**, является векторным. В качестве индекса ему передается идентификатор рабочей книги, видимость которой изменяется.

Объект **Workbook**

Каждый элемент коллекции **Workbooks** является объектом **Workbook**, представляющим открытую рабочую книгу Excel. Рассмотрим основные свойства данного объекта:

- ❑ **ActiveSheet** — объект, который является активным листом в активной рабочей книге;
- ❑ **Sheets** — коллекция типа **Sheets**, содержащая все листы данной рабочей книги (включая диаграммы);

- ❑ **Worksheets** — коллекция типа **Sheets**, содержащая только рабочие листы данной рабочей книги.

Ниже перечислены те из методов объекта **Workbook**, которые представляют наибольший интерес.

```
procedure Activate(Icid: integer);
```

Активизирует окно данной рабочей книги.

```
procedure Close(SaveChanges, FileName:  
RouteWorkbook: OleVariant; Icid: integer);
```

Закрывает рабочую книгу. Первые три параметра являются необязательными и определяют:

- **SaveChanges** — следует ли сохранять изменения;
- **FileName** — имя файла, в котором будет сохранена закрываемая рабочая книга;
- **RouteWorkbook** — посылать (**true**) или нет (**false**) рабочую книгу следующему получателю.

Обязательный параметр **Icid** задает идентификатор рабочей книги.

```
procedure Save(Icid: integer);
```

Сохраняет изменения, внесенные в рабочую книгу.

```
procedure SaveAs(Filename, FileFormat, Password, WriteResPassword,  
ReadOnlyRecommended, CreateBackup: OleVariant;  
AccessMode: OleEnum, ConflictResolution, AddToMru,  
TextCodePage, TextVisualLayout: OleVariant; Icid: integer);
```

Этот метод также предназначен для записи рабочей книги в файл, однако, в отличие от предыдущего метода, он позволяет сохранять текущую рабочую книгу в файле с другим именем и/или в другом формате.

Листы MS Excel

В MS Excel существует два основных типа листов:

- ❑ **рабочие листы** (**worksheets**) содержат ячейки таблицы Excel и представляются объектом **Worksheet**;
- ❑ **листы диаграмм** (**charts**) содержат графики и диаграммы и представляются объектом **Chart**.

Листы рабочей книги располагаются в трех коллекциях:

- ❑ в коллекцию **Sheets** входят все листы рабочей книги, включая рабочие листы и листы диаграмм;
- ❑ коллекция **Worksheets** включает только рабочие листы, содержащиеся в рабочей книге;
- ❑ коллекция **Charts** содержит только диаграммы, расположенные в рабочей книге.

Все перечисленные выше коллекции представляются объектами, имеющими тип **Sheets**. Основным методом объекта **Sheets** является метод **Add**, добавляющий лист в рабочую книгу:

```
function Add(Before, After, Count, Type_: OleVariant;  
Icid: integer): IDispatch;
```

Первые четыре параметра данного метода являются необязательными (то есть могут быть заданы константой `EmptyParam`) и имеют следующий смысл:

- ☐ **Before** — лист, перед которым будет вставлен добавляемый лист;
- ☐ **After** — лист, после которого будет вставлен добавляемый лист;
- ☐ **Count** — количество добавляемых листов (по умолчанию добавляется один лист);
- ☐ **Type_** — тип добавляемого листа (`xlWorksheet`, `xlExcel4MacroSheet` или `xlExcel4IntlMacroSheet`), по умолчанию используется значение `xlWorksheet`.

Из основных свойств следует выделить следующее:

`Item[index: OleVariant] : IDispatch`

Это свойство содержит указатель на интерфейс диспетчеризации элемента коллекции.

Объект **Worksheet**

Объект **Worksheet** представляет собой конкретный рабочий лист. Все объекты **Worksheet** являются элементами коллекции **Worksheets**. В Delphi для работы с этим объектом используется специальный компонент **TExcelWorksheet**.

Работа с рабочими листами (как, впрочем, и с диаграммами) имеет одну особенность. Как вы, наверное, заметили, свойство `Items`, используемое для доступа к элементам коллекции `Sheets`, имеет тип `IDispatch`, а не `Worksheet` или `Chart`. Поэтому в данном случае для работы с элементами коллекции `Sheets` в программе Delphi удобнее всего использовать следующий подход.

1. Поместить на форму компонент **TExcelWorksheet** и задать его опубликованному свойству `ConnectKide` значение `ckAttachToInterface`.
2. Для доступа к объекту **Worksheet** использовать метод `ConnectTo` компонента **TExcelWorksheet**.

Синтаксис метода `ConnectTo` компонента **TExcelWorksheet** следующий:

```
procedure ConnectTo(svrIntf: _Worksheet):
```

В качестве параметра этому методу передается указатель на интерфейс `_Worksheet` сервера, с которым устанавливается соединение. Обратите внимание на то, что параметр `svrIntf` имеет тип `_Worksheet`, а не `IDispatch`. Однако благодаря тому, что интерфейс `_Worksheet` является наследником `IDispatch`, можно использовать оператор приведения типов `as`, который работает с интерфейсами точно так же, как с классами.

ПРИМЕЧАНИЕ

Метод `ConnectTo` можно задействовать для работы с любыми объектами MS Office, имеющими соответствующие VCL-компоненты, например при использовании объекта **TExcelWorkbook**.

Рассмотрим небольшой пример использования метода `ConnectTo`:

```
// Устанавливаем соединение с сервером Excel  
ExcelApplication1.Connect:  
// Создаем новую рабочую книгу  
ExcelApplication1.Workbooks.Add(EmptyParam.0);
```



```
// Делаем окно Excel видимым
ExcelApplication1.Visible[0]:=true;
// Подключаем компонент ExcelWorkbook1 к
// активной рабочей книге
ExcelWorkbook1.ConnectTo(
  ExcelApplication1.ActiveWorkbook);
// Создаем новый рабочий лист
ExcelWorkbook1.Worksheets.Add(EmptyParam, EmptyParam,
  EmptyParam, EmptyParam, 0);
// Подключаем компонент ExcelWorksheet1 к первому
// рабочему листу
index:=1;
ExcelWorksheet1.ConnectTo(
  ExcelWorkbook1.Worksheets.Item[index] as _Worksheet);
```

ПРИМЕЧАНИЕ

Нумерация рабочих листов в Excel ведется с 1.

Рассмотрим основные свойства объекта **Worksheet**:

- ☐ **Cells: Range** — содержит все ячейки данного рабочего листа;
- ☐ **Columns: Range** — содержит все столбцы данного рабочего листа;
- ☐ **Rows: Range** — содержит все строки данного рабочего листа;
- ☐ **Range[Cell1: OleVariant; Cell1: OleVariant]: Range** — это свойство является векторным и представляет ячейку или диапазон ячеек.

Из методов объекта **Worksheet** наиболее важным является метод **Activate**, который делает рабочий лист активным:

```
procedure Activate(Icid: integer);
```

Работа с ячейками

Основным объектом, используемым при работе с ячейками рабочего листа MS Excel, является объект **Range**. Этот объект позволяет изменять все основные атрибуты ячеек, включая граничные линии, шрифт, значения и формулы, а также выполнять ряд других операций.

Наиболее интересными для нас свойствами объекта **Range** являются те, что определяют содержимое ячейки. В первую очередь к ним относятся два следующих свойства:

- ☐ **Value: OleVariant** — определяет содержимое ячейки (если ячейка пустая, то данное свойство имеет значение **EmptyParam**) и может использоваться как для считывания значения из ячейки, так и для задания нового значения;
- ☐ **Formula: OleVariant** — определяет формулу, по которой рассчитывается содержимое данной ячейки.

Объект **Range** обладает большим количеством методов, из которых мы коснемся наиболее важных.

```
function Delete(Shift: OleVariant): OleVariant;
```

Удаляет ячейки, определяемые объектом **Range**. Параметр **Shift** является необязательным и задает способ сдвига ячеек в процессе замены удаленных ячеек. Он может принимать следующие значения:

■ `xlShiftToLeft` — ячейки сдвигаются влево;

■ `xlShiftUp` — ячейки сдвигаются вверх.

```
function Insert(Shift: OleVariant): OleVariant;
```

Вставляет ячейку или диапазон ячеек в рабочий лист. Другие ячейки при этом сдвигаются одним из способов, который определяется аргументом `Shift`:

■ `xlShiftToRight` — ячейки сдвигаются вправо;

■ `xlShiftDown` — ячейки сдвигаются вниз.

```
function Select: OleVariant;
```

Выделяет ячейки, содержащиеся в данном диапазоне `Range`.

```
procedure Merge(Across: OleVariant);
```

Объединяет все ячейки данного диапазона в одну. Параметр `Across` задает способ объединения. Если он принимает значение `true`, то ячейки объединяются построчно, то есть каждая строка диапазона объединяется в одну ячейку; если же он равен `false`, то все ячейки диапазона объединяются в одну. Ячейка, получающаяся в результате объединения, будет содержать значение ячейки, расположенной в левом верхнем углу диапазона.

Пример создания табличного отчета в Excel

Рассмотренные возможности MS Excel можно использовать для построения табличных отчетов. Для этого следует просто выполнить необходимую выборку данных и передать результаты этой выборки в MS Excel.

Рассмотрим небольшой пример. Передадим в Excel сведения, содержащиеся в таблице Сотрудники базы данных `sales` (в формате Access). Чтобы не усложнять пример, будем полагать, что в отчет требуется вывести всю информацию, содержащуюся в таблице.

1. Поместите на форму любой компонент доступа к данным (`Table`, `Query`, `ADOTable` или `ADOQuery`) и подключите его к требуемой таблице.

Для баз данных Access лучше использовать компоненты, работающие с базой данных с применением технологии ADO. А поскольку мы хотим выводить в Excel отчеты, то желательно задействовать компоненты, взаимодействующие с базой данных посредством SQL-запросов. Поэтому в нашем случае оптимальным выбором будет компонент `ADOQuery`.

2. Так как мы собираемся импортировать в Excel всю информацию из этой таблицы, то напишите такой запрос на выборку:

```
select *  
from Сотрудники
```

3. Поместите на форму кнопку (`Button`) и задайте для нее обработчик события `OnClick`, представленный в листинге 17.2.

Листинг 17.2. Обработчик события `OnClick` кнопки

```
procedure TForm1.Button1Click(Sender: TObject);  
const  
// Массив, который будет использоваться для задания
```

```
// имен ячеек при занесении в них данных
CellName : array[0..11] of char = ('A','B',
    'C','D','E','F','G','H','I','J','K','L');
var
    index : OleVariant;
    C1, V : OleVariant;
    i, j : Integer;
begin
    // Подключаемся к серверу автоматизации MS Excel
    ExcelApplication1.Connect;
    // Создаем новую рабочую книгу и делаем ее видимой
    ExcelApplication1.Workbooks.Add(EmptyParam,0);
    ExcelApplication1.Visible[0]:=true;
    ExcelWorkbook1.ConnectTo(
        ExcelApplication1.ActiveWorkbook);
    index:=1;
    ExcelWorksheet1.ConnectTo(
        ExcelWorkbook1.Worksheets.Item[index] as _Worksheet);
    ExcelWorksheet1.Activate(0);
    // Открываем набор данных
    ADOQuery1.Open;
    // Переносим данные в рабочий лист Excel
    try
        for i:=1 to ADOQuery1.RecordCount do begin
            ADOQuery1.RecNo:=i;
            for j:=0 to pred(ADOQuery1.FieldCount) do begin
                C1:=CellName[j]+IntToStr(i);
                V:=ADOQuery1.Fields[j].Value;
                ExcelWorksheet1.Range[C1,C1].Value:=V;
            end;
        end;
    finally
        // Закрываем набор данных
        ADOQuery1.Close;
    end;
end;
```

Работа с диаграммами MS Excel

MS Excel позволяет работать не только с данными, представленными в табличной форме, но и с диаграммами. Эта возможность является, пожалуй, наиболее интересной из всех.

Диаграммы MS Excel могут отображаться как на отдельных листах, так и внедряться в обычные рабочие листы. В первом случае все диаграммы рабочей книги содержатся в коллекции Charts объекта Workbook. Во втором случае диаграммы, внедренные в рабочий лист, содержатся в коллекции ChartObjects объекта Worksheet.

В Delphi для работы с диаграммами Excel имеется специальный компонент TExcelChart. Поскольку элементы коллекций Charts и ChartObjects представлены указателями на интерфейс IDispatch объекта Chart, то при работе с диаграммами из приложения, разработанного в Delphi с помощью компонента TExcelChart, следует использовать метод ConnectTo этого компонента.

Работа с коллекцией Charts практически полностью аналогична работе с коллекцией Worksheets. Так, например, фрагмент кода для создания новой диаграммы будет выглядеть следующим образом:

```
ExcelApplication1.Connect;
ExcelApplication1.Workbooks.Add(EmptyParam,0);
ExcelApplication1.Visible[0]:=true;
ExcelWorkbook1.ConnectTo(
  ExcelApplication1.ActiveWorkbook);
ExcelWorkbook1.Charts.Add(EmptyParam,EmptyParam,
  EmptyParam,EmptyParam,0);
index:=1;
ExcelChart1.ConnectTo(
  ExcelWorkbook1.Charts.Item[index] as _Chart);
ExcelChart1.Activate(0);
```

Как вы можете заметить, данный фрагмент отличается от аналогичного фрагмента, с помощью которого создается новый рабочий лист, только именами объектов (Charts вместо Worksheets) и интерфейсов (_Chart вместо _Worksheet).

Наиболее интересными свойствами объекта Charts (TExcelChart) являются свойства, управляющие внешним видом диаграммы.

- ☐ Legend: Legend — ссылка на объект Legend, который определяет параметры «легенды» диаграммы.
- ☐ ChartTitle: ChartTitle — ссылка на объект ChartTitle, представляющий заголовок диаграммы.
- ☐ ChartType: ToleEnum — тип диаграммы (константы, с помощью которых задается это свойство, для создания основных типов диаграмм приведены в табл. 17.6).
- ☐ HasAxis[index1: OleVariant; index2: OleVariant; Icid: integer] — тип осей на диаграмме. Параметр index1, задающий тип осей, может принимать одно из следующих значений:
 - xlCategory — ось категорий;
 - xlValue — ось значений;
 - xlSeriesAxis — ось рядов данных.

Последнее значение имеет смысл только для трехмерных диаграмм. Параметр index2 определяет группу осей и может принимать одно из двух значений: xlPrimary или xlSecondary. Он имеет смысл только для двухмерных диаграмм.

- ☐ HasDataTable: WordBool — определяет наличие (true) или отсутствие (false) таблицы данных на диаграмме.
- ☐ HasLegend[Icid: integer]: WordBool — определяет наличие (true) или отсутствие (false) «легенды» на диаграмме.
- ☐ HasTitle[Icid: integer]: WordBool — определяет наличие (true) или отсутствие (false) заголовка и осей диаграммы.

Таблица 17.6. Константы ChartType для основных типов диаграмм

Тип	Подтип	Константа
Гистограмма	Обычная	xlColumnClustered
	Объемная	xl3DColumnClustered
	Трехмерная	xl3DColumn

Тип	Подтип	Константа
Гистограмма	С накоплением	xlColumnStacked
	Объемная с накоплением	xl3DColumnStacked
	Нормированная к 100 %	xlColumnStacked100
	Объемная нормированная	xl3DColumnStacked100
Линейчатая	Обычная	xlBarClustered
	С накоплением	xlBarStacked
	Нормированная к 100 %	xlBarStacked100
	Объемная.	xl3DBarClustered
	Объемная с накоплением	xl3DBarStacked
	Объемная нормированная к 100 %	xl3DBarStacked100
График	Обычный	xlLine
	С маркерами	xlLineMarkers
	Объемный	xl3DLine
	С накоплением	xlLineStacked
	С накоплением и маркерами	xlLineMarkersStacked
	Нормированный к 100 %	xlLineStacked100
	Нормированный с маркерами	xlLineMarkersStacked100
Круговая	Обычная	xlPie
	Разрезанная	xlPieExploded
	Объемная	xl3DPie
	Объемная разрезанная	xl3DPieExploded
	Вторичная	xlPieOfPie
	Вторичная гистограмма	xlBarOfPie

Из всех методов объекта Chart мы рассмотрим только один — ChartWizard, который выполняет построение диаграммы с заданными параметрами. В наиболее полном виде синтаксис этого метода выглядит следующим образом:

```
procedure ChartWizard(Source, Gallery, Format, PlotBy,
  CategoryLabels, SeriesLabels, HasLegend, Title,
  CategoryTitle, ValueTitle, ExtraTitle: OleVariant);
```

- ☐ Все параметры данного метода являются необязательными.
- ☐ Source — диапазон ячеек, содержимое которых является источником данных для создаваемой диаграммы. В случае отсутствия данного параметра в качестве диаграммы используется активный лист диаграммы или выделенная диаграмма на активном рабочем листе.
- ☐ Gallery — тип диаграммы. Значение этого аргумента может задаваться одной из следующих констант: xlArea, xlBar, xlColumn, xlLine, xlPie, xlRadar, xlXYScatter, xlCombination, xl3DArea, xl3DBar, xl3DColumn, xl3DLine, xl3DPie, xl3DSurface, xlDoughnut или xlDefaultAutoFormat.

- ❑ **Format** — формат (вид) диаграммы. В зависимости от типа диаграммы значение данного параметра лежит в диапазоне от 1 до 10. Используемое по умолчанию значение зависит от типа диаграммы и источника данных.
- ❑ **PlotBy** — место расположения данных для рядов данных (в строках или столбцах). Задается с помощью констант `xlRows` и `xlColumns`.
- ❑ **CategoryLabels** — число строк или столбцов источника данных, которые содержат подписи категорий. Допустимые значения данного параметра лежат в диапазоне от 0 до значения, равного максимальному количеству категорий или рядов минус единица.
- ❑ **SeriesLabels** — целое число, определяющее количество строк или столбцов источника данных, которые содержат подписи рядов. Допустимые значения данного параметра лежат в диапазоне от 0 до значения, равного максимальному количеству категорий или рядов минус единица.
- ❑ **HasLegend** — определяет, отображать (`true`) или нет (`false`) «легенду».
- ❑ **Title** — текст заголовка диаграммы.
- ❑ **CategoryTitle** — текст заголовка оси категорий.
- ❑ **ValueTitle** — текст заголовка оси значений.
- ❑ **ExtraTitle** — текст заголовка оси рядов данных для трехмерных диаграмм или текст заголовка второй оси значений для двухмерных диаграмм.

Рассмотрим пример построения диаграммы в MS Excel из приложения, разработанного в Delphi. Проще всего проводить построение диаграммы в два этапа.

1. Передать в Excel данные, на основе которых строится диаграмма.
2. Вызвать метод `ChartWizard` и построить диаграмму требуемого типа.

Пример построения гистограммы

Для примера построим гистограмму распределения клиентов по странам. Так же как и в предыдущем примере, для соединения с базой данных будем использовать компонент `ADOQuery`, а построение диаграммы выполнять после щелчка на кнопке, размещенной на форме.

1. Поместите на форму компонент доступа к данным `ADOQuery` и подключите его к требуемой базе данных. Запрос на выборку будет выглядеть следующим образом:

```
SELECT Страна, COUNT([Код клиента]) AS [Количество клиентов]
FROM Клиенты
GROUP BY Страна
```

2. Поместите на форму кнопку (`Button`) и задайте для нее обработчик события `OnClick`, представленный в листинге 17.3.

Листинг 17.3. Обработчик события `OnClick` кнопки

```
procedure TForm1.Button1Click(Sender: TObject);
const
  CellName : array[0..11] of char = ('A','B','C','D','E',
    'F','G','H','I','J','K','L');
var
  index : OleVariant;
```

```

C1. V : OleVariant;
i,j,N : Integer;
begin
// Подключаемся к серверу автоматизации MS Excel
ExcelApplication1.Connect;
// Создаем новую рабочую книгу и делаем ее видимой
ExcelApplication1.Workbooks.Add(EmptyParam,0);
ExcelApplication1.Visible[0]:=true;
ExcelWorkbook1.ConnectTo(
    ExcelApplication1.ActiveWorkbook);
// Создаем новый рабочий лист
ExcelWorkbook1.Worksheets.Add(EmptyParam,EmptyParam,
    EmptyParam,EmptyParam,0);
index:=1;
ExcelWorksheet1.ConnectTo(
    ExcelWorkbook1.Worksheets.Item[index] as _Worksheet);
ExcelWorksheet1.Activate(0);
// Передаем данные в созданный рабочий лист
ADOQuery1.Open;
N:=ADOQuery1.RecordCount;
try
    for i:=1 to N do begin
        ADOQuery1.RecNo:=i;
        for j:=0 to pred(ADOQuery1.FieldCount) do begin
            C1:=CellName[j]+IntToStr(i);
            V:=ADOQuery1.Fields[j].Value;
            ExcelWorksheet1.Range[C1,C1].Value:=V;
        end;
    end;
finally
    ADOQuery1.Close;
end;
// Создаем новую диаграмму на отдельном листе
ExcelWorkbook1.Charts.Add(EmptyParam,EmptyParam,
    EmptyParam,EmptyParam,0);
index:=1;
ExcelChart1.ConnectTo(
    ExcelWorkbook1.Charts.Item[index] as _Chart);
ExcelChart1.Activate(0);
// Создаем индекс для выбора всех данных с рабочего
// листа, полученных из базы данных
C1:='A1:B'+IntToStr(N);
// Строим диаграмму
ExcelChart1.ChartWizard(
    ExcelWorksheet1.Range[C1,EmptyParam],
    xlColumn,EmptyParam,xlColumns);
end;

```

Взаимодействие с MS PowerPoint

Приложение PowerPoint позволяет представлять различного рода данные в виде презентаций, состоящих из последовательности слайдов, которые могут демонстрироваться на экране или распечатываться. Презентации могут использоваться, например, лектором для демонстрации материала слушателям.

По сравнению с такими приложениями, как Word или Excel, приложение PowerPoint значительно менее популярно, поэтому мы лишь кратко рассмотрим вопросы взаимодействия с ним, тем более что область применения PowerPoint в информационных системах пока еще достаточно ограничена.

Так же как остальные приложения MS Office, PowerPoint имеет сложную разветвленную структуру объектов (рис. 17.4). Причем, как и в предыдущих случаях, основными являются лишь несколько из них. В случае PowerPoint можно выделить следующие главные объекты:

- ☐ Application — как и для остальных приложений MS Office, основу PowerPoint составляет данный объект, представляющий само приложение;
- ☐ Presentation — презентация PowerPoint;
- ☐ Slide — слайд презентации (в PowerPoint презентация состоит из набора слайдов).

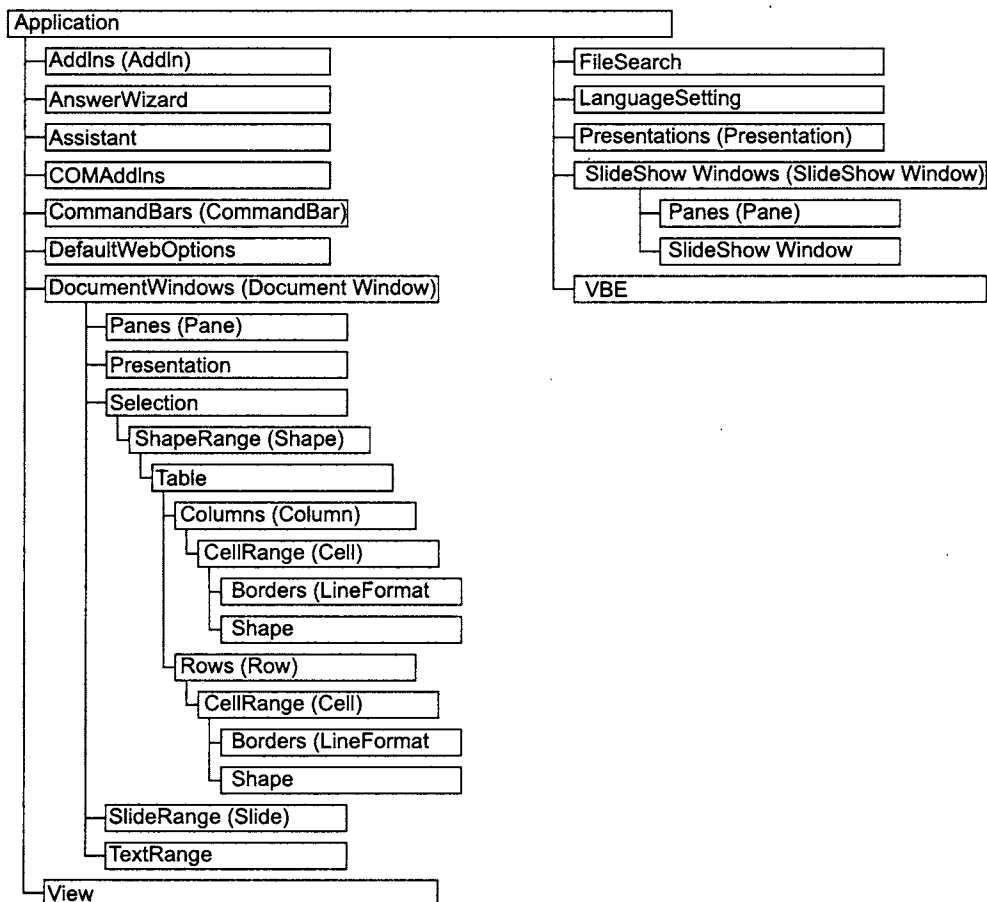


Рис. 17.4. Структура объектов MS PowerPoint

Объекты Presentation и Slide в приложении PowerPoint объединены в коллекции Presentations и Slides соответственно. Коллекция Presentations содержит все презентации, открытые в данный момент. В коллекции Slides находятся все слайды, относящиеся к некоторой конкретной презентации.

Для работы с основными объектами PowerPoint в VCL Delphi имеются специальные компоненты — OLE-серверы:

- ☐ TPowerPointApplication — соответствует объекту Application;
- ☐ TPowerPointPresentation — соответствует объекту Presentation;
- ☐ TPowerPointSlide — соответствует объекту Slide.

Так же как и остальные компоненты OLE-серверов, данные три компонента расположены на странице Servers палитры компонентов IDE Delphi.

Объект Application

Методы и свойства объекта Application в значительной степени повторяют методы и свойства аналогичных объектов Word и Excel. Наибольший интерес представляют следующие методы:

- ☐ Activate — активизирует приложение;
- ☐ Help — вызывает справочную систему;
- ☐ Quit — завершает работу с PowerPoint.

Выполнение последнего метода аналогично выполнению команды **Файл ► Выход**. Если сделанные в процессе редактирования презентации изменения не были сохранены, то перед выходом из приложения будет выдан запрос на сохранение изменений.

Из свойств объекта Application наиболее полезно свойство Visible, управляющее видимостью приложения и полностью аналогичное соответствующему свойству для Word и Excel.

Как уже отмечалось, для работы с объектом Application в приложениях Delphi используется компонент TPowerPointApplication. Для соединения с сервером PowerPoint используется метод Connect данного компонента.

Коллекция Presentations

Коллекция Presentations является одним из свойств объекта Application и содержит все открытые презентации. Для работы с этой коллекцией применяются обычные методы, практически полностью аналогичные тем, которые используются для работы с коллекциями Documents в MS Word и Workbooks в MS Excel. Основными из них являются методы, предназначенные для создания новой презентации и для открытия ранее созданной презентации.

```
function Add(WithWindow: TOleEnum): _Presentation;
```

Создает новую презентацию. Необязательный параметр WithWindow определяет, создается презентация в видимом окне (true) или нет (false). По умолчанию используется значение true.

```
function Open(FileName: WideString;  
  ReadOnly, Untitled, WithWindow: ToleEnum): _Presentation;
```

Открывает ранее созданную презентацию. Параметр `FileName` является обязательным и задает имя открываемого файла. Остальные параметры необязательны:

- **ReadOnly** — если задано значение `true`, позволяет открыть файл только для чтения (по умолчанию используется значение `false`);
- **Untitled** — позволяет открыть файл без названия, что равноценно созданию копии файла (если этот аргумент отсутствует, то имя файла автоматически становится названием открытой презентации);
- **WithWindow** — имеет то же назначение что и для метода `Add`.

```
function Item(index: OleVariant): _Presentation;
```

Обеспечивает доступ к элементу коллекции с порядковым номером или именем `index`.

Объект Presentation

Объект `Presentation` является элементом коллекции `Presentations`. Для работы с данным объектом в приложениях Delphi используется специальный компонент `TPowerPointPresentation`. Подключение компонента `TPowerPointPresentation` к элементу коллекции `Presentations` может быть выполнено с помощью следующего фрагмента кода:

```
var  
  index: OleVariant;  
...  
index:=1;  
PowerPointPresentation1.ConnectTo(  
  PowerPointApplication1.Presentations.Item(index));
```

При этом опубликованному свойству `ConnectKind` компонента `PowerPointPresentation1` должно быть присвоено значение `ckAttachToInterface`.

Объект `Presentation` обладает рядом свойств, которые являются ссылками на соответствующие объекты, встроенные в объект `Presentation`. Кратко рассмотрим главные из них:

- ☐ **ColorSchemes** — коллекция цветовых схем презентации;
- ☐ **Fonts** — коллекция, элементами которой являются все шрифты (объекты `Font`), используемые в презентации;
- ☐ **PageSetup** — объект, который позволяет задать различные атрибуты слайда презентации;
- ☐ **Slides** — коллекция всех слайдов презентации;
- ☐ **SlideShowSettings** — объект, который позволяет задать параметры показа слайдов презентации.

Ниже перечислены основные методы объекта `Presentation`.

```
procedure ApplyTemplate(const FileName: WideString);
```

Применяет к заданной презентации шаблон оформления `FileName`.

procedure Save;

Сохраняет презентацию.

procedure SaveAs(const Filename: WideString;
FileFormat, EmbedFonts: TOLEEnum);

Сохраняет презентацию под другим именем и/или в другом формате. Данный метод использует следующие параметры:

- **FileName** — имя файла презентации;
- **FileFormat** — формат файла;
- **EmbedFonts** — указывает, встраивать ли в файл презентации шрифты TrueType.

procedure Close;

Закрывает презентацию. При вызове этого метода презентация будет закрыта без предупреждения, если даже с момента последнего сохранения в нее были внесены изменения.

Коллекция Slides

Каждый из элементов коллекции **Slides** представляет собой конкретный слайд презентации. Методы работы с коллекцией являются обычными для коллекций MS Office.

function Add(Index: integer; Layout: TOLEEnum): _Slide;

Создает новый слайд и добавляет его в коллекцию слайдов данной презентации. Возвращает указатель на интерфейс **_Slide** созданного слайда. Оба параметра метода являются обязательными. Параметр **Index** определяет порядковый номер создаваемого слайда в коллекции **Slides**. Параметр **Layout** задает разметку создаваемого слайда.

function Paste(Index: integer): SlideRange;

Вставляет слайд из буфера обмена в коллекцию **Slides** данной презентации. Параметр **Index** является необязательным и определяет порядковый номер слайда, перед которым вставляется слайд из буфера обмена. Если этот аргумент не задан, то слайд помещается в конец коллекции.

function Range(Index: OleVariant): SlideRange;

Возвращает объект **SlideRange**, который позволяет работать с диапазоном слайдов. Необязательный параметр **Index** определяет слайд, включаемый в заданный диапазон. Значением данного свойства может являться целое число, которое задает порядковый номер слайда в коллекции, или строка, задающая имя слайда. Наряду с этим может быть задан массив целых чисел или строк, которые определяют слайды, включаемые в диапазон. Если этот аргумент отсутствует, то в диапазон включаются все слайды данной презентации.

function Item(Index: OleVariant): _Slide;

Возвращает элемент коллекции **Slides** с порядковым номером или именем, заданным с помощью параметра **Index**.

Объект Slide

Объект **Slide** представляет собой слайд презентации. Для работы с ним при создании приложений в Delphi используется компонент **PowerPointSlide**. Объект **Slide**

обладает большим количеством свойств, основными из которых являются следующие:

- ☐ **Shapes** — коллекция, которая содержит все рисованные объекты, имеющиеся в слайде или диапазоне слайдов;
- ☐ **Background** — объект **ShapeRange**, который содержит фон слайда (для задания какого-либо значения свойству **Background** следует сначала присвоить значение **False** свойству **FollowMasterBackground**);
- ☐ **ColorScheme** — объект, который представляет собой цветовую схему слайда;
- ☐ **HeadersFooters** — коллекция **HeadersFooters**, которая содержит заголовки, нижние колонтитулы, номер, дату и время создания слайда, мастера автосодержания или диапазона слайдов;
- ☐ **Name** — имя слайда (при добавлении слайда в презентацию ему автоматически присваивается имя Слайд N, где N — целое число, определяемое порядком добавления слайдов);
- ☐ **Master** — объект, который представляет собой мастер автосодержания;
- ☐ **Layout** — определяет разметку слайда.

