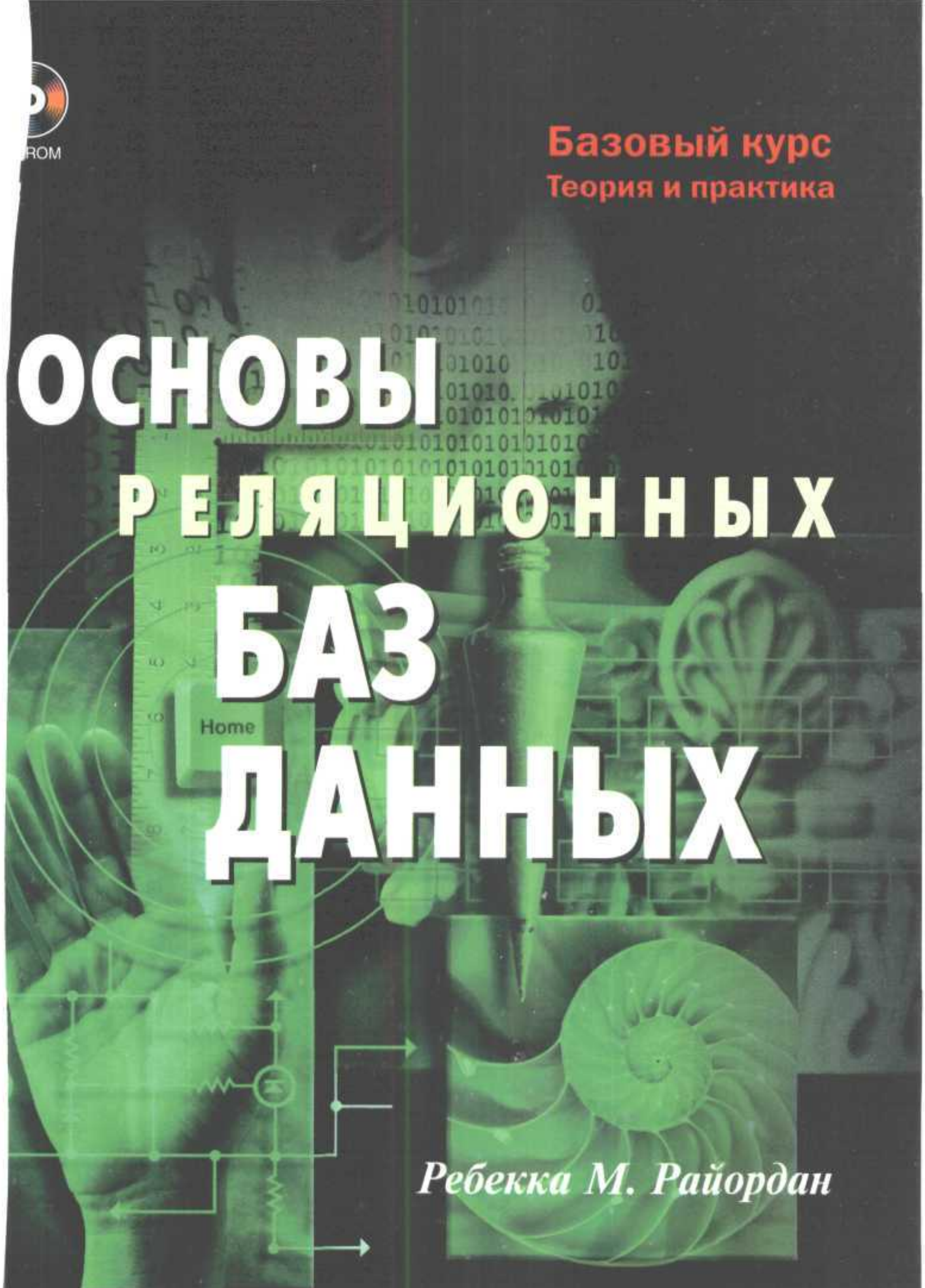


РУССКАЯ РЕДАКЦИЯ

Microsoft Press

Rebecca M. Riordan

DESIGNING **Relational Database Systems**

Microsoft Press

Ребекка Райордан

ОСНОВЫ реляционных баз данных

Москва 2001

УДК 004
ББК 32.973.26-018.2
Р45

Райордан Р.

Р45 Основы реляционных баз данных/Пер. с англ. — М.: Издательско-торговый дом «Русская Редакция», 2001. — 384 с.: ил.

ISBN 5-7502-0150-3

Книга посвящена вопросам проектирования и внедрения современных компьютерных систем, работающих с базами данных. Автор приводит и теоретические сведения, и информацию по разработке и внедрению таких систем. Содержится подробный анализ ситуаций, часто встречающихся на практике. Особое внимание уделено созданию проектной документации, проектированию пользовательского интерфейса и проблемам безопасности. В качестве примеров рассматриваются СУБД Microsoft.

Издание адресовано менеджерам проектов, системным аналитикам и разработчикам баз данных, а также всем, кто интересуется вопросами, связанными с современными системами управления реляционными базами данных.

Книга состоит из 18 глав, словаря терминов и предметного указателя. Прилагается компакт-диск с дополнительными материалами, шаблонами документов и форм, а также базами данных, которые анализируются в книге в качестве примеров.

УДК 004
ББК 32.973.26-018.2

Подготовлено к изданию по лицензионному договору с Microsoft Corporation, Редмонд, Вашингтон, США.

ActiveX, Microsoft, Microsoft Press, Outlook, PowerPoint, Visual Basic, Visual SourceSafe, Visual Studio и Windows являются товарными знаками или охраняемыми товарными знаками корпорации Microsoft в США и/или других странах. Все другие товарные знаки являются собственностью соответствующих фирм.

Все названия компаний, организаций и продуктов, а также имена лиц, используемые в примерах, вымышлены и не имеют никакого отношения к реальным компаниям, организациям, продуктам и лицам.

© Оригинальное издание на английском языке, Ребекка Райордан, 1999
© Перевод на русский язык, Microsoft Corporation, 2001
© Оформление и подготовка к изданию, издательско-торговый дом «Русская Редакция», 200!

ISBN 0-7356-0634-X (англ)
ISBN 5-7502-0150-3

Оглавление

Предисловие XIII
От автора XVI
Введение XVII
О компакт-диске, прилагаемом к книге XIX

Часть I. Теория реляционных баз данных 1

Глава 1. Основные понятия 2
Что такое база данных 3
Инструменты для работы с базами данных 5
Механизм СУБД 6
Объектная модель доступа к данным 7
Средства для разработки клиентской части приложений 8
Реляционная модель 9
Термины, используемые в реляционной теории 11
Модель данных 13
Сущности 13
Атрибуты 16
Домены 21
Связи 23
Диаграмма «сущности - связи» 25
Итоги 27

Глава 2. Структура базы данных 29
Основные принципы 35
Декомпозиция без потерь 35
Ключи-кандидаты и первичные ключи 36
Функциональная зависимость 38
Первая нормальная форма 40
Вторая нормальная форма 43
Третья нормальная форма 44
Дальнейшая нормализация 46
Нормальная форма Бойса-Кодда 46
Четвертая нормальная форма 49

Пятая нормальная форма ... 50
Итоги ... 52
Глава 3. Связи ... 53
Основные понятия и определения ... 53
Моделирование связей ... 56
Связи «один к одному» ... 59
Реализация сущностей как классов-наследников ... 61
Связи «один ко многим» ... 64
Связи «многие ко многим» ... 65
Унарные связи ... 66
Тройные связи ... 67
Связи определенной мощности ... 70
Итоги ... 72
Глава 4. Целостность данных ... 73
Ограничения целостности ... 73
Целостность доменов ... 74
Целостность на уровне переходов ... 76
Целостность на уровне сущности ... 77
Ссылочная целостность ... 79
Целостность на уровне базы данных ... 80
Целостность на уровне транзакций ... 81
Реализация целостности данных ... 82
Неопределенные и несуществующие величины ... 82
Реакции на нарушения целостности ... 85
Декларативная и процедурная целостность ... 86
Целостность на уровне домена ... 86
Целостность на уровне сущности ... 87
Ссылочная целостность ... 92
Другие виды целостности ... 94
Итоги ... 94
Глава 5. Реляционная алгебра ... 95
Значения Null (еще раз о трехзначной логике) ... 96
Реляционные операторы ... 98
Ограничение ... 99
Проекция ... 99
Соединение ... 99
Эквисоединения ... 100
Естественные соединения ... 101

Тета-соединения ... 102
Внешние соединения ... 104
Деление ... 105
Операции над множествами ... 105
Объединение ... 106
Пересечение ... 107
Разность ... 108
Декартово произведение ... 109
Дополнительные реляционные операторы ... 110
Агрегирование ... 110
Расширение ... 111
Переименование ... 111
Оператор TRANSFORM ... 112
Оператор ROLLUP ... 113
Оператор CUBE ... 114
Итоги ... 115

Часть II. Проектирование реляционных систем баз данных ... 117

Глава 6. Процесс проектирования ... 118
Модели жизненного цикла ... 118
Проектирование базы данных ... 123
Определение параметров системы ... 123
Проектирование рабочих процессов ... 123
Построение концептуальной модели данных ... 123
Подготовка схемы базы данных ... 123
Проектирование пользовательского интерфейса ... 123
Замечания о стандартах и технологиях проектирования ... 124

Глава 7. Определение параметров системы ... 125
Цели и границы применения системы ... 125
Определение критериев разработки ... 132
Критерии, выражаемые в измеряемых единицах ... 133
Критерии, определяемые внешним окружением ... 134
Основные направления разработки ... 136
Определение масштаба и границ системы ... 137
Стоимостный анализ ... 138
Итоги ... 142

Глава 8. Определение рабочих процессов 143
Выявление существующих рабочих процессов 144
Беседы с пользователями 144
Определение задач 145
Анализ рабочих процессов 149
Документирование рабочих процессов 151
Пользовательские сценарии 153
Итоги 154

Глава 9. Концептуальная модель данных 155
Определение объектов базы данных 155
Определение связей 160
Мощность связи 163
Обязательность связи 163
Атрибуты связи 163
Дополнительные ограничения 164
Повторный анализ сущностей 164
Связь между сущностью и предметной областью 164
Рабочие процессы, влияющие на сущности 165
Взаимодействие между сущностями 166
Бизнес-правила и ограничения 166
Атрибуты 166
Анализ доменов 168
Выбор типа данных 169
Ограничения на диапазон данных 169
Определение формата 171
Нормализация 171
Итоги 171

Глава 10. Схема базы данных 173
Системная архитектура 173
Программная архитектура 173
Трехуровневая архитектура 174
Четырехуровневая архитектура 176
Программная архитектура и схема базы данных 180
Архитектура данных 183
Одноуровневая архитектура 184
Двухуровневая архитектура 188
Многоуровневая архитектура 190
Интернет и интранет-архитектура 191

Компоненты схемы базы данных 193
Таблицы и связи 193
Ограничения 194
Связи 194
Индексы 195
Представления и запросы 196
Защита данных 198
Уровни защиты данных 199
Отслеживание и регистрация системных событий 201
Итоги 202

Глава 11. Сотрудничество при проектировании 205

Общение с заказчиком 205
Структура документа 206
Введение 206
Обзор системы 208
Рабочие процессы 209
Концептуальная модель данных 210
Схема базы данных 210
Интерфейс пользователя 211
Прототип интерфейса 211
Спецификации интерфейса 212
Контроль за изменениями 213
Специальные средства 213
Итоги 214

Часть III. Проектирование пользовательского интерфейса 215

Глава 12. Интерфейс как посредник между пользователем и системой 216

Роль пользовательского интерфейса в системе 216
Модели интерфейса 218
Уровни подготовки пользователей 220
Начинающий пользователь 220
Опытный пользователь 221
Эксперт 221
Возложите на пользователя ответственность за его действия 222
Не перегружайте память пользователя! 225

Будьте последовательны! ... 227
Итоги ... 230

Глава 13. Архитектура пользовательского интерфейса ... 231

Поддержка рабочих процессов ... 231
Однодокументный и многодокументный интерфейс ... 234
Однодокументная архитектура ... 234
Рабочая книга ... 235
Интерфейс, использующий стиль приложения Microsoft Outlook ... 236
Многодокументная архитектура ... 237
«Классическая» архитектура MDI ... 238
Диалоговая панель управления ... 240
Проект ... 241
Мастер ... 243
Итоги ... 244

Глава 14. Связь между сущностями и формами системы ... 245

Простые сущности ... 246
Связи «один к одному» ... 249
Связи «один ко многим» ... 251
Иерархические структуры ... 255
Связи «многие ко многим» ... 257
Итоги ... 261

Глава 15. Выбор элементов управления пользовательского интерфейса ... 263

Логические значения ... 265
Наборы значений ... 267
Выбор одного значения из диапазона ... 267
Выбор нескольких значений из диапазона ... 270
Числовые данные и даты ... 273
Текстовые данные ... 275
Итоги ... 277

Глава 16. Поддержка целостности базы данных ... 279

Классы ограничений целостности ... 280
Внутренние ограничения ... 282
Ограничения, налагаемые на тип данных ... 283
Ограничения, налагаемые на формат данных ... 283
Ограничения, налагаемые на длину данных ... 283
Значения *Null* ... 285

Ограничения, налагаемые на диапазон возможных значений ... 286
Ограничения на уровне сущностей и ссылочная целостность ... 287
Бизнес-правила ... 291
Случайные ошибки при вводе данных ... 292
Модель системы и реальность ... 293
Итоги ... 296

Глава 17. Создание отчетов ... 297

Сортировка, поиск и использование фильтров ... 298
Сортировка данных ... 299
Фильтр по выделенному фрагменту в одном поле ... 299
Фильтр по заданным значениям в нескольких полях ... 301
Расширенный фильтр ... 302
Средство построения запросов Microsoft English Query ... 303
Стандартные отчеты ... 304
Отчеты в виде списков и подробные отчеты ... 304
Отчеты, использующие агрегированные данные ... 305
Отчеты на основе форм пользовательского интерфейса ... 306
Интерфейс для создания отчетов ... 306
Обработка ошибок принтера ... 308
Печать автоматическая и по команде пользователя ... 310
Пользовательские отчеты ... 311
Средства создания отчетов ... 312
Настраиваемые пользовательские отчеты ... 312
Стандартные письма ... 318
Итоги ... 319

Глава 18. Поддержка пользователя ... 321

Пассивные механизмы поддержки ... 323
Запоминающиеся сочетания клавиш ... 323
Всплывающие подсказки ... 324
Строка состояния ... 326
Реактивные механизмы поддержки пользователя ... 327
Контекстная справка, вызываемая пользователем ... 328
Подсказки типа «Что это такое» ... 329
Звуковые сигналы ... 331
Сообщения об ошибках ... 331
Активная помощь ... 332
Обучение пользователя ... 333
Итоги ... 334

Словарь терминов 335
Рекомендуемая литература 341
Предметный указатель 343
Об авторе 353

Предисловие

Перед вами книга, которая как бы перекидывает мостик между академической теорией и процессом проектирования баз данных в реальных условиях. Вы найдете в ней полезную информацию, которая пригодится на всех стадиях разработки приложений для баз данных - от начального этапа проектирования базы данных до создания пользовательского интерфейса, составления пользовательской документации и планирования курсов подготовки пользователей.

Если вы создаете реальную систему, предназначенную для решения конкретных проблем, то эта книга станет для вас поистине уникальным источником достоверной и полной информации. В ней предлагается множество реальных решений, реализованных на SQL. Даже экспертам в области СУБД будет интересно и полезно прочитать ее.

Вам знакомо понятие «тета-соединение»? Может быть, вы часто слышали этот термин, но не уверены, что он означает и где эти соединения используются? Если вы не прослушали полный курс теории баз данных и хотите наверстать упущенное, или же просто решили поразить коллег глубиной теоретической подготовки, то эта книга окажется вам весьма полезной. К тому же, наряду с общими сведениями она содержит множество практических примеров и решений, непосредственно вытекающих из реляционной теории. Эти примеры — результат многолетнего опыта и анализа ошибок, которые не так уж редки даже у экспертов.

Или возможно, вы получили фундаментальное образование и хотите знать, как наиболее эффективно использовать полученные знания на практике? Предположим, срок реализации системы истекает на следующей неделе, а выбранная вами СУБД не поддерживает реляционный оператор деления, необходимый для создания ключевого

отчета. В этой книге вы найдете совет, как справиться с проблемой, а также ответы на другие вопросы, которые возникают сами собой, стоит лишь перейти от теории к практике.

Реализовать пользовательскую систему на основе СУБД зачастую весьма не просто, но сформулировать практические задачи, которые она должна решить, и добиться, чтобы проектируемая система действительно служила этой цели, — еще сложнее. Вторую часть своей книги Ребекка Райордан посвящает практическим вопросам реализации систем на основе баз данных. Она щедро делится с читателями своим богатым опытом, подсказывает, как избежать «подводных камней», много внимания уделяет работе с пользователями. Прислушайтесь к ее советам, и ваша система действительно будет удовлетворять всем требованиям заказчика и послужит примером рационального подхода к проектированию.

В книге также подробно рассматриваются вопросы архитектуры приложений на основе СУБД, с практической точки зрения сравниваются различные варианты архитектуры — одноуровневой, двухуровневой и многоуровневой. Ребекка не обошла вниманием также разработку систем, предназначенных для работы с Интернетом. Кроме того, обсуждаются вопросы администрирования и безопасности, которым многие разработчики и по сей день не придают должного значения.