Ниже перечислены основные методы объекта **Slide**.

procedure Copy;

Копирует слайд в буфер обмена.

procedure Delete;

Удаляет слайд из коллекции.

procedure Export(FileName, FilterName: WideString;

ScaleWidth, ScaleHeight: integer);

Сохраняет слайд в одном из графических форматов. Параметры метода:

■ **FileName** — имя файла;

■ **FilterName** — графический формат, в котором будет сохранен слайд;

■ **ScaleWidth** и **ScaleHeight** — необязательные параметры, которые позволяют задать высоту и ширину слайда в пикселах.

procedure Select;

Выделяет слайд.

ГЛАВА 18 Создание COM-объектов и элементов ActiveX

Одной из актуальных задач, стоящих перед разработчиками программного обеспечения, является организация взаимодействия между отдельными программами. Платформа Microsoft .NET реализует технологию такого взаимодействия, коренным образом отличающуюся от существовавших ранее технологий. Решение, предлагаемое .NET, заключается в представлении базовых классов, с которыми можно работать на любом языке, поддерживаемом платформой .NET, в том числе Delphi 8.

Однако на момент написания данной книги платформа .NET была не слишком распространена, поэтому в этой и предыдущей главах описаны основы взаимодействия между программами с помощью технологии COM. Технология COM еще долгое время будет использоваться для создания информационных систем, несмотря на активно вытесняющую ее технологию .NET. Кроме того, знакомство с технологией COM полезно ввиду того, что многие применяемые в .NET идеи являются развитием COM.

В рамках одной главы невозможно привести полное описание этой технологии и базирующихся на ней технологий OLE Automation и ActiveX (это тема для отдельной книги), поэтому мы дадим лишь краткое описание основных понятий, связанных с данными технологиями, и рассмотрим примеры создания COM-объектов, объектов автоматизации и элементов ActiveX в Delphi 7.

Основы технологии COM

Модель COM (Component Object Model — модель составных объектов) является независимой от языка программирования спецификацией, которая базируется на объектах. Данная спецификация определяет способ построения составных объектов и предоставления доступа к ним.

ПРИМЕЧАНИЕ

Не следует путать COM-объекты и объекты языка Object Pascal. Хотя они имеют много общего, но также обладают и существенными отличиями.

Составной объект (component object) является экземпляром некоторого класса и поддерживает определенное количество интерфейсов, обычно не менее двух. Интерфейсы используются для получения клиентом доступа к службам объекта. Методом интерфейса является функция или процедура, которая выполняет некоторое действие и может быть вызвана из приложения, применяющего данный объект (такое приложение обычно называется *клиентом объекта*). Клиенты могут получить доступ к службам COM-объекта только путем вызова методов интерфейсов объекта и не имеют непосредственного доступа к данным объекта. Каждому интерфейсу присваивается уникальный идентификатор.

COM-серверы

COM-сервер обычно представляет собой исполняемый файл (или динамическую библиотеку), содержащий, по крайней мере, один COM-объект. Различают три типа серверов:

- ❑ *внутренние* (in-process) *серверы* являются динамическими библиотеками, подключенными к приложению-клиенту и работающие с ним в одном адресном пространстве;
- ❑ *локальные* (local) *серверы* представляют собой отдельные приложения, работающие на том же компьютере, что и приложение-клиент, но, в отличие от внутренних серверов, в собственном адресном пространстве, поэтому такие серверы также называют *внешними* (out-of-process).
- ❑ *удаленные* (remote) *серверы* являются приложениями, функционирующими на другом по отношению к клиенту компьютере.

COM-клиенты

Клиентом составного объекта обычно называется приложение, из которого осуществляется доступ к COM-объекту.

ПРИМЕЧАНИЕ

В качестве клиента может выступать не только приложение, но и динамическая библиотека.

В общем случае клиенту неизвестно, где находится COM-сервер. Обращение к серверу производится по имени COM-объекта. Поиск и запуск сервера производится средствами операционной системы Windows. В общем виде схема получения приложением-клиентом доступа к составному объекту выглядит следующим образом.

1. Клиент запрашивает составной объект.
2. Windows выполняет поиск сервера запрошенного объекта и, в случае если поиск оказывается успешным, выполняет запуск сервера и возвращает приложению-клиенту указатель на запрашиваемый интерфейс.

Информация о местоположении сервера находится в реестре Windows. Следовательно, для того чтобы COM-сервер можно было использовать, необходимо наличие соответствующей записи в реестре.

Идентификация COM-объектов

Каждый COM-интерфейс характеризуется двумя именами. Одно из них представляет собой просто строку символов. По соглашению строковые имена большинства COM-интерфейсов начинаются с буквы I (от слова «Interface»). Различные технологии, основанные на COM, определяют интерфейсы с разными именами, но все они обычно начинаются с буквы I и частично отражают в названии назначение интерфейса. Например, интерфейс, используемый для передачи данных от сервера клиенту, имеет строковое имя IDataObject.

Простые строковые имена интерфейсов удобны при выборе имен переменных и типов для указателей на интерфейсы. Однако они не годятся, когда клиент должен точно показать, какой именно интерфейс объекта ему нужен. Не исключена ситуация, когда два разных интерфейса разных объектов имеют одинаковые имена. В этом случае клиент при запросе указателя на интерфейс может получить неверный указатель. Поэтому для идентификации интерфейса системой Windows используется *глобальный уникальный идентификатор* (Globally Unique Identifier, GUID). Для составных объектов GUID называется *идентификатором класса* и для его обозначения используется аббревиатура CLSID (Class Identifier). GUID для интерфейса называется *идентификатором интерфейса* и обозначается с аббревиатурой IID (Interface Identifier).

Идентификатор GUID представляет собой 16-байтовую величину, которая генерируется автоматически при создании COM-объекта или интерфейса.

ПРИМЕЧАНИЕ

При генерации идентификатора GUID необходимо обеспечить его глобальную уникальность. Это достигается обеспечением уникальности генерируемой величины во времени и пространстве. Уникальность во времени обеспечивается за счет того, что каждый идентификатор GUID содержит метку времени, указывающую, когда он был создан, что гарантирует отличие друг от друга всех идентификаторов GUID, сгенерированных на данной машине. Для обеспечения уникальности в пространстве используется уникальный идентификатор компьютера. В качестве такого идентификатора программа генерации GUID обычно задействует адрес сетевой платы — уникальное значение, уже имеющееся на большинстве компьютеров. Если же на компьютере не установлен сетевой адаптер, то из различных случайных характеристик данной системы генерируется фиктивный идентификатор машины. Но и в этом случае маловероятно, что идентификаторы двух машин окажутся одинаковыми.

Интерфейс IUnknown

Базовым интерфейсом всех COM-объектов является интерфейс IUnknown. Каждый COM-объект должен поддерживать данный интерфейс, в противном случае он не будет COM-объектом. Интерфейс IUnknown имеет всего три метода: QueryInterface,

AddRef и **Release**. Так как все другие интерфейсы являются наследниками интерфейса **IUnknown**, его методы могут быть вызваны через любой из указателей на интерфейсы объекта. Тем не менее **IUnknown** является отдельным самостоятельным интерфейсом с собственным идентификатором **IID**, так что клиент может запросить указатель непосредственно на **IUnknown**. Рассмотрим его методы более подробно.

Метод **QueryInterface**

Метод **QueryInterface** возвращает указатель на интерфейс объекта, идентификатор **IID** которого передается в параметре данного метода. Если запрашивается интерфейс, который не поддерживается объектом, то метод **QueryInterface** возвращает **NULL**. Таким образом, имея указатель на один интерфейс, клиент может получить указатели на другие интерфейсы объекта, методы которых ему необходимо вызывать. Именно эта простая схема позволяет решить важную и сложную проблему контроля версий. Благодаря методу **QueryInterface** удастся устанавливать новые версии объектов с расширенными возможностями, не влияя на работу программ, использующих только прежние возможности. Кроме того, можно обеспечить работоспособность клиента, разработанного с учетом новых возможностей составного объекта, со старой версией COM-объекта.

Пример расширения функциональных возможностей COM-объекта

Рассмотрим небольшой пример. Допустим, имеется некий набор функций для простой статистической обработки данных, реализованный в виде COM-объекта и поддерживающий интерфейс **ISimpleStat**. После установки такого объекта на компьютер объект может использоваться любым приложением-клиентом. Чтобы получить доступ к службам этого объекта, клиент запрашивает указатель на **ISimpleStat**. Если затем модифицировать COM-объект таким образом, чтобы расширить его функциональность и добавить, например, возможности регрессионного анализа, то доступ к этим дополнительным возможностям будет реализован через другой интерфейс с названием, например, **IRegress**. Таким образом, новая версия объекта будет поддерживать как **ISimpleStat**, так и **IRegress**. После установки новой версии составного объекта все приложения-клиенты, разработанные под старую версию объекта, будут работать так же, как и раньше, так как в новой версии объекта интерфейс **ISimpleStat** поддерживается без изменений и, следовательно, клиент при получении указателя на него сможет успешно использовать его методы.

ПРИМЕЧАНИЕ

Технология COM запрещает вносить изменения в существующие интерфейсы.

То что новая версия объекта поддерживает еще и **IRegress**, совершенно неизвестно старой версии клиента, и, следовательно, он никогда не запросит у объекта указатель на этот интерфейс.

Предположим теперь, что на машине установлена новая версия приложения-клиента, поддерживающая работу с интерфейсом **IRegress**. Если на компьютере установлена новая версия COM-объекта, то новая версия клиента сможет получить

указатель на интерфейс IRegress и воспользоваться новыми функциями. Если же версия составного объекта не обновлена, то при запросе указателя на интерфейс IRegress клиент получит в ответ NULL. Достаточно учесть такую возможность при написании обновленной версии клиента, чтобы обеспечить совместимость новой версии клиента со старой версией COM-объекта. При этом, естественно, новые возможности, предоставляемые интерфейсом IRegress новой версии составного объекта, не смогут быть использованы в приложении-клиенте.

Пример расширения и изменения функциональных возможностей интерфейса

Рассмотрим теперь, как при использовании COM-объектов можно изменять или расширять функциональные возможности существующих интерфейсов. Как уже отмечалось, COM не разрешает изменять интерфейсы. Поэтому для повышения функциональности существующих интерфейсов (например, интерфейса ISimpleStat) разработчик составного объекта должен определить новый интерфейс (например, ISimpleStat2) и включить в него необходимые новые или измененные методы. Объект по-прежнему будет поддерживать старый интерфейс ISimpleStat, но теперь он также будет поддерживать и ISimpleStat2. Добавление в объект поддержки ISimpleStat2 ничем не отличается от добавления поддержки любого нового интерфейса. Как и в предыдущем случае с IRegress, клиенты, ничего не знающие о произошедшей модернизации, никогда не запросят указатель на ISimpleStat2, то есть будут продолжать использовать ISimpleStat точно так же, как и прежде.

Таким образом, технология COM позволяет легко проводить обновление версий COM-серверов, не нарушая при этом работы существующих клиентов. Кроме того, она также обеспечивает автоматическое использование клиентами функций новой версии (после обновления самих клиентов). Наличие метода QueryInterface и принцип неизменности COM-интерфейсов позволяют программным компонентам, разрабатываемым независимыми разработчиками, обновляться по отдельности и тем не менее продолжать нормальную совместную работу. Этот принцип является очень важным, и именно поэтому создатели COM иногда называют метод QueryInterface важнейшим элементом модели.

Методы AddRef и Release

Чтобы воспользоваться COM-объектом, клиент должен явно инициировать начало работы экземпляра этого объекта. При этом возникает вопрос: «Когда завершается работа объекта?». Кажущееся очевидным решение — возложить задачу завершения работы объекта на клиента, запустившего объект, является неприемлемым. Дело в том, что данный клиент может оказаться не единственным, кто использует этот объект. Поэтому вполне вероятна ситуация, когда клиент запускает объект, получает указатели на его интерфейсы и затем передает один из них другому клиенту. Последний может задействовать указатель для исполнения методов на том же самом объекте и в свою очередь передать указатель другим клиентам. Если бы первый клиент уничтожил экземпляр запущенного объекта, то работа остальных клиентов была бы нарушена.

Поскольку один объект может использоваться несколькими клиентами одновременно, причем никто из них не в состоянии узнать, когда все остальные завершатся, то разрешение клиенту самому уничтожить объект могло бы привести к сбою. Поэтому только сам объект может знать, когда появится возможность безопасно завершить работу. Этот момент настанет тогда, когда все клиенты сообщат объекту, что они завершили работу с ним. Подобный контроль объекты осуществляют с помощью механизма *подсчета ссылок* (reference counting), поддерживаемого двумя методами интерфейса IUnknown.

Каждый исполняющийся объект поддерживает счетчик ссылок. Выдавая указатель на один из своих интерфейсов, объект всякий раз увеличивает счетчик ссылок на 1.

ПРИМЕЧАНИЕ

Объект может поддерживать отдельные счетчики ссылок для каждого своего интерфейса.

Если один клиент передает другому клиенту указатель на интерфейс, то есть увеличивает число пользователей объекта без ведома последнего, то клиент, получающий указатель, должен вызвать метод AddRef полученного интерфейса. В результате объект увеличивает свой счетчик ссылок.

Независимо от того, как клиент получил указатель на интерфейс, при завершении работы с ним клиент всегда обязан вызвать метод Release. Исполнение этого метода объектом состоит в уменьшении счетчика ссылок на 1. Обычно объект уничтожает сам себя, когда счетчик ссылок становится равным 0.

ПРИМЕЧАНИЕ

Обеспечение правильности подсчета ссылок целиком лежит на программисте. Если не все клиенты следуют правилам подсчета ссылок, то экземпляр объекта может либо существовать неопределенно долго, либо, что еще хуже, может быть преждевременно удален. Поэтому при написании приложений-клиентов следует быть очень внимательным и обеспечивать правильный подсчет ссылок.

Библиотека COM

Для обеспечения работоспособности составных объектов в системе, поддерживающей технологию COM, обязательно имеется некоторая реализация библиотеки COM. Данная библиотека содержит функции, предоставляющие базовые службы объектам и их клиентам. Более того, библиотека предоставляет клиентам механизм запуска серверов объектов. Доступ к службам библиотеки COM осуществляется через вызовы обычных функций, а не через методы интерфейсов COM-объектов. По соглашению, имена функций библиотеки COM начинаются с префикса Co, (например, CoCreateInstance — одна из основных функций библиотеки COM, создающая экземпляр составного объекта).

Фабрика классов

Если клиенту нужен только один составной объект, то легче всего создать его с помощью метода CoCreateInstance. Однако в некоторых случаях клиенту может пона-

добиться несколько экземпляров объекта одного и того же класса. В этом случае следует использовать так называемую *фабрику классов* (class factory) — специальный объект, способный создавать экземпляры других объектов. Фабрики классов являются обычными COM-объектами, доступ к которым осуществляется через интерфейсы. Однако в отличие от обычных составных объектов, фабрика класса обязательно должна поддерживать интерфейс `IClassFactory`. В данном интерфейсе реализованы всего два метода.

- ❑ `CoCreateInstance` — создает новый экземпляр класса, объекты которого может создавать данная фабрика. При вызове этого метода клиент должен указать IID, чтобы получить ссылку на нужный ему интерфейс. Клиент не передает этому методу CLSID в качестве параметра, так как класс объекта неявно определяется самой фабрикой.
- ❑ `LockServer` — оставляет сервер загруженным в память после создания объекта. Так как фабрика классов является COM-объектом, то она поддерживает собственный счетчик ссылок для учета количества использующих ее клиентов. Однако в ряде случаев этого счетчика недостаточно, чтобы сохранить сервер загруженным в память. В этих случаях для гарантированного продолжения работы сервера и используется метод `LockServer`.

Для доступа к фабрике класса предназначен специальный метод `CoGetClassObject` библиотеки COM. Этой функции передается идентификатор CLSID класса объектов, которые будут создаваться фабрикой (но не CLSID самой фабрики!), а также IID интерфейса `IClassFactory`. Метод возвращает указатель на интерфейс. Затем путем вызова метода `CreateInstance` интерфейса `IClassFactory` создается экземпляр COM-объекта.

Создание COM-объектов в Delphi

Понятие объекта как в терминологии модели COM, так и в Delphi имеет практически одинаковый смысл. COM-интерфейс больше напоминает объект Delphi, у которого отсутствуют свойства и имеются лишь виртуальные методы. Список функций интерфейса соответствует виртуальной таблице методов `Object Pascal`. Создать COM-интерфейс можно средствами практически любого языка: достаточно лишь объявить объект с требуемым списком виртуальных методов. Причем задаваемые определения методов должны точно соответствовать определениям функций в самих интерфейсах. Однако из-за необходимости поддержки COM-объектом нескольких интерфейсов возникает проблема, связанная с отсутствием в `Object Pascal` возможности множественного наследования.

Класс `TComObj` и составные классы

Для решения отмеченной проблемы при создании составных объектов в Delphi используется следующий подход. COM-объект создается на основе обычного класса `Object Pascal`, который имеет имя `TComObject` и является непосредственным потомком `TObject`. Все свойства и методы COM-объекта описываются как свойства и методы обычного класса `Object Pascal`, производного от класса `TComObject`. При со-

здании COM-объекта одновременно создается еще один класс, который содержит описание всех интерфейсов COM-объекта. Данный класс называется *составным классом* (Component Class, CoClass) и ему присваивается имя COM-объекта с приставкой Co (например, при создании COM-объекта с именем TestObj будет создан составной класс с именем CoTestObj). Таким образом, класс-наследник TComObj содержит код объекта, а составной класс представляет собой описание экземпляра этого класса в соответствии со спецификацией COM.

Объявление и описание составного класса содержится в библиотеке типов, которая всегда создается при создании COM-объекта.

Для создания COM-объектов с помощью библиотек типов обычно используется класс TTypedComObject. Данный класс является непосредственным наследником класса TComObj.

Описание интерфейса в Delphi

Для описания интерфейса в Delphi используется ключевое слово `interface`. Описание интерфейса в целом подобно описанию класса. Например, описание интерфейса IUnknown выглядит в Delphi следующим образом:

```
IUnknown = interface
  ['{00000000-0000-0000-C000-000000000046}']
  function QueryInterface(const IID: TGUID; out Obj):
    integer; stdcall;
  function _AddRef: integer; stdcall;
  function _Release: integer; stdcall;
end;
```

Обычно код описания интерфейса генерируется автоматически на основании данных, задаваемых в редакторе библиотеки типов.

Создание внутреннего COM-сервера

Как уже отмечалось, внутренний COM-сервер является динамической библиотекой, содержащей в своем составе, по крайней мере, один COM-объект. Процедура создания внутреннего COM-сервера выглядит следующим образом.

1. Создание динамической библиотеки.
2. Создание COM-объекта.
3. Создание интерфейсов COM-объекта.
4. Программирование методов интерфейса.

Рассмотрим каждый из этапов.

Создание динамической библиотеки COM-сервера

Динамическая библиотека, которая является внутренним COM-сервером, создается в виде отдельного проекта Delphi. Для создания такой библиотеки следует выбрать команду `File ► New` в главном меню IDE Delphi и затем дважды щелкнуть на значке `ActiveX Library` на вкладке `ActiveX` открывшегося окна диалога `New Items` (рис. 18.1).

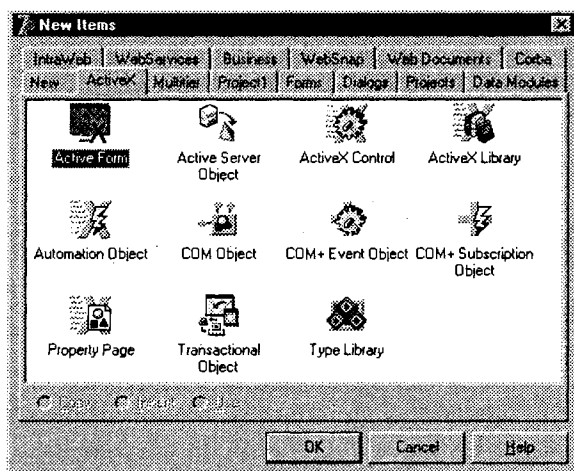


Рис. 18.1. Создание новой динамической библиотеки ActiveX

После этого будет создан новый проект, состоящий из одного файла, которому мы дадим имя `TestComServer.pas`. Данный файл содержит всего несколько строк кода:

```
library TestComServer;
```

```
uses
```

```
    ComServ;
```

```
exports
```

```
    DllGetClassObject,
```

```
    DllCanUnloadNow,
```

```
    DllRegisterServer,
```

```
    DllUnregisterServer;
```

```
{ $R *.RES }
```

```
begin
```

```
end.
```

В нашем примере внесения изменений в данный код не потребуется. Функции, экспортируемые данной библиотекой (включенные в раздел `exports`), обеспечивают ее взаимодействие с COM-объектом и регистрацию сервера в системном реестре.

Создание COM-объекта

После создания динамической библиотеки можно приступить к созданию COM-объекта. Для этого следует выбрать команду `File ► New` в главном меню IDE Delphi и щелчком выделить значок `COM Object` на вкладке `ActiveX` открывшегося окна `New Items` (см. рис. 18.1).

После щелчка на кнопке `OK` в окне `New Items` будет запущен мастер `COM Object Wizard` (рис. 18.2).

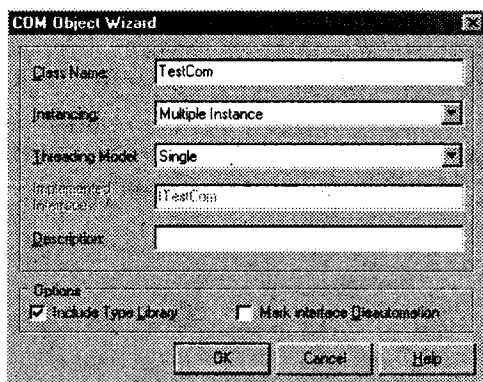


Рис. 18.2. Окно мастера настройки создаваемого COM-объекта

- ☐ Имя класса создаваемого COM-объекта вводится в поле Class Name.
- ☐ Способ создания экземпляров COM-объекта задается с помощью раскрывающегося списка Instancing. В данном списке предлагаются следующие варианты:
 - Internal — объект будет использоваться в процессе;
 - Single Instance — при обращении к объекту нескольких клиентов в общем экземпляре сервера создается несколько экземпляров объекта;
 - Multiple Instance — при обращении к объекту нескольких клиентов для каждого клиента создается отдельный экземпляр сервера.
- ☐ Способ взаимодействия COM-сервера с клиентами задается с помощью раскрывающегося списка Threading Model, в котором предлагаются следующие варианты:
 - Single — сервер обслуживает вызовы клиентов последовательно;
 - Apartment — вызов объекта выполняется только в одном потоке, созданном самим объектом, и сервер может поддерживать одновременную работу нескольких объектов;
 - Free — обращение к объекту может производиться через любой созданный им поток;
 - Both — комбинация вариантов Apartment и Free.
- ☐ Стандартные COM-интерфейсы, которые будут использоваться в объекте, указываются в поле Implemented Interfaces.
- ☐ Описание объекта вводится в поле Description.
- ☐ Флажок Include Type Library указывает, следует ли создавать библиотеку типов для COM-объекта.
- ☐ Установленный флажок Mark interface Oleautomation означает поддержку COM-объектом технологии OLE Automation.

В нашем примере мы зададим следующие параметры создаваемого объекта:

- ☐ имя класса — TestCom;
- ☐ способ создания экземпляров объекта — Multiple Instance;
- ☐ способ взаимодействия сервера и клиентов — Single.

Кроме того, укажем, что следует создать библиотеку типов (путем установки флажка **Include Type Library**) и не будем включать поддержку технологии OLE Automation (оставив сброшенным флажок **Mark interface Oleautomation**).

После щелчка на кнопке ОК будет создан модуль, содержащий код объекта **TestCom**, и откроется окно библиотеки типов. В модуле объекта изначально содержатся описание пустого класса объекта и вызов конструктора фабрики класса созданного объекта в секции инициализации модуля:

```
unit TestCOM;

interface

uses
  Windows, ActiveX, Classes, ComObj, TestComServer_TLB, StdVcl;

type
  TTestCom = class(TTypedComObject, ITestCom)
  protected
    {Declare ITestCom methods here}
  end;

implementation

uses ComServ;

initialization
  TTypedComObjectFactory.Create(ComServer, TTestCom, Class_TestCom,
    ciMultiInstance, tmSingle);
end.
```

Для создания новых интерфейсов и методов COM-объекта лучше всего воспользоваться редактором библиотеки типов.

Создание интерфейсов и методов COM-объекта

Обычно для создания библиотек типов используется специальный язык IDL (Interface Description Language — язык описания интерфейсов), синтаксис которого похож на синтаксис языка C++, однако в Delphi для этой цели обычно используется синтаксис языка Object Pascal.

При разработке библиотек типов в Delphi, как правило, не требуется писать код вручную, так как код библиотеки типов генерируется автоматически, а для внесения необходимых изменений и дополнений используется специальный редактор. Поэтому при разработке COM-объектов с помощью Delphi детальных знаний синтаксиса языка описания интерфейсов не нужно.

ПРИМЕЧАНИЕ

Код библиотеки типов, разработанной в Delphi на основе языка Object Pascal, может быть затем экспортирован в формат IDL.

Окно редактора библиотеки типов приведено на рис. 18.3. В левой части окна расположен иерархический список всех элементов, содержащихся в библиотеке типов, а справа находится своего рода «панель управления» — многостраничный блок-

нот, позволяющий изменять различные параметры элементов библиотеки типов. Вид данного многостраничного блокнота зависит от того, какой элемент является активным (выбран в списке слева). В верхней части окна расположена панель инструментов, содержащая ряд кнопок, с помощью которых создаются новые элементы библиотеки типов и выполняются некоторые общие для всей библиотеки действия.

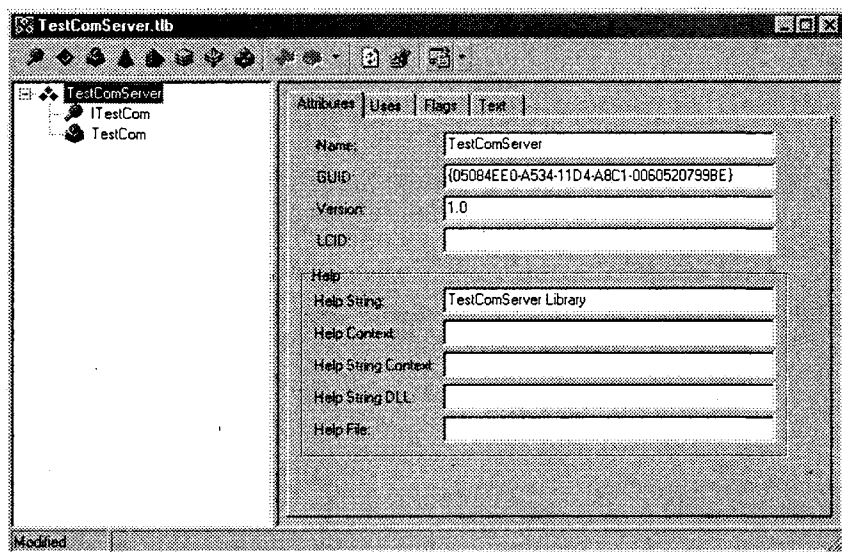


Рис. 18.3. Окно редактора библиотеки типов

Кратко поясним назначение кнопок панели инструментов редактора библиотеки типов. Весь набор кнопок этой панели инструментов разделен на четыре группы. В первую группу включены кнопки, предназначенные для создания различных объектов, включаемых в библиотеку типов:

- ☐ New Interface — создает новый интерфейс;
- ☐ New Disinterface — создает новый интерфейс диспетчеризации;
- ☐ New CoClass — создает новый составной класс для COM-объекта;
- ☐ New Enum — создает новое перечисление;
- ☐ New Alias — создает новый псевдоним типа данных;
- ☐ New Record — создает новую запись (список полей, не имеющих собственных идентификаторов);
- ☐ New Union — создает новое объединение (индексированный список полей);
- ☐ New Module — создает новый модуль.

Вторая группа включает в себя кнопки, предназначенные для создания новых элементов интерфейса:

- ☐ New Method — создает новый метод для выбранного интерфейса;
- ☐ New Property — создает новое свойство для выбранного интерфейса.

Кнопки данной группы активизируются только при выборе в списке какого-либо интерфейса COM-объекта.

Третья группа также содержит две кнопки:

- ☐ **Refresh** — обновляет исходный код библиотеки типов, внося в него изменения, сделанные в редакторе библиотеки типов;
- ☐ **Register** — регистрирует библиотеку типов в системном реестре.

Последняя, четвертая, группа состоит всего из одной кнопки **Export To IDL**, которая выполняет экспорт библиотеки типов в формат языка IDL.

Основные действия по созданию и редактированию библиотеки типов выполняются на страницах многостраничного «блокнота». Полное описание всех возможных вариантов настройки заняло бы слишком много места, поэтому мы рассмотрим лишь те параметры, которые необходимо задать при создании простейшего COM-объекта.

При установке в окне мастера создания COM-объекта флажка **Include Type Library** одновременно с созданием модуля объекта будет сгенерирован код библиотеки типов. Изначально он содержит описание только одного интерфейса и одного составного класса (в текст библиотеки типов также включаются обширные комментарии, которые мы здесь не приводим):

```
unit TestComServer_TLB;

{$TYPEDADDRESS OFF}

interface

uses Windows, ActiveX, Classes, Graphics, OleServer,
    OleCtrls, StdVCL;

const
    TestComServerMajorVersion = 1;
    TestComServerMinorVersion = 0;

    LIBID_TestComServer: TGUID = '{05084EE0-A534-11D4-A8C1-
        0060520799BE}';

    IID_ITestCom: TGUID = '{05084EE1-A534-11D4-A8C1-
        0060520799BE}';
    CLASS_TestCom: TGUID = '{05084EE3-A534-11D4-A8C1-
        0060520799BE}';

type

    ITestCom = interface;

    TestCom = ITestCom;

    ITestCom = interface(IUnknown)
    ['{05084EE1-A534-11D4-A8C1-0060520799BE}']
    end;

    CoTestCom = class
```

```
class function Create: ITestCom;  
class function CreateRemote(const MachineName:  
    string): ITestCom;  
end;  
  
implementation  
  
uses ComObj;  
  
class function CoTestCom.Create: ITestCom;  
begin  
    Result := CreateComObject(CLASS_TestCom) as ITestCom;  
end;  
  
class function CoTestCom.CreateRemote(const MachineName:  
    string): ITestCom;  
begin  
    Result := CreateRemoteComObject(MachineName,  
        CLASS_TestCom) as ITestCom;  
end;  
  
end.
```

Класс **CoTestCom** содержит две функции:

- ☐ **Create** — используется для создания COM-объекта на локальном компьютере;
- ☐ **CreateRemote** — используется для создания COM-объекта на удаленном компьютере.

Каждая из этих двух функций возвращает указатель на интерфейс **ITestCOM**.

После создания «пустого» COM-объекта для него необходимо написать требуемые интерфейсы и методы. Для создания интерфейсов и заголовков методов удобно использовать редактор библиотеки типов. Однако перед тем как приступить к созданию интерфейсов и методов, необходимо представлять, что требуется получить. Поэтому давайте определимся с функциями, выполняемыми нашим COM-объектом. Создадим COM-объект, предназначенный для выполнения простейших статистических расчетов. Пусть объект имеет два интерфейса: первый с набором функций для вычисления математического ожидания и дисперсии случайной величины; второй — с функциями для нахождения минимального и максимального значений случайной величины. Таким образом, каждый из двух интерфейсов объекта будет реализовывать по две функции.

Один интерфейс уже есть — это интерфейс **ITestCom**, второй необходимо создать.