Уделяя основное внимание процессу разработки базы данных, автор не останавливается на этом. Книга содержит немало ценных практических рекомендаций по организации управления проектом в течение всего цикла разработки, эффективному взаимодействию с разработчиками и будущими пользователями системы, а также ряд советов относительно формы и содержания различных документов, создаваемых в процессе разработки. Автор подсказывает, как оптимально выбрать элементы управления для пользовательского интерфейса, рассматривает несколько готовых решений на основе стандартных элементов управления Microsoft Windows, подробно описывает, каким типам данных какой элемент соответствует. В главе 16, посвященной поддержке целостности базы данных, изложен один из наиболее реалистичных подходов к этой сложной проблеме, какие я когда-либо встречал. И, наконец, автор рассказывает о том, как наиболее рационально спланировать и реализовать систему помощи пользователю, применив самые передовые интерактивные средства.

Итак, перед вами — отличная книга, посвященная теории и практике проектирования приложений, работающих с базами данных и написанная человеком, обладающим богатым опытом в этой облас-

ти. Полагаю, что ее оценят все, кто занят разработкой современных приложений на основе реляционных СУБД. Лично я читал ее с большим интересом и рекомендую всем, кто проектирует или собирается проектировать такие системы.

Майкл Ми (Michael Mee)

От автора

На обложке этой книги стоит мое имя. Но своим появлением она обязана не только мне — в работе принимало участие множество людей. Без их помощи книга никогда бы не увидела свет.

Я от всего сердца хочу поблагодарить своих друзей и близких.

Прежде всего, членов моей семьи: мужа, Марка РаЙордана, который верит в меня, в мои силы и возможности, моих родителей Диану и Харлоу Райт — они поддерживали меня и не давали унывать ни при каких обстоятельствах.

Большое спасибо Майку Ми, первым давшему положительный отзыв об этой книге и написавшему предисловие.

Я благодарю Эрика Стру, за все его мучения со мной, неопытным и зачастую неорганизованным начинающим автором.

Приношу благодарность тем, кто терпеливо читал мои каракули и заметки с самого первого дня работы над книгой и высказал немало полезных замечаний: Дэву Эшишу, Джиму Фергюсону, Киму Джакобсону и Аннете Марк.

Элис Тернер, замечательный редактор, с полуслова понимала, что я хотела бы сказать. Элис внесла множество полезных предложений, касающихся структуры книги. Она не просто читала текст и исправляла ошибки — она сделала больше, чем должен был сделать редактор.

Роб Нэнс создал блестящие иллюстрации из моих набросков и рисунков, зачастую выполненных вручную, на листочке бумаги.

Кэри Хардвик не раз приходила на помощь в любое время, даже в поздний час. Воистину, друзья познаются в беде и в работе...

Я хочу также поблагодарить всех авторов интерактивной группы новостей *comp.databases.ms-access*, кто отвечал на мои вопросы. Их великодушное терпение и глубокие знания не раз помогали мне во время работы над книгой.

И наконец, хочу добавить: ***все ошибки, которые читатель заметит в этой книге, принадлежат только мне.***

Еще раз благодарю всех,

Ребекка Райордан (Rebecca Riordan)

Введение

Реляционные базы данных — один из самых сложных типов коммерческих приложений. **Все** остальные типы систем, как правило, имеют более-менее близкие аналогии в реальном мире. С точки зрения практического использования, текстовые процессоры — это усовершенствованная пишущая машинка. Электронные таблицы, несомненно, легко освоит не только бухгалтер, но и любой другой пользователь. А работа с электронной почтой достаточно похожа на обычную отправку корреспонденции. При создании парадигмы Windows Desktop была использована аналогия с письменным столом, за которым работает сотрудник — весьма приблизительная, но все же довольно близкая к реальности.

И только при работе с базами данных от пользователей требуются особые навыки и умения, которые приходят только с опытом. Я бы сравнила системы, работающие с базами данных, с одним из абстрактных разделов математики — они помогают создать модель реального мира, но сами являются абстрактными понятиями и реально не существуют. Вряд ли удастся назвать хоть один объект реального мира, похожий на реляционную базу данных по формальным признакам. Разве что библиотечные каталоги, где на карточках хранятся сведения об авторе, названии и тематике книг, немного напоминают их... И все же библиотечный каталог представляет собой всего лишь отдельные наборы данных, упорядочить и систематизировать которые может только библиотекарь.

Заметьте: я говорю о базах данных, а не о таблицах. Таблицы, несомненно, реальное понятие, и вряд ли найдется хоть один пользователь компьютерной системы, ни разу не слыхавший о них. Простейшие примеры таблиц — телефонные книги и словари. Но реляционные базы данных — это, разумеется, не таблицы; точнее, они содер-

жат таблицы, но сами по себе являются гораздо более сложным и абстрактным понятием.

Эта книга посвящена разработке систем баз данных. Ее цель — дать читателю информацию, с помощью которой он сможет проанализировать сложные, запутанные ситуации и на основе несистематизированных сведений и неупорядоченных данных создать эффективную, хорошо спланированную систему. Я старалась дать вам в руки инструмент для моделирования процессов, происходящих в реальном мире. Таким инструментом являются реляционные базы данных.

В книге три части. В первой кратко изложена теория реляционных баз данных, объясняются основные принципы реляционной модели. Возможно, она покажется вам скучной или чересчур сухой и сложной, Но не бросайте книгу — дальше дело пойдет легче. Вторая часть посвящена разработке баз данных: я расскажу, как создать базу данных в реальной ситуации. В третьей части обсуждаются вопросы, связанные с самой важной частью системы с точки зрения пользователя — пользовательским интерфейсом.

Вопросам реализации также будет уделяться значительное внимание на протяжении всей книги, но все же ее нельзя назвать «книгой о том, как программировать базы данных». Хотя я и включила в нее примеры кода, но постаралась, чтобы книга не была перегружена ими Как правило, это достаточно простые примеры, и даже если вы толком не знаете ни одного языка программирования, вы в них разберетесь. Примеры, которые я привожу, используют базу данных Northwind, поставляемую вместе с Microsoft Access. (Версия базы данных Northwind, поставляемая вместе с SQL Server 7.0, очень похожа на нее).

Надеюсь, прочитав книгу, вы получите достаточно ясное представление о разработке систем баз данных. А кроме того, будете обладать достаточной подготовкой в этой области, чтобы найти в литературе, перечисленной в библиографическом указателе в конце книги, более подробные сведения о методах программирования. И конечно, сможете быть уверены, что созданная вами архитектура данных удовлетворяет критериям целостности и не приведет впоследствии к досадным задержкам или ошибкам в реализации проекта.

Ребекка Райордан

О компакт-диске, прилагаемом к книге

Прилагаемый к книге компакт-диск содержит тематические статьи из обзоров, публикуемых Microsoft, а также из сборника Microsoft Knowledge Base, посвященные разработке баз данных. Кроме того, представлен ряд документов и шаблонов Microsoft Word — они предназначены для создания документации, описывающей процесс разработки баз данных. Вы также найдете там базу данных, используемую в качестве примера для иллюстрации процесса разработки.

Тематические статьи из обзоров Microsoft и Microsoft Knowledge Base

MSDN, Microsoft TechNet и Microsoft Knowledge Base содержат множество статей и обзоров, посвященных различным аспектам разработки баз данных. Некоторые из них включены в прилагаемый компакт-диск, в каталог, структурированный по главам книги. Полный список обзоров и статей — в файле Readme.txt.

Формы

Компакт-диск содержит формы, которые могут использоваться в процессе разработки. Формы представлены в виде документов и шаблонов Word.

Документы Word

Возможно, вы захотите использовать формы, приведенные в качестве примеров в этой книге. Они могут пригодиться вам при обсуждении деталей проекта с заказчиком или просто в качестве черновиков при работе над какой-нибудь похожей системой. Копии всех форм содержит каталог Forms. Для вывода форм на печать используйте Microsoft

Word 97 или Microsoft Word 2000. Если на вашем компьютере не установлен Microsoft Word, то для просмотра и печати форм воспользуйтесь Microsoft Word Viewer. Инструкции по установке и использованию этого приложения находятся в файле Readme.txt.

Шаблоны Word

Кроме документов Word компакт-диск содержит также шаблоны Word, куда включены все формы. Шаблоны Word находятся в каталоге Forms компакт-диска. Для установки шаблонов на вашу систему следуйте инструкциям в файле Readme.txt.

Скопировав шаблоны Word в системный каталог Templates своего компьютера, вы сможете создавать на их основе новые документы. Для этого выберите в меню File команду New, а затем — нужный шаблон. Вы можете изменять шаблоны, например, добавив к ним логотип или модифицировав стили.

База данных, используемая в качестве примера

Каталоги Access 97 Database и Access 2000 Database содержат базу данных, поставляемую вместе с версиями Microsoft Access 97 и Microsoft Access 2000 и отформатированную соответствующим образом Эта база данных использовались в качестве примера в процессе моделирования.

Структура системы, использующей прилагаемую базу данных, построена в соответствии с принципами разработки, изложенными в этой книге. Хотя система вполне работоспособна, советую рассматривать ее не как полнофункциональную систему, а как основу для собственных разработок нуждающуюся в адаптации к процессам и бизнес-условиям вашей организации.

Кроме мелких корректив (например, добавления логотипа компании в отчеты) вы, возможно, захотите внести в базу и более серьезные изменения.

Функциональность системы

Сама по себе система не включает разработку схемы базы данных. Между тем, соответствующие средства вам, скорее всего, понадобятся, если только вы не используете те, что предоставляет механизм СУБД. Решив добавить схему в разрабатываемую базу данных, вы можете автоматически сгенерировать физическую базу данных. Либо создайте базу данных при помощи сценариев SQL или фрагментов кода, выполняемых системой.

Система также не включает средства для выполнения стоимостного анализа.

Если вы работаете над созданием сложных систем совместно с большой группой разработчиков, вам придется предусмотреть средство для контроля версий. Для этого нужно либо расширить функциональность системы, либо интегрировать систему с системой контроля версий. Если пользователи будут работать с базой данных по сети, стоит разделить пользовательские компоненты и компоненты механизма базы данных. Это несложно сделать при помощи Access Database Splitter Wizard.

В настоящее время в системе представлены только ограничения на уровне атрибутов, определенные для каждого атрибута. Поскольку атрибуты наследуют ограничения доменов, на которых они определены, вы можете включить в форму или отчет функцию явного отображения этих ограничений на уровне домена.

Модель данных

Система не включает возможность регистрировать информацию о заказчиках, для которых вы ведете разработку, Вряд ли стоит (да и вряд ли возможно) преобразовать ее в систему регистрации и учета покупателей, но вы, вероятно, все-таки захотите регистрировать хотя бы основные сведения о них — название компании или имя частного лица, а также адрес и контактный телефон.

Во всех таблицах в качестве первичного ключа используются поля *AutoNumber*. Это позволяет использовать повторяющиеся имена сущностей. Разумеется, здесь все зависит от контекста реальной системы, которую вы разрабатываете, а также от вашей личной позиции.

Домены определяются в масштабах всей системы. Кроме того, вы можете добавить для сущности, моделирующей домены, атрибут *ProjectID*, позволяющий идентифицировать домены для отдельных проектов. Обратите внимание на одну особенность пользовательского интерфейса. Не подумайте, что при щелчке кнопки Reference Tables (Справочные таблицы) в форме Projects (Проекты) будут отображены все таблицы, относящиеся к одному проекту. **На** самом деле при щелчке этой кнопки отображаются все справочные таблицы, имеющиеся в системе.

Система позволяет при определении доменов использовать только логические типы данных. Вы можете изменить это правило, позволив использовать другие домены при определении новых доменов.

Дополнительные замечания

База данных, приведенная на компакт-диске, создавалась с учетом последующих значительных изменений. С целью сделать ее структуру понятной без многостраничной документации были предприняты

все усилия, чтобы упростить ее (учитывая сложность модели данных). Разумеется, эту базу данных нельзя считать образцом прекрасного стиля программирования.

Требования к системе

Для работы с базой данных на вашем компьютере должен быть установлен Access 97 или Access 2000.

Чтобы работать с формами в Word в электронном виде, установите также Word 97 или Word 2000. Или воспользуйтесь для просмотра и печати форм Microsoft Word Viewer.

Техническая поддержка

Мы приложили максимум усилий, чтобы обеспечить точность и достоверность материалов этой книге и на прилагаемом к ней компакт-диске. Все исправления, дополнения и комментарии к книгам, выпускаемым издательством Microsoft Press, вы можете найти в Интернете по адресу:

http://mspress. microsofl.com/supporS

Все замечания, вопросы и предложения, касающиеся этой книги и прилагаемого к ней компакт-диска, вы можете отправить по следующим адресам.

Почтовый адрес «Microsoft Press»:

Attn: Designing Relational Database Systems Editor

One Microsoft Way

Redmond, WA 98052-6399

Адрес электронной почты: msinput@microsoft.com

Обратите особое внимание на то, что Microsoft не предоставляет технической поддержки по своим продуктам по данному адресу. Компакт-диск, прилагаемый к книге, содержит базу данных в форматах Microsoft Access 97 и Microsoft Access 2000. Техническая поддержка Microsoft Access доступна по следующему адресу: *http://support.microsoft. com/support/.*

Теория реляционных баз данных

Основные понятия

Итак, что же такое реляционная база данных? Говоря коротко, это средство для рационального и эффективного хранения информации. Иными словами, такая база обеспечивает надежную защиту данных от случайной потери или порчи, экономно использует ресурсы (как людские. так и технические) и снабжена механизмами поиска информации, удовлетворяющимиразумнымтребованиямкпроизводительности,

В теории реляционную базу данных можно создать, не прибегая к помощи специальных инструментов. На практике при разработке реляционных баз данных используют средства систем управления базами данных (СУБД). СУБД иногда называют реляционными СУБД (РСУБД), однако в действительности СУБД должна удовлетворять более чем 300 требованиям, чтобы оправдать это название, и насколько мне известно, практически ни об одной коммерческой СУБД этого сказать нельзя. В этой книге будут рассматриваться две системы управления базами данных: Microsoft Access и Microsoft SQL Server.

Реляционная база данных — это реализация реляционной модели (модели данных) на физическом уровне, и потому важно четко различать эти два понятия: модель данных и базу данных. Как правило, на стадии проектирования невозможно полностью изолироваться от ограничений, налагаемых средой разработки, в то время как в основу проекта рекомендуется закладывать максимально «чистую» модель. Хотя ради увеличения производительности порой стоит пойти на некоторые компромиссы, вы просто обязаны не делать этого при создании модели данных. Приведу конкретный пример. Хранение вычисляемых полей (например, *OrderTotal*) в базовой таблице, строжайше запрещено в теории реляционного проектирования, но часто применяется на практике. Однако независимо от конкретной реализации, создаваемая вами модель не должна содержать вычисляемых полей.

Что такое база данных

Терминология, используемая в области баз данных, включает множество нюансов, столь же тонких, как например, употребление термина «объектно-ориентированное программирование». Само понятие «база данных» может обозначать как отдельный набор данных (например, список телефонов), так и гораздо более сложную систему (например, SQL Server).

Можно привести еще множество примеров, не столь простых, как адресная книга, и не столь сложных, как SQL Server, и тем не менее, объединенных одним общим названием «база данных». Такая нечеткость определений отнюдь не является недостатком — это просто свойство языка.

Попытаемся внести ясность в этот вопрос — на рис. 1-1 показана взаимосвязь между терминами, которые будут обсуждаться далее.

Хотя для реляционных баз данных нет прямых аналогий в реальном мире, большинство их предназначено для моделирования некоторых аспектов реальности. Именно этот «кусочек» реального мира, другими словами, аспект реальности, мы будем называть *предметной областью.*

Предметная область имеет сложную структуру и неупорядочена — и это естественно, ведь если бы она была простой и упорядоченной, нам не понадобилась бы ее реляционная модель. **Но** для успешной реализации проекта необходимо ограничить проектируемую систему определенными рамками, в которые будет входить отдельная, четко определенная совокупность объектов и связей между ними. Только после этого вы сможете правильно оценить масштабы проектируемой системы.

Под термином *модель данных* договоримся понимать концептуальное описание предметной области. Она включает определения сущностей и их атрибутов: например, сущность *Customer* (Покупатель) может иметь атрибуты *Name* (Имя) и *Address* (Адрес). Сюда входят также определяемые для сущностей ограничения: например, *CustomerName* не может допускать «пустых» значений.

Кроме того, модель данных включает в себя описание взаимоотношений между сущностями и ограничения, определенные для этих взаимоотношений: например, ограничение, декларирующее, что для каждого менеджера число отчитывающихся перед ним сотрудников не должно быть более пяти. Модель данных не содержит ссылок и указаний на физическую модель самой системы.

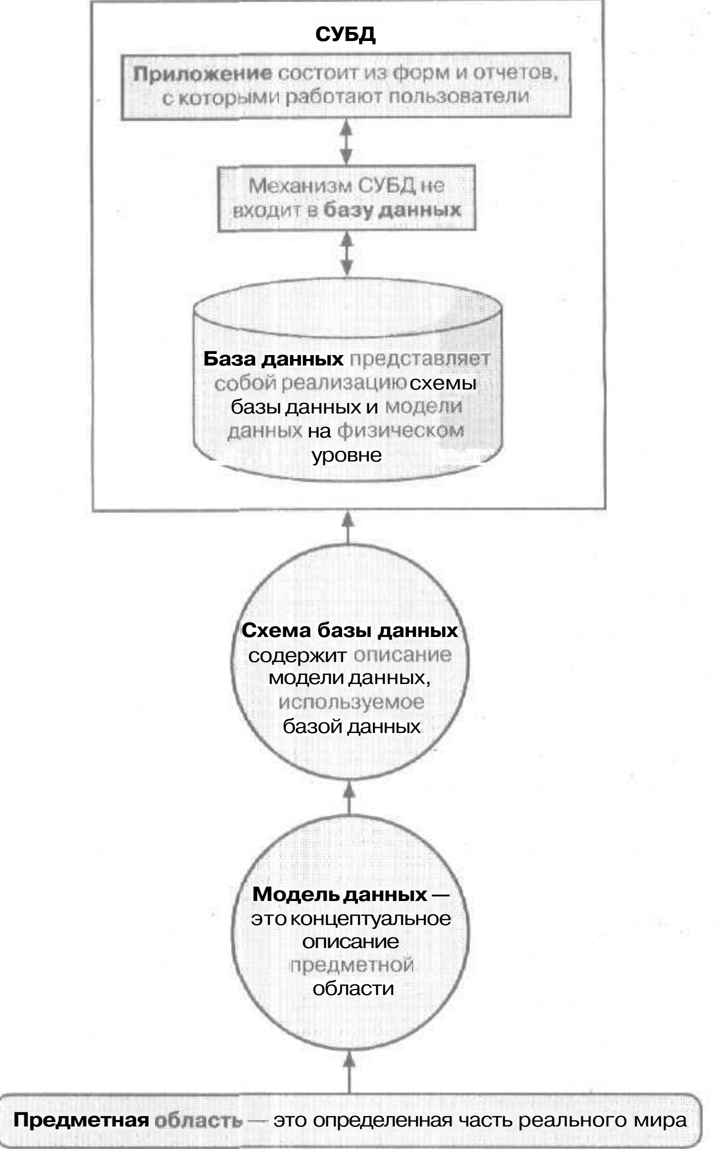


Рис. 1-1. Термины, используемые в ***области реляционных баз данных***

Определение физической модели — создаваемых таблиц и представлений, называется *схемой базы данных* или просто *схемой.* Схема — это перевод концептуальной модели в физическое представление, осуществляемое, как правило, средствами системы управления базами данных. Схема — это понятие, относящееся к концептуальному, а не к физическому уровню. Это все та же модель данных, описываемая в терминах, используемых механизмом СУБД (database engine) — таблицы, триггеры и т. п. Механизм СУБД хорош и тем, что при его использовании не приходится иметь дело с физической реализацией модели; до известного предела вы можете игнорировать такие сущности, как би-деревья и листовые узлы.

Когда вы при помощи программного кода или интерактивной среды, например Microsoft Access, «объясните» механизму СУБД, что же будут представлять собой ваши данные, он создаст физические объекты, в которых эти данные будут храниться. Как правило, такие объекты размещаются на жестком диске, впрочем, это совсем не обязательно. Структура и данные вместе составляют то, что я обычно называю *базой данных.* База данных содержит физические таблицы, представления, запросы, хранимые процедуры, а также правила, используемые механизмом СУБД для зашиты данных.

В понятие «база данных» не входят *приложение,* состоящее, как правило, из форм и отчетов, с которыми работают пользователи, а также средства, обеспечивающие связь между серверной и клиентской частями клиент-серверных приложений (например, связующее программное обеспечение или Microsoft Transaction Server). Кроме того, в базу данных не входит механизм СУБД. Например, файл Access с расширением .mdb — это база данных, а Microsoft Jet — механизм СУБД. На самом деле файл .mdb, помимо базы данных, может содержать множество других объектов (форм, отчетов и т. д.), однако сейчас мы не будем останавливаться на этом вопросе, отложив его для дальнейшего обсуждения.

Для описания всех этих элементов: приложения, базы данных, механизма базы данных, а также связующего программного обеспечения, мы будем использовать термин *система баз данных.* Все программное обеспечение и данные, составляющие реальную эксплуатируемую систему, входят в состав системы баз данных.

Инструменты для работы с базами данных

Эта книга посвящена, главным образом, вопросам разработки, а не реализации баз данных; однако от теории мало толку, если вы не знаете, как ее применять. Поэтому мы уделим внимание построению

реляционных баз данных при помощи средств, предоставляемых корпорацией Microsoft. Этих средств уже немало, и с каждым годом становится все больше. Поэтому давайте остановимся и более внимательно рассмотрим сами средства и их взаимодействие друг с другом (рис. 1-2). Проще всего воспринимать их как инструменты, которыми мы, разработчики, пользуемся, чтобы превратить абстрактную модель в реальную эксплуатируемую систему. На рисунке инструменты для работы с базами данных сгруппированы именно по этому принципу.

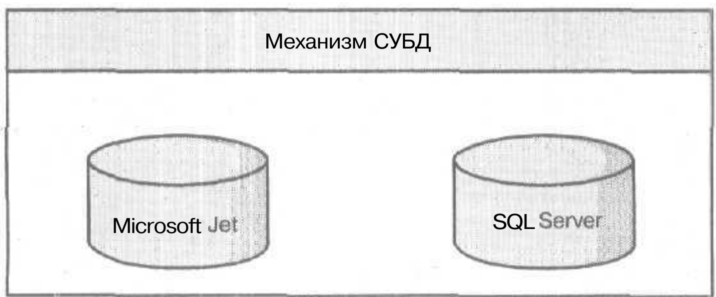


Рис. 1-2. Инструменты для работы с базами данных

Механизм СУБД

На самом нижнем уровне находится механизм СУБД. Его часто называют «серверной частью», однако это неточно, поскольку данный термин подразумевает существование определенной физической архитектуры, о которой подробнее будет рассказываться в главе 10. Механизм базы данных — это специальные средства, предназначенные для физического манипулирования данными: хранением их на

диске и извлечением по запросу. Механизмов СУБД множество, но мы подробно рассмотрим только два из них — Microsoft Jet и SQL Server. Вы удивились, не увидев в этом списке Microsoft Access? Однако здесь нет никакой ошибки. Access использует механизм баз данных Microsoft Jet для манипулирования данными, хранимыми в файлах .mdb, а также может подключаться к любому источнику данных ODBC и манипулировать данными, хранимыми в таком источнике данных, в том числе и в SQL Server. Механизм Microsoft Jet всегда использовался Access, хотя Microsoft и не выделяла механизм баз данных как отдельную сущность до появления Microsoft Visual Basic 3. После того как Access 97 стал поддерживать ODBCDirect, а Access 2000 — SQL Server, Microsoft разделила клиентскую часть Access и механизм баз данных Microsoft Jet. Я полагаю, что в будущих версиях эта тенденция сохранится; однако это лишь мое личное мнение.

Механизмы Microsoft Jet и SQL Server существенно различаются по внутренней архитектуре и назначению, но оба они — замечательные инструменты для хранения данных и манипулирования ими. Microsoft Jet — это «настольный» сервер баз данных, ориентированный на малые и средние системы. SQL Server использует клиент-серверную архитектуру и предназначен для создания систем, от средних до больших. Он прекрасно масштабируется и может поддерживать несколько тысяч пользователей, работающих с важными приложениями (Microsoft Jet пригоден только для создания самых простейших систем.) На протяжении всей книги я буду обращать ваше внимание на различия между двумя этими механизмами. Подробный сравнительный анализ их архитектур — в главе 10.

Объектная модель доступа к данным

И Access, и Visual Basic предоставляют простые и удобные инструментальные средства для непосредственной связи между элементами управления и источником данных. Эти средства позволяют избежать прямого взаимодействия с механизмом баз данных. Однако по различным причинам, которые мы рассмотрим далее, такой способ не всегда реализуем на практике. Порой более эффективно использовать объектные модели доступа к данным для манипулирования данными непосредственно в коде.

Объектная модель доступа к данным представляет собой своего рода «промежуточный слой» между средой программирования и механизмом СУБД. Она содержит набор объектов, обладающих свойствами и методами, которыми можно манипулировать в коде. Сейчас корпорация Microsoft разработала и предлагает три объектных модели доступа к данным:

- **Data** Access **Objects (DAO)** — имеет две разновидности: DAO/Jet и DAO/ODBCDirect;
- **Remote Data Objects (RDO)** — используется в основном для доступа к источникам данных ODBC;
- **Microsoft ActiveX Data Objects (ADO)** — в ближайшем будущем полностью заменит DAO и RDO.

DAO — старейшая из трех перечисленных выше моделей, это собственный интерфейс механизма СУБД Microsoft Jet. RDO похож на DAO, но оптимизирован для доступа к источникам данных ODBC, например SQL Server и Oracle. По сравнению с первыми двумя моделями, ADO использует меньшую иерархию объектов, состоящую всего лишь из четырех основных уровней, и предоставляет некоторые существенные расширения — например, поддержку разобщенных наборов данных (disconnected recordsets) и группы объектов и методов, позволяющие создавать иерархический набор данных.

Поскольку эта книга посвящена разработке, а не внедрению баз данных, мы не будем подробно рассматривать преимущества каждой из этих моделей. Всем, кого интересует данный вопрос, я могу порекомендовать статью Уильяма Вогна (William Vaughn) в «Hitchhiker's Guide to Visual Basic and SQL Server», а также ряд других статей и документов, опубликованных на Web-узле Microsoft. Объектные модели не единственный способ доступа к данным, существует множество альтернативных средств, например Visual Basic Library для SQL Server (VBSQL) и OLE DB.

Средства для разработки клиентской части приложений

Microsoft Jet и SQL Server берут на себя манипулирование данными на физическом уровне, однако необходимо дать им четкие указания, каким образом эти данные должны быть структурированы. Microsoft предоставляет богатейший арсенал средств решения этой задачи, но мы подробно рассмотрим только два: Access и Microsoft Visual Database Tools. Лично я предпочитаю именно их, тем более что остальные методы предоставляют приблизительно те же возможности. Поняв основные механизмы действия Access и Microsoft Visual Database Tools, вы можете выбрать для себя те средства, которые сочтете наиболее удобными для решения конкретной задачи.

При создании структуры базы данных можно также прибегнуть и к непосредственному кодированию, однако я не рекомендую этот метод. Исключение составляют случаи, когда нужно изменить структуру данных уже после того, как приложение начало активно эксплуатироваться. Еще одним, достаточно тривиальным, исключением

могут быть временные таблицы. В большинстве же случаев следует использовать интерактивные средства — они гораздо удобнее, и к тому же экономят время.

После того как проектирование базы данных на физическом уровне будет завершено, вам потребуются инструменты для создания форм и отчетов, с которыми будут работать пользователи. Мы рассмотрим два таких средства: Access и Visual Basic. В главе 10 мы коснемся также средств просмотра Web, однако непосредственно о языке HTML в этой книге говорить не будем.

Реляционная модель

Реляционная модель основывается на математических принципах, вытекающих непосредственно из теории множеств и логики предикатов. Эти принципы впервые были применены в области моделирования данных в конце 60-х гг. доктором Е.Ф. Коддом, в то время работавшим в IBM, а впервые опубликованы — в 1970 г.[1] Реляционная модель определяет способ представления данных (структуру данных), методы зашиты данных (целостность данных), а также операции, выполняемые с данными (манипулирование данными).

Реляционная модель не единственный метод хранения и манипулирования данными. Существуют альтернативные варианты: иерархическая, сетевая, а также звездообразная модели данных. У каждой из них свои преимущества при решении задач определенного типа. Например, применение реляционной модели в обработке данных с иерархической организацией недостаточно хорошо изучено, для решения подобных задач используют специально созданную звездообразную модель данных. Однако гибкость и эффективность реляционной модели делают ее наиболее популярным инструментом для разработки баз данных. В этой книге мы будем рассматривать только реляционную модель, на которой основаны механизмы СУБД Microsoft Jet и Microsoft SQL Server.

В общих чертах основные принципы реляционных систем баз данных можно сформулировать так,

- Все данные на концептуальном уровне представляются в виде упорядоченной организации, определенной в виде строк и столбцов и называемой *отношением.*
- Все значения являются *скалярами.* Это означает, что для любой строки и столбца любого отношения существует одно и только одно значение.

[1] *E. F. Codd.* A Relational Model of Data for Large Shared Data Banks. Communications of the ACM. Vol. 13. No. 6 (June. 1970).

- Все операции выполняются над целым отношением, и результатом выполнения этих операций также является целое отношение. Этот принцип называется *замыканием.*

Если у вас имеется опыт работы с базами данных Microsoft Access, вы, конечно, догадались, что в данном случае представляет собой отношение — это *набор записей* или, в терминах SQL Server, *набор результатов.* Формулируя принципы реляционной модели, доктор Кодд выбрал термин «отношение» (relation), потому что он однозначен (в то время как, например, термин «таблица» имеет множество дополнительных значений). Весьма распространено следующее заблуждение: реляционная модель названа так потому, что она определяет отношения между таблицами. На самом деле название этой модели происходит от отношений, лежащих в ее основе.

В рамках реляционной модели данные представлены в виде отношения на концептуальном уровне; однако при этом совсем не дается никаких указаний, каким образом данные будут реализованы на физическом уровне. Разделение концептуального и логического уровней давно стало привычным для нас; но всего лишь 30 лет назад этот метод произвел настоящий переворот в области программирования баз данных. Ранее программирование баз данных сводилось в основном к написанию программного кода для физического управления устройствами, предназначенными для хранения данных.

В действительности отношения не нуждаются в физическом представлении. Некий набор записей может соответствовать некоей физической таблице, размещенной на диске, а может быть и сформирован из столбцов нескольких десятков таблиц с вычисляемыми полями, значения которых вообще нигде не хранятся. Такой набор записей является отношением, поскольку организован в виде строк и столбцов, и его значения — скаляры. Его существование абсолютно никак не зависит от физической реализации.

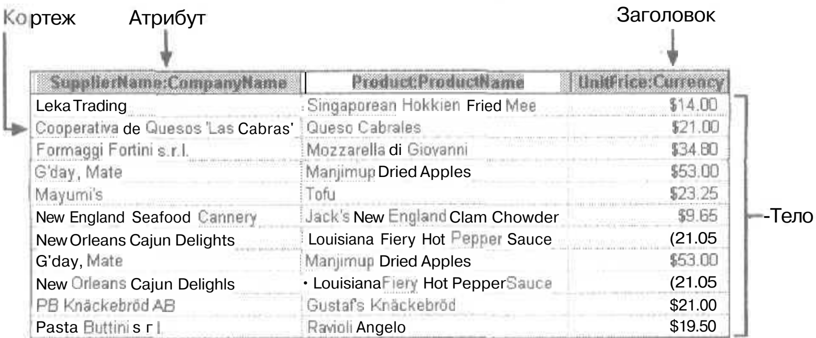
Принцип замыкания заключается в том, что и базовые таблицы, и результаты операций над ними на концептуальном уровне представляются как отношения. Он позволяет непосредственно использовать результаты одной операции в качестве исходных данных для выполнения другой. Таким образом, и Microsoft Jet, и SQL Server дают возможность использовать результаты одного запроса для составления нового. Этот принцип реализует в области разработки баз данных функциональность, аналогичную подпрограммам в процедурном программировании — возможность инкапсуляции сложных или часто повторяющихся операций для повторного использования.

Предположим, вы составили запрос с именем *FullNameQuery*, выполняющий операцию конкатенации над данными и помещающий имя и фамилию физического лица в вычисляемое поле *FullName*. Вы можете составить следующий запрос, в котором *FullNameQuery* будет выступать в качестве источника. Вычисляемое поле *FullName* используется в этом случае точно так же, как любое другое поле базовой таблицы. Заново выполнять вычисления, в результате которых будет получено имя физического лица, не нужно.

Требование, чтобы все значения в отношении являлись скалярами, может иногда создавать дополнительные трудности и соблюдается не абсолютно строго. Принцип существования одного и только одного значения для любой строки и любого столбца субъективен и зависит от семантики модели данных. Например, имя и фамилия физического лица в одной модели могут быть представлены как одно значение, а в других моделях — разбиты на несколько отдельных значений (например, имя и фамилию или обращение, имя, отчество и фамилию). С точки зрения абстрактной теории ни один из этих вариантов не является более правильным, чем остальные; представление данных зависит от выбранного варианта реализации системы.

Термины, используемые в реляционной теории

На рис. 1-3 приведен пример отношения и наглядно представлены формальные названия его основных выделенных компонентов. Читатели, знакомые с основами реляционной теории, могут заметить, что данное отношение не приведено к нормальной форме. Тем не менее, оно все равно остается отношением, поскольку данные представлены в виде строк и столбцов и все значения являются скалярами.



SupplierName:CompanyName	Product:ProductName	UnitPrice:Currency
Leka Trading	Singaporean Hokkien Fried Mee	$14.00
Cooperativa de Quesos 'Las Cabras'	Queso Cabrales	$21.00
Formaggi Fortini s.r.l.	Mozzarella di Giovanni	$34.80
G'day, Mate	Manjimup Dried Apples	$53.00
Mayumi's	Tofu	$23.25
New England Seafood Cannery	Jack's New England Clam Chowder	$9.65
New Orleans Cajun Delights	Louisiana Fiery Hot Pepper Sauce	(21.05
G'day, Mate	Manjimup Dried Apples	$53.00
New Orleans Cajun Delighls	• Louisiana Fiery Hot Pepper Sauce	(21.05
PB Knäckebröd AB	Gustaf's Knäckebröd	$21.00
Pasta Buttini s.r.l.	Ravioli Angelo	$19.50

Рис. 1-3. Компоненты отношения

Как мы условились ранее, *отношением* называется вся структура в целом. Каждая строка, содержащая данные, является *кортежем.* Строго говоря, каждая строка является *n*-кортежем, однако «n-», как правило, опускается. Число кортежей в отношении определяет *мощность* отношения- В приведенном на рис. 1-3 примере мощность отношения равна П. Каждый столбец отношения называется *атрибутом.* Число атрибутов в отношении определяет *размерность* этого отношения, для проиллюстрированного примера она равняется трем.