1. Щелкните на кнопке **New Interface** панели инструментов редактора библиотеки типов. После этого будет создан новый интерфейс, который отобразится в иерархическом списке в левой части окна редактора.
2. Измените имя интерфейса. Это необходимо сделать для того, чтобы интерфейс имел осмысленное название (при создании ему по умолчанию присваивается имя **Interface1**). Изменить имя интерфейса можно либо прямо в иерархическом списке, либо в поле **Name** на вкладке **Attributes** многостраничного «блокнота» (при этом в иерархическом списке должен быть выделен вновь созданный интерфейс). В нашем примере зададим для второго интерфейса имя **IMinMax**.

Помимо имени интерфейса, на вкладке **Attributes** могут быть изменены и некоторые другие параметры интерфейса.

- ❑ Поле **GUID** содержит уникальный идентификатор интерфейса, который генерируется автоматически. Изменять его не следует.
- ❑ Поле **Version** позволяет задать номер версии.
- ❑ Раскрывающийся список **Parent Interface** используется для выбора родительского интерфейса. В нашем случае здесь следует выбрать интерфейс **IUnknown**.

На вкладке **Flags** многостраничного «блокнота» настройки интерфейса выполняется установка некоторых дополнительных параметров с помощью флажков. При создании простого COM-объекта все флажки на этой странице должны быть сброшены.

После создания второго интерфейса зададим для каждого из существующих интерфейсов необходимые методы.

1. Выберите нужный интерфейс в иерархическом списке.
2. Щелкните на кнопке **New Method** панели инструментов.
3. Задайте параметры созданного метода.

Остановимся на последнем шаге более подробно. Параметры метода задаются на вкладках многостраничного «блокнота». При выборе в иерархическом списке одного из методов интерфейса справа отображаются четыре вкладки. Наибольший интерес из них представляют две — **Attributes** и **Parameters**. Самым важным на вкладке **Attributes** является изменение имени метода. Для этого используется поле **Name**.

На вкладке **Parameters** указывается тип результата, возвращаемого методом (метод интерфейса обязательно является функцией), а также параметры, передаваемые в метод при вызове (параметры вызова функции). Тип результата задается с помощью раскрывающегося списка **Return Type** (рис. 18.4).

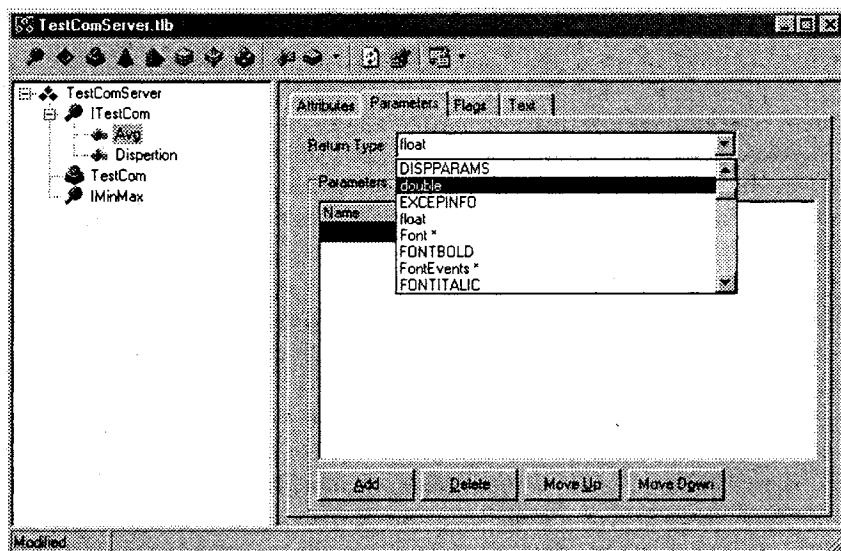


Рис. 18.4. Изменение типа результата, возвращаемого методом

Для задания параметров вызова метода используются список и кнопки, расположенные в нижней части вкладки **Parameters**. При щелчке на кнопке **Add** в список заносится новый параметр. После включения параметра в список можно изменять его свойства (рис. 18.5):

- ☐ в столбце **Name** задается имя параметра;
- ☐ в столбце **Type** указывается тип параметра;
- ☐ в столбце **Modifier** указываются некоторые дополнительные свойства, в том числе входной параметр или выходной, обязательный или нет и т. п. (может быть также задано значение параметра по умолчанию).

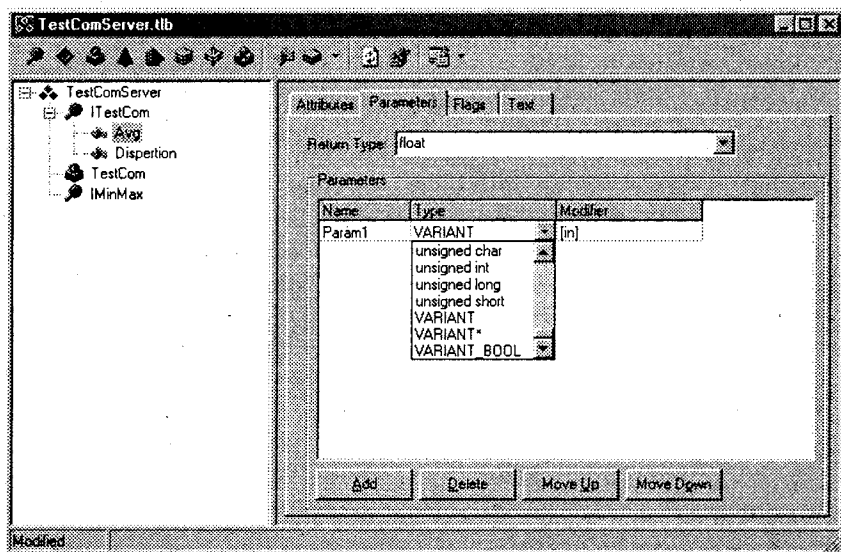


Рис. 18.5. Настройка параметров вызова метода

Для нашего примера необходимо создать по два метода для каждого интерфейса. Функции интерфейса **ITestCom** назовем **Avg** (для вычисления математического ожидания) и **Variance** (для вычисления дисперсии). Методам интерфейса **IMinMax** присвоим имена **Min** (для вычисления минимальной величины) и **Max** (для вычисления максимальной величины).

Тип результатов, возвращаемых всеми методами, должен быть вещественным. Поэтому для всех функций следует выбрать в списке **Return Type** пункт **float**. Параметры вызова всех методов также одинаковы — при вызове передается массив, содержащий значения случайной величины. В списке типов, которые можно задавать параметрам, тип массива отсутствует. Однако для передачи массива можно использовать вариантный тип. Поэтому для всех методов укажите по одному параметру типа **Variant**. Никакой другой настройки методов в нашем примере делать не требуется.

После создания дополнительных интерфейсов и задания параметров методов необходимо выполнить привязку вновь созданных интерфейсов к COM-объекту. Для

этого в иерархическом списке следует выбрать составной класс (в нашем примере это объект `TestCom`). Затем на вкладке `Implements` многостраничного «блокнота» с помощью команды `Insert Interface` контекстного меню надо добавить необходимый интерфейс (в нашем случае это интерфейс `IMinMax`).

Осталось только обновить исходный код библиотеки типов для внесения в него всех выполненных нами изменений. Для этого щелкните на кнопке `Refresh` панели инструментов редактора библиотеки типов.

Рассмотрим, какие изменения произошли в коде библиотеки типов в результате наших действий:

```
unit TestComServer_TLB;

{$TYPEDADDRESS OFF}

interface

uses Windows, ActiveX, Classes, Graphics, OleServer,
    OleCtrls, StdVCL;

const
    TestComServerMajorVersion = 1;
    TestComServerMinorVersion = 0;

LIBID_TestComServer: TGUID = '{05084EE0-A534-11D4-
    A8C1-0060518799BE}';

IID_ITestCom: TGUID = '{05084EE1-A534-11D4-A8C1-
    0060518799BE}';
IID_IMinMax: TGUID = '{1AD49218-A5FB-11D4-A8C1-
    0060518799BE}';
CLASS_TestCom: TGUID = '{05084EE3-A534-11D4-A8C1-
    0060518799BE}';

type

    ITestCom = interface;
    IMinMax = interface;

    TestCom = ITestCom;

    ITestCom = interface(IUnknown)
    ['{05084EE1-A534-11D4-A8C1-0060520799BE}']
    function Avg(Param1: OleVariant): Single; stdcall;
    function Variance(Param1: OleVariant): Single;
    stdcall;
    end;

    IMinMax = interface(IUnknown)
    ['{1AD49220-A5FB-11D4-A8C1-0060520799BE}']
    function Min(Param1: OleVariant): Single; stdcall;
    function Max(Param1: OleVariant): Single; stdcall;
    end;

    CoTestCom = class
    class function Create: ITestCom;
```

```

class function CreateRemote(const MachineName:
string): ITestCom;
end;

implementation

uses ComObj;

class function CoTestCom.Create: ITestCom;
begin
    Result := CreateComObject(CLASS_TestCom) as ITestCom;
end;

class function CoTestCom.CreateRemote(const MachineName:
string): ITestCom;
begin
    Result := CreateRemoteComObject(MachineName,
    CLASS_TestCom) as ITestCom;
end;

end.

```

Как видно из приведенного листинга, в библиотеку типов добавлено объявление еще одного интерфейса, и каждый интерфейс содержит по два объявления функций. Параметры и типы результатов функций соответствуют заданным в редакторе библиотеки типов.

При обновлении исходного кода путем щелчка на кнопке **Refresh** в редакторе библиотеки типов изменения вносятся не только в исходный код библиотеки (файл **TestComServer_TLB.pas**), но и в исходный код модуля COM-объекта:

```

unit TestCOM;

interface

uses

Windows, ActiveX, Classes, ComObj, TestComServer_TLB, StdVcl;

type
    TTestCom = class(TTypedComObject, ITestCom, IMinMax)
    protected
        function Avg(Param1: OleVariant): Single; stdcall;
        function Variance(Param1: OleVariant): Single;
        stdcall;
        function Max(Param1: OleVariant): Single; stdcall;
        function Min(Param1: OleVariant): Single; stdcall;
        {Declare ITestCom methods here}
    end;

implementation

uses ComServ;

function TTestCom.Avg(Param1: OleVariant): Single;
begin

end;

```

```
function TTestCom.Variance(Param1: OleVariant): Single;
begin
end;

function TTestCom.Max(Param1: OleVariant): Single;
begin
end;

function TTestCom.Min(Param1: OleVariant): Single;
begin
end;

initialization
  TTypedComObjectFactory.Create(ComServer, TTestCom,
    Class_TestCom, ciMultiInstance, tmSingle);
end.
```

Здесь в описание класса объекта **TTestCom** добавлены объявления созданных нами методов. Кроме того, в секции **Implementation** модуля **COM**-объекта включены шаблоны этих методов, так что нам остается только написать для каждого метода необходимый код.

ПРИМЕЧАНИЕ

В описании класса **TTestCom** никак не отмечена принадлежность его методов разным интерфейсам. Это разделение выполняется в библиотеке типов. В заголовке объявления класса указано, что данный класс имеет два интерфейса — **ITestCom** и **IMinMax**.

Программирование методов интерфейса

Программирование методов интерфейсов никаких особенностей не имеет. Достаточно просто написать необходимый текст между ключевыми словами **begin** и **end** для каждого из объявленных методов. Наибольшие сложности в нашем примере может вызвать лишь использование вариантных массивов. При разработке **COM**-объектов довольно часто применяют вариантные переменные. А передачу массива **COM**-объекту проще всего реализовать с типом **Variant**. Поэтому рассмотрим работу с вариантными массивами более подробно.

Для создания вариантного массива используются две функции.

```
function VarArrayCreate(const Bounds: array of Integer;
  varType: Integer): Variant;
```

Выделяет область памяти для вариантного массива и возвращает указатель на него. Параметр **Bounds** задает индексацию массива, **varType** — тип элементов массива.

```
function VarArrayOf(const Values: array of Variant): Variant;
```

Выделяет область памяти для вариантного массива и инициализирует созданный массив значениями, переданными в параметре **Values**.

При использовании вариантных массивов важно знать минимальное и максимальное значения его индексов. Для этого тоже определены две функции.

```
function VarArrayLowBound(const A: Variant; Dim: Integer): Integer;
```

Возвращает минимальное значение индекса измерения Dim массива A.

```
function VarArrayHighBound(const A: Variant; Dim: Integer): Integer;
```

Возвращает максимальное значение индекса измерения Dim массива A.

ПРИМЕЧАНИЕ

Вариантные массивы могут быть многомерными, причем размерность различных измерений в общем случае может быть разной. Поэтому функции VarArrayLowBound и VarArrayHighBound требуют указания измерения, для которого определяются границы индекса.

Нумерация измерений в вариантных массивах ведется с 1. Из этого следует, что для получения границ индекса одномерного массива параметр Dim должен принимать значение 1.

Набор действий, которые должны выполняться при вызове каждого из методов в рассматриваемом примере, достаточно прост: сначала необходимо определить пределы изменения индекса вариантного массива, а затем произвести тривиальные вычисления:

```
function TTestCom.Avg(Param1: OleVariant): Single;
```

```
var
  N1,N2,i : Integer;
  A : Single;
begin
  A:=0;
  N1:=varArrayLowBound(Param1,1);
  N2:=varArrayHighBound(Param1,1);
  for i:=N1 to N2 do
    A:=A+Param1[i];
  result:=A/(N2-N1+1);
end;
```

```
function TTestCom.Variance(Param1: OleVariant): Single;
```

```
var
  N1,N2,i : Integer;
  A,D : Single;
begin
  A:=Avg(Param1);
  D:=0;
  N1:=varArrayLowBound(Param1,1);
  N2:=varArrayHighBound(Param1,1);
  for i:=N1 to N2 do
    D:=sqr(Param1[i]-A);
  result:=D/(N2-N1);
end;
```

```
function TTestCom.Max(Param1: OleVariant): Single;
```

```
var
  N1,N2,i : Integer;
  M : Single;
begin
  N1:=varArrayLowBound(Param1,1);
  N2:=varArrayHighBound(Param1,1);
```



```
M:=Param1[N1];
for i:=N1 to N2 do
  if Param1[i]>M then M:=Param1[i];
result:=M;
end;

function TTestCom.Min(Param1: OleVariant): Single;
var
  N1,N2,i : Integer;
  M : Single;
begin
  N1:=varArrayLowBound(Param1,1);
  N2:=varArrayHighBound(Param1,1);
  M:=Param1[N1];
  for i:=N1 to N2 do
    if Param1[i]<M then M:=Param1[i];
  result:=M;
end;
```

После задания кода реализации методов создание COM-объекта завершено. Осталось только выполнить компиляцию динамической библиотеки и регистрацию созданного COM-сервера.

Компиляция динамической библиотеки, являющейся внутренним COM-сервером, ничем не отличается от компиляции обычного приложения. Следует просто выбрать в главном меню команду Project ► Compile или нажать комбинацию клавиш Ctrl + F9. Если в коде не содержится синтаксических ошибок, будет выполнена компиляция и создан файл TestCOMServer.dll. Чтобы созданный COM-объект можно было использовать в приложениях, следует выполнить регистрацию COM-сервера. Регистрация внутреннего сервера может быть выполнена несколькими способами:

- ☐ выполнением в среде Delphi команды Run ► Register ActiveX Server главного меню;
- ☐ запуском специальной утилиты RegSvr32;
- ☐ внесением необходимой записи в реестр вручную;
- ☐ созданием специального приложения, которое загрузит динамическую библиотеку, являющуюся сервером, и вызовет функцию DllRegisterServer, экспортируемую данной библиотекой.

В нашем случае проще всего использовать первый способ. Однако следует иметь в виду, что при переносе COM-сервера на другие компьютеры этот способ, как правило, непригоден. Поэтому если вы собираетесь распространять созданные вами COM-серверы, то необходимо позаботиться о том, чтобы пользователь мог без особых затруднений выполнить их регистрацию.

Разработка клиентского приложения для внутреннего сервера

Создание COM-объекта не является самоцелью — функции, выполняемые этим объектом, используются в другом приложении. Приложение, вызывающее методы COM-объекта, обычно называется *клиентом COM-объекта*. Рассмотрим пример создания клиентского приложения для созданного нами сервера.

Приложение, являющееся COM-клиентом, практически ничем не отличается от обычного приложения. Поэтому его разработку следует начинать с выполнения команды **File ► New Application** главного меню IDE Delphi. После создания нового приложения разработаем его главную форму. На форму нужно выводить исходные данные, используемые для расчета. В нашем случае это одномерный массив вещественных чисел. В качестве примера возьмем фиксированный массив из девяти элементов [1, 2, 3, 4, 5, 6, 7, 8, 9]. Отообразим их на форме с помощью списка (компонента **TListBox**). Помимо списка на форме должны быть элементы управления, с помощью которых дается команда на выполнение расчетов, и элементы для вывода результатов. В качестве первых будем использовать кнопки (**TButton**), в качестве вторых — метки (**TLabel**). Так как наш COM-объект имеет два интерфейса, то разделим все элементы управления, размещенные на форме, на две группы. Группировку выполним с помощью компонента **TGroupBox**. Примерный внешний вид формы приложения приведен на рис. 18.6.

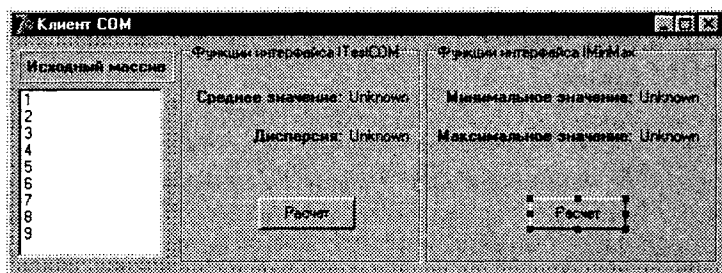


Рис. 18.6. Главная форма клиентского приложения

После разработки формы займемся созданием необходимого кода. Чтобы обеспечить взаимодействие приложения с COM-объектом, в раздел **uses** главного модуля приложения следует включить библиотеку типов (модуль **TestComServer_TLB**).

ПРИМЕЧАНИЕ

В нашем случае файл библиотеки типов в синтаксисе языка Object Pascal создается при разработке COM-объекта. В случае использования каких-либо других COM-серверов (которые могут быть написаны на любом алгоритмическом языке) такого файла может не быть. Однако файл библиотеки типов в синтаксисе языка Object Pascal может быть сгенерирован Delphi. Для этого используется команда меню **Project ► Import Type Library**. При выборе этой команды открывается окно диалога, содержащее список всех зарегистрированных в системе серверов ActiveX. Для генерации файла библиотеки типов следует выбрать в этом списке необходимый сервер и щелкнуть на кнопке **Install**.

Чтобы вызывать методы COM-объекта в приложении, следует объявить переменные, которые будут являться указателями на соответствующие интерфейсы объекта. Объявление таких переменных удобнее всего поместить в раздел **private** или **public** класса формы:

```
TForm1 = class(TForm)
...
private
```

```

Stat: ITestCom;
MinMax : IMinMax;
public
end;

```

При загрузке нашего приложения необходимо создать экземпляр СОМ-объекта и получить ссылку на его интерфейс. Эти действия лучше всего выполнить с помощью метода-обработчика события OnShow главной формы приложения. В этом случае будет гарантия того, что к моменту отображения главного окна приложения на экране экземпляр СОМ-объекта уже будет создан.

Для создания экземпляра СОМ-объекта используется метод Create составного класса объекта. Данный метод возвращает указатель на основной интерфейс СОМ-объекта (в нашем случае это интерфейс ITestCom). Для получения указателя на какой-либо другой интерфейс следует воспользоваться методом QueryInterface основного интерфейса. Код обработчика события OnShow, создающего экземпляр СОМ-объекта и получающего указатели на его интерфейсы, для нашего примера имеет следующий вид:

```

procedure TForm1.FormShow(Sender: TObject);
begin
    Stat:=CoTestCom.Create;
    Stat.QueryInterface(IMinMax,MinMax);
end;

```

Осталось только задать код методов-обработчиков события OnClick кнопок, щелчках на которых вызываются методы интерфейсов. Вызов методов интерфейсов СОМ-объекта ничем не отличается от вызова методов обычного объекта Delphi.

Приведем полный листинг главного модуля клиентского приложения:

```

unit ClientUnit;

interface

uses
    Windows, Messages, SysUtils, Classes, Graphics,
    Controls, Forms, Dialogs, TestComServer_TLB,
    StdCtrls, ExtCtrls;

type
    TForm1 = class(TForm)
        ListBox1: TListBox;
        Panel1: TPanel;
        GroupBox1: TGroupBox;
        Button1: TButton;
        Label1: TLabel;
        Label2: TLabel;
        Label3: TLabel;
        Label4: TLabel;
        GroupBox2: TGroupBox;
        Label5: TLabel;
        Label6: TLabel;
        Label7: TLabel;
        Label8: TLabel;
        Button2: TButton;
        procedure FormShow(Sender: TObject);
    end;

```

```

    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    private
// Объявляем переменные — указатели на интерфейсы
    Stat: ITestCom;
    MinMax : IMinMax;
    public
    { Public declarations }
    end;

var
    Form1: TForm1;

implementation
.
{$R *.DFM}

procedure TForm1.FormShow(Sender: TObject);
begin
// Создаем экземпляр COM-объекта и получаем указатель
// на интерфейс ITestCom
    Stat:=CoTestCom.Create;
// Получаем указатель на интерфейс IMinMax
    Stat.QueryInterface(IMinMax.MinMax);
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    V: OleVariant;
    i: Integer;
begin
// Создаем вариантный массив
    V:=varArrayCreate([0,
        ListBox1.Items.Count-1].varSingle);
// Заполняем созданный массив значениями из списка
    for i:=0 to ListBox1.Items.Count-1 do
        V[i]:=StrToFloat(ListBox1.Items[i]);
// Вызываем методы интерфейса ITestCom и выводим
// результаты на форму
    Label3.Caption:=FloatToStrF(Stat.Avg(V),ffGeneral,4,2);
    Label4.Caption:=FloatToStrF(Stat.Variance(V),
        ffGeneral,2,2);
end;

procedure TForm1.Button2Click(Sender: TObject);
var
    V: OleVariant;
    i: Integer;
begin
// Создаем вариантный массив
    V:=varArrayCreate([0,
        ListBox1.Items.Count-1].varSingle);
// Заполняем массив значениями из списка ListBox1
    for i:=0 to ListBox1.Items.Count-1 do
        V[i]:=StrToFloat(ListBox1.Items[i]);
// Вызываем методы интерфейса IMinMax и выводим

```

```
// результаты на форму
Label7.Caption:=FloatToStrF(MinMax.Min(V),
    ffGeneral,4,2);
Label8.Caption:=FloatToStrF(MinMax.Max(V),
    ffGeneral,2,2);
end;

end.
```

Основы автоматизации

Технология автоматизации (OLE Automation) разработана на основе технологии COM. Так же как и COM, автоматизация позволяет использовать функции одних приложений в других приложениях. *Автоматизация* — это механизм обмена информацией между процессами в операционной системе Windows, с помощью которого одна прикладная программа может управлять другой. Технология OLE Automation обеспечивает выполнение базовых функций, позволяющих обособленным программным модулям связываться и обмениваться информацией.

Многие используемые в технологии автоматизации термины подобны тем, которые применяются при описании технологии COM.

Объект автоматизации не имеет принципиальных отличий от COM-объекта и представляет собой отдельный, самодостаточный объект, разработанный для выполнения специфической задачи или функции. Отличие от COM-объекта заключается только в том, что доступ к объектам автоматизации осуществляется через специальные интерфейсы, называемые *интерфейсами диспетчеризации*. Интерфейсы диспетчеризации похожи на обычные COM-интерфейсы, но позволяют упростить обращение к объекту автоматизации.

Сервер автоматизации

Сервер автоматизации представляет собой исполняемый модуль, который может включать в себя несколько объектов автоматизации.

Серверы автоматизации подразделяются на внутренние (in-process) и внешние (out-of-process). *Внутренние серверы* являются динамическими библиотеками, которые выполняются в адресном пространстве клиента автоматизации. *Внешние серверы* представляют собой самостоятельные приложения, которые выполняются в отдельных процессах по отношению к клиенту автоматизации. Каждый из типов серверов имеет свои преимущества. Главным достоинством внутренних серверов является высокая скорость обмена данными с клиентом, которая достигается за счет того, что DLL-библиотеки работают в одном процессе с клиентом. Кроме того, DLL-серверы обычно более просты в отладке и тестировании, чем серверы-приложения.

С другой стороны, внешние серверы более выгодны с точки зрения изоляции ошибок. В случае использования внешних серверов даже аварийное завершение работы клиента не приведет к сбою в работе сервера.

Контроллер автоматизации

Контроллером автоматизации принято называть клиентское приложение, управляющее объектом автоматизации.

Контроллер автоматизации для управления объектом автоматизации использует методы интерфейсов этого объекта. Для получения информации об интерфейсах служат библиотеки типов.

Библиотеки типов

Информация о типах обычно хранится в файлах формата TLB (Type Library — библиотека типов) или OLB (Object Library — библиотека объектов). В библиотеках типов содержится следующая информация:

- ☐ данные о перечислениях;
- ☐ описания обычных интерфейсов и интерфейсов диспетчеризации;
- ☐ описания классов составных объектов;
- ☐ ссылки на описания типов из других библиотек типов.

Для доступа к объектам автоматизации не требуется информация о типах. Однако наличие информации о типах позволяет производить проверку синтаксиса кода, используемого для обращения к методам интерфейсов объектов автоматизации, на стадии компиляции приложения.

Интерфейс IDispatch

Базовый интерфейс автоматизации (интерфейс диспетчеризации) называется IDispatch и представляет собой COM-интерфейс, являющийся наследником интерфейса IUnknown. Отличие интерфейса диспетчеризации от обычного COM-интерфейса заключается в способе вызова методов интерфейса, которые вызываются с помощью специального метода Invoke. Объект автоматизации обязательно должен иметь дополнительный интерфейс, в котором указываются все методы, доступные через метод Invoke. Каждому методу этого интерфейса соответствует идентификатор, который называется *идентификатором диспетчеризации* (dispatch identifier, dispid). Реализация этого метода подобна оператору case: на основе указанного идентификатора диспетчеризации выбирается метод, который должен быть выполнен. Благодаря такой организации объекту автоматизации требуется всего одна таблица виртуальных методов — для интерфейса IDispatch. Все остальные методы вызываются посредством метода Invoke этого интерфейса.

Создание серверов автоматизации в Delphi

Создание сервера автоматизации в Delphi в целом подобно созданию COM-сервера. Однако объекты автоматизации обладают рядом дополнительных функциональных возможностей, а их создание имеет ряд особенностей.

Объект автоматизации в Delphi создается на основе класса TAutoObject, который является потомком класса TTypedCOMObject.

Рассмотрим пример создания сервера автоматизации. В качестве примера создадим объект автоматизации, выполняющий те же функции, что и рассмотренный ранее СОМ-объект. Данный объект автоматизации включим в состав внешнего сервера.

Создание внешнего сервера автоматизации в Delphi включает выполнение следующих шагов.

1. Создание нового приложения-сервера.
2. Создание объекта автоматизации.
3. Создание интерфейсов, а также заголовков методов и свойств интерфейсов объекта автоматизации.
4. Программирование методов интерфейса.

Как видите, процесс создания сервера автоматизации включает те же действия, что и процесс создания СОМ-сервера.

Создание приложения-сервера

Приложение-сервер автоматизации представляет собой обычное приложение, включающее в себя, по крайней мере, один объект автоматизации. Создание такого приложения не имеет никаких особенностей. Как и в предыдущем случае, процесс создания сервера автоматизации начинается с команды File ► New Application главного меню IDE Delphi.

Создание объекта автоматизации

После того как приложение-сервер создано, следует включить в него новый объект автоматизации. Для этого выберите команду File ► New, затем в открывшемся окне диалога New Items перейдите на вкладку ActiveX и щелкните на значке Automation Object (см. рис. 18.1).

После щелчка на кнопке OK в окне диалога New Items производится запуск мастера создания объекта автоматизации, в окне которого необходимо задать параметры создаваемого объекта (рис. 18.7).

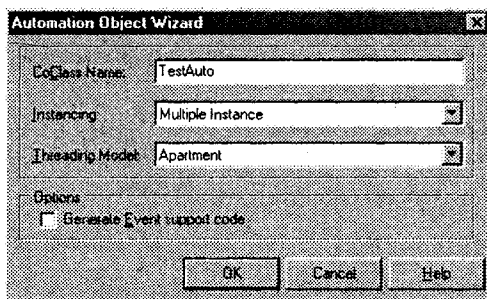


Рис. 18.7. Окно мастера создания объекта автоматизации

- ❑ В поле CoClass Name вводится имя класса объекта автоматизации (в нашем примере назовем класс TestAuto).

- ❑ В списке **Instancing**, как и в случае COM-сервера, выбирается способ создания объекта автоматизации.
- ❑ В списке **Threading Model**, как и в случае COM-сервера, выбирается вариант работы сервера с потоками.
- ❑ Флажок **Generate event support code** определяет, должен ли создаваемый объект автоматизации поддерживать обработки событий.

После щелчка на кнопке **OK** в окне **Automation Object Wizard** будет создан новый объект автоматизации и откроется окно редактора библиотеки типов. Если просмотреть текст модуля объекта автоматизации (в нашем примере назовем его **AutoObjectUnit**), то даже при беглом взгляде видно, что содержащийся в нем код не сложнее, чем в модуле «пустого» COM-объекта:

```
unit AutoObjectUnit;

interface

uses
  ComObj, ActiveX, AutoServer_TLB, StdVcl;

type
  TTestAuto = class(TAutoObject, ITestAuto)
  protected
    { Protected declarations }
  end;

implementation

uses ComServ;

initialization
  TAutoObjectFactory.Create(ComServer, TTestAuto,
    Class_TestAuto, ciMultiInstance, tmApartment);
end.
```

ПРИМЕЧАНИЕ

В данном примере при создании объекта автоматизации не была включена поддержка обработки событий. Программирование событий в некоторой степени усложняет код модуля объекта автоматизации и представляет собой нетривиальную задачу, однако ввиду ограниченного объема книги данные вопросы здесь не рассматриваются.

Код библиотеки типов вновь созданного объекта автоматизации, в отличие от COM-объекта, включает описание нескольких интерфейсов:

```
unit AutoServer_TLB;

{$TYPEDADDRESS OFF}

uses Windows, ActiveX, Classes, Graphics, OleServer,
  OleCtrls, StdVCL;

const
  AutoServerMajorVersion = 1;
  AutoServerMinorVersion = 0;
```



```
LIBID_AutoServer: TGUID = '{62D7BF80-A76C-11D4-A8C1-0060520799BE}';
```

```
IID_ITestAuto: TGUID = '{62D7BF81-A76C-11D4-A8C1-0060520799BE}';
```

```
CLASS_TestAuto: TGUID = '{62D7BF85-A76C-11D4-A8C1-0060520799BE}';
```

type

```
ITestAuto = interface;
```

```
ITestAutoDisp = dispinterface;
```

```
TestAuto = ITestAuto;
```

```
ITestAuto = interface(IDispatch)
```

```
['{62D7BF81-A76C-11D4-A8C1-0060520799BE}']
```

```
end;
```

```
ITestAutoDisp = dispinterface
```

```
['{62D7BF81-A76C-11D4-A8C1-0060520799BE}']
```

```
end;
```

```
CoTestAuto = class
```

```
class function Create: ITestAuto;
```

```
class function CreateRemote(const MachineName:  
    string): ITestAuto;
```

```
end;
```

implementation

uses ComObj;

```
class function CoTestAuto.Create: ITestAuto;
```

```
begin
```

```
    Result := CreateComObject(CLASS_TestAuto) as ITestAuto;
```

```
end;
```

```
class function CoTestAuto.CreateRemote(const MachineName:  
    string): ITestAuto;
```

```
begin
```

```
    Result := CreateRemoteComObject(MachineName,  
        CLASS_TestAuto) as ITestAuto;
```

```
end;
```

```
end.
```

Интерфейс **ITestAuto** называется дуальным и предназначен для работы с объектом **TTestAuto**.

ПРИМЕЧАНИЕ

Дуальные интерфейсы являются наследниками интерфейса **IDispatch** и обеспечивают как связывание с объектом во время компиляции (раннее связывание через таблицу виртуальных методов), так и позднее связывание, используемое в технологии автоматизации.