Каждое отношение можно разделить на две части — *заголовок* и *тело.* Тело отношения состоит из кортежей, в то время как заголовок не имеет более мелких компонентов структуры. Обратите внимание — название каждого из атрибутов состоит из двух терминов, разделенных двоеточием (например, *UnitPrice:Currency*).Первая часть названия — непосредственно имя атрибута, вторая — имя домена. *Домен* атрибута — это «вид» данных, которые представляет данный атрибут (в приведенном примере — валюта). Понятие «домен» не эквивалентно понятию «тип данных». Различие между этими двумя понятиями будет подробно обсуждаться далее в этой главе. На практике домен в заголовках часто не указывается.

Тело отношения состоит из неупорядоченного набора кортежей (число кортежей может быть любым, от 0 и более). Остановимся на некоторых важных моментах. Во-первых, отношение не упорядочено. Понятие «номер строки» не применимо к отношению. Для отношений не существует никакого внутреннего порядка. Во-вторых, отношение может иметь нулевое число кортежей (это так называемое пустое отношение, которое, тем не менее, является отношением). В третьих, отношение представляет собой набор. Элементы в этом наборе по определению уникально идентифицируемые. Поэтому чтобы таблица являлась отношением, каждая ее строка должна быть уникально идентифицируемой, записи в ней не должны повторяться.

Читателей, знакомых с документацией к Access или *SQL* Server, возможно, удивит, что они не встречали там терминов, используемых в этой книге. Однако следует заметить, что здесь я использую терминологию, общепринятую в технической литературе; терминология, используемая Microsoft, несколько отличается от нее. (Я специально ввела эти термины в книгу, чтобы читатели не смущались, услышав где-нибудь, например, об n-кортежах и третьей размерности). Ни в коем случае не следует забывать, что отношения определяются исключительно на концептуальном уровне. Как только речь заходит о конкретных примерах из области баз данных, отношения становятся *наборами записей* (для Microsoft Jet) или *наборами результатов* (для

SQL Server). И для Microsoft Jet, и для SQL Server атрибут «превращается» в *поле, а* кортеж, соответственно — в *запись.* Эти соотношения практически взаимнооднозначны; однако нужно помнить, что отношения существуют на концептуальном уровне, в то время как наборы записей и наборы результатов — на физическом.

Модель данных

Модель данных, то есть концептуальное описание предметной области — самый абстрактный уровень проектирования баз данных. Элементами описания модели данных являются сущности, атрибуты, домены и отношения, Рассмотрим подробно каждый из них.

Сущности

Весьма непросто дать точное формализованное описание сущности, однако основная идея достаточно прозрачна: *сущность* — это нечто такое, о чем нужно хранить информацию в разрабатываемой системе.

Составить первоначальный список сущностей не составляет труда. Когда вы ведете деловую беседу с заказчиком, большинство существительных и часть глаголов, используемых в разговоре, и есть кандидаты на эту роль.

Приведем простейший пример: «Покупатели покупают товары. Сотрудники продают товары покупателям. Поставщики поставляют товары». Существительные «покупатели», «товары», «сотрудники» и «поставщики», вне всякого сомнения, будут являться сущностями. События, описываемые глаголами «покупать» и «продавать», также являются сущностями; однако здесь есть несколько тонкостей, незаметных на первый взгляд. Так, глагол «продавать» описывает два разных события: продажу товара покупателю и приобретение товаров компанией у поставщика. В данном случае различие между этими двумя процессами очевидно, однако иной раз заметить подобные нюансы достаточно трудно, особенно если вы не очень хорошо знаете предмет беседы.

Другая коварная ловушка полностью противоположна первой: два разных глагола («покупать» в первом предложении и «продавать» во втором) используются для описания одного и того же события, а именно приобретения товара покупателем. Это далеко не очевидный факт, и он чаще ускользает от внимания проектировщиков, чем первый. Если заказчик использует два разных глагола для описания, как вам кажется, одного и того же события, не торопитесь с выводами — на самом деле речь может идти о двух совершенно разных вещах. Например, если заказчик — ателье, то фразы «мистер N покупает костюм» и «мистер N заказывает костюм» на первый взгляд кажутся

двумя вариантами описания одного и того же события, результатом которого является приобретение мистером N костюма. Однако в первом случае (покупка мистером N костюма у ателье) — это продажа уже готового костюма, а но втором (заказ костюма) — индивидуальный пошив. Это разные процессы, которые нельзя смешивать при создании модели.

Как правило, чтобы составить список сущностей, одной беседы с заказчиком недостаточно. Нужно просмотреть как можно больше документов, имеющих отношение к предметной области. Заполняемые бланки, отчеты, инструкции для персонала — это настоящая сокровищница, из которой и следует извлекать «кандидаты» в сущности. Причем анализировать документы следует предельно внимательно. Как правило, печатные документы достаточно инертны и могут не отражать самые последние изменения в инструкциях и иных внутренних правилах компании-заказчика. Будьте внимательны, если, например, столкнулись с сущностью, о которой заказчик в беседе ни разу не упомянул. Не следует сразу же предполагать, что он просто что-то забыл. Возможно, это просто «наследие прошлого», уже не имеющее никакого отношения к текущему состоянию дел в компании. Вам обязательно следует это проверить.

После того как составлен первоначальный вариант списка сущностей, следует обязательно проверить его на полноту и связность. Кроме того, нужно выявить «дубли», то есть повторяющиеся сущности, и сущности, на самом деле разные, но ошибочно представленные в списке как одна, Мощный инструмент для такого анализа — концепция подтипов сущностей. Вернемся к примеру с ателье: покупка и заказ одежды представляют одно событие — приобретение мистером N одежды, однако это разные *типы* приобретения. Другими словами, *продажа готовой одежды* и *заказ одежды* будут являться подтипами сущности *приобретение одежды.*

Атрибуты, общие для обоих типов сущности *приобретение одежды,* связываются с надтипом (в данном случае таким подтипом является *приобретение*). Атрибуты, присущие только одному конкретному подтипу (в нашем примере это *продажа готовой одежды* и *заказ одежды)* специально вводятся только для этого подтипа. Такой подход позволяет интерпретировать оба типа событий как *приобретение одежды* в одном случае (например, при подсчете общей прибыли от продаж) или как отдельные типы приобретения (например, при сравнении числа и объемов продаж готовых изделий и изделий, выполняемых на заказ).

Может оказаться, что некоторые подтипы сущностей обладают одинаковым набором атрибутов. Тогда лучше определить *TypeOfSale* (вид продажи), *TypeOfCustomer* (тип клиента) и т. п. как атрибуты надтипа, а не моделировать подтипы как отдельные сущности. В примере с ателье для моделирования пошива одежды может оказаться необходимой информация о виде одежды и цвете, выбранном клиентом, а для моделирования продажи готовых изделий — наименование фирмы — производителя одежды. В таком случае следует использовать подтипы для моделирования этих сущностей. Однако если единственное, что вам нужно учитывать: была ли приобретена готовая одежда или сшитая на заказ, лучше ввести атрибут *TypeOfSale*.

Подтипы одного типа являются взаимоисключающими, но далеко не всегда. Рассмотрим, например, базу данных, в которую заносятся сведения о сотрудниках некой фирмы. Для всех сотрудников существуют некоторые общие атрибуты: дата поступления на работу, отдел, в котором работает сотрудник, номер внутреннего телефона. Но лишь часть сотрудников — торговые агенты, и у них свои специфические атрибуты, например размер комиссионного вознаграждения и план продаж. И лишь некоторые сотрудники будут членами баскетбольной команды этой фирмы. Разумеется, торговому агенту, как и любому другому сотруднику, ничто не мешает играть в баскетбол.

Большинство сущностей моделируют объекты или события реального мира, примерами могут служить клиенты, товары, или звонки в службу продаж. Это *конкретные сущности.*

Сущности также могут моделировать и абстрактные понятия. Здесь наиболее яркий пример — сущность, моделирующая отношения между сущностями: скажем, тот факт, что некий торговый атент отвечает за определенного клиента, или что некий студент записан на определенный курс лекций.

Иногда необходимо моделировать только сам факт наличия отношения, иногда — хранить дополнительную информацию об этом отношении (дата возникновения данного отношения, а также его дополнительные характеристики). К примеру, если вы планируете создать заповедник, нужно знать, что между *гиенами* и *шакалами* существует отношение *конкуренция, в* то время как между *гиенами* и *антилопами* — отношение *хищничество* (гиены охотятся на антилоп).

Вопрос, нужноли моделировать отношения, у которых отсутствуют атрибуты, как отдельные сущности, не столь тривиален. Лично я считаю, что, представляя подобные отношения как отдельные сущности, вы ничего не выигрываете, в то время как построение схемы базы данных на основе модели данных существенно усложняется. И

все же не следует забывать, что отношения играют в модели данных ничуть не меньшую роль, чем сущности.

Атрибуты

В разрабатываемой системе будут храниться записи об определенных параметрах каждой из сущностей. Эти параметры называются атрибутами сущностей. Например, если в вашей системе присутствует такая сущность, как *Customer* (Покупатель), вам, скорее всего, потребуется хранить имена и фамилии, и возможно, род деятельности клиентов. При моделировании такого события, как звонок в службу технической поддержки потребуется знать, кто звонил, время звонка, и удалось ли успешно разрешить проблему, с которой обратился клиент.

Определение атрибутов, которые нужно включить в разрабагываемую модель — это семантический процесс. Решая эту задачу, нужно основываться на том, что реально означают хранимые данные и как они будут использоваться. Возьмем простейший пример — адрес. Определите ли вы адрес как одну сущность *(Address)* или как несколько сущностей (*HouseNumber* - номер дома, *Street* - улица, *City* - город, *ZipCode* - почтовый индекс)? Большинство разработчиков баз данных (в том числе и я) автоматически разобьют адрес на несколько атрибутов — ведь структурированными данными обычно легче манипулировать. Но порой это не так, а значит, данное правило отнюдь нельзя считать непреложным.

Предположим, мы создаем базу данных адресов для местного клуба любителей рок-музыки. В ней должны храниться адреса членов клуба, чтобы их можно было распечатать на конвертах или наклейках при отправке корреспонденции. Поскольку все члены клуба живут в одном городе, имеет смысл хранить их адреса в формате больших двоичных объектов, представляющих собой отдельные фрагменты текста, которые могут содержать несколько строк. Эти несколько строк выводятся целиком в ответ на запрос.

Ну а если клиент - американская компания, предлагающая покупателям свои товары через Интернет? Чтобы определить размер налога с продаж, нужно знать, в каком штате живет покупатель, заказавший товар. Если использовать тот же подход, что и в случае с базой данных для местного клуба любителей рок-музыки, то конечно, мы столкнемся с непростой задачей: как извлечь информацию о штате из единого фрагмента текста. Поэтому вполне естественно моделировать одну из составных частей адреса (штат) как отдельную сущность. Но следует ли дробить оставшуюся часть адреса на более мелкие фрагменты, и если да, то на какие? В Соединенных Штатах код штата имеет четко определенный формат, и моделировать этот код как отдельную

сущность не составляет труда. Но моделирование составных частей адреса клиента может оказаться не таким уж простым делом.

На первый взгляд кажется, что простого набора атрибутов: *House-Number* (номер дома), *Street* (название улицы), *City* (город), *State* (штат), *ZipCode* (почтовый индекс), — вполне достаточно. Но ведь существуют еще номера квартир многоэтажных жилых домов и номера почтовых ящиков, на которые приходит корреспонденция для частных лиц или небольших фирм. А как учесть возможность заявки на приобретение товара на чужое имя? И конечно же, нужно подумать о Потенциальных возможностях расширения бизнеса, Что если клиентом компании станет человек, проживающий за пределами США, или иностранная компания? В последнем случае недостаточно знать только название страны и почтовый код клиента — форматы почтового кода в США и в других странах могут существенно различаться. Не исключено, что потребуется существенно изменить уже имеющиеся сущности. Например, в большинстве стран Европы при написании адреса номер дома указывается после названия улицы. Подобную проблему легко решить при вводе данных, но есть головоломки и посложнее. Например, многие ли пользователи вашей системы знают, что в австралийском адресе «4/32 Griffen Avenue, Bondi Beach, Australia» цифры 4/32 означают номер квартиры (4) и номер дома (32)?

Все эти примеры я привела здесь не затем, чтобы показать, насколько сложно смоделировать обычный почтовый адрес, а для наглядной иллюстрации следующего тезиса: невозможно сделать общие предположения о том, как следует моделировать конкретный вид данных. Сложная схема, которую разрабатывают для обработки заказов, поступающих от клиентов из разных стран, будет совершенно непригодна для базы данных адресов местного рок-клуба.

Говорят, что Матисс считал картину полностью завершенной, когда в ней ничего нельзя было ни добавить, ни убрать. При моделировании сущностей применяется похожий принцип. Но где следует поставить точку? Когда моделирование сущностей можно признать законченным? К сожалению, достоверных способов узнать это не существует. Современный уровень технологии не позволяет создать модель базы данных, которую можно было бы назвать абсолютно правильной, опираясь на неопровержимые доказательства. В каждом конкретном случае вы можете доказать, что в данной реализации допущены такие-то и такие ошибки, однако невозможно доказать, что ошибки в реализации отсутствуют.

Тем не менее, проектируя базы данных, разработчики пользуются общими стратегическими подходами. Первое правило: начните с ре-

зультата и старайтесь по возможности упрощать модель, а не усложнять ее.

На какие вопросы должна отвечать ваша база данных? В рассмотренном примере с местным рок-клубом единственный вопрос формулировался так: «Какой адрес нужно написать на конверте при отправке корреспонденции члену клуба?», — и для данного случал модель, использующая единственный атрибут, вполне подходила. **Во** втором примере, где разрабатывалась база данных для компании, специализирующейся на услугах «товары — почтой», требовалось ответить еще на один вопрос: «В каком штате живет клиент?», поэтому пришлось реализовать другую структуру.

При проектировании баз данных нужна определенная гибкость реализуемой системы, заложенная в ней возможность не просто давать ответы на вопросы, которые пользователь задает сегодня, но и предвидеть, какую информацию он захочет получить завтра. Вернемся опять к примеру с местным рок-клубом — весьма вероятно, что в течение первого года эксплуатации спроектированной базы данных вас попросят реализовать возможность сортировать адреса по почтовому индексу, чтобы воспользоваться скидками на отправку отсортированной корреспонденции.

Кроме того, следует уделить особое внимание вопросам, которые пользователь мог задать, если бы знал, что это принципиально возможно. Это особенно важно, когда вы проектируете систему, автоматизирующую действия, ранее выполнявшиеся вручную. Попробуйте, например, обратиться к библиотекарю с просьбой дать справку: какое количество книг из 4 млн., находящихся в его ведении, было выпущено в Чикаго до 1900 г. Скорее всего, библиотекарь предложит вам выяснить это самостоятельно, воспользовавшись картотекой. А хорошо спланированная база данных позволит получить ответ за несколько секунд.

Один из признаков профессионализма разработчика — скрупулезный анализ потенциальных вопросов, ответ на которые могут захотеть получить пользователи разрабатываемой системы. От неопытных аналитиков часто слышишь, что пользователи сами не знают, чего хотят. И это вполне естественно! В том и заключается работа аналитика — помочь пользователю понять, чего же он хочет.

Впрочем, здесь вы рискуете угодить в ловушку. Как правило, стремление сделать систему как можно более гибкой неизбежно приводит к ее усложнению — вспомните пример с почтовыми адресами. Разбивая адрес на все более и более мелкие фрагменты, приходится иметь дело со все возрастающим количеством исключений — случа-

ев, не укладывающихся в рамки определяемых форматов. И наконец, может наступить момент, когда все преимущества автоматизации сойдут на нет просто потому, что разрабатываемая система станет чересчур сложной.

Итак, мы вплотную подошли ко второму правилу: следует выявлять исключения, с которыми предстоит иметь дело. Здесь следует обращать внимание на два аспекта: во-первых, при разработке системы важно определить все исключения, и во-вторых, заложить в нее обработку максимально возможного числа исключений (то есть тех, которые вы в состоянии обработать, не запутав при этом пользователя). Давайте посмотрим, как это выглядит на практике.

Если одной из функций разрабатываемой системы будет составление и отправка корреспонденции, следует уделить особое внимание правильности имен адресатов, Например, лично я, получив от незнакомого отправителя письмо, на котором совершенно правильный адрес, но перепутаны фамилия или инициалы, отправлю конверт в мусорную корзину, даже не вскрыв.

На первый взгляд, правильно указать имя не такая уж сложная задача. Ведь в этом случае большинство данных укладываются приблизительно в одну и ту же схему. Например, имя *Ms. Jane Q. Public* состоит из обращения (Ms.), имени (Jane), первой буквы второго имени (Q) и фамилии (Public). Значит, при разработке системы можно ограничиться четырьмя сущностями — *Title* (обращение), *First Name* (имя), *Middle Initial* (первая буква второго имени) и *LastName* (фамилия), не так ли?

Нет, не так. Во-первых, в разных странах существуют разные правила, что следует указывать первым — имя или фамилию. Поэтому лучше использовать сущности *GivenName* (имя, данное при рождении) и *Surname* (фамилия или прозвище). Во-вторых, как в таком случае ввести в базу данных, например, такое громкое имя как сэр Джеймс Педдингтон Смит, лорд Данстэйбл (*Sir James Peddington Smythe, Lord Dunstable*)? Можно ли считать, что Педдингтон Смит *(Peddington Smythe)* — это фамилия лорда? Или, может быть, Педдингтон *(Peddington)* — это второе имя? И что, наконец, делать с этой неудобной последней частью имени: «лорд Данстэйбл» *(Lord Dunstable)*? А что вы скажете о певце по имени Стинг (*Sting*)? Какая из сущностей будет реально представлять собой имя Стинг — *GivenName* (имя, данное при рождении) или *Surname* (фамилия или прозвище)? И наконец, как быть с таким сложным именем, как *The Artist Formerly Known as Prince* (Художник-который-ранее-был-известен-как-Принц)? Как вы собираетесь выходить из такого положения?