Интерфейс диспетчеризации ITestAutoDisp обеспечивает взаимодействие с объектом на основе технологии автоматизации.

Создание интерфейсов и методов объекта автоматизации

Так же как и при разработке COM-объекта, интерфейсы и методы объекта автоматизации создаются с помощью редактора библиотеки типов. Для упрощения примера создадим только один интерфейс, который будет содержать методы вычисления среднего значения и дисперсии. Следует отметить, что создание методов интерфейса для объекта автоматизации имеет одну особенность, которая заключается в том, что результат, возвращаемый функцией, может иметь только тип HRESULT. В этом случае метод является процедурой, поэтому для передачи рассчитанного значения в клиентское приложение следует использовать параметр вызова процедуры.

Таким образом, в нашем примере при вызове метода требуются два параметра: первый используется для передачи данных объекту автоматизации, а второй — для получения результата от объекта автоматизации. Создание нового метода интерфейса и изменение его имени производится точно так же, как и при разработке COM-объекта. Тип результата HRESULT присваивается методу по умолчанию, поэтому обычно изменять значение в списке Return Type не требуется. Создание первого параметра метода также полностью аналогично примеру для COM-объекта. А вот задание второго параметра имеет особенности, связанные с тем, что через этот параметр объект автоматизации возвращает результат клиентскому приложению. Поэтому данный параметр должен передаваться по ссылке, а не по значению, и иметь атрибут out. Чтобы указать, что параметр передается по ссылке, следует просто добавить символ звездочки (*) к имени типа параметра в столбце Type списка Parameters (рис. 18.8).

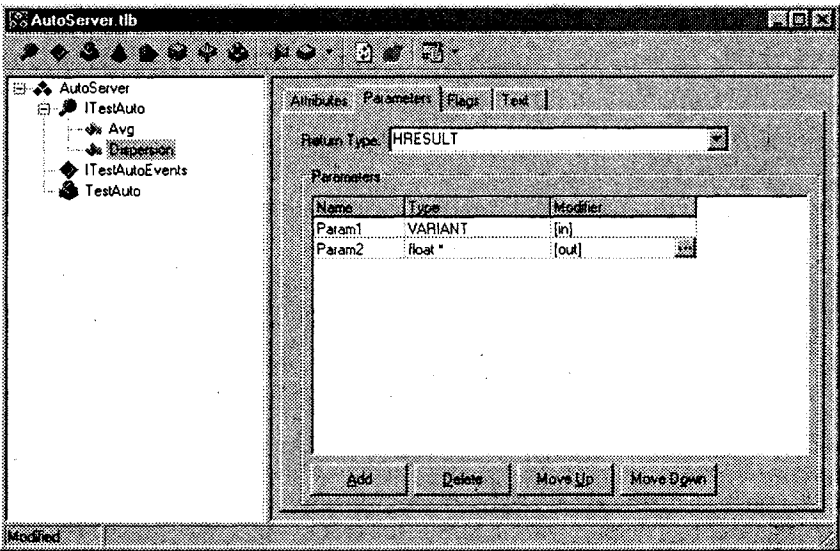


Рис. 18.8. Задание параметров метода объекта автоматизации

Чтобы задать параметру атрибут `out` (говорящий о том, что параметр является выходным), используется столбец `Modifier` списка `Parameters`. При щелчке в этом столбце списка в поле ввода появляется кнопка с многоточием (см. рис. 18.8). Щелчок на этой кнопке открывает окно диалога `Parameter Flags`, в котором устанавливаются дополнительные атрибуты параметра (рис. 18.9). В нашем примере следует установить флажок `Out`.

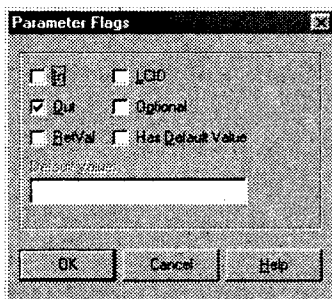


Рис. 18.9. Окно диалога `Parameter Flags`

Параметры вызова обоих методов в рассматриваемом примере одинаковы и описываются в редакторе библиотеки типов так, как показано ранее рис. 18.8.

После создания всех методов и описания всех параметров щелкните на кнопке `Refresh` панели инструментов редактора библиотеки типов. При этом внесенные изменения будут отражены в файле библиотеки типов:

```
unit AutoServer_TLB;

{$TYPEDADDRESS OFF}

uses Windows, ActiveX, Classes, Graphics, OleServer,
    OleCtrls, StdVCL;

const
    AutoServerMajorVersion = 1;
    AutoServerMinorVersion = 0;

    LIBID_AutoServer: TGUID = '{62D7BF80-A76C-11D4-A8C1-0060520799BE}';

    IID_ITestAuto: TGUID = '{62D7BF81-A76C-11D4-A8C1-0060520799BE}';
    CLASS_TestAuto: TGUID = '{62D7BF85-A76C-11D4-A8C1-0060520799BE}';

type

    ITestAuto = interface;
    ITestAutoDisp = dispinterface;
    TestAuto = ITestAuto;

    ITestAuto = interface(IDispatch)
```

```

['{62D7BF81-A76C-11D4-A8C1-0060520799BE}']
procedure Avg(Param1: OleVariant;
  out Param2: Single); safecall;
procedure Dispersion(Param1: OleVariant;
  out Param2: Single); safecall;
end;

ITestAutoDisp = dispinterface
['{62D7BF81-A76C-11D4-A8C1-0060520799BE}']
procedure Avg(Param1: OleVariant;
  out Param2: Single); dispid 1;
procedure Dispersion(Param1: OleVariant;
  out Param2: Single); dispid 2;
end;

CoTestAuto = class
class function Create: ITestAuto;
class function CreateRemote(const MachineName:
  string): ITestAuto;
end;

implementation

uses ComObj;

class function CoTestAuto.Create: ITestAuto;
begin
  Result := CreateComObject(CLASS_TestAuto) as ITestAuto;
end;

class function CoTestAuto.CreateRemote(const MachineName:
  string): ITestAuto;
begin
  Result := CreateRemoteComObject(MachineName,
    CLASS_TestAuto) as ITestAuto;
end;

end.

```

Обратите внимание на то, что в описании каждого метода интерфейса диспетчеризации присутствует ключевое слово `dispid`, за которым следует целое число. Так методам присваиваются идентификаторы диспетчеризации, которые используются при вызове методов интерфейса с помощью метода `Invoke`.

Заголовки методов интерфейса будут добавлены и в описание класса `ITestAuto`. Кроме того, в модуль описания класса объекта автоматизации будут добавлены заготовки для написания кода реализации методов, заголовки которых мы создали в редакторе библиотеки типов.

Программирование методов интерфейса

Код реализации методов интерфейса незначительно отличается от кода, написанного нами в предыдущем примере (где мы рассматривали создание COM-объекта) для аналогичных методов. Различия связаны только с тем, что результат возвращается не через значение функции, а через параметр процедуры:

```

unit AutoObjectUnit;

interface

uses
  ComObj, ActiveX, AutoServer_TLB, StdVcl;

type
  TTestAuto = class(TAutoObject, ITestAuto)
  protected
    { Protected declarations }
    procedure Avg(Param1: OleVariant;
      out Param2: Single); safecall;
    procedure Dispersion(Param1: OleVariant;
      out Param2: Single); safecall;
  end;

implementation

uses ComServ;

procedure TTestAuto.Avg(Param1: OleVariant;
  out Param2: Single);
var
  N1,N2,i : Integer;
  A : Single;
begin
  A:=0;
  N1:=varArrayLowBound(Param1,1);
  N2:=varArrayHighBound(Param1,1);
  for i:=N1 to N2 do
    A:=A+Param1[i];
  Param2:=A/(N2-N1+1)
end;

procedure TTestAuto.Dispersion(Param1: OleVariant;
  out Param2: Single);
var
  N1,N2,i : Integer;
  A,D : Single;
begin
  Avg(Param1,A);
  D:=0;
  N1:=varArrayLowBound(Param1,1);
  N2:=varArrayHighBound(Param1,1);
  for i:=N1 to N2 do
    D:=sqr(Param1[i]-A);
  Param2:=D/(N2-N1);
end;

initialization
  TAutoObjectFactory.Create(ComServer, TTestAuto,
    Class_TestAuto, ciMultiInstance, tmApartment);
end.

```

На этом разработку объекта автоматизации можно считать завершённой. Осталось только откомпилировать приложение. Выполнять регистрацию в случае внешне-

го сервера автоматизации не требуется — сервер регистрируется автоматически при первом запуске приложения-сервера.

Разработка клиента автоматизации

Использование объекта автоматизации в приложениях не вызывает особых сложностей. Мы уже довольно подробно обсуждали этот вопрос в предыдущей главе, рассматривая взаимодействие с приложениями MS Office.

При позднем связывании создание экземпляра объекта автоматизации и получение указателя на его интерфейс выполняется с помощью функции

```
CreateOleObject (const ClassName: string): IDispatch
```

В качестве параметра данной функции передается строка, содержащая имена сервера и объекта автоматизации. Для сервера, рассмотренного в приведенном выше примере, эта строка выглядит следующим образом:

```
'AutoServer.TestAuto'
```

Для работы с объектами автоматизации используется специальный тип переменных — `OleVariant`. Таким образом, чтобы получить доступ к методам объекта автоматизации, следует объявить переменную типа `OleVariant` и создать объект с помощью функции `CreateOleObject`:

```
var  
  S: OleVariant;  
...  
S:=CreateOleObject('AutoServer.TestAuto');
```

После этого методы объекта автоматизации можно вызывать как обычные методы любого объекта `ObjectPascal`:

```
S.Method1(Param1,Param2);
```

Однако при этом следует помнить, что компилятор не может проверить корректность вызова методов, так как не имеет о них никакой информации. Поэтому предпочтительнее использовать позднее связывание. Для этого в модуле приложения-клиента следует объявить модуль библиотеки типов объекта автоматизации и работать с ним как с обычным COM-объектом.

Элементы ActiveX

Первоначально элементы ActiveX назывались элементами OLE или OCX. Затем корпорация Microsoft внесла некоторые изменения в элементы OCX, обеспечив ряд новых возможностей, сделавших эти элементы более адаптированными для технологий Интернета (например, элемент ActiveX может хранить свои данные на веб-сервере либо загружаться с веб-сервера и затем выполняться на стороне клиента). Чтобы отразить наличие новых возможностей, название «элемент OLE» было заменено на «элемент ActiveX».

С точки зрения модели COM элементы ActiveX являются внутренними серверами, поддерживающими технологию OLE Automation. Таким образом, платформа ActiveX по существу представляет собой адаптацию к Web существующих техно-

логий Microsoft, базирующихся на механизмах OLE и COM. Первоначально термин «ActiveX» относился именно к технологиям, связанным с Интернетом. Однако со временем значение термина «ActiveX» было расширено, и его стали использовать для обозначения разнообразных технологий Microsoft на основе COM, ранее известных под именем OLE (OLE Automation, OLE Documents, OLE Controls и т. п.). Тем не менее не совсем правильно отождествлять технологии OLE и ActiveX — элементы ActiveX являются модификацией элементов OLE, но не их полными аналогами. Библиотеки элементов ActiveX имеют меньший объем по сравнению с библиотеками элементов OLE и обеспечивают более высокую скорость взаимодействия с клиентами. Сохранилась и совместимость — любой программный компонент OLE будет работать с библиотеками ActiveX.

ActiveX и компонентное программирование

Элемент ActiveX представляет собой сервер, поддерживающий технологию автоматизации, реализованный в виде динамической библиотеки, исполняемый в адресном пространстве вызвавшего его приложения и допускающий возможность визуального редактирования. Разработчики программ могут использовать такие элементы в приложениях, улучшая их функциональные возможности. Таким образом, приложение строится из готовых частей. Такой подход к разработке программных продуктов называется *компонентным программированием*. В виде элемента ActiveX может быть реализовано все что угодно — от обычной кнопки до полнофункциональной электронной таблицы.

Компонентное программирование имеет много достоинств. При использовании компонентного подхода приложение (которое может представлять собой большую и сложную систему) разбивается на более мелкие и простые для понимания части. В результате:

- ☐ систему можно наращивать постепенно, по мере создания частей;
- ☐ значительно упрощается процесс отладки, так как каждый компонент системы можно отлаживать отдельно;
- ☐ систему легче адаптировать под требования заказчика, так как можно изменять не всю систему, а лишь некоторые ее компоненты;
- ☐ компоненты системы можно повторно использовать в последующих разработках;
- ☐ компоненты системы могут находиться в любом доступном через сеть месте, в том числе и на другом компьютере;
- ☐ компоненты могут быть написаны на разных языках программирования;
- ☐ значительно упрощается модернизация системы, так как систему можно модифицировать по частям, по мере готовности новых версий компонентов.

Технология ActiveX хорошо соответствует идее компонентного программирования и в настоящее время является своего рода стандартом компонентного подхода к разработке Windows-приложений. Возможность создания и использования элементов ActiveX реализована практически во всех современных средствах разра-

ботки, включая Visual Basic, Visual C++, Borland Delphi и Borland C++ Builder, а также во всех средах разработки на языке Java.

Несмотря на многочисленные достоинства, элементы ActiveX не лишены и некоторых недостатков.

- ❑ Полноценная поддержка элементов ActiveX обеспечивается только в операционной системе Windows. В настоящее время этот недостаток, вероятно, не очень существен ввиду монополии Windows как операционной системы для настольных компьютеров. Однако не исключено, что в скором времени ситуация может измениться, и появятся операционные системы, способные составить конкуренцию Windows в этом секторе рынка программного обеспечения.
- ❑ Элементы ActiveX не позволяют в полной мере реализовать объектно-ориентированный подход. У элементов ActiveX (как и у COM-объектов) отсутствует механизм наследования и, естественно, не реализуется полиморфизм. Поэтому программы, построенные на основе иерархии COM-объектов, имеют очень сложную структуру (вспомните, например, структуру объектов приложений MS Office, которую мы рассматривали в предыдущей главе).

Использование и создание элементов ActiveX в Delphi

Среда разработки Delphi позволяет работать с элементами ActiveX точно так же, как с VCL-компонентами. В стандартную поставку Delphi входит несколько элементов ActiveX, которые располагаются на странице ActiveX палитры компонентов. Пожалуй, единственным отличием элементов ActiveX от VCL-компонентов является то, что их нельзя использовать в качестве предков при создании новых компонентов.

Использование существующих элементов ActiveX

Существует громадное количество различных элементов ActiveX, созданных многочисленными разработчиками. В Интернете можно найти множество как коммерческих, так и бесплатных компонентов, реализующих самые разные функции. Поэтому если при разработке приложений возникает необходимость в каком-либо нестандартном элементе управления, то не всегда имеет смысл разрабатывать его самостоятельно — лучше сначала выяснить, не существует ли уже что-нибудь похожее на то, что вам требуется. (Тем не менее к VCL-компонентам и элементам ActiveX, загруженным из Интернета, следует относиться с осторожностью, так как они могут стать источником трудно обнаруживаемых ошибок.)

Рассмотрим процедуру подключения к Delphi нового элемента ActiveX. В качестве примера разберем установку элементов ActiveX из сервера pdf.osx, который устанавливается при установке программы Adobe Acrobat Reader.

Установка нового элемента ActiveX происходит в два этапа.

1. Регистрация элемента в системном реестре.
2. Размещение элемента в палитре компонентов IDE Delphi.

Регистрация элемента ActiveX выполняется, как правило, при его установке. Все зарегистрированные в текущий момент серверы ActiveX отображаются в списке, который выводится в окне диалога Import ActiveX (рис. 18.10). Это окно открывается при выборе в меню команды Component ► Import ActiveX Control. Поскольку сервер pdf.osx уже присутствует в списке окна диалога Import ActiveX, то выполнять его регистрацию не требуется.

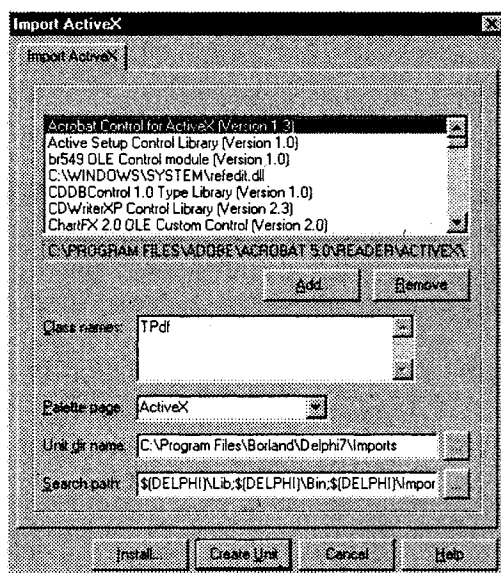


Рис. 18.10. Окно диалога Import ActiveX

ПРИМЕЧАНИЕ

Даже если вы не устанавливали ни одного элемента управления ActiveX, список в окне Import ActiveX будет содержать обширный перечень зарегистрированных в системе элементов. Все они устанавливаются при установке Windows, Microsoft Office и других приложений (например, Adobe Acrobat Reader).

Если в списке отсутствует необходимый вам элемент, но вы точно знаете, что он установлен на вашем компьютере, это означает, что не была выполнена регистрация элемента. Зарегистрировать элемент ActiveX можно прямо из окна Import ActiveX. Для этого следует щелкнуть на кнопке Add и затем выбрать нужный файл в окне диалога открытия файлов. Естественно, при этом необходимо знать имя файла, содержащего элемент ActiveX, и его местоположение на диске.

ПРИМЕЧАНИЕ

Элементы ActiveX обычно находятся в файлах, имеющих расширение osx или dll.

После того как в списке окна Import ActiveX выбран нужный сервер ActiveX, в поле Class Name появляется перечень всех классов, содержащихся в выбранном сервере.

Например, сервер pdf.ocx содержит только один класс, который имеет имя TPdf. Информацию в поле Class Name можно редактировать. Поэтому, если вас по какой-либо причине не устраивают заданные по умолчанию имена классов, вы можете их изменить.

Для установки компонента следует щелкнуть на кнопке Install. При этом откроется окно диалога Install, в котором требуется указать имя файла пакета, который будет содержать устанавливаемый элемент (рис. 18.11). Обычно удобнее производить установку каждого элемента в отдельный пакет. Для этого перейдите в окне Install на вкладку Into new package и укажите в поле File name имя создаваемого пакета. В нашем примере пакет назван pdf.dpk. В поле Description окна Install можно задать описание пакета.

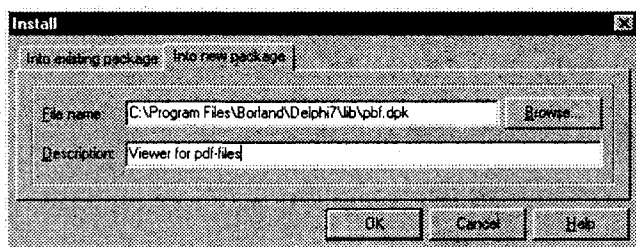


Рис. 18.11. Создание нового пакета

После щелчка на кнопке OK в окне диалога Install будет выполнена генерация файла библиотеки типов в синтаксисе языка Object Pascal. Имя созданного при этом файла состоит из имени файла сервера ActiveX, к которому прибавляется суффикс _TLB. Файл имеет расширение pas.

Затем будет выполнена компиляция пакета и все элементы ActiveX, содержащиеся в сервере, будут добавлены в палитру компонентов Delphi IDE на страницу ActiveX. В нашем примере это будет только один элемент — Pdf.

После размещения элемента ActiveX в палитре компонентов работа с ним ничем не отличается от работы с VCL-компонентами.

Преобразование VCL-компонента в элемент ActiveX

Элементы ActiveX в Delphi создаются на основе VCL-компонентов или форм. Во втором случае создаваемая форма называется ActiveForm и представляет собой набор визуальных и не визуальных компонентов.

Наиболее простым способом создания элемента ActiveX является преобразование существующего VCL-компонента в элемент ActiveX. Таким образом, если вам требуется создать какой-либо новый элемент ActiveX, то вначале следует создать VCL-компонент, а затем преобразовать его в ActiveX.

Преобразование VCL-компонента в элемент ActiveX выполняется очень просто. Для этого используется специальный мастер, который запускается двойным щелчком на значке ActiveX Control на вкладке ActiveX окна диалога New Items (напомним, что это окно открывается при выборе команды File ► New в главном меню IDE Delphi).

ПРИМЕЧАНИЕ

Напрямую в ActiveX могут быть преобразованы только те VCL-компоненты, которые являются потомками класса `TWinControl`. В принципе, можно создать элемент ActiveX на основе любого визуального VCL-компонента, но здесь мы этот вопрос рассматривать не будем. Такая задача, во-первых, довольно сложна, и ее рассмотрение потребовало бы значительного увеличения объема данной главы. Во-вторых, этот вопрос лежит вне темы данной книги, так как представляет собой чисто программистскую задачу.

Параметры создаваемого элемента задаются в окне мастера `ActiveX Control Wizard` (рис. 18.12).

- ☐ В раскрывающемся списке `VCL Class Name` выбирается имя класса библиотеки VCL, на основе которого будет создан элемент ActiveX.
- ☐ В поле `New ActiveX Name` вводится имя создаваемого элемента ActiveX, в поле `Implementation Unit` — имя файла модуля, в поле ввода `Project Name` — имя файла проекта.
- ☐ В раскрывающемся списке `Threading Model` выбирается вариант работы сервера с потоками.
- ☐ Дополнительные параметры задаются с помощью группы флажков `ActiveX Control Options`:
 - `Include Design-Time License` — включение в элемент ActiveX лицензионной информации;
 - `Include Version Information` — добавление в проект информации о версии;
 - `Include About Box` — организация окна диалога с информацией о разработчике.

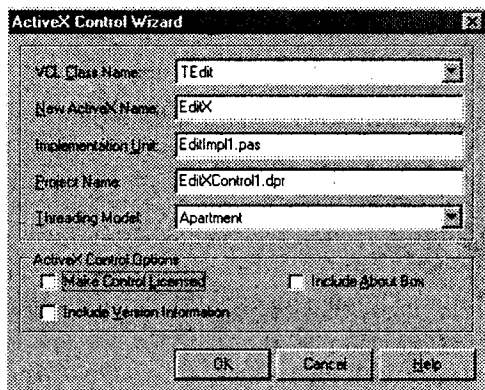


Рис. 18.12. Мастер создания элементов ActiveX

После задания всех параметров и щелчка на кнопке `OK` происходит автоматическое создание всех необходимых файлов: файла проекта (элементы ActiveX всегда входят в состав внутреннего сервера, поэтому генерируется файл проекта динамической библиотеки), библиотеки типов и модуля реализации. Для завершения создания элемента ActiveX следует просто откомпилировать проект.

Создание форм ActiveForm

В элемент ActiveX можно преобразовать не только VCL-компонент, но и целую форму. Причем в этом случае на форму можно помещать и невизуальные компоненты. Следовательно, в виде форм ActiveForm можно использовать компоненты для доступа к данным.

Никаких ограничений на действия, выполняемые в формах ActiveForm, не накладывается. Единственное, что необходимо учитывать, — клиентам будут недоступны свойства, методы и события компонентов, размещенных на форме. Если такой доступ необходим, то следует добавить в класс формы свойства и методы, через которые клиент сможет изменять свойства внутренних компонентов формы.

В качестве примера создадим форму ActiveForm для вывода в табличном виде информации из таблицы базы данных. Будем использовать таблицу **Физические лица** из базы данных **sales** (такой пример, конечно, не имеет большого практического значения, но зато позволяет продемонстрировать механизм использования как визуальных, так и невизуальных компонентов для доступа к данным при создании форм ActiveForm).

1. Выберите в меню команду **File ► New** и в открывшемся окне диалога **New Items** на вкладке **ActiveX** выберите значок **ActiveForm**. После щелчка на кнопке **OK** откроется окно мастера создания форм ActiveForm.
2. Задайте имя создаваемого объекта ActiveForm в окне мастера **ActiveForm Wizard**. Данное окно практически не отличается от окна мастера создания элементов ActiveX, приведенного на рис. 18.12. Единственное отличие состоит в том, что для ActiveForm не нужно задавать имя класса VCL, на основе которого создается элемент ActiveX, поэтому список **VCL Class Name** в окне ActiveForm Wizard недоступен. Присвоим создаваемому элементу ActiveForm имя **TableX**.

После щелчка на кнопке **OK** в окне ActiveForm Wizard будет создан новый проект, содержащий три файла:

- **TableProj1.dpr** — файл проекта с заголовком библиотеки DLL;
- **TableImpl1.pas** — модуль, содержащий код реализации элемента ActiveForm;
- **TableProj1_TLB.pas** — файл библиотеки типов в синтаксисе языка Object Pascal.

Форма, входящая в проект, является основой, на которой будет создаваться форма ActiveForm.

3. Поместите на форму **TTable**, **TDataSource** и **TDBGrid**. Им будут присвоены имена **Table1**, **DataSource1** и **DBGrid1** соответственно. Подключите компонент **Table1** к таблице **Физические лица** базы данных **sales**. Установите свойство **DataSet** компонента **DataSource1** равным **Table1**, а свойство **DataSource** компонента **DBGrid1** равным **DataSource1** (таким образом мы настраиваем компонент **DBGrid1** на отображение данных из таблицы **Физические лица**).

Теперь следует задать свойства, обеспечивающие открытие и закрытие базы данных из программы, использующей форму ActiveForm. Сделать это можно по-разному. Мы создадим одно дополнительное свойство **ActiveDb**, с помощью которого будем выполнять операции открытия и закрытия базы данных (при

присваивании этому свойству значения true база данных будет открываться, а при присваивании значения false — закрываться). Свойство ActiveDb фактически станет интерфейсом, с помощью которого мы будем обращаться к свойству Active компонента Table1 (напомним, что прямой доступ к свойствам компонентов, размещенных на форме ActiveForm, из программы, в которой этот компонент используется, невозможен).

4. Для создания свойства переключитесь в окно редактора библиотеки типов, выберите в иерархическом списке интерфейс ITableX, щелкните на стрелочке, расположенной рядом с кнопкой New Property на панели инструментов, и в раскрывшемся меню выберите команду Read ► Write. Присвойте созданному свойству имя ActiveDb. Добавленное свойство отобразится в списке как пара свойств — одно для чтения, второе для записи (рис. 18.13). Тип каждого свойства задайте равным VARIANT_BOOL.

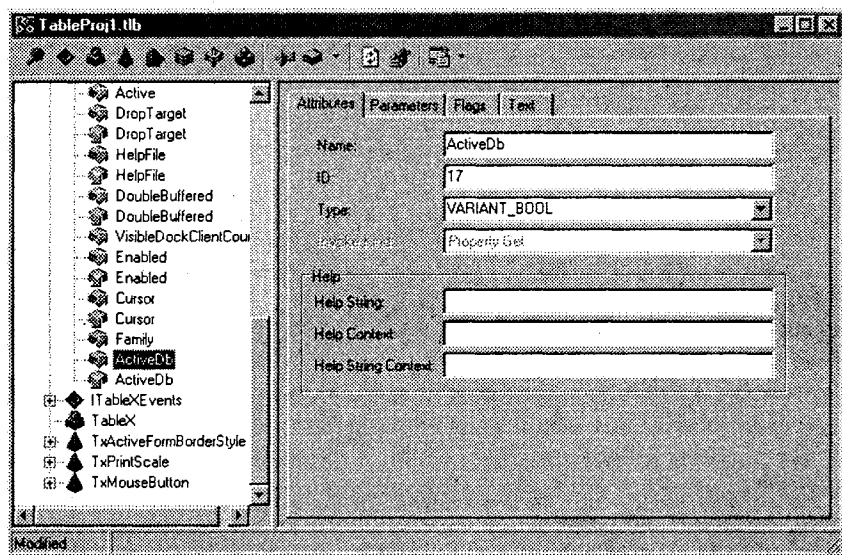


Рис. 18.13. Включение новых свойств в элемент ActiveForm

5. Щелкните на кнопке Refresh Implementation панели инструментов. При этом в файлы проекта будут внесены изменения, соответствующие модификациям, произведенным в редакторе библиотеки типов: к интерфейсу ITableX будет добавлено свойство ActiveDb и пара методов Get_ActiveDb и Set_ActiveDb соответственно для чтения и записи свойства. В модуль реализации также будут добавлены заголовки этих методов (в описание класса TTableX). Реализацию этих методов напишите вручную:

```
function TTableX.Get_ActiveDb: WordBool;
begin
    result:= Table1.Active;
end;

procedure TTableX.Set_ActiveDb(Value: WordBool);
```

```
begin
  Table1.Active:=Value;
end;
```

6. Теперь необходимо задать свойства, через которые будет производиться передача информации от компонента Table1 приложению, использующему элемент ActiveForm. Чтобы не усложнять наш пример, будем передавать только один параметр — текущее значение поля Фамилия. Для этого нам понадобится одно дополнительное свойство интерфейса ITableX, которому мы присвоим имя Family. Предположим, что нам не требуется модифицировать информацию в базе данных, поэтому это свойство будет только для чтения. Создается оно точно так же, как и рассмотренное свойство ActiveDb, только при его создании следует в меню кнопки New Property выбрать команду Read Only. Тип данного свойства сделайте равным BSTR (он соответствует типу WideString Object Pascal).

7. Щелкните на кнопке Refresh Implementation на панели инструментов, переключитесь в окно редактора кода и задайте следующий код реализации метода Get_Family в модуле:

```
function TTableX.Get_Family: WideString;
begin
  result:= Table1.FieldName('Фамилия').AsString;
end;
```

8. Последнее, что мы сделаем, — реализуем возможность задания обработчика события AfterScroll компонента Table1, размещенного на форме ActiveForm. Это позволит в приложении определить, что положение курсора данных в таблице Физические лица изменилось, и следует обновить отображаемую информацию. Обработчик события создается просто добавлением метода к интерфейсу ITableXEvents. Переключитесь в окно редактора библиотеки типов, выберите в иерархическом списке этот интерфейс, добавьте к нему новый метод и измените имя добавленного метода на OnScroll. После щелчка на кнопке Refresh Implementation в файлы проекта будут внесены необходимые изменения. Однако после этого следует также произвести вручную некоторые модификации в файле библиотеки типов. Это связано с тем, что по умолчанию считается, что событие, добавляемое к интерфейсу ITableXEvents, имеет тип TNotifyEvent. Однако в случае события AfterScroll компонента TTable это не так — данное событие имеет тип TDataSetNotifyEvent. Поэтому вручную измените тип поля и свойства, соответствующие созданному событию:

```
TTableX = class(TOLEControl)
private
  ...
  FOnScroll: TDataSetNotifyEvent;
  ...
public
  ...
  property OnScroll: TDataSetNotifyEvent read FOnScroll
  write FOnScroll;
  ...
end;
```

9. Кроме того, добавьте к списку используемых модулей в разделе uses файла библиотеки типов модуль Db с описанием типа TDataSetNotifyEvent:

```
uses
  Windows, ActiveX, Classes, Graphics, OleServer,
  OleCtrls, StdVCL, Db;
```

10. Затем следует вручную внести некоторые изменения в файл реализации формы `ActiveForm`. Прежде всего, добавьте к разделу `private` описания класса `TTableX` заголовок процедуры, точно соответствующий заголовку метода-обработчика события `AfterScroll` компонента `TTable`. Название добавляемой процедуры не играет роли, но желательно, чтобы из названия было понятно ее назначение. Назовем ее `ScrollEvent`:

```
TTableX = class(TActiveForm, ITableX)
DataSource1: TDataSource;
Table1: TTable;
DBGrid1: TDBGrid;
private
{ Private declarations }
FEvents: ITableXEvents;
...
procedure ScrollEvent(DataSet: TDataSet);
protected
{ Protected declarations }
...
function Get_ActiveDb: WordBool; safecall;
procedure Set_ActiveDb(Value: WordBool); safecall;
public
{ Public declarations }
procedure Initialize; override;
end;
```

11. Отредактируйте код реализации метода `Initialize`:

```
procedure TTableX.Initialize;
begin
  inherited Initialize;
  OnActivate := ActivateEvent;
  OnClick := ClickEvent;
  OnCreate := CreateEvent;
  OnDbClick := DbClickEvent;
  OnDeactivate := DeactivateEvent;
  OnDestroy := DestroyEvent;
  OnKeyPress := KeyPressEvent;
  OnPaint := PaintEvent;
  Table1.AfterScroll := ScrollEvent;
end;
```

Здесь мы добавили последнюю строку. Данный метод выполняется при инициализации формы `ActiveForm`. Благодаря добавленной нами строке обработчиком события `AfterScroll` компонента `Table1` будет определенная нами процедура `ScrollEvent`.