Последний вопрос был задан совсем не случайно. Скорее всего, письмо, адресованное сэру Джеймсу Педдингтону Смиту (Sir James Peddington Smythe), не вызовет раздражения у самого обладателя титула и не собьет с толку его слугу или личного секретаря, разбирающего почту. Однако нельзя обратиться к упомянутому джентльмену *«сэр Смит»*; правильное обращение — *«сэр Джеймс»* или *«лорд Данстэйбл»*. Однако давайте выясним у нашего заказчика, много ли у него клиентов, которые носят титулы лордов или леди. Большинство компаний все же не могут похвастаться обилием титулованной клиентуры. Во всяком случае, тот рок-клуб, о котором мы уже говорили, вряд ли поблагодарит вас за систему ввода и поиска информации, если та будет содержать форму, подобную показанной на рис. 1-4.

Рис. 1-4. Чрезмерно усложненная форма ввода адреса.

Итак, приступая к разработке системы, всегда помните, что большая гибкость достигается, как правило, за счет увеличения сложности. Конечно, нужно стремиться выявить и обработать как можно большее количество исключений, Но иной раз стоит остановиться и подумать: а стоит ли вообще заниматься этим исключением? Если его обработка чересчур усложнит систему, или маловероятно, что пользователи когда-либо с этим исключением встретятся — то нет.

Иногда довольно сложно провести различие между сущностями и атрибутами, Возвращаясь к примеру с моделированием адресов, мы видим, что конкретное решение зависит от проблемной области. Некоторые разработчики считают вполне приемлемым, когда в системе,

хранящей адреса клиентов, адрес реализуется как одна сущность. С точки зрения реализации системы, такой подход имеет свои преимущества — он обеспечивает инкапсуляцию и позволяет повторно использовать элементы кода.

И все же преимущества подобного способа весьма сомнительны. Маловероятно, что адреса сотрудников и клиентов будут использоваться одними и теми же пользователями в одних и тех же целях. Вряд ли, например, массовая рассылка сообщений сотрудникам компании будет производиться с помощью обычной почты, а не корпоративная сети. А значит, и моделирование адресов сотрудников и клиентов компании должны принципиально различаться. Сложная форма ввода адреса, показанная на рис. 1-4, вполне оправдана для хранения только адресов клиентов. Однако если как одна сущность будут реализованы все адреса (и сотрудников, и клиентов), вам поневоле придется использовать ту же форму для ввода адресов сотрудников компании, что вряд ли имеет смысл.

Домены

В самом начале этой главы я приводила пример, в котором название каждого из атрибутов в заголовке отношения состояло из двух терминов, разделенных двоеточием — *Имя_атрибута:Имя_домена.*Мы уже говорили, что домен определяет «вид» данных, которые представляет данный атрибут. Если дать более четкое определение, то домен — это набор всех допустимых значений, которые может содержать данный атрибут.

Понятие «домен» часто путают с понятием «тип данных». Необходимо четко различать эти два понятия. Тип данных — это физическая концепция, а домен — логическая. Например, «целое число» — это тип данных, а «возраст» — это домен. Приведу еще один пример. Сущности *StreetName* (название улицы) и *Surname* (фамилия) могут быть реализованы как текстовые поля; однако совершенно очевидно, что это разные виды текстовых полей, и принадлежат они к разным доменам.

Кроме того, понятие домена намного шире понятия «тип данных», поскольку определение домена включает в себя более детальное описание допустимых значений данных. Поясним это различие на примере: возьмем домен *Degree Awarded,* представляющий ученые степени, присваиваемые выпускникам университетов. На схеме базы данных этот атрибут может быть определен как текстовое поле, длиной в три символа (*Text[3]*)И все же это не просто любая комбинация символов, число которых не превосходит трех, а одно из следующих значений; {BA, BS, MA, MS, PhD, LLD, MD}.

Безусловно, не все домены можно определить, просто перечислив допустимые значения. Например, домен *Age* (возраст) содержит около сотни значений, если речь идет о человеке, и несколько тысяч - если мы говорим о музейных экспонатах. В подобных случаях в определении домена, как правило, используются правила, проверяющие принадлежность конкретного значения к области допустимых значений. Например, домен *PersonAge* (возраст физического лица) можно определить как целое число в интервале от 0 до 120, а *ExhibitAge*(возраст выставочного экспоната) — просто как неотрицательное целое число, большее 0.

Значит, домен — это тип данных и логические правила, определенные для данной сущности? Почти правильно. Но необходимо одно уточнение: логические правила — это один из механизмов реализации целостности данных, а отнюдь не элемент их описания. Например, логическое правило, реализующее проверку допустимых значений для почтового индекса, может ссылаться на атрибут *State*, в то время как домен *ZipCode*(почтовый индекс) представляет собой строку, длина которой не превышает шести символов.

Вы, очевидно, уже заметили, что все эти определения ссылаются на вид хранимых данных (число или строка), Это описание очень похоже на тип данных, однако между ними существуют различия. Как я уже упоминала, типы данных — это физические концепции, они определяются и реализуются в терминах механизма СУБД. Если при разработке модели данных вы определите домен как *varchar(30)*(символьную константу переменной длины, с числом символов, не превосходящим 30) или *Long Integer*(длинное целое), это будет ошибкой. поскольку и varchar(30), и Long Integer — это описания, относящиеся к механизму СУБД.

Для любых двух доменов можно сравнивать определенные для них атрибуты, и более того, выполнять логические операции, например, соединения (будут подробно обсуждаться в главе 5). Если над атрибутами двух доменов можно выполнять логические операции, то перед вами домены, имеющие совместимый типы данных (*type-compatible*). Если рассматривать два отношения, представленных на рис. 1-5, то несомненно, имеет смысл связать их по *EmployeeID= SalespersonID* (например, чтобы получить список счетов конкретного сотрудника). Домены *EmployeelD* и *SalespersonID* имеют совместимые типы. Но попытавшись соединить отношения по *EmployeeID= OrderDate*, вы вряд ли получите осмысленный результат, даже если в определения этих двух доменов заложен один и тот же тип данных.

Orders

OrderID	Customer	SalespersonID	OrderDate
10248	Vins et alcools Chevalier	5	04-Aug-94
10249	Toms Spezialitäten	B	05-Aug-94
10250	Hanari Carnes	4	08-Aug-94
10251	Victuailles en stock	3	08-Aug-94
10252	Suprêmes délices	4	09-Aug-94

Employees

EmployeeID	LastName	FirstName	Title
1	Davolio	Nancy	...Sales Representalive
2	Fuller	Andrew	Vice President, Sales
3	Leverling	...Janet	Sales Representative
4	Peacock	Margaret	Sales Representative
5	Buchanan	Steven	Sales Manager
6	Suyama	Michael	Sales Representative

*Рис. 1-5. Отношения **Employees** и **Orders***

К сожалению, механизмы СУБД Microsoft Jet и SQL Server не предоставляют более строгой внутренней поддержки доменов, чем типы данных. Однако даже там, где речь идет о типах данных, ни один из упомянутых механизмов СУБД не обеспечивает строгой проверки. оба они выполняют преобразования данных незаметно для пользователя. Рассмотрим конкретный пример, обратившись к базе данных *Northwind*, поставляемой Microsoft Access в качестве примера. Если вы определили *EmployeeID* в таблице *Employees* как длинное целое, а *InvoiceTotal*в таблице *Invoices* — как *currency* (валюта), то можете написать запрос, соединяющий эти две таблицы при помощи критерия *WHERE EmployeeID* = *InvoiceTotal*. Microsoft Jet успешно выполнит этот запрос и вернет вам результат — список сотрудников, идентификационный номер (*EmployeeID*)которых будет совпадать с суммой, указанной в счете *(InvoiceTotal)*. Совершенно очевидно, что два эти атрибута не являются совместимыми по типу данных, однако мы никак не можем «объяснить» это механизму СУБД Microsoft Jet,

Итак, стоит ли вообще иметь дело с доменами? Ответ однозначный: да, стоит. Во второй части книги мы рассмотрим домены более подробно, и увидим, что это чрезвычайно мощные и эффективные инструменты. При разработке баз данных часто возникают вопросы: «Являются ли эти атрибуты взаимозаменяемыми?» или «Определены ли правила, применимые к одному атрибуту, но не применимые к другому?». Анализ доменов позволяет получить ответы.

Связи

Кроме атрибутов каждой сущности модель данных должна определять связи между сущностями. На концептуальном уровне *связи* представ-

ляют собой простые ассоциации между сущностями. Например, утверждение «Покупатели покупают продукты» указывает, что между сущностями *Customers* (Покупатели) и *Products* (Продукты) существует связь, и такие сущности называются ***участниками*** этой связи. Число участников определяет ***размерность*** связи. Определение размерности связи похоже на определение размерности отношения, но они между собой не эквивалентны.

Большинство связей — двойные, в них два участника. Пример такой связи — связь между сущностями *Customers* (Покупатели) и *Products* (Продукты). Однако существуют и другие виды связей, например, на существование неявной тройной связи указывает утверждение «Сотрудники фирмы продают товары покупателям». Определение двух двойных связей не позволяет определить, кто именно из сотрудников продал товары покупателям, и какие именно товары каким именно покупателям он продал. Для этого нужно определять тройныесвязи.

Весьма интересен особый случай — это связь, в которой сущность является участником связи с самой собой. Такую связь часто называют связью типа «спецификация товаров» и используют для представления иерархических структур. Наглядный пример — связь между сотрудником и менеджером: любой сотрудник может одновременно сам являться менеджером и подчиняться другому менеджеру.

Существует несколько типов связей между двумя сущностями: это связи «один к одному», «один ко многим» и «многие ко многим». Связи «один к одному» встречаются достаточно редко, в основном, между сущностями надтипов и подтипов. Возвращаясь к рассмотренному нами примеру, связь между сотрудником и информацией о торговом агенте, относящейся к данному сотруднику, будет связью «один к одному».

Связи «один ко многим» более часты. «В счете перечислено множество товаров», «Торговый агент выписывает много счетов» — вот примеры таких связей,

Связи «многие ко многим», хотя и не столь широко распространены, как «один ко многим», также не редкость. Например, каждый покупатель покупает разные товары, и определенные виды товаров также приобретаются разными покупателями. Преподаватели в университете читают лекции многим студентам, и каждый конкретный студент посещает лекции нескольких преподавателей. Связи «многие ко многим» невозможно непосредственно реализовать в реляционной модели, однако их косвенная реализация вполне проста и однозначна (об этом будет подробно рассказано в главе 3).

Участие каждой сущности в определенной связи может быть *частичным* или *полным*. Если существование данной сущности полностью определяется ее участием в связи, то такое участие будет полным, в противном случае — частичным.

Например, сущность, определяющая информацию об агенте по сбыту, не может существовать, если она не связана с конкретным сотрудником. Обратное утверждение неверно: сотрудник может не являться торговым агентом, а занимать какую-либо другую должность в компании, и в этом случае для хранения данных о сотруднике может быть выбрана другая сущность. Этому сотруднику не будет соответствовать ни одна запись, содержащая информацию об агентах по сбыту. Таким образом, участие сущности *Employee* (Сотрудник) в рассматриваемой нами связи будет мастичным, а сущности *Salesperson* (Торговый агент) — полным.

Один из самых важных и тонких моментов в процессе создания модели данных — схема должна содержать верные определения связей для каждой сущности на протяжении всего срока эксплуатации системы.

Поясним это на конкретном примере. Компании, специализирующиеся на реализации товаров, довольно часто меняют поставщиков отдельных товаров. Поэтому если участие сущности *Products* (Продукты) в связи «Поставщики поставляют продукты» будет определено как полное, то впоследствии окажется невозможным удалить текущего поставщика какого-либо продукта, не удаляя при этом всю остальную информацию об этом продукте.

Диаграмма «сущности - связи»

В 1976 г. Питер Пин Шань Чен (Peter Pin Shan Chen) внес существенный вклад в теорию моделирования данных, разработав модель «сущности — связи», в которой реализовано описание данных в терминах сущностей, атрибутов и связей[2].

Одновременно он Предложил новый метод построения диаграмм — диаграммы «сущности — связи» (Entity Relationship diagrams, или E/R diagrams), который вскоре стал широко применяться разработчиками баз данных. На диаграммах «сущности — связи» сущности изображаются в виде прямоугольников, атрибуты — эллипсов, а отношения — ромбов (рис. 1-6).

[2] *Peter Pin Shan Chen.* The Entity Relationship Model—Toward a Unified View **of Data? ACM** TODS I. №. I (March. 1976).

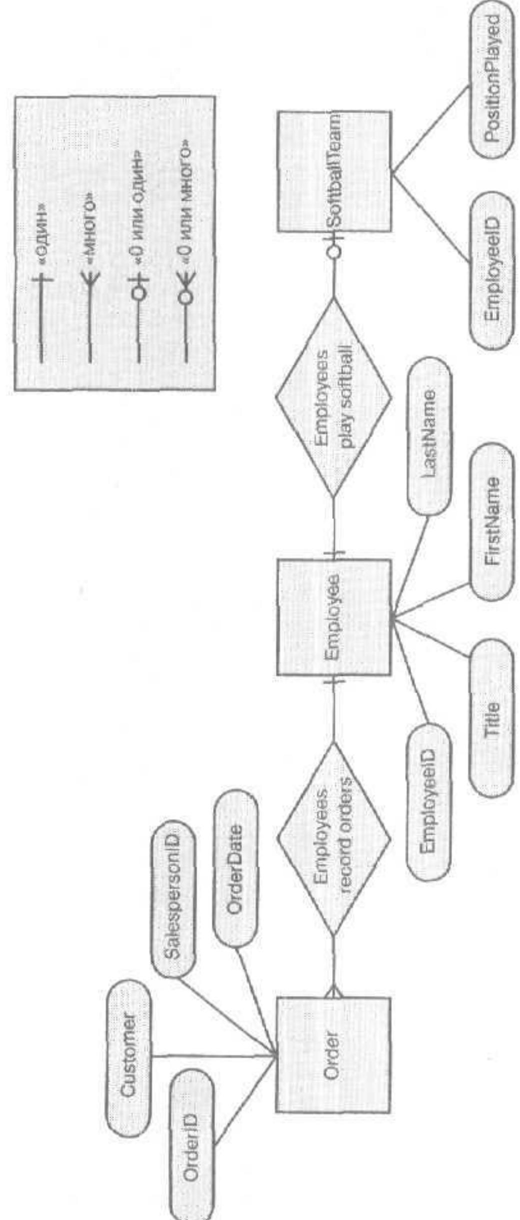


Рис. 1-6. Диаграмма «сущности — связи»

Вид связи между сущностями («один к одному», «один ко многим» или «многие ко многим») на разных диаграммах может изображаться по-разному. Некоторые разработчики используют обозначения 1 и M или 1 и со для обозначения понятий «один» и «много»; символ ∞ означает «бесконечно много». Лично я использую технику представления связей между сущностями, которую разработчики баз данных называют «птичья лапа». Эти обозначения использованы при составлении диаграммы на рис. 1-6.

Огромное преимущество техники диаграмм «сущности — связи» — простота восприятия, такие диаграммы легко составлять и читать. Но проектируя реальную систему, я составляю диаграмму атрибутов отдельно от диаграммы «сущности — связи», поскольку атрибуты предполагают отдельный уровень детализации.

Обычно разработчики баз данных при проектировании системы концентрируют внимание либо на сущностях данной модели и связях между ними, либо на атрибутах конкретной сущности, но не на том и том одновременно.

Итоги

Мы познакомились с компонентами системы баз данных и определили основные понятия, которые будут подробно раскрыты далее. Начав с описания предметной области как определенной части реального мира, мы перешли к понятию концептуальной модели данных — описанию предметной области в терминах сущностей, атрибутов, и доменов, на которых они определены. Физической реализацией модели данных является схема базы данных, представляющая собой описание реализации конкретной базы данных. Механизм базы данных выполняет все физические операции с базой данных, предоставляя результаты приложению, с которым работают пользователи. Пользовательское приложение, как правило, состоит из форм и отчетов.

В следующей главе будут подробно рассмотрены структура базы данных и принципы нормализации.

Структура базы данных

В этой главе обсуждается начальная стадия проектирования моделей реляционных баз данных — создание структуры отношений. На этой стадии проектирования главное — добиться, чтобы модель максимально корректно и полно описывала предметную область, для которой создается. Кроме того, нужно минимизировать избыточность данных и устранить связанные с этим проблемы.

Избыточность данных есть зло не только потому, что приводит к бесполезному расходованию ресурсов, она еще и осложняет жизнь тем, кто работает с базой. Возьмем в качестве примера набор записей, изображенный на рис. 2-1 и представляющий собой информацию о счетах-фактурах компании (предположим, что этот набор данных — содержимое таблицы, реально хранящейся в базе данных, а не просто результат выполнения какого-либо запроса).

OrderID	EmployeeName	HireDate	TelephoneExtension	CompanyName	ProductName	Quantity
10368	Buchanan Steven	10/17/93	3453	La maison d'Asie	Sasquatch Ale	10
10507	Buchanan Steven	10/17/93	3453	Save-a-lot Markets	Mozzarella di Giovanni	12
10714	Buchanan Steven	10/17/93	3453	Save-a-lot Markets	Zaanse koeken	50
10437	Callahan Laura	3/5/94	2344	Wartian Herkku	Perth Pasties	15
10383	Callahan Laura	3/5/94	2344	Around the Horn	Konbu	20
10773	Davolio Nancy	6/1/92	5467	Ernst Handel	Gorgonzola Telino	70
10325	Davolio Nancy	6/1/92	5467	Königlich Essen	Mozzarella di Giovanni	40
10773	Davolio Nancy	6/1/92	5467	Ernst Handel	Rhönbräu Klosterbier	7
10992	Davolio Nancy	6/1/92	5467	The Big Cheese	Mozzarella di Giovanni	2
10916	Davolio Nancy	6/1/92	5467	Rancho grande	Ravioli Angelo	20
10727	Fuller Andrew	8/14/92	3457	Reggiani Caseifici	Gnocchi di nonna Alice	10
10727	Fuller Andrew	8/14/92	3457	Reggiani Caseifici	Raclette Courdavault	10
10597	King Robert	1/2/94	465	Piccolo und mehr	Guaraná Fantástica	35
10642	King Robert	1/2/94	465	Simons bistro	Sir Rodney's Scones	30
10523	King Robert	1/2/94	465	Seven Seas Imports	Gravad lax	18
10856	Leverling Janet	4/1/92	3355	Antonio Moreno Taquería	Singaporean Hokkien Fried Mee	20
10856	Leverling Jane!	4/1/92	3355	Antonio Moreno Taquería	Chang	20
10544	Leverling Janet	4/1/92	3355	Wellington Importadora	Spegesild	21
10554	Peacock Margaret	5/3/93	5176	Ottilies Käseladen	Tarte au sucre	20
10654	Peacock Margaret	5/3/93	5176	Ottilies Käseladen	Tunnbröd	20
10657	Suyama Michael	10/17/93	428	Lonesome Pine Restaurant	Perth Pasties	3
10365	Suyama Michael	10/17/93	428	Die Wandernde Kuh	Gorgonzola Telino	30
10701	Suyama Michael	10/17/93	428	Hungry Owl All-Night Grocers	Lakkalikööri	35
10271	Suyama Michael	10/17/93	428	Split Rail Beer & Ale	Geitost	24

Рис. 2-1. *Набор записей содержит избыточные данные*

Как видите, в этой таблице значения *HireDate* и *TelephoneExtension* одинаковы в записях, в которых совпадает имя работника компании. Такая организация данных имеет несколько последствий.

Во-первых, каждый раз при создании новой записи вы должны вместе с именем сотрудника занести значения в поля *HireDate* и *TelephoneExtension*, даже если уже вносили их раньше (разумеется, каждый раз вы рискуете ошибиться). Например, на основании данных на рис. 2-2, сможете ли вы определить, в каком году сотрудник по имени Стивен Букэнон (Steven Buchanan) был принят на работу — в 1989 или 1998 г.?

Кроме того, такая структура данных не позволяет ввести дату найма и внутренний телефон сотрудника, до тех пор пока он не совершит хотя бы одной сделки. И наконец, если счета-фактуры за определенный год удаляются из базы данных и архивируются, то данные о телефоне и дате найма определенных сотрудников могут быть утеряны.

OrderID	EmployeeName	HireDate	TelephoneExtension	CompanyName	ProductName	Quantity
10358	Buchanan Steven	10/17/89	3453	La maison d'Asie	Sasquatch Ale	10
10607	Buchanan Steven	10/17/98	3453	Save-a-lot Markets	Mozzarella di Giovanni	12
10714	Buchanan Steven	10/17/98	3453	Save-a-lot Markets	Zaanse koeken	50

Рис. 2-2. Дублирование данных может привести в том числе и к их рассогласованию

Такие проблемы обычно называют аномалиями обновления. Они могут приводить к еше более серьезным последствиям, если избыточные данные хранятся не в одном, а одновременно в нескольких отношениях.

Рассмотрим пример на рис. 2-3 (как и прежде мы видим содержимое реально существующих таблиц, а не результаты выполнения запросов). Если телефонный номер компании «Around the Horn» изменился, вам придется поменять его в наборе записей *Customer* и не забыть обновить телефонный номер в каждой записи набора *Invoice*, относящейся к этой компании.

В этом примере данных очень мало, и их аккуратное обновление представляет собой невыполнимую задачу. Проблема в том, чтобы не забыть сделать все, что необходимо. И даже если вы лично никогда ничего не забываете, уверены ли вы в том, что другой программист, который будет поддерживать вашу программу через полгода, хорошо осведомлен об этой избыточности данных и о том, как с ней правильно обращаться? Намного лучше избежать избыточности данных и всех связанных с ней проблем.

Отношение *Customers*

CustomerID	CompanyName	Phone
ALFKI	Alfreds Futterkiste	030-0074321
ANATR	Ana Trujillo Emparedados y helados	(5) 555-4729
ANTON	Antonio Moreno Taquería	(5) 555-3932
AROUT	Around the Horn	(171) 555-7788
BERGS	Berglunds snabbköp	0921-12 34 65

Отношение *Invoices*

OrderID	CompanyName	Phone
10952	Alfreds Futterkiste	030-0074321
10952	Alfreds Futterkiste	030-0074321
10625	Ana Trujillo Emparedados y helados	(5) 555-4729
10625	Ana Trujillo Emparedados y helados	(5) 555-4729
10625	Ana Trujillo Emparedados y helados	(5) 555-4729
10856	Antonio Moreno Taquería	(5) 555-3932
10558	Around the Horn	(171) 555-7788
10558	Around the Horn	(171) 555-7788
10558	Around the Horn	(171) 555-7788
10558	Around the Horn	(171) 555-7788
10572	Berglunds snabbköp	0921-12 34 65
10875	Berglunds snabbköp	0921-12 34 65
10875	Berglunds snabbköp	0921-12 34 65
10875	Berglunds snabbköp	0921-12 34 65

Рис. 2-3. Дублированные данные могут присутствовать в нескольких отношениях

Однако, предположив, что некоторые атрибуты в отношении являются избыточными, не торопитесь делать поспешные выводы. Необходимо убедиться, что атрибуты, которые кажутся избыточными, являются таковыми на самом деле.

Рассмотрим рис. 2-4. На первый взгляд может показаться, что атрибуты *UnitPrice* в отношениях содержат избыточные данные. На самом же деле, в каждом из отношений они представляют разные по смыслу величины.

Атрибут *UnitPrice* в отношении *Products* отражает текущую цену продажи товара. Атрибут *UnitPrice* в отношении *Orders* — это цена, по которой товар продан. Для товара *Tofu*, например, в *Orders* существует запись, где значение *UnitPrice* равно $18,60, а в отношении *Products* — запись, где *UnitPrice* равно $23,25. То, что товар *Tofu* продается сейчас по цене $23,25, никак не может изменить то, что некоторое время назад он продавался по $18,60. Оба атрибута определены в одном и том же домене, но являются различными по смыслу величинами.

Отношение *Products*

ProductID	ProductName	UnitPrice
11	Queso Cabrales	S21 00
14	Tofu	$23.25
22	Gustaf's Knäckebröd	$21.00
41	Jack's New England Clam Chowder	$9.65
42	Singaporean Hokkien Fried Mee	$14.00
51	Manjimup Dried Apples	(53.00
57;	Ravioli Angelo	$19.50
65	Louisiana Fiery Hot Pepper Sauce	$21.05
66	Louisiana Hot Spiced Okra	$17.00
72	Mozzarella di Giovanni	$34.80

Отношение *Orders*

OrderID	ProductName	UnitPrice	Quantity	OrderDate
10248	Singaporean Hokkien Fried Mee	$9.80	10	04-Aug-94
10248	Queso Cabrales	$14DD	12	04-Aug-94
10248	Mozzarella di Giovanni	$34BO	5	04-Aug-94
10249	Tofu	$18.60	9	05-Aug-94
10249	Manjimup Dried Apples	$42.40	40;	05-Aug-94
10250	Manjimup Dried Apples	$42.40	35	08-Aug-94
10250	Louisiana Fiery Hot Pepper Sauce	$16.80	15	08-Aug-94
10250	Jack's New England Clam Chowder	(7.70	10	08-Aug-94
10251	Ravioli Angelo	$15.60	15!	08-Aug-94
10251	Louisiana Fiery Hot Pepper Sauce	$16.80	20!	08-Aug-94
10251	Gustaf's Knäckebröd	$16.80	6	08-Aug-94

Рис. 2-4. Данные, которые кажутся избыточными, на самом деле не являются таковыми

Способность модели данных отвечать на поставленные вопросы определяется прежде всего полнотой хранящихся в ней данных (никакая база данных не может обеспечить пользователей той информацией, которой не содержит), и только во вторую очередь — ее структурой. Но вот легкость, с которой можно получить необходимые данные, во всех случаях определяется исключительно структурой базы. Принципиально важно, что собрать в единое целое информацию из отдельных атрибутов и отношений достаточно легко, а вот провести дальнейшую детализацию уже занесенной в базу информации очень сложно (рис. 2-5).

Обратимся вновь к примеру из главы 1, где речь шла об именах адресатов — допустим, именно их список отображен на рис. 2-5. Полное имя из значений полей верхнего отношения получить легко: достаточно записать выражение:

```
TitleOfCourtesy & " " & FirstName S " " LastName &_
    ", " & Title
```

EmployeeID	TitleOfCourtesy	FirstName	LastName	Title
1	Ms.	Nancy	Davolio	Sales Representative
2	Dr.	Andrew	Fuller	Vice President, Sales
3	Ms.	Janet	Leverling	Sales Representative
4	Mrs.	Margaret	Peacock	Sales Representative
5	Mr.	Steven	Buchanan	Sales Manager

EmployeeID	FullName
1	Ms. Nancy Davolio, Sales Representative
2	Dr. Andrew Fuller, Vice President, Sales
3	Ms. Janet Leverling, Sales Representative
Л	Mrs. Margaret Peacock, Sales Representative
5	Mr Steven Buchanan, Sales Manager

Рис. 2-5. Полное имя адресатов представлено в виде отдельных атрибутов — в верхнем отношении, и как один атрибут — в нижнем

А вот для того, чтобы выделить фамилию из поля *FullName* нижнего отношения, нужно обработать строковые данные:

```
Function GetLastname (FullName as String) as String (
  Dim lastname as String
  ' убрать титул
  lastname - Left(FullName, InStr(FullName, ", ")-1)
  ' убрать вежливое обращение
  lastname - Right(lastname, Len(lastname) - _
    InStr(lastname, " "))
  ' убрать имя
  lastname = Right(lastname, Len(lastname) - _
    InStr(lastname, " "))

  GetLastname = lastname

End Function
```

Уязвимое место такого подхода — его чувствительность к изменениям содержимого поля *FullName;* например, естественно ожидать, что для имени *Billy Rae Jones* в качестве фамилии должно возврашаться значение *Rae Jones*; но вы, вероятно, получите вместо этого строку *Jones.* В результате список, составленный в виде набора значений (*LastName, FirstName*), будет выглядеть довольно странно.

Второй принцип, используемый при создании модели данных, способной эффективно предоставлять необходимую информацию, — избегать дублирования одной и той же информации во множестве полей (рис. 2-6 и 2-7),

Отношение *Enrollments*

StudentID	FirstName	LastName	Period1	Period2	Period3	Period4	Period5	Period6
1	Nancy	Davolio	Biology	French	English	History	Chemistry	Physical Education
2	Andrew	Fuller	Physical Educa	Biology	French	English	Chemistry	History
3	Janet	Levening	Physical Educa	Biology	French	English	Chemistry	History
4	Margaret	Peacock	French	Physical Educa	History	English	Biology	Chemistry
5	Steven	Buchanan	Biology	French	Physical Educa	History	English	Chemistry
6	Michael	Suyama	French	Physical Educa	Biology	History	English	Chemistry
7	Robert	King	History	French	English	Biology	Chemistry	Physical Education

Рис. 2-6, Такая структура затрудняет получение ответов на некоторые вопросы

Отношение *Enrollments*

StudentID	FirstName	LastName	Period	Class
1	Nancy	Davolio	1	Biology
2	Andrew	Fuller	1	Physical Education
3	Janet	Levelling	1	Physical Education
4	Margaret	Peacock	1	French
5	Steven	Buchanan	1	Biology
6	Michael	Suyama	1	French
7:	Robert	King	1	History
1	Nancy	Davolio	2	French
2.	Andrew	Fuller	2	Biology
3	Janet	Leverling	2.	Biology
A	Margaret	Peacock	2.	Physical Education
5	Steven	Buchanan	2	French
6	Michael	Suyama	2	Physical Education
7	Robert	King	2	French
1	Nancy	Davolio	Э	English
2.	Andrew	Fuller	3	French

Рис. 2- 7. Данная структура содержит больше записей, но для нее легче формулировать запросы

Чтобы получить ответ на вопрос «Кто из студентов в этом году изучает биологию?» с помощью первого отношения, придется искать записи со значением *Biology* в шести различных полях. Соответствующее выражение SELECT языка SQL будет выглядеть так:

```
SELECT StudentID FROM Enrollments WHERE Period1 = "Biology"
   OR Period2 = "Biology" OR Period3 = "Biology"
   OR Period4 = "Biology" OR Period5 = "Biology"
   OR Period6 = "Biology"
```

А при использовании отношения, показанного на рис. 2-7, требуется выполнить поиск только в одном поле *Class*:

```
SELECT StudentID FROM Enrollments WHERE Class = "Biology"
```

Оба этих подхода вполне работоспособны, но второй очевидно проще, и вероятность ошибки при его реализации меньше, не говоря уже о том, что его не нужно долго обдумывать.

Устранить избыточность и упростить структуру данных, чтобы облегчить их получение — вот главные ориентиры при построении модели. Все остальное — только попытки формализовать эти базовые принципы. Но если вы когда-либо создавали большую (или не очень большую) модель данных, то знаете, что как бы ни были эти принципы просты, на их практическом использовании очень легко «поскользнуться». Они напоминают канцелярский скоросшиватель для бумаг: как им пользоваться — совершенно очевидно, но только когда вам это уже показали.

Основные принципы

Принципы нормализации, обсуждаемые далее в этой главе, являются таким же инструментом контроля структуры данных, как и канцелярский скоросшиватель, не дающий разлететься стопке бумаг. Нормальные формы определяют возрастающую строгость правил, которым подчиняются структуры отношений. Мы рассмотрим шесть таких форм. Каждая последующая форма расширяет предыдущую, устраняя при этом возможность возникновения аномалий обновления определенного типа.

Помните, что нормальные формы не являются руководством для создания «правильной» модели данных. Модель данных может быть совершенной с точки зрения теории нормализации, но созданная на ее основе реальная база данных будет выдавать требуемые результаты столь медленно и неуклюже, что ее просто не удастся использовать. Тем не менее, если ваша модель данных нормализована (то есть удовлетворяет принципам, принятым для реляционных структур) шансы получить в результате эффективную модель данных увеличиваются. Перед тем, как мы вернемся к нормализации, ознакомимся с несколькими принципами, лежащими в ее основе.

Декомпозиция **без потерь**

Реляционная модель позволяет различным образом соединять отношения, связывая их через атрибуты. Процесс получения полностью нормализованной модели данных включает в себя устранение избыточности. Для этого отношение, содержащее избыточные данные, разбивают на несколько других отношений. Нужно сделать это так, чтобы получившиеся в результате отношения можно было бы вновь соединить и получить точную копию структуры и данных исходного отношения. Это и есть принцип декомпозиции без потерь. Например, вы можете получить из отношения (рис. 2-8), два других отношения (рис. 2-9).

OrderID	OrderDate	RequiredDate	CompanyName	Address	City	PostalCode
10248	04-Aug-94	01-Sep-94	Vins et alcools Chevalier	59 rue de l'Abbaye	Reims	51100
10249	06-Aug-94	16-Sep-94	Toms Spezialitäten	Luisenstr. 48	Munster	44087
10250	08-Aug-94	05-Sep-94	Hanari Carnes	Rua do Paço, 67	Rio de Janeiro	05454-876
10251	08-Aug-94	05-Sep-94	Victuailles en stock	2V rue du Commerce	Lyon	69004
10252	09-Aug-94	06-Sep-94	Supremes délices	Boulevard Tirou, 255	Charleroi	B-6000

Рис. 2-8. Ненормализованное отношение

Отношение *Customers*

CustomerID	CompanyName	Address	City	PostalCode
ALFKI	Alfreds Futterkiste	Obere Str. 57	Berlin	12209
ANATR	Ana Trujillo Emparedados y helados	Avda. de la Constitución 2222	México D.F.	05021
ANTON	Antonio Moreno Taquería	Mataderos 2312	México D.F.	05023
AROUT	Around the Horn	120 Hanover Sq.	London	WA1 1DP
BERGS	Berglunds snabbköp	Berguvsvägen 8	Luleå	S-958 22

Отношение *Invoices*

OrderID	CustomerID	OrderDate	RequiredDate
10248	VINET	8/4/94	9/1/94
10249	TOMSP	8/5/94	9/16/94
10250	HANAR	8/8/94	9/5/94
10251	VICTE	8/8/94	9/5/94
10252	SUPRD	8/9/94	9/6/94

Рис. 2-9. Отношение с рис. 2-8 можно разбить на эти два отношения без потери данных

Использование двух отношений вместо одного устраняет избыточность адресных данных заказчика, но позволяет при необходимости найти адрес, связав отношения *Customers* и *Invoices* по содержащемуся в них полю *CustomerID*.