12. Последнее, что осталось сделать, — задать код реализации процедуры `ScrollEvent`:

```
procedure TTableX.ScrollEvent(DataSet: TDataSet);
begin
  if FEvents <> nil then FEvents.OnScroll;
end;
```

На этом разработка нашего элемента `TableX` завершается. Остается только выполнить его компиляцию (в результате которой будет создан файл `TableProj1.ocx`). По-

сколько большая часть кода генерируется автоматически, то ошибки компиляции могут возникнуть только в небольших фрагментах программы, которые мы добавляли вручную, и их обнаружение не представляет сложности. После компиляции проекта следует зарегистрировать элемент в системе и установить его в палитру компонентов IDE Delphi. Эти процедуры полностью аналогичны соответствующим процедурам для обычного элемента ActiveX, уже рассмотренным нами в этой главе. Разработанный элемент ActiveForm можно использовать как обычный компонент Delphi при создании любых приложений. Более того, этот компонент можно задействовать не только в Delphi, но и в любых других средствах разработки, поддерживающих элементы ActiveX (например, в Visual Basic).

Часть VI

Программирование для Интернета

ГЛАВА 19 Особенности Интернет-приложений

В данной главе приводятся основные сведения о способах взаимодействия компьютеров через Интернет, рассматриваются основные типы веб-приложений и способы публикации данных в Интернете.

Основные сведения об Интернете

Рассмотрим некоторые основные сведения об Интернете и дадим определения ряда терминов, связанных с программированием для Интернета.

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. Основным принципом 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 — протокол Интернета).

ПРИМЕЧАНИЕ

Под протоколом понимаются правила и описания работы сети, включая правила установления и поддержания связи в сети, описания IP-пакетов, правила обращения с IP-пакетами и их обработки.

Сеть проектировалась таким образом, чтобы для работы в ней не требовалось никакой информации о конкретной структуре сети. Для передачи информации по сети компьютер-источник помещал данные в некий «конверт», указывая на этом «конверте» конкретный адрес компьютера-приемника и передавал получившийся в результате этих действий пакет в сеть.

Именно протокол IP, являющийся очень удачным средством 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 Интернета 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 сетей, не использующих протокол IP. Поэтому приведенное определение сейчас уже не совсем корректно.

Многоуровневая сетевая модель

Для управления сетевым обменом данными используются 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 передачу пакета. 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, определяющая четыре уровня обмена данными:

- ☐ уровень сетевого доступа;
- ☐ межсетевой уровень;
- ☐ транспортный уровень;
- ☐ уровень приложений.

Нижние уровни трактуют пакеты высших уровней как данные, к которым добавляется служебная информация 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. Данный уровень 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 логическим каналом).

MAC-уровень управляет доступом к сети (с передачей маркера в сетях Token Ring или распознаванием конфликтов в сетях Ethernet) и обеспечивает управление сетью. LLC-уровень, действующий поверх MAC-уровня, 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 с 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 оптимального движения в точку назначения.

ПРИМЕЧАНИЕ

Процесс определения пути пакета называется маршрутизацией.

Для осуществления маршрутизации каждый узел имеет таблицу, называемую таблицей маршрутизации. В таблице маршрутизации адресу точки назначения ставится в соответствие адрес узла, на который следует послать данные. В Интернете составление и модификация таблиц маршрутизации (этот процесс тоже является частью маршрутизации и также называется маршрутизацией) определяются протоколами ICMP (Internet Control Message Protocol — протокол управляющих сообщений Интернета), RIP (Routing Internet Protocol — протокол маршрутизации в Интернете) и OSPF (Open Shortest Path First — открытие кратчайшего пути первым). Узлы, выполняющие функции маршрутизации, называются *маршрутизаторами*.

Адресация пакетов на межсетевом уровне обеспечивается протоколом IP. В заголовке пакета помещается информация, называемая IP-адресом, которой достаточно, чтобы определить, куда и как доставить пакет данных. IP-адрес состоит из че-

тырех байтов. 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 собственные протоколы сетевого уровня (например, локальные сети Ethernet для отыскания Ethernet-адреса по IP-адресу компьютера, находящегося в данной сети, использует протокол ARP).

Информация, пересылаемая по IP-сетям, делится по границам байтов на пакеты. Размер пакета обычно лежит в диапазоне от 1 до 1500 байт.

Транспортный уровень

Транспортный уровень определяет правила обслуживания сетевых соединений. Типичным протоколом транспортного уровня является протокол TCP (Transmission Control Protocol — протокол управления передачей). Протокол TCP занимается проблемой пересылки больших объемов информации, основываясь на возможностях протокола IP. TCP делит информацию, которую надо переслать, на несколько частей и нумерует их, чтобы обеспечить возможность последующего восстановления. Каждая порция информации вместе с номером образует TCP-пакет. TCP-пакет затем помещается в отдельный IP-пакет, с которым сеть уже «умеет» обращаться.

Получатель (TCP-процесс) распаковывает IP-пакеты и получает TCP-пакеты, далее распаковывает их и объединяет данные. Если какой-то информации недостает, TCP 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 TCP решает и эту проблему. Для этого используется специальная система кодов, исправляющих ошибки. Наиболее простым примером таких кодов является код, добавляющий к каждому пакету поле контрольной суммы (а к каждому байту — бит контроля четности). При создании TCP-пакета вычисляется контрольная сумма, которая записывается в TCP-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 пакет следует переслать заново.

Таким образом, протокол TCP обеспечивает гарантированную доставку пакетов, освобождая прикладные процессы от необходимости для надежности использовать режим ожидания и повторные передачи.

ПРИМЕЧАНИЕ

Как уже отмечалось, протокол TCP тесно связан с протоколом IP. Поэтому эти протоколы вместе часто именуют стеком TCP/IP. Этим термином обычно называют все, что связано с протоколами TCP и IP. Стек TCP/IP охватывает целое семейство протоколов, прикладные программы и даже саму сеть.

Уровень приложений

На *уровне приложений* определяется, как компьютер обрабатывает полученные данные. Для поддержки уровня приложений разработано несколько протоколов:

- ☐ HTTP (Hypertext Transfer Protocol — протокол передачи гипертекста);
- ☐ FTP (File Transfer Protocol — протокол передачи файлов);
- ☐ SMTP (Simple Mail Transfer Protocol — простой протокол передачи почты);
- ☐ POP (Post Office Protocol — протокол почтового отделения).

Особенностью протоколов уровня приложений является то, что обмен служебной информацией между ними производится в символьном виде.

Адресация в Интернете

Как уже отмечалось, компьютеры в Интернете идентифицируются по IP-адресу, уникальному в пределах всего Интернета. Однако пользователям крайне неудобно обращаться к требуемому серверу по IP-адресу, так как для человека IP-адреса не имеют никакого смысла и трудны для запоминания. Поэтому серверам Интернета присваивают *символьные адреса*. Все Интернет-приложения наряду с числовыми IP-адресами поддерживают символьные имена.

Доменная система имен

При присваивании серверу символьного имени используется так называемая *доменная система имен* (Domain Name System, DNS), основанная на иерархии доменов. В соответствии с ней домен на каждом уровне определяет имена подчиненных уровней. Доменный адрес имеет вид нескольких идентификаторов, разделенных точками:

домен_п. домен_2. домен_1

Чем дальше (правее) расположен в адресе домен, тем шире охватываемая им область. Домен высшего уровня (самый правый) представляет собой либо двухбуквенный код 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 первоначально применялась исключительно в США, но в настоящее время множество подобных имен принадлежат компаниям и организациям, расположенным за пределами США. Трехбуквенные домены имеют следующий смысл:

- ☐ com — коммерческие организации (эта международная доменная зона 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 делается запрос к так называемому DNS-серверу — узлу, обладающему 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 не обязательно 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 Интернет-службы, 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 применяются протоколы SMTP, POP3, IMAP и т. д.

Для упорядочения работы каждой службе отведен отдельный *порт*, представляющий собой число от 0 до 65 534. Для наиболее популярных служб зарезервированы стандартные номера портов. Так, для FTP — это 21, для HTTP — 80, для SMTP — 25, для POP3 — 110. Однако это всего лишь общепринятые значения, поэтому владелец узла может настроить эти службы на работу с совершенно другими портами. Часто таким образом удается легко решать некоторые проблемы, например, обеспечить поддержку различных кодировок кириллицы в Web. Для этого достаточно предусмотреть автоматическую перекодировку документа на сервере в зависимости от того, с каким портом общается клиентское приложение.

Унифицированный указатель ресурсов

Унифицированный указатель ресурсов (Uniform Resource Locator, URL) предназначен 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 — применяется 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 отсутствует, используется порт, выделенный по умолчанию для данного протокола);
- ☐ путь — путь на сервере от его корневого или 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. Дан также обзор наиболее распространенных типов веб-приложений и особенностей их использования.

Основные понятия и термины

В настоящее время наиболее развитой частью Интернета является Всемирная паутина (World Wide Web, WWW, или Web) — система публикации ресурсов, представленных в виде гипертекстовых документов. Под *публикацией* обычно понимается размещение 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 трактовок. Так обычно называют компьютер, предназначенный для публикации гипертекстовых документов, или программный продукт для доступа к гипертекстовым документам, расположенным на компьютере. В последнем случае веб-сервер обеспечивает обработку запросов, поступающих от клиентов по протоколу HTTP. Во избежание путаницы в дальнейшем изложении под сервером мы будем понимать только программный продукт.

Для просмотра гипертекстовых документов (которые часто называются также веб-страницами) используются специальные программы, называемые *браузерами*. На сегодняшний день наиболее известными являются браузеры Microsoft Internet

Explorer и Netscape Navigator. Программа-браузер выполняет интерпретацию команд языка разметки гипертекста (HTML) и отображает содержимое HTML-документа.

По структуре веб-страницы можно разделить 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 (например, информацию из базы данных), которые могут изменяться без изменения самого HTML-документа.

Для создания динамических HTML-страниц обычно используют специальные серверные расширения, называемые *сценариями* (иногда также 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 реализуются, например, возможности поиска в базе данных и выборки из базы данных именно той информации, которую запрашивает пользователь.

Веб-дизайн и веб-программирование

Реализация и поддержка веб-серверов предполагает решение трех основных задач.

1. Подготовка материалов к веб-публикации, редактирование, дизайн, соблюдение единого стиля и единообразного оформления веб-страниц, поддержание связности веб-документов и т. п.
2. Обеспечение динамического представления информации на веб-странице: создание интерактивных веб-страниц, организация разных видов поиска информации, ведение статистики посещений страницы, регистрация пользователей, регламентация доступа к страницам, организация доступа к базам данных.
3. 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 используются специальные визуальные средства разработки HTML-документов (например, Microsoft FrontPage, можно применять для этих целей и Delphi 8).

ПРИМЕЧАНИЕ

В настоящее время для оформления веб-страниц, помимо языка HTML, широко применяются языки Java, JavaScript, VBScript, предоставляющие гораздо более широкие возможности по форматированию информации, чем язык HTML.

Вторая задача подразумевает разработку средств, расширяющих возможности веб-сервера, и поэтому решается специалистами в области программирования. Цель состоит в реализации возможности динамического изменения содержимого страниц в сочетании с возможностью интерактивного режима работы. Данная задача решается с помощью 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. Наиболее распространенными из них являются Delphi и Visual Basic (практически все современные средства разработки Windows-приложений обеспечивают возможность разработки веб-приложений).

В процессе создания 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 разработку сценариев, необходимых для организации динамических документов.

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, протоколы уровня приложений могут обмениваться только текстовой информацией. Для передачи двоичных файлов по протоколу HTTP используется спецификация MIME (Multipurpose Internet Mail Extension — многоцелевые расширения почты Интернета). 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 HTTP-сервером в наиболее общем виде состоит из нескольких шагов.

1. Установление TCP-соединения.
2. Запрос клиента.
3. Ответ сервера.
4. Разрыв TCP-соединения.

Запрос клиента представляет собой просто требование на передачу HTML-документа или какого-либо иного 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 методом GET сервер должен включить код ресурса в ответ клиенту, причем ресурс не обязательно 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 в теле запроса.
- ☐ Content-Length — число символов 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. Обычно 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 важнее, 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>

- ☐ Версия протокола задается в том же формате, что и в запросе клиента.
- ☐ Код ответа представляет собой трехзначное десятичное число, обозначающее результат обработки запроса клиента 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, и 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. 19.1.

Таблица 19.1. 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 Windows. С помощью нее можно установить связь с любым сервером, имя которого указывается пользователем (для установления связи с веб-сервером следует задействовать 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 должен знать этот язык. Результатом выполнения сценария, как правило, является HTML-документ. 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 годится 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 тегов. 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:

- ☐ раздел заголовка определяется тегом `HEAD`;
- ☐ тело включает собственно содержимое документа и определяется тегом `BODY`.

В общем виде HTML-документ 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 рассмотрено далее);
- ☐ `<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 — это `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 специальные теги обеспечивают форматирование текста документа и включают в него изображения, таблицы и формы.

Теги форматирования текста

Приведем описание основных тегов, используемых для форматирования 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 задать 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 абзацев можно задать способ выравнивания текста с помощью атрибута `ALIGN`.

Если требуется начать текст с новой строки в пределах одного абзаца, то используется тег `<BR>`. При этом тип выравнивания не меняется.

Списки

HTML-документ может содержать как маркированные, так и нумерованные списки. Для создания маркированных списков используются теги `<UL>` и `</UL>`, нумерованных — `<OL>` и `</OL>`. В обоих случаях каждый элемент списка помещается между тегами `<LI>` и `</LI>`. Допускается создание вложенных списков.

Выделение фрагментов текста

Язык HTML позволяет выделять отдельные слова и даже символы документа. Приведем основные теги, используемые при выделении:

- ☐ `<B> ... </B>` — выделение полужирным шрифтом;
- ☐ `<I> ... </I>` — выделение курсивом;
- ☐ `<U> ... </U>` — выделение подчеркиванием.

Указанные теги задают способ выделения фрагмента текста явным образом. Наряду с ними имеются теги, которые лишь указывают, что текст должен быть выделен, не обозначая способа выделения. В этом случае выбор способа выделения определяется браузером:

- ☐ `<EM> ... </EM>` — выделенному тексту должно уделяться внимание;
- ☐ `<STRONG> ... </STRONG>` — 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 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 Internet Explorer показан на рис. 19.1.

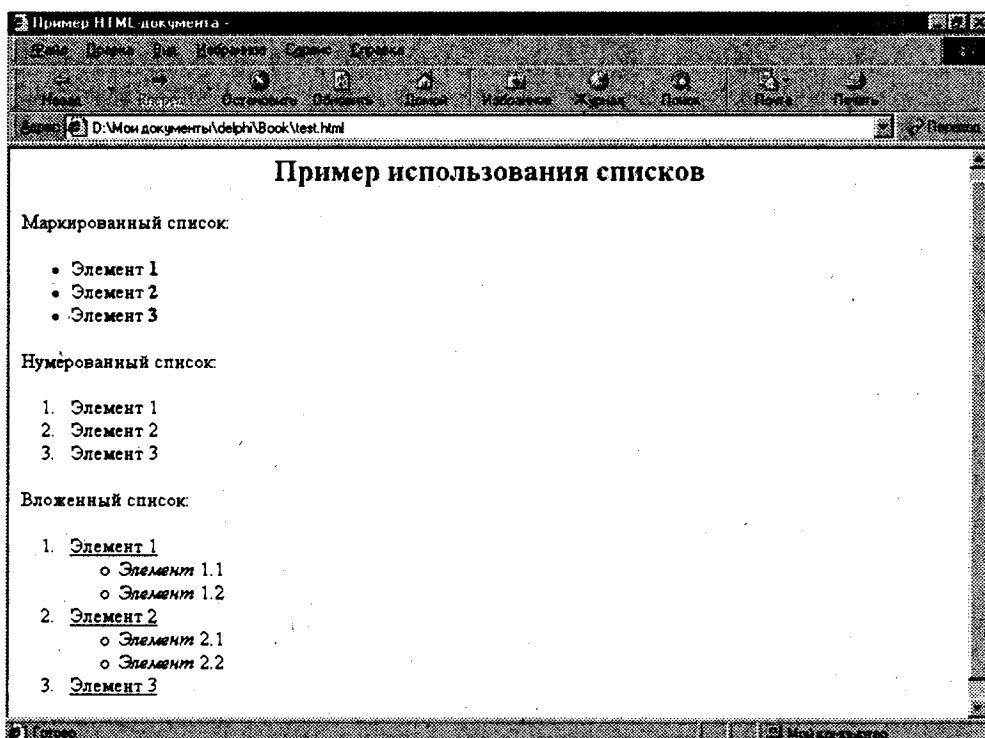


Рис. 19.1. Пример HTML-документа

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 `</A>`. Заключенный между ними фрагмент HTML-документа при просмотре будет отображаться как гиперссылка. Тег `<A>` обязательно должен указываться совместно с атрибутом `HREF`. С его помощью 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 абсолютным и относительным.

❑ *Абсолютный URL-адрес* уже был нами рассмотрен. В нем содержится полная информация о местоположении ресурса и протоколе обращения к ресурсу.

❑ *Относительный URL-адрес* задает расположение ресурса относительно местоположения текущего HTML-документа. Например, следующая строка создает гиперссылку, указывающую на гипертекстовый документ, содержащийся в файле `doc1.html`, который размещен в том же каталоге, что и текущий документ:

```
<A HREF=doc1.html> Переход к документу 1 </A>
```

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 — помимо создания гиперссылок, он позволяет устанавливать маркеры для организации переходов по гиперссылкам в пределах одного HTML-документа. 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 `</A>`, 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 интерактивного взаимодействия 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. Данный атрибут может принимать одно из двух значений: `GET` или `POST`.

Основные элементы управления создаются с помощью тега `<INPUT>`, который используется без закрывающего тега. Тип элемента управления задается с помощью атрибута `TYPE` данного тега. Помимо 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 сразу после загрузки документа;
- ☐ `SIZE` — максимальное количество отображаемых символов;
- ☐ `MAXLENGTH` — максимальное количество символов, которые могут быть введены с помощью поля.

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 — поля, предназначенные для ввода пароля. Для создания такого поля ввода атрибуту `TYPE` следует задать значение `"PASSWORD"`. Все символы, вводимые в этом поле, отображаются 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 элемент управления, подобный флажку. Однако в отличие от флажков в установленном состоянии может находиться только один из переключателей, 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 переключателей, входящих в одну группу);
- ☐ **VALUE** — значение, передаваемое серверу при установленном переключателе;
- ☐ **CHECKED** — атрибут, указывающий, какой из переключателей должен быть установлен в группе при загрузке документа.

Следующий фрагмент HTML-кода содержит описание группы из трех переключателей:

```
<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** возвращает все элементы управления в исходные состояния.

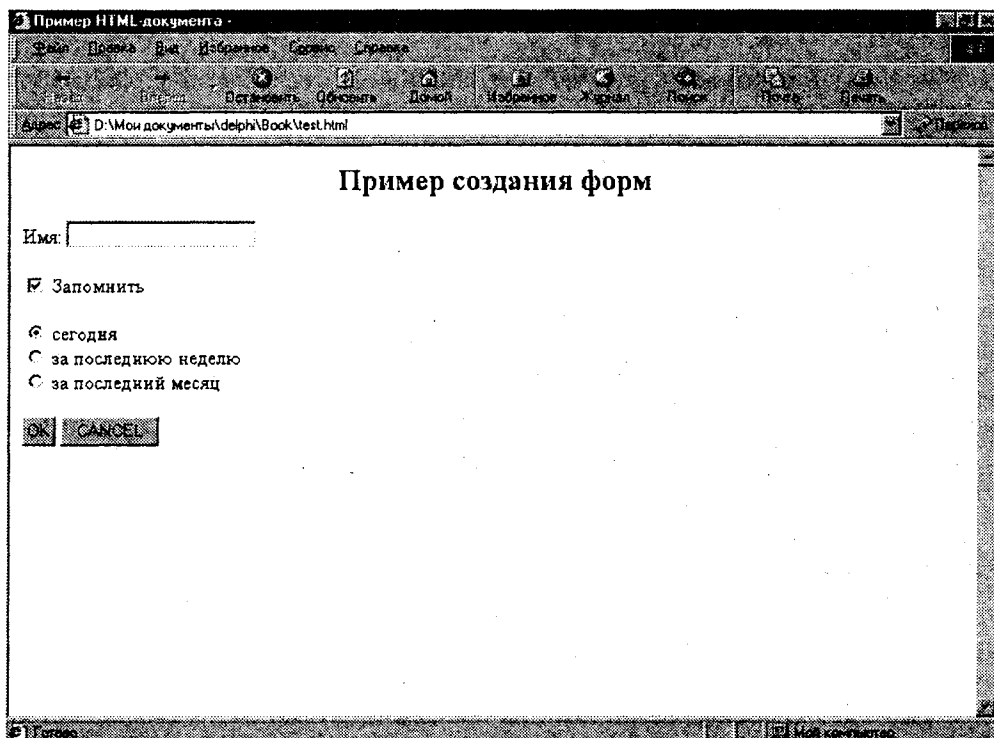


Рис. 19.2. Пример HTML-формы

Для создания кнопок атрибуту TYPE присваивается либо значение "SUBMIT", либо значение "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 Internet Explorer показан на рис. 19.2.

ПРИМЕЧАНИЕ

Среда проектирования Delphi 8 предоставляет возможности визуального проектирования HTML-страниц. Для этого необходимо выполнить команду File ► New ► Others, а затем выбрать на соответствующей вкладке значок HTML-документа.

Язык XML

Расширяемый язык разметки (Extensible Markup Language, XML) является дальнейшим развитием языков разметки веб-документов. В настоящее время XML — общепринятый открытый стандарт.

С помощью языка XML разработчики получили возможность создавать 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.

Как и HTML, язык XML основан на применении тегов разметки. Для примера определим правила своей разметки путем *описания типа документа* (Document Type Definition, DTD). В DTD указывается, каким образом пользоваться новой языковой конструкцией, какие элементы могут содержать другие элементы, каковы значения атрибутов и т. д.

Простой DTD-шаблон, описывающий классификационный язык для учащихся института, мог бы выглядеть, например, так:

```
<!-- 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= "М" 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.

Область применения языка XML постоянно расширяется. Это связано с тем, что XML позволяет представлять данные не в двоичном, а в текстовом формате (что очень удобно для их передачи и обработки), а также описывать сколь угодно сложно организованные данные и их структуру.

Очень полезна возможность преобразования наборов данных из различных внутренних форматов (например, баз данных) в стандартный формат XML (текстовый) и наоборот. Эта возможность позволяет обеспечивать совместную 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 передаче клиенту запрашиваемых ресурсов по протоколу HTTP. 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 много разнообразных приложений, использующихся в качестве расширений веб-серверов. Мы рассмотрим только основные из них, которые могут создаваться средствами разработки приложений, функционирующих под управлением операционной системы Windows:

- ☐ CGI-сценарии;
- ☐ ISAPI-расширения;
- ☐ ASP-страницы.

CGI-сценарии

CGI-сценарии можно отнести к классике Интернет-приложений — это первый и общепринятый интерфейс для создания расширений веб-серверов (аббревиатура CGI расшифровывается как Common Gateway Interface — интерфейс общего шлюза). 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 скорость взаимодействия с сервером довольно низка;
 - ❑ производить обмен данными через переменные окружения довольно неудобно.
- Указанные недостатки преодолены в других типах Интернет-приложений, в частности, таких как ISAPI-расширения и ASP-страницы. Но тем не менее CGI-сценарии до сих пор очень широко распространены в WWW. Это объясняется их универсальностью — они поддерживаются практически всеми существующими веб-серверами, работающими на любых платформах. Кроме того, CGI-сценарий 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 технология CGI имеет широкое распространение.

ПРИМЕЧАНИЕ

Что касается серверов, работающих под управлением операционной системы Windows NT, то для них технология CGI считается устаревшей и, вероятно, вскоре будет вытеснена другими технологиями Интернет-приложений.

ISAPI-расширения

Спецификация ISAPI (Internet Server Application Programming Interface — прикладной программный интерфейс для Интернет-серверов), 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 ISAPI-расширение представляет собой динамическую библиотеку (DLL), которая при вызове загружается не как отдельный процесс, а как программный поток, принадлежащий веб-серверу. Благодаря этому ISAPI-расширения обладают тремя существенными преимуществами перед CGI:

- ❑ программный поток требует существенно меньших ресурсов, чем отдельный процесс, что приводит к меньшей загрузке сервера;
- ❑ ISAPI-расширение выполняется в адресном пространстве веб-сервера, поэтому работает быстрее, чем отдельный процесс;
- ❑ в отличие от CGI, ISAPI-расширение может оставаться постоянно загруженным в память, а не загружаться каждый раз при поступлении нового запроса, как CGI, благодаря чему снижается загрузка сервера 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 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 достоинствами ISAPI-расширений, поскольку обусловлен тем, что ISAPI-расширения выполняются в одном адресном пространстве с веб-сервером.

ASP-страницы

ASP (Active Server Pages — активные серверные страницы) является наиболее современной из рассматриваемых технологий. В принципе, технология ASP не предлагает новых концепций — сценарий выполняется на сервере, а клиенту отправляется формируемый им HTML-файл. Входные данные для ASP-страниц поступают от клиентских приложений с помощью методов GET и POST.

Наиболее существенным отличием технологии ASP от технологий CGI и ISAPI является то, что сценарий ASP-страницы формирует не полный HTML-документ, а лишь его часть, добавляемую к исходному документу, из которого производится вызов сценария. Более того, в одном HTML-документе может содержаться несколько обращений к разным ASP-серверам. Посылаемый клиенту результирующий HTML-документ формируется на основе откликов всех ASP-сценариев.

Код ASP-сценариев, написанный на языках C#, Visual Basic (VBScript) или JScript (вариант JavaScript от Microsoft), может включаться непосредственно в текст HTML-документа. Начало ASP-сценария указывается с помощью тега `<%`, окончание — с помощью тега `%>` (блок генерации кода).

Технология ASP получила дальнейшее развитие в платформе .NET. Технология ASP.NET упрощает разработку веб-страниц за счет программирования на базе форм. В ASP.NET эти формы называются веб-формами (web forms). Они во многом похожи на формы Delphi.

Создание приложения ASP.NET начинается с выполнения команды **File ► New ► ASP.NET Forms Application**.

ПРИМЕЧАНИЕ

В отличие от сценариев, созданных по технологии ASP и имеющих расширение `asp`, приложения ASP.NET имеют расширение `aspx`.

Delphi предложит выбрать тип сервера, для которого создается приложение, ввести имя приложения (рис. 19.3). Если на момент разработки приложения выбранный тип сервера недоступен, Delphi предложит его эмулировать.

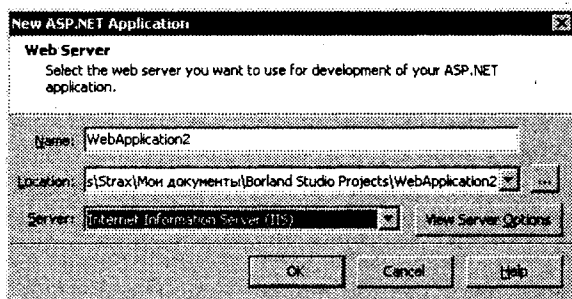


Рис. 19.3. Создание приложения ASP.NET

Свойства приложения можно задать в окне настройки проекта, выбрав команду **Project ► Options** (рис. 19.4).

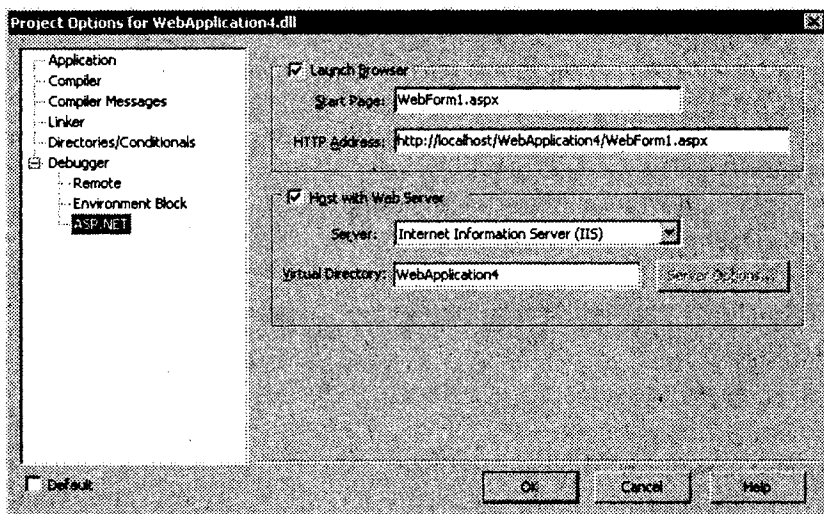


Рис. 19.4. Настройка свойств проекта ASP.NET

Программирование приложения ASP.NET базируется на событиях. С помощью мыши мы можем создавать сколь угодно сложный интерфейс пользователя, перетаскивая плавучие элементы управления из палитры компонентов на форму и задавая их свойства. Однако в плане задания свойств элементов управления разработка приложения ASP.NET существенно отличается от разработки обычного приложения.

ПРИМЕЧАНИЕ

Следует помнить, что, создавая приложение ASP.NET, мы работаем с библиотекой стандартных классов .NET, а не VCL.

Веб-формы в Delphi можно конструировать с помощью редактора форм и редактора кода. В редакторе кода веб-формы (ASP-страницы) представляют собой текстовые файлы, состоящие из HTML-тегов и управляющих тегов, таких как директивы и блоки сценариев. Пример описания веб-формы приведен в листинге 19.1.

Листинг 19.1. Описание веб-формы

```
<%@ Page language="c#" Debug="true"
Codebehind="WebForm1.pas" AutoEventWireup="false"
Inherits="WebForm1.TWebForm1" %>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">

<html>
  <head>
    <title></title>
    <meta name="GENERATOR"
      content="Borland Package Library 7.1">
  </head>

  <body ms_positioning="GridLayout">
```

```

<form runat="server">
  <div style="DISPLAY: inline; Z-INDEX: 1; LEFT: 118px;
    WIDTH: 323px; POSITION: absolute; TOP: 62px;
    HEIGHT: 126px"
    ms_positioning="FlowLayout">
    <font color=#0000ff size=7><strong>
      Hello. Word!</strong></font>
    </div>
  </form>
</body>
</html>

```

Существует десять различных синтаксических элементов ASP.NET:

- ☐ директивы;
- ☐ блоки объявления кода;
- ☐ блоки генерации кода;
- ☐ синтаксис элементов управления HTML;
- ☐ синтаксис нестандартных элементов управления;
- ☐ выражения для привязки данных;
- ☐ теги серверных объектов;
- ☐ директивы включения файлов на стороне сервера;
- ☐ серверные комментарии;
- ☐ текстовые литералы.

Остановимся на некоторых из них.

Директивы имеют формат

```
<%@ directive [атрибут=значение] %>
```

Например, следующая директива означает, что в веб-форме возможен вызов сценария на языке C# (произносится си-шарп):

```
<%@ Page Language="C#" %>
```

А эта директива добавляет на страницу ссылку на пространство имен System.Data:

```
<%@ Import Namespace="System.Data" %>
```

После этого в коде страницы можно обращаться ко всем классам и интерфейсам импортированного пространства имен. Для упрощения разработки автоматически импортируются во все страницы ASP.NET следующие пространства имен:

```

System
System.Collections
System.Collections.Specialized
System.Configuration
System.IO
System.Text
System.Text.RegularExpressions
System.Web
System.Web.Caching
System.Web.Security
System.Web.SessionState

```

```
System.Web.UI
System.Web.UI.HtmlControls
System.Web.UI.WebControls
```

Блоки объявления кода определяют код страницы, который должен быть проанализирован и запущен. В этих блоках атрибут `runat` указывает, клиентским или серверным является код. Для сервера значением атрибута является `server`. Если разработчик проигнорирует атрибут `runat`, IIS-сервер интерпретирует этот блок как клиентский.