Ключи-кандидаты и первичные ключи

В главе 1 я определила содержимое отношения как неупорядоченное множество, состоящее из 0 или более кортежей, и указала, что по определению каждый элемент множества кортежей должен быть уникален. В таком случае для любого отношения должна существовать комбинация атрибутов, однозначно определяющая каждый кортеж. Такой набор из одного или более атрибутов называют ключом-кандидатом.

Для любого отношения может существовать более одного ключа-кандидата, но в любом случае каждый ключ-кандидат должен однозначно определять каждый кортеж, и не только для любого специфического множества кортежей, а для всех возможных кортежей в любой момент времени. Кстати, обратный принцип тоже верен: если взять два кортежа с одинаковыми значениями ключа-кандидата, то оба этих кортежа должны представлять одну и ту же сущность. Подтекст в том, что вы не можете определить ключ-кандидат эмпирически. Нельзя гарантировать, что некое поле или набор полей будет уникальным для всех кортежей лишь на том основании, что такой набор уникально идентифицирует некое ограниченное множество кортежей.

Между тем однозначная идентификация любого кортежа является непременным требованием к ключу-кандидату. Повторю еще раз: вы должны понимать специфику предметной области, чтобы правильно определить ключевой набор атрибутов.

Рассмотрим отношение *Invoices* (рис. 2-9). Поле *Customer ID* является уникальным в этом примере, но весьма сомнительно, что оно будет оставаться уникальным в дальнейшем — ведь компания стремится продать каждому заказчику как можно больше различных товаров! Несмотря на кажущуюся очевидность выводов, специфика предметной области говорит, что это поле не является ключом-кандидатом.

Если буквально следовать определению, то каждое отношение должно иметь, по меньшей мере, один ключ-кандидат — набор всех атрибутов, составляющих кортеж. Ключ-кандидат может состоять из единственного атрибута *(простой ключ)* или множества атрибутов (*составной ключ*). Тем не менее, дополнительное требование к ключу-кандидату такое: он должен состоять из минимального набора атрибутов, однозначно идентифицирующих кортеж, поэтому полный набор атрибутов отношения не обязательно является ключом-кандидатом. Для отношения, показанного на рис. 2-10, атрибут *Category* Л) является ключом-кандидатом, а набор *{CategoryID, CategoryName}* хотя и уникален для каждого кортежа, не является им, так как атрибут *CategoryName* необязателен для однозначной идентификации кортежа.

CategoryID	CategoryName	Description
1	Beverages	Soft drinks, coffees, teas, beers, and ales
2	Condiments	Sweet and savory sauces, relishes, spreads, and seasonings
3	Confections	Desserts, candies, and sweet breads
4	Dairy Products	Cheeses
5	Grains/Cereals	Breads, crackers, pasta, and cereal
6	Meat/Poultry	Prepared meats
7	Produce	Dried fruit and bean curd
8	Seafood	Seaweed and fish

Рис. 2-10. Ключ-кандидат должен быть минимальным, чему соответствует CategoryID, но не соответствует набор {CategoryID, CategoryName}

Иногда, хотя и не часто, отношение имеет несколько возможных ключей-кандидатов. В этом случае проектировщик должен по своему усмотрению выбрать один ключ-кандидат в качестве *первичного ключа,* а оставшиеся ключи-кандидаты будут являться *альтернативными ключами.* Это особенность логической модели довольно удобна. (Напомню, что логическая модель данных — абстрактное понятие). Чтобы различать модель и ее физическую реализацию, я предпочитаю

использовать термин «ключ-кандидат» для логической модели данных и приберечь «первичный ключ» для использования в физической реализации модели.

ПРИМЕЧАНИЕ Когда единственно возможный ключ-кандидат чересчур громоздок (например, состоит из слишком многих атрибутов или слишком велик), вы можете использовать специальный тип данных, который механизм баз данных Microsoft Jet и Microsoft SQL Server поддерживают для создания искусственных ключей. В таких искусственных ключах хранятся значения, генерируемые самой системой. Называемые *AutoNumber* в Microsoft Jet и *Identity* в SQL Server, поля этого типа очень удобны для создания идентификаторов строк. При этом подразумевается, что вы не будете пытаться связать поле этого типа с какой-нибудь конкретной сущностью предметной области. Такие поля не более чем ярлыки. Ничто не гарантирует, что значения содержащихся в них величин будут строго последовательными, вы практически не можете контролировать процесс их генерации системой. Так что не пытайтесь использовать их для чего-либо еще, кроме нумерации, иначе столкнетесь с массой проблем.

Хотя выбор ключа-кандидата является семантической проблемой, не надейтесь, что атрибуты, с помощью которых вы моделируете сущности реального мира, всегда позволят вам построить подходящий ключ-кандидат. Люди обычно отзываются на свои имена; но, внимательно просмотрев страницы телефонной книги, вы убедитесь, что имена вряд ли можно рассматривать в качестве уникального идентификатора личности.

Разумеется, скомбинировав имя с рядом других атрибутов, мы получим в результате ключ-кандидат, но этот путь может оказаться довольно извилистым. Однажды я работала в офисе, где из двадцати сотрудников двоих звали Ларри Симон и еще одного — Лари Симон. Они имели прозвища «Коротышка Лари», «Ларри-немец» и «Ларри-блондин»; то есть рост, национальность и цвет волос совместно с именем выступали, и вполне удачно, в роли ключа-кандидата. В подобных ситуациях наилучший способ получить ключ-кандидат — использовать идентификатор, генерируемый системой, такой, например, как поле *AutoNumber или Identify*. Но помните: не следует пытаться использовать это поле для чего-либо еще кроме идентификации!

Функциональная зависимость

Концепция функциональной зависимости чрезвычайно удобна для анализа структуры данных. Рассматривая любой кортеж T и два на-

бора атрибутов этого кортежа {X1,...,Xn} и {Y1,...,Yn} (множества X и Y не обязательно являются взаимоисключающими) говорят, что Y функционально зависит от X, если для каждого возможного значения X существует единственно возможное соответствующее значение Y.

Например, в отношении на рис. 2-10, все кортежи с одним и тем же значением *{CategoryID}*будут иметь одинаковое значение *{CategoryName, Description}*. Поэтому мы вправе сказать, что атрибут *CategoryID*функционально определяет множество *{CategoryName, Description}*. Учтите, что обратной функциональной зависимости может и не быть: зная величину *{CategoryName,Description}*, нельзя однозначно определить величину *CategoryID*.

Вы можете указать на функциональную зависимость между атрибутами так, как это сделано на рис. 2-11. В тексте следует записать функциональную зависимость в виде выражения X → Y, что читается как «X функционально зависит от Y».

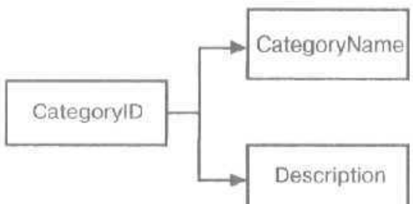


Рис. 2-11. Диаграмма функциональной зависимости чрезвычайно информативна и интуитивно понятна

Функциональная зависимость привлекает ученых-математиков, поскольку это понятие является основой для развития математических подходов к моделированию данных. Если вы склонны к теоретическим изысканиям, то можете исследовать, например, свойства рефлексивности и транзитивности функциональной зависимости.

На практике функциональная зависимость — удобный способ отразить тот очевидный факт, что для любого отношения существует некий набор атрибутов, уникальный для каждого кортежа данного отношения, и зная этот набор, можно определить значения других, неуникальных атрибутов.

Если {X} является ключом-кандидатом, то все атрибуты {Y} функционально зависимы от {X}: это следует из определения ключа-кандидата. Если {X}не является ключом-кандидатом, и функциональная зависимость нетривиальна (то есть {Y} не является подмножеством {X}), тогда отношение обязательно будет содержать избыточные данные и дальнейшей нормализации не избежать. До известной степени

нормализация — это процесс, конечным результатом которого является диаграмма функциональной зависимости, подобная изображенной на рис. 2-11, на которой все стрелки выходят из ключей-кандидатов.

Первая нормальная форма

Отношение находится в первой нормальной форме, если домены, в которых определены его атрибуты, являются скалярными величинами. Понятие скалярной величины — одновременно самое простое и самое сложное в моделировании данных. Принцип таков: каждый атрибут кортежа должен содержать отдельную величину. Но что это означает? В отношении, изображенном на рис. 2-12, атрибут *Items* несомненно содержит составные величины и поэтому отношение не находится в первой нормальной форме. Но окончательный вывод не всегда столь очевиден.

OrderID	CustomerID	OrderDate	Items	OrderTotal
1	CACTU	1/1/99	3 Zaanse koeken, 1 Tarte au sucre	$89.70
2	BSBEV	1/5/98	4 Mozzarella di Giovanni	$139.20
3	SUPRD	5/2/99	3 Ravioli Angelo, 6 Tofu	$198.06

***Рис.** 2-12. **Атрибут Items** в данном **отношении** не является скалярным*

Не так просто определить, является ли атрибут скалярным — мы сталкивались с этим, когда рассматривали моделирование имен и адресов в главе 1. Можно привести и другие примеры «коварных» видов доменов. Например, дата состоит из трех различных компонентов: дня, месяца и года. Как лучше хранить дату: как три различных атрибута или как единое целое? Как всегда, ответ зависит от особенностей предметной области, которую вы моделируете. Если в вашей системе дата используется исключительно или даже просто в большинстве случаев как отдельная величина, то она — скалярная величина. Но если система часто работает с отдельными составляющими даты, лучше хранить ее в качестве набора из трех разных атрибутов (например, если вас интересует только месяц и год, а не конкретный день: или только месяц и день, но не год). Такие случаи не очень часто встречаются, но нельзя сказать, что их нет совсем.

Арифметические манипуляции с датами утомительны, так что в большинстве случаев лучше использовать для хранения даты поля типа *DateTime*, чтобы переложить большую часть вычислительной работы с датами на саму среду разработки. Тем не менее, такой подход может привести к ошибкам, если сравнивать данные только в той части, которая содержит дату. Ошибки практически неизбежны, если игнорировать часть поля, отвечающую за хранение времени. Допус-

тим, вы присваиваете содержимому поля *DateCreated* значения, возвращаемые функцией *Now* (VBA): величину, содержащую и дату, и время. Если потом вы попытаетесь сравнить содержимое поля с результатом, возвращаемым функцией *Date()*, то получите неверный результат (эта функция возвращает только дату). Даже если вы никогда не показываете значение времени пользователям, оно все равно хранится вместе с датой, а величины «1/1/1999 12:30:19 AM» и «1/1/1999» очевидно не совпадают.

Часто проблемы, связанные с составными данными, возникают при моделировании кодов и флагов. Множество компаний используют для ведения учета ссылочные номера, часто представляющие собой некоторую последовательность символов, например REF0010398. Это может означать, например, что учетная запись с таким номером является первой записью в марте 1998 г. Так как маловероятно, что вам удастся изменить учетную политику компании-заказчика, вряд ли стоит манипулировать отдельными компонентами учетного номера в модели данных.

Значительно проще и перспективнее хранить компоненты номера как отдельные величины: (Ref#, Case#, Month, Year). В этом случае определение номера следующей учетной записи или учетных записей, созданных в определенном году выполняется с помощью простого запроса значения одного из атрибутов и не требует выполнять сложныеоперации.

Это важно для обеспечения высокой производительности приложений, в частности, когда реализуется модель клиент-сервер. Ведь для извлечения части данных из середины содержимого поля может потребоваться локальная обработка записи на компьютере-клиенте, вместо того чтобы возложить эту обязанность на сервер.

Другой тип составных данных, также вызывающий сложности при моделировании — битовый флаг. В принятой повсеместно практике программирования множество булевых величин сохраняют в виде значений индивидуальных битов в слове, а затем используют побитовые операции, чтобы проверять и извлекать эти значения. Например, программирование с помощью Windows API в значительной степени основывается на такой технологии. В общем случае такой подход вполне оправдан, но в реляционной модели данных это не так. Здесь он не только нарушает принципы первой нормальной формы — вы просто не сможете его использовать, так как ни механизм СУБД Jet, ни версии языка SQL для SQL Server не поддерживают битовые операции. Выполняйте такие операции, используя собственные функции в приложениях баз данных, но только в Mic-

rosoft Access (Microsoft Visual Basic не позволяет использовать в запросах функции, написанные самим пользователем). Такой подход требует дополнительно обрабатывать результаты выполнения запросов на локальном компьютере.

К сожалению, подобные ограничения часто налагаются предысторией разработки: использовавшимися в прошлом средами разработки, корпоративной политикой и т. д.. Но если у вас есть какой-то выбор, сохраняйте в каждом отдельном атрибуте только одну логически неделимую часть информации. Если вы используете унаследованную информацию, то всегда можете разделить данные на логически независимые составляющие и хранить обе версии данных — старую и модифицированную.

Существуют другие виды составных данных, о которых следует позаботиться, когда речь идет о приведении к первой нормальной форме: это повторяющиеся группы данных. На рис. 2-13 изображено отношение *Invoices.* Кто-то где-то решил, что заказчикам нельзя позволить покупать более пяти наименований товара за один раз. (Я буду страшно удивлена, если окажется, что этот кто-то согласовал свое решение с менеджером по продажам). А если серьезно, то такое ограничение всегда налагается на систему искусственно, оно не обусловлено особенностями ведения бизнеса. Искусственное ограничение всегда вредно, а в данном случае еще и несомненно ошибочно.

OrderID	CustomerID	Item1	Qty1	Item2	Qty2	Item3	Qty3	Item4	Qty4	Item5	Qty5
1	ANTON	Queso Cabrales	4	Tofu	3	Ravioli Angela	1	0	0	0	0
2	BLAUS	Louisiana Fiery Hot Pepper Sauce	2		0		0		0		0

Рис. 2-13. Эта модель данных ограничивает число покупок, которые может сделать покупатель

Другой пример повторяющихся групп приведен на рис. 2-14. Ошибочность этого подхода не столь очевидна, и на его основе было создано множество вполне работоспособных баз данных. Однако это всего лишь вариант подхода к организации данных, показанного на рис. 2-13, и он имеет те же самые недостатки. Чтобы убедиться в этом, попытайтесь составить запрос с целью определить, для какого из товаров результаты продаж превысили запланированную величину на 10% в I квартале.

Product	Year	TargetJan	ActualJan	TargetFeb	ActualFeb	TargetMar	ActualMar
Aniseed Syrup	1999	$1,000.00	$1,300.00	$0.00	$0.00	$0.00	$0.00
Chai	1999	$4,000.00	$2,000.00	$0.00	$0.00	$0.00	$0.00
Chang	1999	$3,000.00	$8,022.00	$0.00	$0.00	$0.00	$0.00
Chef Anton's Cajun Seasoning	1999	$7,000.00	$7,300.00	$0.00	$0.00	$0.00	$0.00
Chef Anton's Gumbo Mix	1999	$3,000.00	$3,231.00	$0.00	$0.00	$0.00	$0.00

Рис. 2-14. Здесь присутствуют повторяющиеся группы

Вторая нормальная форма

Отношение находится во второй нормальной форме, если оно находится в первой нормальной форме, и кроме того, все его атрибуты зависят от полного набора атрибутов ключа-кандидата. Например, ключом отношения на рис. 2-15, является {*ProductName*, *SupplierName*}, но поле *SupplierPhoneNumber* определяется только значением поля *SupplierName*, а не полным набором атрибутов составного ключа.

ProductName	SupplierName	CategoryName	SupplierPhoneNumber
Chai	Exotic Liquids	Beverages	(171) 555-2222
Chang	Exotic Liquids	Beverages	(171) 555-3222
Aniseed Syrup	Exotic Liquids	Condiments	(171) 555-2222
Chef Anton's Cajun Seasoning	New Orleans Cajun Delights	Condiments	(100) 555-4822
Chef Anton's Gumbo Mix	New Orleans Cajun Delights	Condiments	flrJO) 555-4822

Рисунок 2- 75. Все атрибуты отношения должны зависеть от полного ключа

Мы уже видели, что такая зависимость между атрибутами приводит к избыточности данных, а избыточность, в свою очередь, к возможным неприятностям в процессе поддержки базы. Более совершенной с этой точки зрения является модель, изображенная на рис. 2-16.

Отношение *Products*

ProductID	ProductName	CategoryName
1	Chai	Beverages
2	Chang	Beverages
3	Aniseed Syrup	Condiments
4	Chef Anton's Cajun Seasoning	Condiments
5	Chef Anton's Gumbo Mix	Condiments

Отношение *Suppliers*

SupplierID	SupplierName	SupplierPhoneNumber
1	Exotic Liquids	(171) 555-2222
2	New Orleans Cajun Delights	(100) 555-4822
3	Grandma Kelly's Homestead	(313) 555-5735
4	Tokyo Traders	(03) 3555-5011
5	Cooperativa de Quesos 'Las Cabras'	(98) 598 76 54
6	Mayumi's	(06) 431-7877
7	Pavlova, Ltd.	(03) 444-2343
8	Specialty Biscuits, Ltd.	(161) 555-4448
9	PB Knäckebröd AB	031-987 65 43

Рис. 2-16. Эти два отношения находятся во второй нормальной форме

Данная модель является следствием декомпозиции исходного отношения на два независимых отношения: *Products* и *Suppliers.* При такой декомпозиции вы не только устраняете избыточность данных, но также позволяете записать информацию о поставщике, до того как

станет известно что-либо о поставляемых им продуктах. Этого нельзя было сделать в исходном отношении, так как ни одна из составляющих первичного ключа не может быть пустой.

Существует и другая проблема, связанная со второй нормальной формой. Она возникает при попытке опереться в ходе проектирования модели данных на факты и обстоятельства, которые справедливы сейчас, но в дальнейшем могут кардинально измениться. Например, отношение на рис. 2-17 создано с расчетом, что у поставщика на текущий момент только один адрес. Однако это вовсе не значит, что в будущем у кого-нибудь из поставщиков не окажется несколько адресов, что потребует переделки уже готовой системы.