Блоки объявления кода имеют формат

```
<script runat= "server"
    [language= "язык_программирования"]
    [src="имя внешнего файла кода" ]/>
```

```
Код
</script>
```

Для блоков объявления как клиентского, так и серверного кода можно задать атрибут `src`, указывающий на внешний исходный файл кода, что помогает отделить код HTML-страницы от кода сценариев. Значением атрибута может быть как относительный путь, так и URL-адрес.

Элементы управления HTML ASP-страниц схожи со стандартными элементами HTML, но располагаются не на стороне клиента, а на стороне сервера и имеют атрибуты `id` и `runat`. Например, следующий код представляет собой описание серверного элемента управления HTML (точнее, кнопки OK):

```
<input id="but1" runat="server" type="button" value="ok" />
```

При обнаружении такого элемента ASP.NET создает соответствующий объект `HTMLControl` типа `HTMLInputButton` с идентификатором `"but1"`. С событием такого элемента управления можно связать обработчик, который будет обрабатывать сообщения от него, например сообщение `onclick`.

Выражения привязки данных связывают серверные элементы управления с источниками данных и имеют синтаксис:

```
<%# выражение привязки %>
```

Например, следующий код на языке C# выполняет привязку содержимого элемента управления `Label` к открытому свойству `Test` веб-формы:

```
<html>
<head><title>Пример привязки данных</title></head>
<body>
<script language="C#" runat=server>
/* объявление переменной test типа string */
public string Test;
/* Обработчик события Load объекта Page (веб-формы) */
/* фигурные скобки в языке C обозначают составной
оператор (begin - end в Delphi) */
void Page_Load(Object oSender, EventArgs oEvent){
/* определение значения переменной */
Test = "Hello, World!\n";
/* выполнение привязки */
Page.DataBind();
}
```

```
<asp:Label text='<## Test %>' runat= server />
</body>
</html>
```

В ходе выполнения этого сценария ASP.NET при обработке события Load веб-формы атрибут text элемента управления Label примет значение "Hello, World!", что и отразится на экране.

ПРИМЕЧАНИЕ

Атрибут text можно заполнить из любого источника данных.

Следующая строка является объявлением элемента управления Label:

```
<asp:Label text='<## Test %>' runat= server />
```

Помимо пользовательского интерфейса, веб-форма имеет код, обрабатывающий события, связанные с расположенными на форме элементами управления (в приведенном примере это процедура Page_Load на языке C#). Совместное их размещение неудобно — в достаточно сложных приложениях возникает эффект «спагетти» (пользовательский интерфейс и код обработки событий перемешиваются). Поэтому код обработки событий выносится в отдельный файл, на который дается ссылка в свойстве Codebehind директивы Page. В листинге 19.1 это строка:

```
Codebehind = "WebForm1.pas"
```

В Delphi 8 мы можем обрабатывать события веб-формы с использованием синтаксиса Object Pascal. Например:

```
var Test : String;
procedure TWebForm1.Page_Load(sender: System.Object;
    e: System.EventArgs):
begin
    Test := "Hello, World!\n";
end;
```

События класса Page стандартной библиотеки FCL платформы .NET приведены в табл. 19.2.

Таблица 19.2. События класса Page

Имя события	Описание
Init	Генерируется для предоставления возможности инициализации страницы. Расположенные на странице элементы управления на данный момент еще не созданы. Это событие возникает один раз для каждого пользователя страницы
Load	Следует за событием Init. Возникает каждый раз при запросе страницы. Когда оно возникает, все дочерние элементы управления веб-формы должны быть загружены и доступны. Необходимо иметь возможность извлечь данные и заполнить элементы управления, чтобы они могли отобразить свое содержимое на странице при отправке ее клиенту
PreRender	Происходит непосредственно перед генерацией страницы и отправкой ее клиенту
Unload	Происходит, когда страница выгружается из памяти. Как правило, используется для финальной очистки

Доступ к базам данных через Интернет

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 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, имеющим право доступа к этому серверу.

Благодаря всем этим достоинствам доступ к базам данных на основе веб-технологии нашел применение и в локальных сетях. Такие сети, использующие технологию Web для доступа к данным, носят название *интрасетей* (intranets).

Модель взаимодействия с базой данных в рамках технологий доступа к базе данных, размещенной на стороне сервера, можно изобразить в виде схемы, приведенной на рис. 19.5.

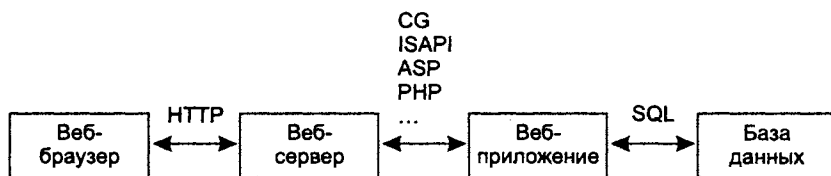


Рис. 19.5. Модель доступа к базе данных через Web

При обеспечении доступа к базам данных через Web возможен ряд технологических и организационных решений (рис. 19.6). Выбор конкретных решений зависит от специфики СУБД и ряда других факторов, таких как наличие специалистов, способных с минимальными издержками освоить определенную ветвь технологических решений, или существование других баз данных, доступ к которым через Web осуществляется с минимальными дополнительными затратами.

Для доступа к базам данных через Web наиболее часто используется один из двух подходов:

- ☐ однократное или регулярное преобразование содержимого базы данных в статические гипертекстовые документы;
- ☐ динамическое создание гипертекстовых документов на основе информации, содержащейся в базе данных, и информации, переданной клиентом веб-серверу.

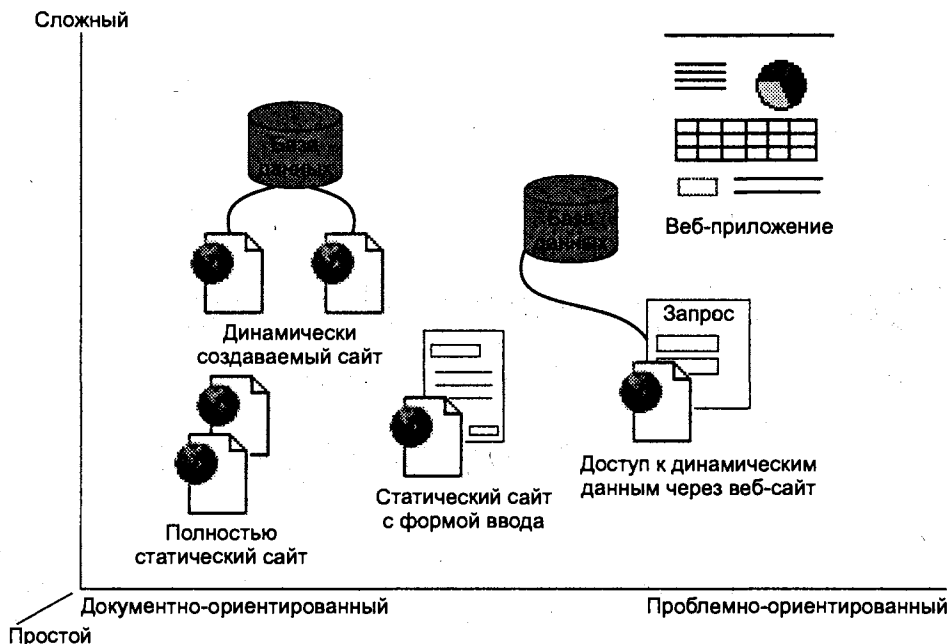


Рис. 19.6. Типы веб-сайтов

В случае однократного или регулярного преобразования содержимого базы данных в статические гипертекстовые документы база данных периодически просматривается специальной программой, создающей множество связанных HTML-документов, содержащих информацию из базы данных. Полученные HTML-файлы размещаются на одном или нескольких компьютерах, выполняющих функции веб-серверов. Доступ к ним осуществляется как к статическим гипертекстовым документам (полностью статический сайт на рис. 19.6). Этот вариант характеризуется минимальными начальными расходами. Он достаточно эффективен при работе с небольшими редко обновляемыми базами данных, имеющими простую структуру, а также при невысоких требованиях к актуальности данных, предоставляемых через Web.

В случае динамического создания гипертекстовых документов на основе информации, содержащейся в базе данных, и информации, переданной клиентом веб-серверу, доступ к базе данных обеспечивается специальным приложением (CGI, ISAPI, ASP, PHP и т. п.), вызываемым веб-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 20 Разработка Интернет- приложений

Современный подход к проектированию приложений для Интернета предполагает, что подобное проектирование 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.

Начиная с версии 7, в Delphi появилась группа компонентов Indy (сокращенно от Internet Direct), значительно упрощающих разработку приложений для Web и позволяющих разработчику не вникать в особенности Интернет-протоколов. Набор этих компонентов перенесен фирмой Borland и в Delphi версии 8. К сожалению, в Delphi 8, по крайней мере в варианте Architect, отсутствуют компоненты Indy для создания серверных составляющих (TIdTCPServer, TIdHTTPServer), поэтому нам придется воспользоваться возможностями, предоставляемыми Delphi 7.

Восьмая версия Delphi ориентирована на использование технологии Web Forms платформы .NET, поэтому значительная часть компонентов VCL Delphi 7 в версию 8 также не перенесена (TwebModule, TdataSetPageProducer и др.).

В данной главе рассматривается разработка приложений для Интернета с использованием компонентов Delphi версий 7 и 8.

Разработка CGI-сценариев

Для начала рассмотрим небольшой пример CGI-приложения, иллюстрирующий процесс передачи данных посредством методов GET и POST.

ПРИМЕЧАНИЕ

Пример адаптирован для версии Delphi 7.

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 HTML-тегов, которые были рассмотрены в предыдущей главе.

Запуск CGI-приложения

CGI-приложение может запускаться двумя способами:

- ☐ щелчком на кнопке SUBMIT формы (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 CGI-приложение 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 входящим в стандартную поставку Windows 98 сервером PWS (Personal Web Server — персональный веб-сервер).

После установки PWS на диске C: (по умолчанию) создается каталог Webshare, предназначенный для хранения публикуемых в Web документов и расширений сервера. Данный каталог содержит несколько вложенных каталогов, 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.

Разработка простейшего CGI-приложения

Как уже отмечалось, CGI-сценарии представляют собой традиционные консольные приложения, поэтому для их разработки не требуется никаких специальных средств. Для вывода результатов выполнения CGI-сценария 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 CGI-приложением. В большинстве случаев достаточно одного поля — Con-

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 строки «Hello, world!»).

1. Выберите в главном меню IDE Delphi команду **File ► New**, в открывшемся окне хранилища объектов выделите значок **Console Application** и щелкните на кнопке **OK**. Будет создан шаблон консольного приложения, имеющий следующий вид:

```
program Project1;
```

```
{$APPTYPE CONSOLE}
```

```
uses SysUtils;
```

```
begin
```

```
  // Insert user code here
```

```
end.
```

2. Введите следующий код:

```
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

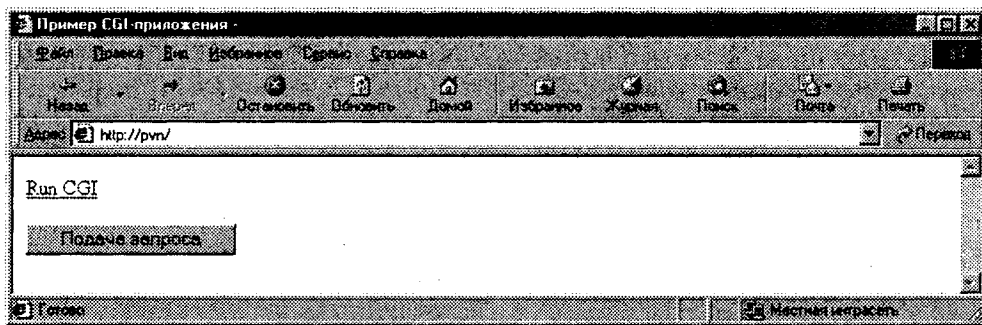
В результате компиляции будет создан исполняемый файл, по умолчанию имеющий расширение **exe**. PWS-сервер различает форматы запускаемых сценариев по расширению, поэтому исполняемый файл следует переименовать, присвоив ему расширение **cgi**.

4. Для тестирования полученного приложения необходимо создать HTML-документ, из которого будет производиться вызов CGI-приложения. Поскольку в на-

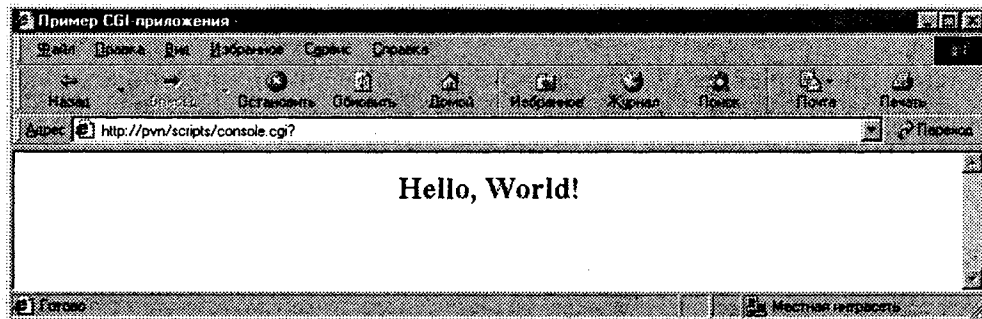
шем примере не требуется получать какие-либо данные от пользователя, то неважно, какой способ вызова использовать — кнопку или обычную ссылку. Создадим HTML-документ, в котором имеются оба средства вызова сценария:

```
<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 адресной строке либо имя `localhost`, либо IP-адрес `127.0.0.1`, либо имя своего компьютера. В окне браузера отобразится документ, содержащий одну ссылку и одну кнопку (рис. 20.1, а). При щелчке на любом из этих элементов в окне браузера отобразится документ, соответствующий ответу запущенного CGI-сценария. В нашем случае это просто строка текста «Hello, World!», выведенная по центру стилем заголовка второго уровня (рис. 20.1, б).



а



б

Рис. 20.1. Тестовый HTML-документ и результат выполнения простого CGI-сценария

Рассмотренный 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 элемента и его значение, разделенные знаком равенства. Различные (относящиеся к разным элементам управления) параметры разделяются в строке символами `&`.

Если символ `=` или `&` входит в состав имени или значения элемента управления, то он кодируется последовательностью из трех знаков: первый знак — `%`, за ним следуют две шестнадцатеричные цифры, являющиеся кодом символа (например, символ `=` кодируется как `%3D`, а символ `&` — как `%26`). Помимо этих двух знаков, трехсимвольными последовательностями обычно кодируются все знаки, за исключением латинских букв, цифр и символа пробела. Символ пробела заменяется символом плюс (`+`).

Полученную строку параметров прежде всего следует декодировать.

1. Разделение строки на пары
идентификатор_N=значение_N
2. Выделение в каждой паре идентификатора и значения.
3. Замена символа плюс (`+`) в каждом идентификаторе и каждом значении пробелом.
4. Преобразование каждой трехсимвольной последовательности, начинающейся со знака `%`, в символ ASCII.

Таким образом, алгоритм декодирования довольно прост и сводится к нескольким строковым операциям.

Методы передачи и получения строки параметров

Строка параметров может передаваться веб-серверу либо методом GET, либо методом 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, а для разделения URL-адреса и строки параметров используется символ вопроса (?);
- ❑ 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 CGI-приложению через стандартный поток ввода консольной программы, и длина строки в этом случае может быть определена через переменную окружения `CONTENT_LENGTH`.

Считывание строки параметров методом GET

Для получения строки параметров в CGI-приложении методом GET можно использовать следующую функцию Win32 API, возвращающую значение переменной окружения с заданным именем:

```
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 строку Паскаля
    St1 := StrPas(buff);
    // Освобождаем память
    FreeMem(buff);
    ...
end;
```

Считывание строки параметров методом 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, оказывается сложнее, так как требует двух процедур: обращения к переменной окружения и считывания данных из стандартного потока ввода:

```
var
  buff: PChar;
  ContentLength, i: Integer;
  Stl: 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
  Stl := '';
  for i := 1 to ContentLength do begin
    Read(C);
    Stl := Stl + 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, запустившем сценарий. Ряд переменных окружения, содержащих наиболее важную информацию, приведен в табл. 20.1.

Перед получением строки параметров обычно следует проверить, какой метод передачи информации использован. Это обеспечит правильность считывания передаваемой информации в любом случае.

Таблица 20.1. Основные переменные окружения

Название	Описание
REQUEST_METHOD	Метод передачи информации (GET или POST)
SERVER_NAME	IP-адрес или имя сервера
SERVER_PORT	Номер порта, используемый при обращении к серверу
SERVER_PROTOCOL	Название и версия протокола, по которому был передан запрос
PATH_INFO	Строка параметров, расположенная в запросе после имени сценария, но до данных запроса
REMOTE_ADDR	IP-адрес узла, с которого был послан запрос
REMOTE_HOST	Доменное имя узла, с которого поступил запрос

Проиллюстрируем это на примере. Объединим приведенные выше фрагменты программ таким образом, чтобы обеспечить корректное получение данных любым методом. 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;
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 формы, 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:

```

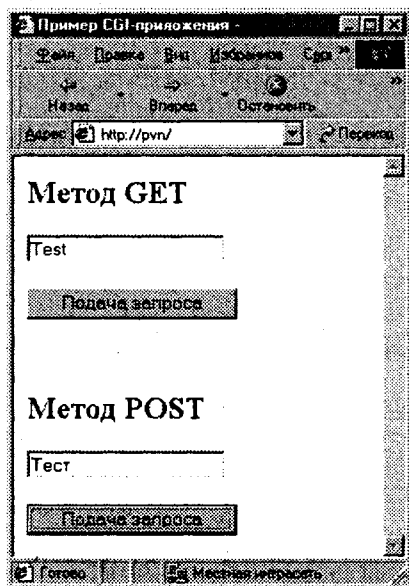
<HTML>
<HEAD>
  <TITLE> Пример CGI-приложения </TITLE>
</HEAD>
<BODY>
<H2> Метод GET </H2>
<FORM METHOD="GET" ACTION="/scripts/console.cgi">
<INPUT TYPE="TEXT" NAME="Edit1" VALUE="Test"><BR><BR>
<INPUT TYPE="SUBMIT">
</FORM>
<BR><BR>
<H2> Метод POST </H2>
<FORM METHOD="POST" ACTION="/scripts/console.cgi">
<INPUT TYPE="TEXT" NAME="Edit1" VALUE="Test"><BR><BR>
<INPUT TYPE="SUBMIT">
</FORM>
</BODY>
</HTML>

```

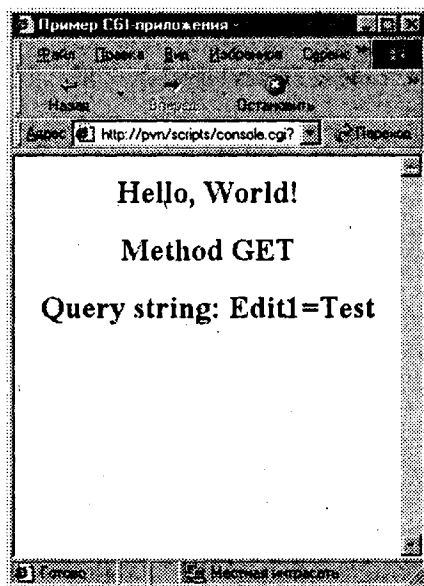
Напомним, что данный документ должен называться default.htm и располагаться в каталоге Wwwroot, а откомпилированный файл CGI-сценария — в каталоге Scripts.

Если теперь запустить веб-браузер и набрать в адресной строке имя вашего компьютера (localhost или 127.0.0.1), то в окне браузера отобразится документ, показан-

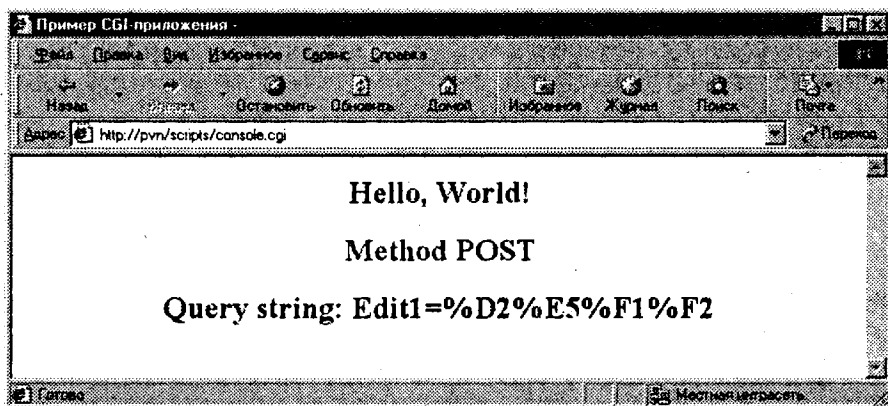
ный на рис. 20.2, а. При щелчке на кнопке SUBMIT, относящейся к форме, использующей метод GET, в окне браузера отобразится документ, приведенный на рис. 20.2, б, а при щелчке на кнопке, относящейся к форме, использующей метод POST — документ, приведенный на рис. 20.2, в.



а



б



в

Рис. 20.2. Пример получения данных, введенных пользователем в CGI-сценарии

Обратите внимание, что во втором случае (при использовании метода POST), в поле ввода был введен кириллический текст. В полученной строке параметров все символы кириллицы закодированы трехсимвольными последовательностями (см. рис. 20.2, в).

Рассмотренный пример позволяет лишь получить строку параметров в CGI-сценарии. Как правило, этого недостаточно. Полученную строку необходимо декодировать и затем обработать данные пользователя. Такая задача достаточно проста и может быть решена без использования каких-либо специальных средств.

Разработка веб-приложений специальными средствами Delphi

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 CGI-приложении в зависимости от используемого метода передачи;
- ☐ формирование сценария 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 специализированных средств позволяет в значительной степени упростить разработку веб-приложений, особенно в тех случаях, когда осуществляется разработка приложений для работы с базами данных.

Компонент TWebModule

Компонент TWebModule является основой любых веб-приложений, разрабатываемых в Delphi, будь то CGI-сценарий или ISAPI-расширение. С помощью этого компонента приложение выполняет интерпретацию HTTP-запросов.

ПРИМЕЧАНИЕ

Компонент TWebModule доступен в Delphi версии 7.

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 собственными свойствами.

- ☐ PathInfo: String — указывает, при обработке какой строки 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 значения: mtGet, mtPost, mtHead, mtPut и 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 обрабатывать 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 рассматриваются далее.

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, с помощью объекта `Request` можно легко получить все данные, введенные пользователем на форме, а также определить ряд параметров клиента. При этом 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 компонента TWebModule

Для 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 четырех событий: `AfterDispatch`, `BeforeDispatch`, `OnCreate` и `OnDestroy`.

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 будет установлено соответствие HTTP-запроса 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 экземпляра `TWebModule`. Обычно используется для освобождения объектов, созданных динамически. При работе с базами данных в обработчике этого события можно, например, разрывать соединение с базой данных.

Пример создания CGI-приложения

Рассмотрим пример создания простого CGI-приложения с использованием средств, предоставляемых компонентом `TWebModule`. Наше приложение будет выполнять

процедуру идентификации пользователя, то есть анализировать введенные им имя и пароль.

Разработку веб-приложения, создаваемого на основе компонента TWebModule, можно разделить на три этапа.

1. Выбор типа 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.

Выбор типа создаваемого приложения

Для создания нового CGI-приложения, использующего компонент TWebModule, необходимо выполнить описанную ниже процедуру.

1. Выберите в главном меню Delphi IDE команду File ► New и на вкладке New открывшегося окна диалога New Items дважды щелкните на значке Web Server Application.
2. В открывшемся окне диалога New Web Server Application с помощью переключателя выберите необходимый тип приложения (рис. 20.3). Так как мы создаем CGI-приложение, установите переключатель CGI Stand-alone executable. Щелкните на кнопке OK. В результате будет создано новое CGI-приложение, содержащее компонент TWebModule.

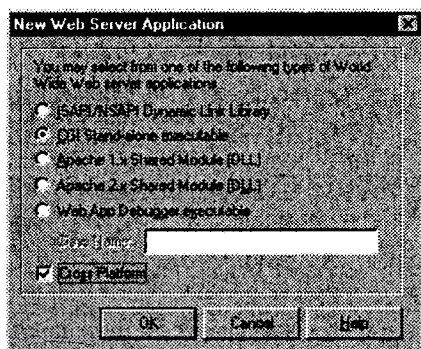


Рис. 20.3. Окно диалога New Web Server Application

Компонент TWebModule объединяет в себе возможности модуля данных (TDataModule) и диспетчера HTTP-запросов. На форму TWebModule можно помещать компоненты доступа к данным и специальные компоненты для формирования ответа сервера на HTTP-запрос клиента.

Задание действий, обрабатывающих запросы клиента

При использовании компонента TWebModule обязательно надо задать хотя бы одно действие (action), которое обеспечит обработку запроса клиента.

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 клавишу Ins).
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 свойств 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.

В запросе 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 формируется ответ.

5. 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, каким методом выполнялась передача строки параметров — GET или POST. Это необходимо, поскольку разными методами 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. Таким образом, введите следующий код обработчика запроса:

```
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;

```

Компиляция и тестирование созданного приложения

Для компиляции проекта выберите в меню IDE Delphi команду Project ► Compile (или нажмите клавиши Ctrl + F9). Если в тексте приложения не содержится ошибок, то проект будет откомпилирован и в результате компиляции создан исполняемый файл, имеющий расширение exe. Измените расширение на cgi и перепишите файл в каталог Scripts.

Для проверки разработанного сценария необходимо подготовить HTML-документ, из которого будет производиться вызов сценария. Документ должен содержать форму с двумя полями 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, вторая — POST. Исходный HTML-код такого документа мог бы выглядеть 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 HTML-документов.

Компоненты для формирования ответа в формате HTML

Компоненты для формирования HTML-документов располагаются на странице Internet палитры компонентов IDE Delphi 7. 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 позднее, а пока познакомимся с единственным компонентом, относящимся ко второй группе.

Компонент `TPageProducer`

Компонент `TPageProducer` — это простейший компонент для генерации HTML-документа на основе некоторого заданного шаблона. Шаблон задается с помощью 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 задать только одно из них.

Компонент `TPageProducer` обрабатывает документ с помощью 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, помимо обычных HTML-тегов, могут включаться специальные теги, имеющие следующий формат:

```
<#name param1=value1 param2=value2 ...>
```

Когда при обработке шаблона `TPageProducer` встречается с такими тегами, возникает 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` определяет 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 областями срабатывания);
- ☐ `tgObject` — тег `OBJECT` (объект `ActiveX`);
- ☐ `tgEmbed` — тег `EMBED` (встраиваемая библиотека);
- ☐ `tgCustom` — тег, определяемый пользователем (имя этого тега не совпадает ни с одним из предопределенных).

Имя тега содержится в аргументе `TagString`. Параметры тега передаются через аргумент `TagParams` в виде строк вида `имя=значение`. Через аргумент `ReplaceText` возвращается строка, на которую должен быть заменен обрабатываемый тег.

Модифицируем предыдущий пример, используя компонент `TPageProducer`. В ответе дополнительно выведем информацию об IP-адресе клиента, пославшем запрос, а также отобразим имя, которое ввел пользователь. Для этого потребуются внести в проект некоторые изменения.

1. Поместите на форму компонента `TWebModule` 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 достоинства 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 Web, часто удобно хранить в базах данных. Кроме того, как уже отмечалось, 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 7 имеется ряд компонентов, специально предназначенных для генерации HTML-документов на основе информации, извлекаемой из базы данных. Таких компонентов три: `TDatasetPageProducer`, `TDataSetTableProducer` и `TQueryTableProducer`.

Рассмотрим каждый из них.

Компонент `TDatasetPageProducer`

Компонент `TDatasetPageProducer` 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 представляется в табличной форме. Это своего рода генератор табличных отчетов для публикации в Web. Перечислим основные свойства компонента TDataSetTableProducer.

- ☐ property DataSet: TDataSet — набор данных, на основе которого будет формироваться выводимая таблица.
- ☐ property Dispatcher: TCustomWebDispatcher — 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 таблицы.
- ☐ property Columns: THTMLTableColumns — поля, включаемые в формируемую таблицу, а также атрибуты отображения полей в таблице. Для установки этого свойства используется специальный редактор, открывающийся при щелчке на кнопке с многоточием в поле ввода свойства Columns окна инспектора объектов или при двойном щелчке на значке компонента TDataSetTableProducer, размещенного на форме.
- ☐ property MaxRows: Integer — максимальное количество строк (записей), выводимых в таблице.
- ☐ property TableAttributes: THTMLTableAttributes — атрибуты отображения таблицы. Класс THTMLTableAttributes имеет следующие свойства:
 - property Align: THTMLAlign — способ выравнивания таблицы относительно окна браузера (доступные 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 (доступные значения: 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:
 - **property VAlign: THTMLVAlign** — вертикальное выравнивание текста в ячейке таблицы (доступные значения: **haVDefault** — выравнивание определяется браузером, **haTop** — выравнивание по верхней границе ячейки, **haMiddle** — выравнивание по центру ячейки, **haBottom** — выравнивание по нижней границе ячейки);
 - **property Custom: String** — дополнительные атрибуты вывода строки.

Рассмотрим пример вывода информации из таблицы базы данных в виде HTML-таблицы. Выведем данные из таблицы **Физические лица** базы данных **sales**.

1. Создайте новое веб-приложение.
2. Поместите на форму **TWebModule** два компонента: **TTable** и **TDataSetTableProducer**. По умолчанию им будут присвоены имена **Table1** и **DataSetTableProducer1**.
3. Выполните подключение компонента **Table1** к таблице **Физические лица** базы данных **sales**.
4. Установите свойство **DataSet** компонента **DataSetTableProducer1** равным **Table1**.
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. 20.4). При указании полей таблицы базы данных, включаемых в HTML-таблицу, удобно вначале подключить все поля, а затем удалить лишние.

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. 20.4. Измените значение свойства **Border**, сделав его равным `1` (в этом случае при выводе будут отображаться линии таблицы).

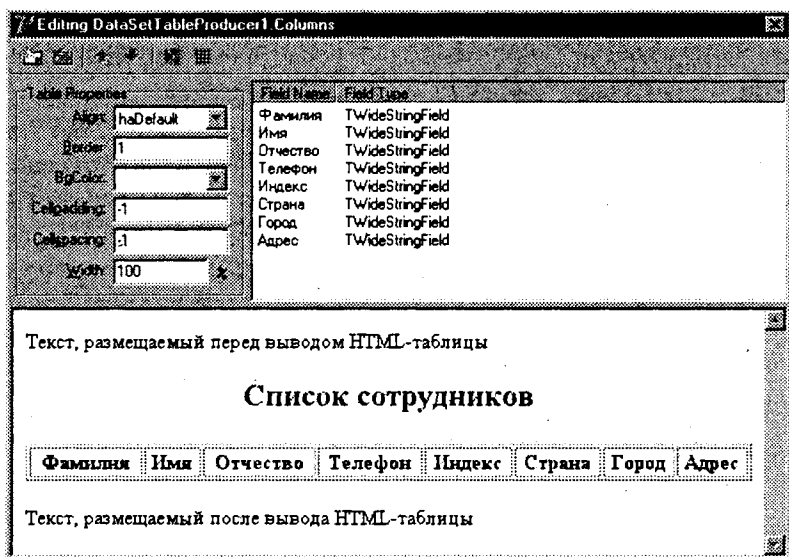


Рис. 20.4. Редактор столбцов компонента TDataSetTableProducer

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 Table1, во втором — закрыть его:

```
procedure TWebModule1.WebModuleCreate(Sender: TObject);
begin
  Table1.Open;
end;

procedure TWebModule1.WebModuleDestroy(Sender: TObject);
begin
  Table1.Close;
end;
```

Осталось только откомпилировать веб-приложение, выводящее информацию из таблицы базы данных в виде HTML-таблицы, изменить расширение полученного файла на cgi и перенести его в каталог Scripts.

Для тестирования полученного сценария следует создать простую HTML-форму. 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. 20.5.

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.

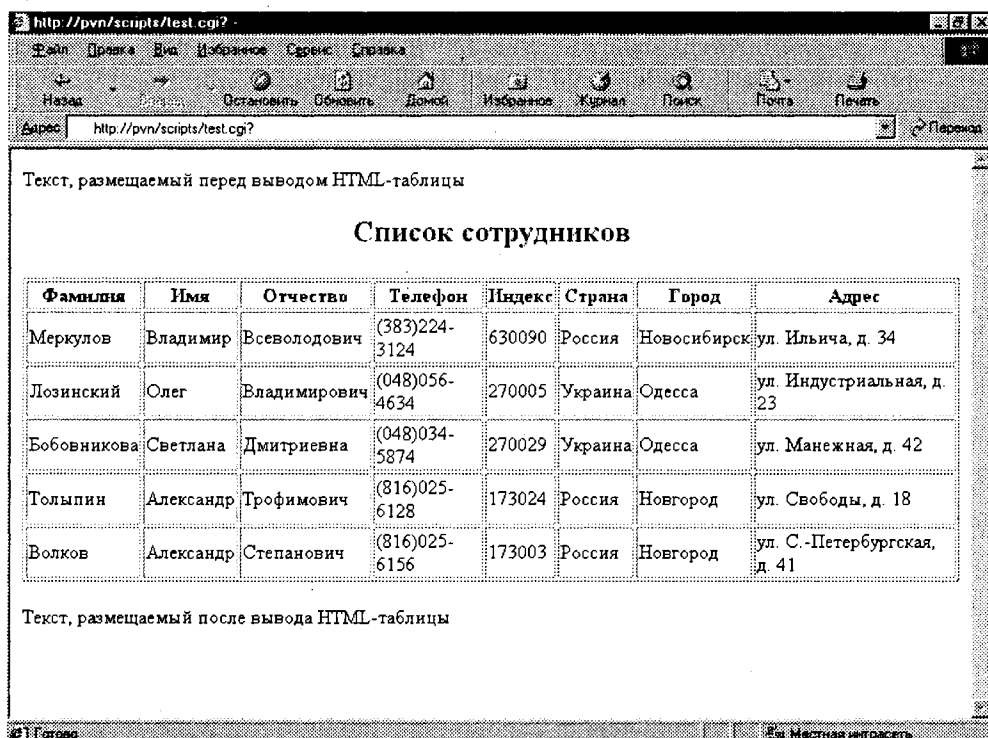


Рис. 20.5. Пример вывода информации из таблицы базы данных в окне браузера


```

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 будут отображаться на светло-сером фоне, четные — на белом (рис. 20.6):

```

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;

```



Рис. 20.6. Пример изменения внешнего вида таблицы путем обработки события **OnFormatCell**

Компонент TQueryTableProducer

Компонент TQueryTableProducer 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 с полученной с его помощью строкой параметров. С помощью данного компонента можно предоставить пользователю возможность задавать критерии выборки данных, на основе которых будет формироваться HTML-таблица.

На основе предыдущего CGI-приложения разработаем сценарий, выводящий список только тех сотрудников, которые проживают в городе, указанном пользователем.

1. Замените в предыдущем приложении компонент TTable компонентом TQuery, а компонент TDataSetTableProducer — компонентом TQueryTableProducer. Настройка свойств компонента TQueryTableProducer выполняется точно так же, как и свойств компонента TDataSetTableProducer, за исключением того, что набор данных подключается через свойство Query.

ПРИМЕЧАНИЕ

В свойстве Query компонента TQueryTableProducer можно ссылаться только на компонент TQuery. С наборами данных TADOQuery и TIBQuery компонент TQueryTableProducer не взаимодействует.

2. Поскольку мы изменили компонент, генерирующий HTML-код, то следует внести изменения в обработчик события OnAction действия, заданного в TWebModule. Кроме того, необходимо внести изменения в обработчики событий OnCreate и OnDestroy компонента TWebModule (так как мы используем другой компонент набора данных):

```
procedure TWebModule1.WebModule1WebActionItem1Action(
  Sender: TObject; Request: TWebRequest;
  Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := '<HEAD><TITLE>' +
    'Пример использования TQueryTableProducer' +
    '</TITLE></HEAD>' + QueryTableProducer1.Content;
end;
```

```
procedure TWebModule1.WebModuleCreate(Sender: TObject);
begin
  Table1.Open;
  Query1.Open;
end;
```

```
procedure TWebModule1.WebModuleDestroy(Sender: TObject);
begin
  Table1.Close;
  Query1.Close;
end;
```

В свойстве SQL задайте следующий запрос:

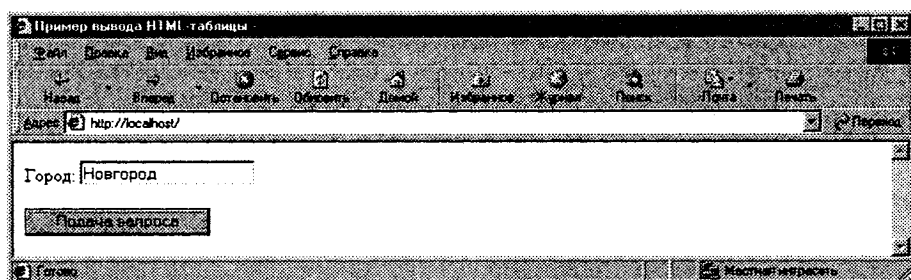
```
SELECT *
FROM [Физические лица]
WHERE (Город=:City)
```

3. Если теперь на HTML-форме, из которой производится вызов CGI-приложения, разместить текстовое поле с именем City, то данные, введенные в это поле,

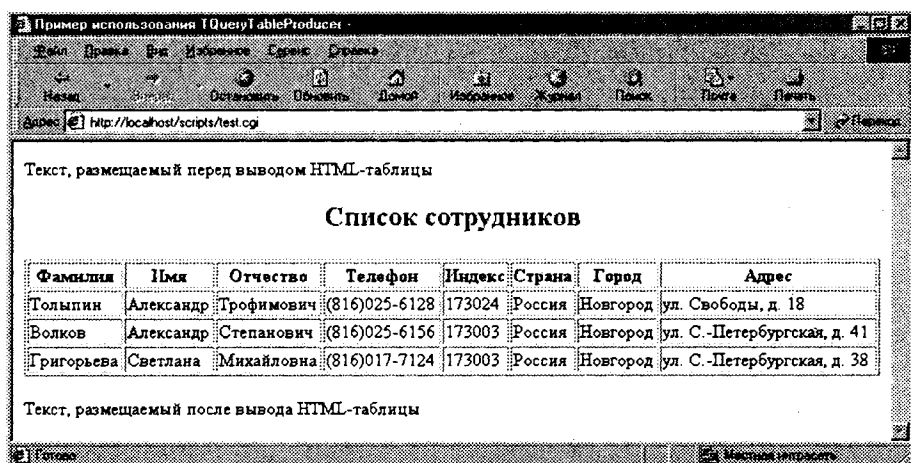
будут передаваться в SQL-запрос. Соответственно, в таблицу будут включаться только те записи, в которых значение, содержащееся в поле Город, окажется равным значению, заданному пользователем в текстовом поле. Поэтому следует изменить код HTML-формы, из которой выполняется вызов сценария, добавив элемент управления TEXT с именем CITY:

```
<HTML>
<HEAD>
  <TITLE> Пример вывода HTML-таблицы </TITLE>
</HEAD>
<BODY>
  <FORM METHOD="POST" ACTION="/scripts/test.cgi">
    Город: &nbsp;   <input TYPE="TEXT" NAME="CITY"><BR><BR>
    <input TYPE="SUBMIT">
  </FORM>
</BODY>
</HTML>
```

Вид формы, из которой выполняется запуск сценария, приведен на рис. 20.7, а, а результат выполнения сценария — на рис. 20.7, б.



а



б

Рис. 20.7. HTML-форма, использующая компонент TQueryTableProducer, и результат выполнения приложения

Компоненты Indy

Компоненты Indy (Internet Direct) появились в Delphi, начиная с версии 7.

К сожалению не все компоненты Indy, в частности, для создания серверных частей, доступны в версии Delphi 8 комплектации Architect, поэтому при создании серверной части приложения мы воспользуемся Delphi 7, а при создании клиентской части — Delphi 8.

Пример создания простого веб-сервера

В качестве основы при создании сервера возьмем компонент `TIdHTTPServer`. С его помощью мы создадим не просто программный модуль, исполняемый на сервере и выполняющий конкретные функции вроде формирования HTML-страниц, а полноценный HTTP-сервер, обрабатывающий HTTP-запросы любого клиентского приложения, поддерживающего протокол HTTP.

1. Создайте новое приложение. Поместите на форму компонент `TIdHTTPServer`.
2. Свойству `Bindings` присвойте IP-адрес своего компьютера (127.0.0.1) и порт обработки HTTP-запросов (как правило, 80).
3. Опишите функциональность сервера в обработчике события `OnCommandGet`. Это событие генерируется, когда к серверу поступает запрос от клиентского приложения. За параметры соединения несет ответственность `AThread`, за параметры запроса — `ARequestInfo`, за параметры ответа — `AResponseInfo`. Код должен выглядеть следующим образом:

```
procedure TForm1.IdHTTPServer1CommandGet (AThread:
  TIdPeerThread; ARequestInfo: TIdHTTPRequestInfo;
  AResponseInfo: TIdHTTPResponseInfo);
begin
  Form1.IdHTTPServer1.ServeFile(AThread.AResponseInfo,
    'C:\MyServer\root'+ARequestInfo.Document);
end;
```

Отправка HTML-запроса реализуется методом `ServeFile` компонента `TIdHTTPServer`.

4. Остается перевести объект в активное состояние в момент создания формы путем программирования события `OnCreate` формы:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Form1.IdHTTPServer1.Active:=True;
end;
```

Для тестирования приложения создадим в Delphi HTML-страницу, на которую поместим надпись «Главная страница моего сервера» и небольшую картинку. Ниже приведен текст тестовой страницы:

```
<!DOCTYPE HTML PUBLIC "-//W3C//
DTD HTML 4.0 Transitional//EN">
<html>
  <head> <title></title>
  <meta name="GENERATOR"
  content="Borland HTML Designer 7.1"></head>
  <body ms_positioning="gridlayout">
    <p> </p>
```

```

<div style="DISPLAY: inline; Z-INDEX: 102; LEFT: 62px;
WIDTH: 211px; POSITION: absolute; TOP: 22px; HEIGHT:
42px" ms_positioning="FlowLayout">
Главная страница тестового сервера </div>

</body>
</html>

```

Откомпилируем наш проект и запустим его (рис. 20.8).

В качестве клиентской части используем браузер Internet Explorer, в качестве адреса укажем 127.0.0.1. Результат обращения к серверу представлен на рис. 20.9.

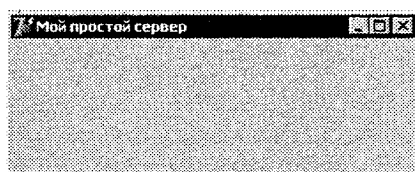


Рис. 20.8. Вид окна сервера

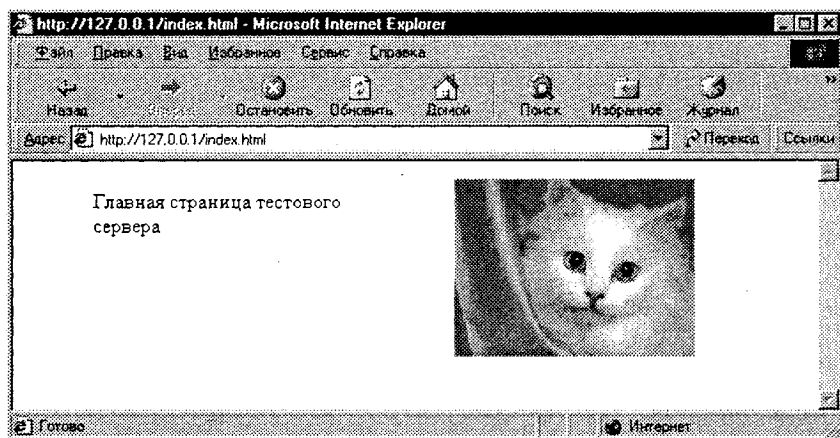


Рис. 20.9. Результат работы сервера

Пример клиентского приложения

С помощью компонентов Indy в Delphi 8 можно быстро создавать клиентские приложения для Интернета.

Компонент `TIdTCPClient` предназначен для создания приложений на основе протокола TCP, компоненты `TIdPOP3` и `TIdSMTP` реализуют функции приема и отправки электронной почты.

Воспользуемся готовым примером использования компонента `IdHTTP`, входящего в стандартную поставку Delphi 8 (см. каталог `Borland\Bds\2.0\Demos\IndyNet\HTTP\VCLForms`):

```

unit IdHTTPDemoVCLForm;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ExtCtrls, System.ComponentModel;
type
  TForm1 = class(TForm)
    Panel1: TPanel;
    Label1: TLabel;
    Label2: TLabel;
    Edit1: TEdit;
    Edit2: TEdit;
    Button1: TButton;
    Memo1: TMemo;
    procedure Button1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var Form1: TForm1;
implementation
uses IdHTTP;
{$R *.DFM}
procedure TForm1.Button1Click(Sender: TObject);
var
  LClient : TIdHTTP;
begin
  LClient := TIdHTTP.create(nil);
  try
    if Edit2.Text <> '' then
      begin
        LClient.ProxyParams.ProxyServer :=
          copy(Edit2.Text, 1, pos(':', Edit2.Text)-1);
        if pos(':', Edit2.Text) <> 0 then
          LClient.ProxyParams.ProxyPort :=
            StrToInt(copy(Edit2.Text,
              pos(':', Edit2.Text) + 1, $FF))
            else
              LClient.ProxyParams.ProxyPort := 80;
        end;
        Memo1.Lines.Text := LClient.Get(Edit1.Text);
      finally
        LClient.Free;
      end;
    end;
  end;
end.

```

В данном примере компонент TIdHTTP подключен вручную путем добавления в секцию реализации модуля IdHTTP и объявления переменной LClient типа TIdHTTP.

Откомпилируем и запустим проект. Для успешной работы примера сервер, созданный ранее, должен быть запущен. В адресной строке введите адрес главной страницы нашего сервера <http://127.0.0.1/index.html>. Результат представлен на рис. 20.10.

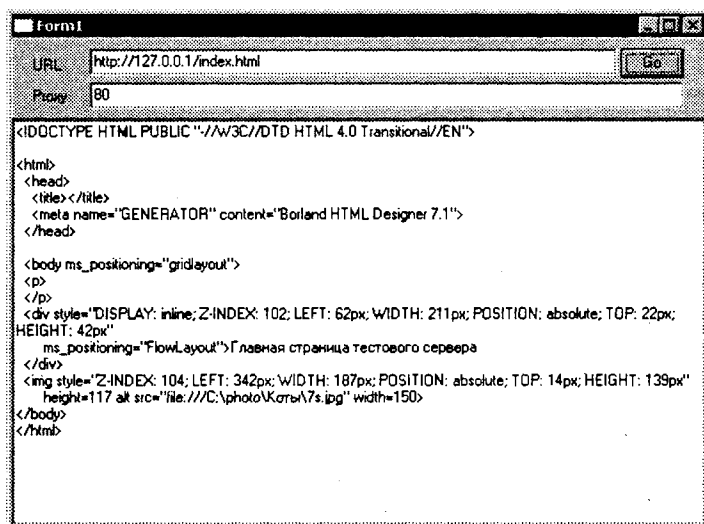


Рис. 20.10. Результат выполнения клиентского приложения

В качестве URL-адреса можно использовать любой Интернет-адрес, например www.microsoft.com.

Заключение

Современные программные системы становятся все сложнее, претендуя на решение глобальных задач, например, таких, как создание единой системы управления предприятием. При создании таких систем важно хорошо представлять современные подходы к разработке информационных систем и основные сложности, возникающие при этом.

Объектно-ориентированные визуальные средства разработки успешно используются для создания сложных систем в самых разных областях. Потребность в сложных программных системах все время растет. По мере того как растет производительность и падает цена вычислительной техники, выполняется автоматизация все более сложных процессов. Основная ценность объектно-ориентированного визуального проектирования состоит в том, что при создании сложных информационных систем оно позволяет свести к минимуму трудоемкую рутинную работу и сосредоточиться на решении действительно творческих задач.

В этой книге мы постарались затронуть различные аспекты проектирования информационных систем, начиная от этапов их жизненного цикла и заканчивая реализацией отдельных операций на базе многообразия существующих технологий.

Рекомендуемая литература

С одной стороны, средства проектирования и стандарты разработки информационных систем постоянно изменяются, с другой — разработка информационных систем выполняется для конкретной организации, под конкретный проект, в этой связи неизбежно возникнут вопросы, не нашедшие отражения в настоящем издании.

Если возникают вопросы по применению средств разработки приложений, во-первых, рекомендуем обратиться к справочным системам используемых средств проектирования. Несмотря на критическое отношение к ним со стороны многих пользователей, ответы на большинство вопросов можно найти именно там.

Во-вторых, советуем ознакомиться с документацией фирм-производителей, доступной через Интернет. Хотя большая ее часть представлена на английском языке, встречаются и русифицированные варианты.

В-третьих, очень много полезной информации можно найти на всевозможных форумах в Интернете. Достоинством их является то, что большинство форумов ведутся на русском языке. Существенным их недостатком является отсутствие системы в представлении информации и, как следствие, трудоемкость поисков.

Отдельные статьи в журналах и в Интернете также содержат весьма полезные сведения о проектировании.

Но какой бы подробной ни была документация по средствам разработки приложений, в ней вы не найдете сведений о фундаментальных процессах и методиках разработки информационных систем. Систематизированные, фундаментальные знания могут дать только книги. Мы можем посоветовать следующие издания по темам, затронутым в этой книге.

☐ Общие вопросы разработки информационных систем.

- *Константин Л., Локвуд Л.* Разработка программного обеспечения. СПб.: Питер, 2004.
- *Буч Г.* Объектно-ориентированный анализ и проектирование с примерами приложений на C++. М.: Бином, 2001.

☐ Основы платформы Microsoft .NET.

- *Троэлсен Э.* C# и платформа .NET». СПб.: Питер, 2004.
- *Тай Т., Лэм Х. К.* Платформа .NET. Основы. Символ, 2003.

- ❑ Программирование в Delphi.
 - *Флеков М.* Библия Delphi. СПб.: БХВ-Петербург, 2004.
 - *Архангельский А.* Программирование в Delphi 7. М.: Библио-Пресс, 2003.
 - *Бобровский С.* Delphi 7. Учебный курс. Информ-Пресс; Питер, 2003.
- ❑ По технологии COM доступна книга: *Елманова Н., Трепалин С., Тенцер А.* Delphi и технология COM. СПб.: Питер, 2003.
- ❑ В качестве фундаментального пособия по проектированию веб-приложений рекомендуем книгу: *Пауэлл Т. А.* Web-дизайн. СПб.: БХВ-Петербург, 2004.
- ❑ Вопросы программирования для Интернета.
 - *Макдональд М.* ASP.NET. СПб.: БХВ-Петербург, 2003.
 - *Феррара А., Макдональд М.* Программирование Web-серверов для .NET. Библиотека программиста. СПб.: БХВ; Питер, 2003.

Алфавитный указатель

А

abstract, 98, 235
access, 98
active, 111
active object, 111
ActiveForm, 566, 568
ActiveX, 406, 529, 563
AddRef, 534
ADO, 283, 295
alias, 263
ALL, 355
ALTER TABL, 153
ALTER TABLE, 153, 154, 155, 157, 158
 ADD, 153
 DROP, 153
 MODIFY, 153
AND, 357
ANSI SQL 92, 342
ANSI SQL-92, 148, 150, 151, 159
AnsiChar, 206
AnsiString, 209
ANY, 356
array, 209
as, 263
ASC, 162, 358
ASP, 598
ASP-страницы., 596
association, 111
automated, 237
AVG, 362

В

BDE, 247, 283, 295, 489
BETWEEN...AND, 352
BIT, 151
BIT VARYING, 151
BLOB, 128
Boolean, 206

Borland Delphi, 65
break, 220
Byte, 205
ByteBool, 206

С

Cardinal, 205
CASCADE, 165, 175
CASCADES, 160
case, 212
CASE-средства, 48, 60, 63, 67, 68, 176,
 177, 178, 180, 181, 191, 192, 196
CASE-технология, 179, 180, 181
CDM, 78
 особенности, 81
 структура, 79
CGI-приложение, 606, 607, 615, 617, 630
 запуск, 606
CGI-сценарии, 596
CGI-сценарий, 606, 610, 615
 передача данных серверу, 609
Char, 206
CHARACTER, 149
CHECK, 154, 160
class, 226
CLSID, 531
CoClass, 536
COM, 237, 486, 487, 529, 553
COM-интерфейс, 535
COM-клиент-объекта, 549
COM-объект, 531, 542, 546, 547, 549, 550, 556
COM-сервер, 530
Comp, 208
complete, 102
composite state, 104
concurrency, 97
concurrent, 97
Const, 204
CONSTRAINT, 160

constructor, 236
continue, 220
COUNT, 362
CRC-карточки, 89
CREATE INDEX, 161, 162, 163
CREATE PROCEDURE, 167
CREATE TABLE, 152, 155, 170
CREATE TRIGGER, 168
CREATE UNIQUE INDEX, 162
CREATE VIEW, 165, 375
CreateOleObject, 490, 562
Currency, 208

D

DATE, 151
DEFAULT, 161
default, 232
DELETE, 169
DELETE FROM, 172
DELETE OF, 160
Delphi, 246, 256
Delphi IDE, 248
 главное меню, 248
 меню Component, 256
 меню Edit, 250
 меню File, 248
 меню Project, 255
 меню Run, 256
 меню Search, 251
 меню Tools, 257
 меню View, 252
палитра компонентов, 258
 страница BDE, 259
 страница Data Access, 259
 страница Data Controls, 259
 страница Dialogs, 259
 страница Indy, 259
 страница InterBase, 259
 страница InternetExpress, 259
 страница Standard, 259
 страница System, 259
 страница Win 3.1, 259
 страница Win32, 259
панель инструментов, 248, 257, 258
derive, 98
DESC, 162, 358
destructor, 236
disjoint, 102
dispid, 554, 560
Dispose, 214
DISTINCT, 350, 362, 363, 364
DMT, 234
DNS-сервер, 579
DOUBLE, 151
Double, 207, 208
DROP INDEX, 163

DROP PROCEDURE, 167
DROP TABLE, 154
DROP TRIGGER, 169
DROP VIEW, 165, 166, 377
dynamic, 234

E

EAbstractError, 235
ER-диаграммы, 178
 атрибут, 179
 связь, 179
 сущность, 178
ER-модель, 178
Exception, 238
EXECUTE, 167
EXISTS, 354, 355
Extended, 207, 208

F

finalization, 225
Finalize, 211
FLOAT, 151
FloatToStr, 268
FloatToStrF, 268
for, 219
FOREIGN KEY, 157
FreeAndNil, 214
FreeMem, 214
FULL OUTER JOIN, 370

G

GetEnvironmentVariable, 610
GetLongHint, 446
GetMem, 214
GetShortHint, 446
global, 112
goto, 217
GRANT, 174
GROUP BY, 363, 364, 365, 374
guarded, 97
GUID, 531

H

HAVING, 365, 366, 374
HTML Help, 465
 основные элементы, 466
HTML Help Workshop, 469, 470, 473, 477
HTML-документ, 581, 586, 587, 608, 613
 абзацы, 588
 выделение фрагментов текста, 589
 заголовки, 588
 раздел заголовка, 587
 списки, 589
 тело документа, 588
HtmlHelp, 448, 481

HTTP-запрос, 616, 617
HTTP-протокол, 596
HTTP-сервер, 582
 запрос клиента, 583
 коды ответов, 584
 ответ сервера, 584
 сеанс взаимодействия, 582

I

IClassFactory, 535
 методы, 535
 CoCreateInstance, 535
 LockServer, 535
IDispatch, 486, 516, 554, 557
Idispatch, 516
IDL, 539
IID, 531
implémentation, 225
import, 98
IN, 352
incomplete, 102
inherited, 236
initialization, 225
INSERT, 169
INSERT INTO, 169, 170, 171, 174
Int64, 205
INTEGER, 150
Integer, 205
interface, 102, 225, 536
Interface Description Language, 539
INTERVAL, 151
IntToStr, 268
Invoke, 554, 560
IP-адрес, 576, 578, 579, 583
IP-адресом, 577
IP-пакеты, 577
IS NULL, 352
ISAPI-расширения, 596
IUnknow, 486
IUnknown, 531, 532, 534, 536, 554
 методы, 531
 AddRef, 532
 QueryInterface, 531
 Release, 532

K

Kylix, 489

L

Label, 202
LEFT OUTER JOIN, 370
library, 113
LIKE, 353
local, 111

LongBool, 206
LongInt, 205
LongWord, 205

M

MAX, 362
Microsoft Office, 486
MIME, 582, 583, 585
MIN, 362

N

NDR-поток, 390
New, 214
nil, 213
NOT, 358
NOT NULL, 154
Null, 215

O

Object Pascal, 201, 246
ODBC, 489
OLE Automation, 486, 529, 553
OleVariant, 562
OMG, 86
OR, 357
ORDER BY, 358, 359, 364, 371, 374
OUTER JOIN, 370
overlapping, 102
overload, 236
override, 234

P

parameter, 111
Personal Web Server, 606
pointer, 213
Power Designer, 184
 модификация модели, 197
 определение атрибутов сущности, 187
 преобразование концептуальной модели
 в физическую, 194
 проверка модели, 191
 создание доменов, 186
 создание нового проекта, 184
 создание связи между сущностями, 189
 создание структуры базы данных, 195
 создание сущностей, 186
PRIMARY KEY, 155
private, 96, 237
procedure, 221
Program, 202
property, 231
protected, 96, 237
public, 96, 237
published, 237

Q

query, 97
QueryInterface, 532, 533, 551

R

RAD, 62
 недостатки, 69
 фазы жизненного цикла, 66
 анализ и планирование
 требований, 66
 внедрение, 68
 построение, 68
 проектирование, 67
raise, 239
Rational Rose, 91
Rave Reports, 382
read, 231, 232
Real, 207, 208
Real48, 207, 208
record, 211
REFERENCE, 174
refine, 98
Release, 534
repeat... until, 219
RESTRICT, 165, 175
RESTRICTED, 160
result, 223, 224
REVOKE, 174, 175
RIGHT OUTER JOIN, 370

S

SELECT, 165, 166, 170, 174, 348, 349
self, 112
sequential, 97
set, 211
SetLength, 210
ShortInt, 205
ShortString, 209
Single, 207, 208
SMALLINT, 150
SmallInt, 205
SQL-запрос, 89, 283
SQL-запросы, 342
SQL-запросы с параметрами, 377
SQL-сервер, 342
String, 209
StrToFloat, 272
StrToInt, 272
StrToIntDef, 272
SUM, 362

T

TabOrder, 325
TabStop, 325

TAction, 400
 события
 OnExecute, 400
TActionList, 399
 свойства, 399
 Images, 399
 Name, 399
 Tag, 399
TADOCCommand, 343
TADODataset, 343
TADOStoredProc, 343
TApplication, 408, 411
 методы, 412
 BringToFront, 412
 HelpCommand, 479
 HelpContext, 479
 HelpJump, 479
 Minimize, 412
 ProcessMessages, 412
 Restore, 412
 Terminate, 412
 свойства, 411
 Active, 412
 CurrentHelpFile, 412
 ExeName, 412
 Handle, 412
 HelpFile, 412
 Hint, 412
 HintColor, 446
 HintHidePause, 446
 HintPause, 446
 Icon, 412
 MainForm, 412
 ShowHint, 446
 события, 412
 OnActionExecute, 412
 OnActivate, 412
 OnDeactivate, 412
 OnHelp, 479
 OnMessage, 412
 OnShortCut, 412
 OnShowHint, 446
TApplicationEvents, 412
TArrayField, 292
TAutoObject, 554
TBlobField, 292, 299
TBooleanField, 292
TButton, 265, 266
 свойства
 Anchors, 265
 Cancel, 265
 Caption, 265
 Default, 265
 Enable, 265
 Font, 265
 Left, 266
 ModalResult, 265

- TButton (*продолжение*)
 - TabOrder, 265
 - TabStop, 265
 - Top, 266
 - Visible, 265
 - события
 - OnClick, 266
- TCheckBox, 269, 403
 - свойства
 - AllowGrayed, 269
 - Caption, 269
 - Checked, 269
 - State, 269
- TColorDialog, 282
 - свойства, 282
 - Color, 282
 - CustomColors, 282
 - Options, 282
- TColumn, 297
 - свойства, 297
 - Alignment, 298
 - Color, 298
 - DropDownRows, 298
 - FieldName, 298
 - Font, 298
 - PickList, 298
 - PopupMenu, 298
 - Showing, 298
 - Title, 298
 - Visible, 298
 - Width, 298
- TComboBox, 277, 401
 - свойства, 277
 - DropDownCount, 277
 - DroppedDown, 277
 - ItemIndex, 277
 - Items, 277
 - MaxLength, 277
 - SelfText, 277
 - Sorted, 277
 - Style, 277
- TComObject, 535
- TComponent, 265, 302
- TControl, 303
- TControlBar, 404, 405
- TCoolBand, 404
- TCoolBands, 404
- TCoolBar, 404, 405
- TCP-пакеты, 577
- TCP-соединение, 582
- TCustomControl, 302
- TDataLink, 311
 - методы, 311
 - UpdateRecord, 312
 - свойства, 311
 - Active, 312
 - ActiveRecord, 312
 - DataSet, 312
- TDataSet, 284
- TDataSetField, 292
- TDataSetPageProducer, 624
 - свойства, 624
 - DataSet, 624
- TDataSetState, 288
- TDataSetTableProducer, 625, 630
 - свойства, 625
 - Caption, 625
 - CaptionAlignment, 625
 - Columns, 625
 - DataSet, 625
 - Dispatcher, 625
 - Footer, 625
 - Header, 625
 - MaxRows, 625
 - RowAttributes, 626
 - TableAttributes, 625
- TDataSetTableProducer, 624
 - события, 628
 - OnCreateContent, 628
 - OnFormatCell, 629
- TDataSource, 295
 - методы
 - Edit, 295
 - IsLinkedTo, 295
 - свойства
 - AutoEdit, 295
 - DataSet, 295
 - Enabled, 295
 - State, 295
 - события
 - OnDataChange, 295
 - OnStateChange, 296
 - OnUpdateData, 296
- TDataSourceLink, 311
- TDBCheckBox, 299
- TDBComboBox, 299
- TDBEdit, 298
- TDBGGrid, 297, 332
 - свойства, 297
 - Columns, 297
 - DataSource, 297
 - DefaultDrawing, 297
 - FieldCount, 297
 - Fields, 297
 - Options, 297
 - ReadOnly, 297
 - SelectedField, 297
 - SelectedIndex, 297
- TDBImage, 300
- TDBListBox, 299
- TDBMemo, 299
- TDBNavigator, 300, 326, 333
 - события, 301
 - BeforeAction, 301
 - OnClick, 301

- TDBRadioGroup, 299
- TDBText, 298
- TeamSource
 - закладки, 442
 - работа с локальной копией проекта, 435
 - работа с проектом, 435
 - работа с хранилищем версий, 437, 441
 - сборка проекта, 443
- TEdit, 271, 401
 - свойства, 271
 - AutoSelect, 271
 - CharCase, 271
 - ReadOnly, 271
 - SelText, 271
 - Text, 271
 - события
 - OnChange, 271
- TExcelWorksheet, 516
- TField, 289
 - методы, 291
 - Assign, 291
 - AssignValue, 291
 - Clear, 291
 - IsBlob, 291
 - IsValidChar, 291
 - SetFieldType, 291
 - свойства, 289
 - Alignment, 290
 - AsBoolean, 290
 - AsCurrency, 290
 - AsDateTime, 290
 - AsFloat, 290
 - AsInteger, 290
 - AsString, 290
 - AsVariant, 290
 - Calculated, 290
 - CanModify, 290
 - ConstraintErrorMessage, 290
 - CurValue, 290
 - CustomConstraint, 290
 - DataSet, 290
 - DataSetSize, 290
 - DataType, 290
 - DefaultExpression, 290
 - DisplayLabel, 290
 - DisplayText, 290
 - FieldKind, 290
 - IsIndexField, 290
 - IsNull, 290
 - KeyFields, 290
 - Lookup, 290
 - NewValue, 290
 - OldValue, 291
 - ReadOnly, 291
 - ValidChars, 291
 - Value, 291
 - Visible, 291
- TFieldDataLink, 311, 312
 - методы, 312
 - Edit, 312
 - Modified, 312
 - Reset, 312
 - свойства, 312
 - CanModify, 312
 - Field, 312
 - FieldName, 312
 - события
 - OnActiveChange, 312
 - OnDataChange, 312
 - OnEditingChange, 312
 - OnUpdateData, 312
- TFloatField, 292
- TFontDialog, 281
 - свойства, 281
 - Device, 281
 - Font, 281
 - MaxFontSize, 281
 - MinFontSize, 281
 - Options, 281
 - события, 281
 - OnApply, 281
 - OnClose, 281
 - OnShow, 281
- TForm, 317, 318, 407
 - методы, 319
 - Close, 319
 - CloseQuery, 319
 - FocusControl, 319
 - GetFormImage, 319
 - Hide, 319
 - Print, 319
 - Release, 319
 - ShowModal, 319
 - свойства, 317, 318
 - Active, 318
 - ActiveControl, 317
 - ActiveMDIChild, 318
 - AutoScroll, 317
 - AutoSize, 317
 - BorderIcons, 317
 - BorderStyle, 317, 318
 - BorderWidth, 317
 - Canvas, 318
 - Caption, 317
 - ClientHeight, 317
 - ClientWidth, 317
 - Color, 317
 - Constraints, 317
 - Ctl3D, 317
 - Cursor, 317
 - FormStyle, 317
 - Height, 318
 - Icon, 318
 - Left, 318

TForm (продолжение)

- MDIChildCount, 319
- MDIChildren, 319
- Menu, 318
- ModalResult, 319
- Name, 318
- PopupMenu, 318
- Tag, 318
- Top, 318
- Visible, 318
- Width, 318
- WindowState, 318

события

- OnActivate, 319
- OnClick, 319
- OnClose, 319
- OnCloseQuery, 319
- OnCreate, 319
- OnDblClick, 319
- OnDeactivate, 319
- OnDestroy, 319
- OnKeyPress, 320
- OnMouseDown, 320
- OnMouseMove, 320
- OnMouseUp, 320
- OnPaint, 319
- OnShow, 319

TFrames, 321

TGraphic, 278

TGraphicControl, 302, 303

TGraphicField, 292

TGridDataLink, 311

TGroupBox, 272, 273

TImage, 277

свойства, 277

- AutoSize, 277
- Center, 277
- Picture, 277
- Stretch, 277

TIME, 151

TIMESTAMP, 151

TIntegerField, 292

TLabel, 267

свойства

- Alignment, 267
- AutoSize, 267
- Layout, 267
- Transparent, 267
- WordWrap, 267

TLargeField, 292

TListBox, 276

свойства, 276

- Columns, 276
- ItemIndex, 276
- Items, 276
- MultiSelect, 276
- SelCount, 276

TListBox (продолжение)

- Selected, 276
- Sorted, 276

TListSourceLink, 311

TMainMenu, 394

методы, 394

- Merge, 394
- Unmerge, 395

свойства, 394

- AutoHotkeys, 394
- AutoMerge, 394
- Images, 394
- OwnerDraw, 394

TMasterDataLink, 311

TMemo, 276, 299

свойства

- Alignment, 276
- CaretPos, 276
- Lines, 276
- SelText, 276

TMemoField, 292, 299

TMenuItem, 394, 395, 400

методы, 396

- Add, 396
- Clear, 396
- Click, 396
- Find, 396
- IndexOf, 396
- Insert, 396
- InsertNewLineAfter, 396
- InsertNewLineBefore, 396
- IsLine, 396
- Remove, 396

свойства, 395

- Action, 395
- Bitmap, 395
- Break, 395
- Caption, 395
- Checked, 395
- Default, 395
- Enabled, 395
- GroupIndex, 395
- ImageIndex, 395
- MenuIndex, 395
- RadioItem, 395
- ShortCut, 395

события, 396

- OnAdvancedDrawItem, 396
- OnClick, 396
- OnDrawItem, 396
- OnMeasureItem, 396

TNavDataLink, 311

TObject, 230

TOpenDialog, 279

TOpenPictureDialog, 279

- TPageControl, 335
 - свойства, 335
 - ActivePage, 335
 - ActivePageIndex, 335
 - HotTrack, 335
 - MultiLine, 335
 - PageCount, 335
 - Pages, 335
 - Style, 335
- TPageProduce
 - методы, 621
 - Content, 621
- TPageProducer, 621, 624
 - события, 622
 - OnHTMLTag, 622
- TpageProducer
 - свойства, 621
 - HTMLDoc, 621
 - HTMLFile, 621
- TPanel, 272
 - свойства
 - BevelInner, 272
 - BevelOuter, 272
- TPicture, 277
 - методы, 278
 - LoadFromClipboardFormat, 278
 - LoadFromFile, 278
 - SaveToClipboardFormat, 278
 - SaveToFile, 278
 - свойства
 - Bitmap, 278
 - Height, 278
 - Icon, 278
 - Metafile, 278
 - Width, 278
- TPopupMenu, 400
- TPowerPointApplication, 525
- TPowerPointPresentation, 525
- TPowerPointSlide, 525
- TPrintDialog, 282
 - свойства, 282
 - Collate, 282
 - Copies, 282
 - FromPage, 282
 - MaxPage, 282
 - MinPage, 282
 - Options, 282
 - PrintRange, 282
 - PrintToFile, 282
 - ToPage, 282
- TPrinterSetupDialog, 282
- TQuery, 283, 284, 343, 630
 - методы, 343
 - ExecSQL, 344
 - ParamByName, 344
 - Prepare, 344
 - UnPrepare, 344
- TQuery (*продолжение*)
 - свойства, 343
 - Constrained, 343
 - DataSource, 343
 - ParamCheck, 343
 - ParamCount, 343
 - Params, 343
 - Prepared, 343
 - RowsAffected, 343
 - SQL, 343
 - Text, 343
- TQueryTableProducer, 624, 630
- TRadioButton, 270, 403
 - свойства
 - Caption, 270
 - Checked, 270
 - события
 - OnClick, 270
- TRvCustomConnection, 384
- TRvNDRWriter, 390
- TRvProect
 - метод ExecuteReport, 390
- TRvProject, 383
- TRvQueryConnection, 384
- TRvRenderPreview, 390
- TRvRenderPrinter, 390
- TRvTableConnection, 384
- try...except, 238
- try...finally, 239
- TSaveDialog, 279
- TSavePictureDialog, 279
- TSmallIntField, 292
- TStatusBar, 447
- TStatusPanel, 448
 - свойства, 448
 - Bevel, 448
 - Style, 448
 - Text, 448
 - Width, 448
- TStoredProc, 343
- TStringField, 292
- TTable, 283, 284, 343
 - методы, 285
 - AddIndex, 285
 - ApplyRange, 285
 - Cancel, 286
 - CancelRange, 286
 - DeleteTable, 286
 - Edit, 286
 - EditKey, 286
 - EditRangeEnd, 286
 - EditRangeStart, 286
 - EmptyTable, 286
 - FieldByName, 286
 - FindKey, 286
 - FindNearest, 286
 - First, 286

TTable (продолжение)

- GotoKey, 286
- GotoNearest, 286
- Insert, 286
- Last, 287
- Locate, 287
- LockTable, 287
- MoveBy, 287
- Next, 287
- Post, 287
- Prior, 287
- RenameTable, 287
- SetKey, 287
- SetRange, 287
- SetRangeEnd, 287
- SetRangeStart, 287
- UnlockTable, 287
- свойства, 284
 - Active, 284
 - BOF, 284
 - DatabaseName, 284
 - DefaultIndex, 284
 - EOF, 284
 - Exclusive, 284
 - Exists, 285
 - FieldCount, 285
 - Fields, 285
 - IndexDefs, 285
 - IndexFieldCount, 285
 - IndexFieldNames, 285
 - IndexFields, 285
 - IndexFiles, 285
 - IndexName, 285
 - KeyExclusive, 285
 - KeyFieldCount, 285
 - MasterFields, 285
 - MasterSource, 285
 - Modified, 285
 - ReadOnly, 285
 - RecordCount, 285
 - TableLevel, 285
 - TableName, 285
 - TableType, 285
- события, 288
 - AfterCancel, 289
 - AfterClose, 289
 - AfterDelete, 289
 - AfterEdit, 289
 - AfterInsert, 289
 - AfterOpen, 289
 - AfterPost, 289
 - AfterScroll, 289
 - BeforeCancel, 289
 - BeforeClose, 289
 - BeforeDelete, 289
 - BeforeEdit, 289
 - BeforeInsert, 289

TTable (продолжение)

- BeforeOpen, 289
- BeforePost, 289
- BeforeScroll, 289
- OnCalcFields, 289
- OnNewRecord, 289
- TTabsheet, 335
- TToolBar, 400, 401, 402, 403, 404
 - свойства, 401
 - ButtonHeight, 401
 - ButtonWidth, 401
 - DisabledImages, 401
 - Flat, 401
 - HotImages, 401
 - Images, 401
 - Indent, 401
 - List, 401
 - ShowCaptions, 401
 - Transparent, 401
 - Wrapable, 401
 - события, 401
 - OnDockDrop, 401
 - OnDockOver, 401
 - OnDragDrop, 401
 - OnDragOver, 401
 - OnEndDock, 401
 - OnEndDrag, 401
 - OnStartDock, 401
 - OnStartDrag, 401
- TToolButton, 401, 403
 - методы, 402
 - CheckMenuDropdown, 402
 - Click, 403
 - свойства, 401
 - Action, 402
 - AllowAllUp, 402
 - AutoSize, 402
 - Caption, 402
 - Down, 402
 - DropDownMenu, 402
 - Grouped, 402
 - Hint, 402
 - ImageIndex, 402
 - Indeterminate, 402
 - Marked, 402
 - MenuItem, 402
 - ShowHint, 402
 - Style, 402
 - события, 403
 - OnClick, 403
- TTypedCOMObject, 554
- TTypedComObject, 536
- TWebModule, 615, 618, 621
 - свойства, 615
 - Action, 615
 - события, 617
 - AfterDispatch, 617

TWebModule (*продолжение*)

BeforeDispatch, 617

OnCreate, 617

OnDestroy, 617

TWebRequest, 616

TWebRequest

свойства, 616

Content, 616

ContentFields, 616

Method, 616

Query, 616

QueryFields, 616

RemoteAddr, 616

RemoteHost, 616

TWebResponse, 617

TWebResponse

свойства, 617

Content, 617

ContentLength, 617

ContentStream, 617

ContentType, 617

TWinControl, 302

TWordField, 292

Type, 203

U

UML, 86

call, 109

create, 109

destroy, 109

return, 109

send, 109

абстрактный класс, 96

вариант использования, 91

диаграмма

активности, 107

классов, 91, 95

компонентов, 113

кооперации, 91

последовательности, 91, 108

прецедентов, 91

развертывания, 115

состояний, 103

сотрудничества, 110

сотрудничества уровня примеров, 110

сотрудничества уровня

спецификаций, 110

дорожки, 108

интерфейс, 92

исключающая ассоциация, 99

исторические состояния, 104

конечное состояние, 103

кратность, 96

метамодель, 90

начальное состояние, 103

UML (*продолжение*)

отношение

агрегации, 99

ассоциации, 93, 99

включения, 93

зависимости, 98, 114

композиции, 100

обобщения, 93, 100

расширения, 93

реализации, 114

отношение зависимости, 114

пакеты, 95

параллельное состояние, 104

прецедент, 91

синхронизирующие состояния, 106

составное состояние, 104

состояние действия, 107

состояния, 103

сторожевое условие, 105

строка-свойство, 97

узел, 115

Unassigned, 215

UNION, 372, 373

UNION ALL, 372, 373

UNIQUE, 154, 156, 355

unit, 224, 225

UPDATE, 169, 171, 172, 174

UPDATE OF, 160

URL, 579, 580

Uses, 202

uses, 224, 225

V

Var, 203

VarArrayCreate, 215

VarArrayLock, 216

VarArrayOf, 216

VarArrayRedim, 216

VARCHAR, 150

variant, 214

VarType, 215

virtual, 234

Visual Basic, 65

VMT, 234

W

WHERE, 171, 173, 350, 364

while...do, 219

WideChar, 206

WideString, 209

WinHelp, 448, 477

WITH GRANT OPTION, 175

Word, 205

WordBool, 206

write, 231, 232

Х**XML**

описание типа документа, 595

А

Аббота метод, 89
абсолютный URL-адрес, 591
абстрактные методы, 98
абстрактный класс, 95
автоматизация, 486, 553
активный объект, 111
алгоритм, 107
аномалии ввода, 142
аномалии обновления, 142
аномалии удаления, 142
артефакт, 114
архитектура
 клиент-сервер, 30
 файл-сервер, 29
архитектура клиент-сервер, 79
атрибут, 127
атрибуты, 129, 134

Б

база данных, 117
безопасность информационной системы, 37
библиотека
 COM, 534
 визуальных компонентов, 261
 типов, 554
блокировки, 421
браузер, 580
Буча метод, 89
быстрая разработка приложений, 68

В

вариантные записи, 212
вариантные типы, 205, 214
веб-дизайн, 582, 586
веб-документ, 581
веб-приложения, 574, 580–582
веб-программирование, 582, 584
веб-сервер, 580, 581, 582, 585, 596, 603, 606
 переменные окружения, 611
 CONTENT_LENGTH, 611
 PATH_INFO, 612
 QUERY_STRING, 610
 REMOTE_ADDR, 612
 REMOTE_HOST, 612
 REQUEST_METHOD, 612
 SERVER_NAME, 612
 SERVER_PORT, 612
 SERVER_PROTOCOL, 612
веб-страницы, 581
 динамические, 581
 статические, 581

верификация проекта, 181
версии проекта, 442
версия проекта, 421
ветвление, 107
вещественные типы с плавающей точкой, 150
вещественные типы с фиксированной точкой, 150
вид параметра, 97
визуальное программирование, 65
визуальные средства разработки, 65
 специализированные, 65
 универсальные, 65
внешние серверы, 553
внешние соединения, 370
внешний ключ, 157, 158, 159
внешний сервер, 562
внутренний сервер, 530, 536, 549, 553, 562
всплывающие подсказки, 393, 445
вычисляемые поля, 291

Г

гибкость информационной системы, 36
гиперссылки, 591
главная таблица, 136, 338
глобальный уникальный идентификатор, 531
группа проектов, 414
 управление, 414
группировка данных, 363
групповая информационная система, 26

Д

двоичные строки переменной длины, 150
двоичные строки фиксированной длины, 150, 151
действительные типы, 207, 208
действия, 399, 400
декомпозиция проекта, 61
деструктор, 236
детерминант, 143
диаграмма
 сущность-связь, 88
диаграммы сущность-связь, 178
диапазонные типы, 207
директивы компилятора, 416, 418
домен, 127, 128, 129, 578
доменная система имен, 578
доменный адрес, 578
дуальный интерфейс, 557

Ж

жизненный цикл информационных систем, 39, 45, 48, 61, 62, 66, 79, 82, 196
вспомогательные процессы, 47

жизненный цикл информационных систем

(*продолжение*)

организационные процессы, 47

основные процессы, 46

разработка, 46

сопровождение, 46

эксплуатация, 46

структура, 45, 48

конструирование, 49

начальная стадия, 49

передача в эксплуатацию, 49

уточнение, 49

журнал изменений базы данных, 120

З

заголовок программы, 202

записи, 208, 211

значение по умолчанию, 97

значения по умолчанию, 161

И

идентификатор интерфейса, 531

иерархия объектов, 88

избыточность данных, 141

изображения, 277

импорт библиотеки типов, 492

индекс, 139

индексы, 138

одно столбцовый индекс, 162

создание, 161

составные индексы, 162

типы индексов, 140

удаление, 161, 163

уникальные индексы, 162

инкапсуляция, 64, 228, 229, 246

инспектор объектов, 260

интегрированная среда разработки, 241, 247

Интернет, 574

Интернет-приложения, 574, 605

интерфейс, 102, 536

интерфейс диспетчеризации, 486, 553, 554

интерфейс пользователя, 392, 393

интрасети, 603

информационная система, 22, 60, 63, 66, 77

безопасность, 37

гибкость, 36

групповая, 26

корпоративная, 26, 39, 46

надежность, 36

одиночная, 26

ошибки, 38

сложность, 38

фазы проектирования, 42

эффективность, 36

информационно-справочная система, 28

информационные системы, 420

информационные технологии, 60

исключительная ситуация, 97

исключительные ситуации, 238

итерационная модель разработки, 63

итерационный процесс разработки, 56

К

кардинальное число, 130

каскадная модель, 50

возврат на более ранние стадии, 53, 55

высокий уровень риска, 55

задержка получения результатов, 53

информационная перенасыщенность, 54

основные этапы разработки, 50

сложность параллельного ведения работ, 54

сложность управления проектом, 55

каскадное обновление, 160

каскадное удаление, 135

клавиатурные сокращения, 394, 397

клавиши ускоренного доступа, 397

класс, 95

абстрактный, 95

примесный, 101

классификация информационных систем,

25, 27

классы, 208, 213, 226

клиент автоматизации, 487

клиент объекта, 530

ключ, 127, 130

альтернативный, 132

внешний, 134

вторичный, 133

естественный, 131

искусственный, 131

первичный, 130, 131, 143

потенциальный, 132

простой ключ, 131

сложный, 131

составной, 131

суррогатный, 131

кнопки, 265

Кодд, 124, 126, 127

коллективная разработка приложений, 420

коллективная разработка проектов, 181

команды запуска приложения, 419

команды компиляции, 419

комбинированные списки, 277

компиляция приложения, 418

компиляция файла справки, 457

компонент, 113

компонентное программирование, 563

компонентный подход к

программированию, 58

компоненты, 240, 246

- компоненты Delphi, 250, 261
 - визуальные, 264
 - невизуальные, 264
- компоненты VCL, 406
- компоненты для формирования документов
 - HTML, 621
- компьютерная инфраструктура, 24
- константы, 204
- конструктор, 236
- контекстная справка, 444
- контекстное меню, 400
- контроллер автоматизации, 554
- контрольная сумма, 577
- конфликт методов, 102
- концептуальная модель, 177, 182, 184, 193
 - объектно-ориентированная модель, 177
 - семантическая модель, 177
- кооперация, 110
- корпоративная информационная система, 22, 26
- корпоративная сеть, 24
- корпоративный стандарт, 77
- кортеж, 127, 129, 132

Л

- линия синхронизации, 106
- логические типы, 206
- локальные переменные, 224
- локальные серверы, 530

М

- макрокоманды WinHelp, 450
 - ALink, 451
 - AppendMenu, 451
 - BrowseButtons, 451
 - ControlPanel, 451
 - CreateButton, 451
 - ExecFile, 451
 - Find, 451
 - InsertMenu, 451
 - KLink, 451
 - MPrintID, 451
 - ShellExecute, 451
 - ShortCut, 451
- манипулирование данными, 169
- маршрутизатор, 576
- маршрутизация, 576
- массивы, 208, 209
 - динамические массивы, 210
 - статические массивы, 209
- менеджер проекта, 252, 415
- меню, 392, 393
 - главное меню, 394
 - подменю, 398
 - разделители, 398

- реакция на выбор команды, 398, 399
- редактор меню, 396
- метакласс, 103
- метки, 202, 218
- метод, 97
- метод GET, 583, 610, 620
- метод POST, 583, 610, 611, 614, 620
- методология быстрой разработки
 - приложений, 62
- методы, 233
 - абстрактные, 235
 - виртуальные, 234
 - динамические, 234
 - перегружаемые, 236
 - статические, 233
- методы класса, 226
- многодокументный интерфейс, 316
- многомерные массивы, 209
- многотабличные базы данных, 338
- многоуровневая сетевая модель
 - межсетевой уровень, 575, 576
 - транспортный уровень, 575, 577
 - уровень приложений, 575, 578
 - уровень сетевого доступа, 575, 576
 - LLC, 576
 - MAC, 576
- множества, 208, 211
- множественное наследование, 101
- модальные формы, 316
- модель
 - COM, 529, 562
 - жизненного цикла, 50, 84, 86
 - каскадная, 50, 52
 - спиральная, 50, 56
 - жизненного цикла информационной системы, 50
 - сущность-связь, 178
 - модули, 202, 224
 - блок завершения, 225
 - блок инициализации, 225
 - блок интерфейса, 225
 - блок реализации, 225
 - заголовок, 225
 - модули данных, 296
 - мощность отношения, 130
 - мультиобъект, 111

Н

- набор данных, 284, 287
- надежность информационной системы, 36
- надписи, 267
- наследование, 125, 228, 229, 246
- неключевой атрибут, 143, 145
- немодальные формы, 316
- нормализация данных, 140

нормальные формы, 142
 вторая нормальная форма, 143
 нормальная форма Бойса—Кодда, 143
 первая нормальная форма, 143
 пятая нормальная форма, 143
 третья нормальная форма, 143, 145
 четвертая нормальная форма, 143

О

области видимости, 237
 автоматизация, 237
 защищенные, 237
 личные, 237
 общие, 237
 опубликованные, 237
 область действия метода, 98
 обновление локальных копий файлов проекта, 426, 431
 обобщение, 110
 обработка исключительных ситуаций, 246
 обработчик события, 66, 261
 объединение запросов, 372
 объект, 64, 66, 102, 227
 объект COM, 534
 объект автоматизации, 556, 558, 561, 562
 объект базы данных, 152
 объектная модель, 125
 объектно-ориентированная модель, 200
 объектно-ориентированное программирование, 64, 65, 87, 200, 226
 объектно-ориентированное проектирование, 63
 объектно-ориентированные СУБД, 125
 объектно-ориентированный анализ
 CRC-карточки, 89
 метод Аббота, 89
 объектные типы, 226
 объекты, 227
 ограничения, 154
 ограничение CHECK, 154, 160
 ограничение NOT NULL, 154, 155, 156
 ограничение UNIQUE, 154, 156, 157
 ограничение внешнего ключа, 154, 157, 159
 ограничение первичного ключа, 154, 155
 ограничения целостности, 134
 ограничительные условия, 134, 152
 одиночная информационная система, 26
 однокластерный интерфейс, 316
 окна диалога, 278, 280
 методы, 279
 Execute, 279
 свойства, 279
 DefaultExt, 279
 FileEditStyle, 279
 FileName, 279
 Files, 279
 Filter, 279
 окна диалога (*продолжение*)
 FilterIndex, 279
 HistoryList, 279
 InitialDir, 279
 Options, 279
 Title, 279
 события, 279
 OnClose, 279
 OnCloseQuery, 279
 OnFolderChange, 279
 OnSelectionChange, 279
 OnShow, 279
 OnTypeChange, 279
 фильтры, 280
 оператор
 break, 220
 case...of, 218
 continue, 220
 if, 217
 безусловного перехода, 217
 присваивания, 216
 операторы Object Pascal, 216
 операторы цикла, 218
 оптимизация быстродействия, 113
 ответственность разработчика информационных систем, 38
 открытые информационные системы, 70
 открытые массивы, 223
 относительный URL-адрес, 587, 591
 отношение
 основные свойства, 137
 отношения, 127
 отчеты, 381
 отчеты в свободной форме, 382

П

пакеты, 263, 416, 574, 575
 пакеты времени выполнения, 264
 пакеты времени разработки, 264
 панели инструментов, 392, 393, 400
 параметризированный класс, 103
 параметры процедуры, 221
 параметры-константы, 222
 параметры-переменные, 222
 первичный ключ, 66
 передача параметров, 221
 по значению, 221
 по ссылке, 222
 переключатели, 270
 перекомпиляция, 113
 перекрестные ссылки, 449
 перекрытие методов, 236
 переменные, 203
 перечисляемые типы, 207
 планирование приложения, 392

повторное использование компонентов, 58
подзапросы, 371
подсистема, 24
подсчет ссылок, 534
подчиненная таблица, 136, 338
позднее связывание, 234, 489, 562
поле, 289
полиморфизм, 125, 228, 230, 246
пользователь, 91
поля записи, 212
поля класса, 226, 231
порт, 579
порядковые типы, 205
предметная область, 64, 88, 132, 177
представление, 163
представления, 374
 области применения, 164
 создание, 165
 удаление, 165
привилегии пользователей, 173
 объектные, 173, 174
 системные, 173
признак оригинальности программы, 90
примесный класс, 101
примечание, 91
проблема коммуникации, 87
программа Object Pascal, 201
программирование
 визуальное, 65
 объектно-ориентированное, 65
 событийное, 66
проект, 40
 классификация, 42
 настройка параметров, 415
 основные отличительные признаки, 40
 свойство управляемости, 41
 техничко-экономические показатели, 41
проект Delphi, 406
 главный файл проекта, 408
 добавление форм и модулей, 410
 модуль формы проекта, 407
 удаление форм и модулей, 410
 файл описания формы, 409
 файлы проекта, 407
производный класс, 229
пространства имен, 262
простые индексы, 140
простые типы, 204, 205
протокол, 574, 575
 FTP, 578, 579
 HTTP, 578, 579, 580, 582, 606
 IP, 574, 575, 577
 POP, 578
 SMTP, 579
 TCP, 577

 протокол (*продолжение*)
 TCP/IP, 577
 WAL, 120
протоколирование, 119, 120
протоколы SMTP, 578
прототип, 63, 64, 65, 67
прототипы, 63, 181
профиль жизненного цикла информационных систем, 77
профиль информационной системы, 70
профиль информационных систем
 структура, 73
процедурно-ориентированная модель, 200
процедуры, 221
псевдоним, 263, 293
псевдонимы полей, 361
публикация, 580

Р

раннее связывание, 489
редактор кода, 261
редактор форм, 261
рекурсия, 110
реляционная база данных, 137
реляционная модель данных, 126, 127, 176
реляционная система управления базами данных, 138
реляционные базы данных, 116, 124, 125
реляционные СУБД, 126, 132, 135
репозиторий, 180, 181
родительская таблица, 157
родительский ключ, 157, 159
роль, 110

С

сбои, 119
 аппаратные, 119
 жесткие, 119, 120
 мягкие, 119, 120
 программные, 119
свойства, 231
 векторные, 232
 значение по умолчанию, 232
 только для записи, 232
 только для чтения, 232
свойства класса, 226, 227
связи между таблицами, 136
 многие к одному, 136
 многие ко многим, 136
 один ко многим, 136
 связь один к одному, 136
связь, 133
селектор, 218
семантическое моделирование, 177
сервер автоматизации, 487, 553, 555
сеть
 корпоративная, 24

символьные строки переменной
длина, 149, 150

символьные строки фиксированной длины, 149

символьные типы, 206

синхронизация версий, 62

синхронизация документации, 54

система

- информационно-справочная, 28
- обработки транзакций, 27
- поддержки принятия решений, 28
- управления базами данных, 62, 65, 117

системы контроля версий проектов, 421

сложность информационной системы, 38

событийное программирование, 66

события, 66

содержание справочной системы, 449

соединение неравенства, 369

соединение равенства, 367

соединения таблиц, 367

сокращение избыточности, 101

сообщение, 227

составной класс, 536

составной объект, 111, 530

составной оператор, 220

составные запросы, 372

составные индексы, 140

состояния набора данных, 287

спецификация ISAPI, 597

спиральная модель, 50

- итерации, 56

список, 276

справочная система, 444

среда визуальной разработки, 241

средства

- быстрой разработки приложений, 62, 240
- визуального программирования, 64, 65, 68, 241
- групповой разработки, 422, 423

ссылочная целостность, 66

ссылочная целостность данных, 159

стандарт

- ISO/IEC 12207, 45
- ISO/IEC 12207, 50
- ISO/IEC 12207 1995-08-01, 82
- ISO/IEC 12207 1995-08-01, 78
- виды, 77
- корпоративный, 77
- на интерфейсы, 77
- открытых систем, 77

степень отношения, 129

стереотип, 98, 109

сторожевое условие, 105, 112

строка состояния, 445, 447

строки, 208

строковые типы, 209

структурные типы, 203, 204, 208

СУБД

- основные свойства, 117
- основные функции, 117

суперключ, 131

схема базы данных, 129

схема отношения, 129

сценарий, 581

Т

таблица виртуальных методов, 234

таблица динамических методов, 234

табличные отчеты, 382

табличные формы, 329

тег, 587, 589

- A, 591
- B, 589
- BODY, 587
- BR, 589
- FORM, 592
- FRAMESET, 588
- HEAD, 587
- I, 589
- INPUT, 592, 593
- LI, 589
- LINK, 587
- OL, 589
- P, 588
- U, 589
- UL, 589

текстовые поля, 271

текущая запись, 136

темы, 448, 451, 468

- атрибуты, 452
- изображения, 457
- кнопки, 456
- перекрестные ссылки, 454
- текст, 453

технология проектирования, 61

тип данных, 127

типизированные константы, 204

типизированные поля, 292

типы, 203

типы данных SQL/92, 149

- для представления даты и времени, 149, 151
- строковые, 149
- числовые, 149, 150

транзакции, 118

транзакция, 89

- откат, 118
- фиксация изменений, 118

триггер, 66, 79, 105, 148

- создание, 168
- удаление, 169

триггеры, 168

У

- удаленные серверы, 530
- указатели справочной системы, 449
- указательные типы, 205, 213
- универсальный язык моделирования, 86
- уникальные индексы, 140
- унифицированный указатель ресурсов, 579
- управление
 - безопасностью базы данных, 173
 - доступом к базе данных, 174
 - конфигурацией, 63
 - объектами базы данных, 152
 - проектами, 40
 - реляционными базами данных, 147
 - транзакциями, 118
- условный оператор, 217
- устройства внешней памяти, 118

Ф

- фабрика классов, 535
- фазы проектирования информационной системы, 42
 - ввод системы в эксплуатацию, 44
 - концептуальная фаза, 43
 - подготовка технического предложения, 43
 - проектирование, 44
 - разработка, 44
- файловый тип, 212
- файлы, 208
- фактические параметры, 221
- физическая модель, 193, 195
- физическое проектирование системы, 68
- фокус управления, 108
- формальные параметры, 221, 223
- формы, 264, 316, 412
 - HTML, 592
 - для ввода, 325
 - с вкладками, 335
 - управление формами, 412
- фреймы, 321
- функции, 223
 - агрегирования, 362
- функциональная зависимость, 143
- функциональная связь является транзитивной транзитивная, 145

Х

- хранилище версий, 441
- хранилище объектов, 250, 421, 424

- храняемая процедура, 31, 79, 166
 - выполнение, 167
 - выполняемые процедуры, 167
 - процедуры выбора, 166
 - создания, 167
 - удаление, 167

Ц

- целостность данных, 134
 - категорийная целостность, 134
 - ссылочная целостность, 134, 135
- целочисленные типы, 150
- целые типы, 205
- цикл
 - for...do, 218
 - repeat...until, 219
 - while...do, 219

Ч

- члены класса, 228

Э

- экземпляр класса, 227
- элемент ActiveX, 562, 563, 566
- элементы ActiveX, 564
- эффективность информационной системы, 36

Я

- язык
 - HTML, 582, 586
 - OCL, 91
 - QBE, 124, 147
 - QUEL, 147
 - SQL, 121, 124, 147, 284, 342, 343, 348
 - XHTML, 595
 - XML, 595
- запросов, 149
- манипулирования данными, 148
- определения данных, 148
- основных типов команд, 148
 - команд администрирования баз данных, 149
 - команды управления транзакциями, 149
- управления данными, 148
- описания интерфейсов, 539
- баз данных, 121
 - манипулирования данными, 121
 - определения схем данных, 121

Избачков Юрий Сергеевич,
Петров Владимир Николаевич

Информационные системы: Учебник для вузов
2-е издание

Заведующий редакцией
Руководитель проекта
Литературный редактор
Иллюстрации

Художник
Корректор
Верстка

*А. Кривцов
Ю. Суркис
А. Жданов
Л. Родионова,
В. Шендерова, М. Шендерова
Л. Адуевская
И. Смирнова, Н. Филатова
А. Келле-Пелле*

Лицензия ИД № 05784 от 07.09.01.

Подписано к печати 19.01.06. Формат 70×100/16. Усл. п. л. 52,89. Доп. тираж 3500. Заказ 507
ООО «Питер Принт», 194044, Санкт-Петербург, пр. Б. Сампсониевский, д. 29а.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.

Отпечатано с готовых диапозитивов в ОАО «Техническая книга»
190005, Санкт-Петербург, Измайловский пр., 29