SupplierID	Address	City	Region	PostalCode
1	49 Gilbert St.	London		EC1 4SD
2	P.O. Box 78934	New Orleans	LA	70117
3	707 Oxford Rd	Ann Arbor	MI	48104
4	9-8 Sekimai	Tokyo		100
5	Calle del Rosal 4	Oviedo	Asturias	33007
6	92 Setsuko	Osaka		545
7	74 Rose St	Melbourne	Victoria	3058
8	29 King's Way	Manchester		M14 GSD
9	Kaloadagatan 13	Göteborg		S-345 67
10	Av. das Americanas 12.890	São Paulo		5442
11	Tiergartenstraße 5	Berlin		10785
12	Bogenallee 51	Frankfurt		60439
13	Frahmredder 112a	Cuxhaven		27478
14	Viale Dante, 75	Ravenna		48100
15	Hatlevegan 5	Sandvika		1320
16	3400 - 8th Avenue	Bend	OR	97101
17	Brovallavagen 231	Stockholm		S-123 45
18	203, Rue des Francs-Bourgeois	Paris		75004
19	Order Processing Dept.	Boston	MA	02134
20	471 Serangoon Loop, Suite #402	Singapore		0512
21	Lyngbysild	Lyngby		2800
22	Verkoop	Zaandam		9999 ZZ
23	Valtakatu 12	Lappeenranta		53120
24	170 Prince Edward Parade	Sydney	NSW	2042
25	2960 Rue St. Laurent	Montréal	Québec	H1J 1C3

Рис. 2-17. ***Каждый из поставщиков может иметь*** *несколько адресов*

Третья нормальная форма

Отношение находится в третьей нормальной форме, если оно находится во второй нормальной форме, и кроме того, все неключевые атрибуты совершенно независимы. Давайте возьмем в качестве примера некую компанию, расположенную в США, у которой в каждом из штатов имеется единственный офис продаж, а в каждом из офисов работает один сотрудник. В отношении, показанном на рис. 2-18, существует зависимость между атрибутами *Region* и *Salesperson,* но ни один из этих атрибутов не является ключом-кандидатом данного отношения.

OrderID	CompanyName	Region	Salesperson
10389	Bottom-Dollar Markets	BC	Margaret Peacock
10290	Comércio Mineiro	SP	Laura Callahan
10347	Familia Arquibaldo	SP	Laura Callahan
10386	Familia Arquibaldo	SP	Laura Callahan
10423	Gourmet Lanchonetes	SP	Laura Callahan
11061	Great Lakes Food Market	OR	Michael Suyama
10528	Great Lakes Food Market	OR	Michael Suyama
10785	GROSELLA-Restaurante	DF	Nancy Davolio
10266	GROSELLA-Restaurante	DF	Nancy Davolio
10253	Hanari Carnes	RJ	Janet Leverling
10925	Hanari Carnes	RJ	Janet Leverling
10981	Hanari Carnes	RJ	Nancy Davolio
11052	Hanari Carnes	RJ	Janet Leverling
10415	Hungry Coyote Import Store	OR	Michael Suyama
10375	Hungry Coyote Import Store	OR	Michael Suyama
10394	Hungry Coyote Import Store	OR	Michael Suyama

***Рис. 2-18. Ни атрибут Region, ни атрибут Salesperson не являются ключами-кандидатами, хотя и полностью зависят** друг от друга*

Расскажу о третьей нормальной форме подробней. Например, для большинства населенных пунктов вы можете задать в качестве идентификатора величину *PosatalCode* (Почтовый индекс), основанную на сочетании величин *City* (Город) и *Region* (Район), так что отношение, показанное на рис. 2-19, не находится в третьей нормальной форме.

CompanyName	Address	City	Region	PostalCode
Alfreds Futterkiste	Obere Str. 57	Berlin		12209
Ana Trujillo Emparedados y helados	Avda. de la Constitución 2222	México D.F.		05021
Antonio Moreno Taquería	Mataderos 2312	México D.F.		05023
Around the Horn	120 Hanover Sq.	London		WA1 1DP
Berglunds snabbköp	Berguvsvägen 8	Luleå		S-958 22
Blauer See Delikatessen	Forsterstr. 57	Mannheim		68306
Blondel père et fils	24, place Kléber	Strasbourg		67000
Bolide Comidas preparadas	C/ Araquil, 67	Madrid		28023
Bon app'	12, rue des Bouchers	Marseille		13008
Bottom-Dollar Markets	23 Tsawassen Blvd.	Tsawassen	BC	T2F 8M4

Рис. 2-19. Отношение не находится в третьей нормальной форме

Два отношения на рис. 2-20 технически более корректны с точки зрения теории нормализации, чем то, что на рис. 2-19. На самом деле единственным преимуществом, которое вы получите в результате произведенной декомпозиции исходного отношения, будет возможность автоматического поиска нужного почтового индекса, избавляющая пользователей от нескольких лишних нажатий клавиш. Это не такое уж незначительное преимущество, к тому же достичь его можно и иными способами, не требующими выполнять соединение отношений каждый раз, когда нужно получить адрес.

CompanyName	Address	City	Region
Bottom-Dollar Markets	23 Tsawassen Blvd.	Tsawassen	BC
Comércio Mineiro	A» dos Lusíadas, 25	São Paulo	SP
Familia Arquibaldo	Rua Orós, 92	São Paulo	SP
Gourmet Lanchonetes	Av. Brasil, 442	Campinas	SP
Great Lakes Fco: Market	2732 Baker Blvd	Eugene	OR
GROSELLA-Restaurante	5ª Ave. Los Palos Grandes	Caracas	OF
Hanrn Carnas	Rua do Paço, 67	Rio de Janeiro	RJ
HILARIÓN-Abastos	Carrera 22 con Ave. Carlos Soublette #8-35	San Cristobal	Táchira
Hungry Coyote Import Store	City Center Plaza	Elgin	OR
Island Trading	Garden House	Cowes	Isle of Wight

City	Region	PostalCode
Tsawassen	BC	T2F 8M4
São Paulo	SP	05432-043
São Paulo	SP	05442-030
Campinas	SP	04876-786
Eugene	OR	97403
Caracas	DF	1081
Rio de Janeiro	RJ	05454-876
Sin Cristóbal	Táchira	5022
Elgin	OR	97827
Cowes	Isle of Wight	PO31 7PJ

Рис. 2-20. Эти два отношения находятся в третьей нормальной форме

Решение, когда и как вводить в модель третью нормальную *форму,* может приниматься только с учетом семантики строящейся модели (впрочем, как и любое другое решение в ходе построения модели данных). Как правило, отдельные отношения проектируются только для моделирования наиболее важных сущностей предметной области, или когда предполагается частое изменение каких-то данных, или когда вы получаете в результате создания конкретного отношения серьезные технологические выгоды. Почтовые индексы изменяются, но не часто; кроме того, они не являются существенно важными элементами большинства систем.

Дальнейшая нормализация

Первые три нормальные формы были введены доктором Коддом в его оригинальной работе по реляционной теории, и в подавляющем большинстве случаев это все, о чем вам следует беспокоиться. Вспоминаю пословицу, которую я слышала в университете: «Ключ, целый ключ и ничего кроме ключа, и да поможет мне Кодд!»

Другие нормальные формы — Бойса-Кодда, четвертая и пятая, были разработаны для специальных случаев, которые редко встречаются на практике.

Нормальная форма Бойса-Кодда

Нормальная форма Бойса-Кодда рассматривается как вариант третьей нормальной формы. Она имеет дело со специальной разновидностью отношений, для которых существует несколько ключей-кан-

дидатов. Фактически, чтобы применять нормализацию по Бойсу-Кодду, нужно сочетание нескольких условий:

- отношение должно иметь не менее двух ключей-кандидатов;
- по крайней мере два ключа-кандидата должны быть составными;
- ключи-кандидаты должны иметь перекрывающиеся атрибуты.

Простейший способ понять, что такое нормальная форма Бойса-Кодда — использовать функциональные зависимости. Нормальная форма Бойса-Кодда, в сущности, равносильна утверждению, что между ключами-кандидатами отношения нет функциональных зависимостей[1]. Возьмем, к примеру, отношение, изображенное на рис. 2-21. Оно находится в третьей нормальной форме (если предположить, что имя поставщика уникально), но все еще содержит существенно избыточные данные.

SupplierID	SupplierName	ProductID	Quantity	UnitPrice
5	Cooperativa de Quesos 'Las Cabras'	11	12	$14.00
14	Formaggi Fortini s r l	72	5	$34.80
20	Leka Trading	42	10	$9.80
6	Mayumi's	14	9	$18.60
24	G'day, Mate	51	40	$42.40

Рис. 2-21. ***Отношение находится в третьей нормальной форме,*** *но* ***не удовлетворяет нормальной форме Бойса-Кодда***

В данном случае имеется два ключа-кандидата *{SupplierID, ProductID}* и *{SupplierName, ProductID}*, а диаграмма функциональной зависимости показана на рис. 2-22[2].

[1] К сожалению, это утверждение неточно. Обычно пользуются более формальным определением: отношение R находится в нормальной форме Бойса-Кодда тогда и только тогда, когда для любой нетривиальной функциональной зависимости {X} а {Y} между атрибутами R множество атрибутов {X} (называемое детерминантом функциональной зависимости) всегда содержит какой-нибудь из ключей-кандидатов. При этом на {Y} не налагается никаких ограничений. Поэтому в нормальной форме Бойса-Кодда ключи-кандидаты могут функционально зависеть друг от друга или часть атрибутов одного ключа-кандидата может функционально зависеть от другого ключа-кандидата. — *Прим. перев.*

[2] На диаграмме есть ошибка и отсутствует важная деталь. Вертикальная стрелка между ключами-кандидатами должна быть двунаправленной, как и между атрибутами *SupplierID* и *SupplierName.* Это означает, что существуют две функциональные зависимости: *{SupplierID}* → *{SupplierName}* и *{SupplierName}* → *{SupplierID}*. Ни одна из этих функциональных зависимостей не является тривиальной и не содержит ключа-кандидата в детерминанте, следовательно, согласно данной выше формулировке, каждая из этих зависимостей нарушает условие Бойса-Кодда. — *Прим. перев.*

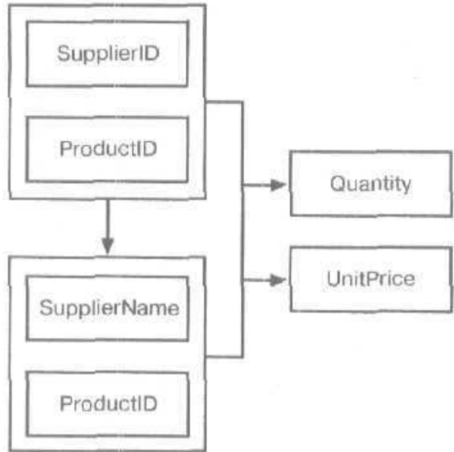


Рис. 2-22. Диаграмма функциональной зависимости для отношения, показанного на рис. 2-21

Как вы можете видеть, существует функциональная зависимость {SupplierID} →{SupplierName}, которая нарушает нормальную форму Бойса-Кодда. Корректная модель показана на рис. 2-23.

Отношение *Suppliers*

SupplierID	SupplierName
20	Leka Trading
5	Cooperativa de Quesos 'Las Cabras'
14	Formaggi Fortini s.r.l.
&	Mayumi's
24	G'day, Male

Отношение *Products*

SupplierID	ProductID	Quantity	UnitPrice
20	42	10	$9.80
5	11	12	$14.00
14	72	5	$34.80
6	14	9	$18.60
24	51	40	$42.40

Рис. 2-23. Полностью нормализованный вариант модели, изображенной на рис. 2-21

Нарушения нормальной формы Бойса-Кодда легко избежать, если уделить внимание логическому смыслу проектируемого отношения. Если отношение на рис. 2-21 содержит информацию о производимой продукции, то туда не следует включать информацию о поставщике (разумеется, верно и обратное утверждение).

Четвертая нормальная форма

Четвертая нормальная форма подводит теоретическую базу под интуитивно очевидный принцип: независимые повторяющиеся группы данных не следует размещать в одном и том же отношении. Предположим, что собственная продукция компании «Northwind Traders» упаковывается в тару нескольких размеров, что эта тара поставляется несколькими поставщиками, и что каждый поставщик обеспечивает компанию тарой всех необходимых размеров. Ненормализованный вариант отношения *Products* мог бы выглядеть так, как показано на рис. 2-24.

ProductName	SupplierName	PackSize
Chai	Exotic Liquids	8 oz, 16 oz, 32 oz
Chef Anton's Cajun Seasoning	New Orleans Cajun Delights	8 oz, 16 oz, 32 oz
Pavlova	Pavlova, Ltd	8 oz, 16 oz, 32 oz

Рис. 2-24. ***Отношение не является нормализованным***

Первым шагом нормализации будет устранение составного атрибута *PackSize*, в результате чего мы получим отношение, показанное на рис. 2-25.

ProductName	SupplierName	PackSize
Chai	Exotic Liquids	16 oz
Chai	Exotic Liquids	12 oz
Chai	Exotic Liquids	8 oz
Chef Anton's Cajun Seasoning	New Orleans Cajun Delights	16 oz
Chef Anton's Cajun Seasoning	New Orleans Cajun Delights	12 oz
Chef Anton's Cajun Seasoning	New Orleans Cajun Delights	8 oz
Pavlova	Pavlova, Ltd.	16 pr
Pavlova	Pavlova, Ltd.	12 oz
Pavlova	Pavlova, Ltd	8 oz

Рис. 2-25. ***Вариант отношения, показанного на рис.*** *2-24,* *в нормальной* ***форме Бойса-Кодда***

Забавно то, что отношение на рис. 2-25 находится в нормальной форме Бойса-Кодда, так как полный набор ее атрибутов образует ключ. Однако данные в отношении явно избыточны, и поддержание их целостности может показаться ночным кошмаром тому, кто этим будет заниматься. Решение проблемы базируется на концепции многозначных зависимостей и четвертой нормальной формы.

Многозначная зависимость — это два совершенно независимых множества атрибутов. На рис. 2-24 многозначная зависимость записывается как {ProductName} ->> {PackSize} | {SupplierName}, что означает: «Существуют две множественные зависимости поставщиков упаковки и размеров упаковки от наименования продукта». Упрощен-

но, нормализация до четвертой нормальной формы состоит в выделении многозначных зависимостей в разные отношения (рис. 2-26). Формально отношение находится в четвертой нормальной форме, если оно находится в нормальной форме Бойса-Кодда, и кроме того, все многозначные зависимости являются также функциональными зависимостями от ключей-кандидатов.

Приведение отношения к четвертой нормальной форме актуально, только если между атрибутами существуют многозначные связи. Если бы каждый продукт упаковывался в тару только одного размера, или вся тара поступала от единственного поставщика, приведения к четвертой нормальной форме не потребовалось бы. Точно так же, если два множества атрибутов не являются совершенно независимыми, это скорее всего говорит о том, что отношение не находится во второй нормальной форме.

ProductName	PackSize
Chai	16 oz
Chai	12 oz
Chai	8 oz
Chef Anton's Cajun Seasoning	16 oz
Chef Anton's Cajun Seasoning	12 oz
Chef Anton's Cajun Seasoning	8 oz
Pavlova	16 oz
Pavlova	12 oz
Pavlova	8 oz

ProductName	SupplierName
Chai	Exotic Liquids
Chai	Exotic Liquids
Chai	Exotic Liquids
Chef Anton's Cajun Seasoning	New Orleans Cajun Delights
Chef Anton's Cajun Seasoning	New Orleans Cajun Delights
Chef Anton's Cajun Seasoning	New Orleans Cajun Delights
Pavlova	Pavlova, Ltd.
Pavlova	Pavlova, Ltd.
Pavlova	Pavlova, Ltd.

Рис. 2-26, Отношения, содержащие многозначные зависимости, должны быть подвергнуты декомпозиции

Пятая нормальная форма

Пятая нормальная форма имеет дело с чрезвычайно редко встречающимся случаем *зависимостей соединения.* Зависимости соединения подчиняются следующему принципу: «Если сущность 1 зависит от сущности 2, сущность 2 зависит от сущности 3, а сущность 3 в свою очередь зависит от сущности 1, то все три сущности обязательно должны входить в один и тот же кортеж».

Если перевести это на нормальный язык, то получится следующее: «Если поставщик поставляет данный товар своим заказчикам и определенный заказчик заказывает данный товар у поставщиков, а упомянутый поставщик поставляет нечто упомянутому заказчику (то есть заказчик иногда заказывает что-то у поставщика), то это значит, что данный поставщик поставляет упомянутый товар упомянутому заказчику».

В общем случае это рассуждение ошибочно. Поставщик не обязательно поставляет заказчику именно этот конкретный товар, он может поставлять ему другие имеющиеся у него продукты. Зависимость соединения существует, только когда на модель налагается дополнительное ограничение, делающее подобное рассуждение справедливым. Зависимость соединения в нашем случае можно выразить так: «Если поставщик поставляет товар, этот товар интересует заказчика, и заказчик работает с поставщиком, то заказчик непременно получает товар у поставщика».

В такой ситуации недостаточно ограничиться созданием одного-единственного отношения {*Supplier*, *Product*, *Customer*} (Поставщик, Продукт, Покупатель), так как при обновлении данных могут возникнуть проблемы, связанные с избыточностью. Например, если рассматривать связь, изображенную на рис. 2-27 и придерживаться сформулированного выше принципа о связи поставщика, продукта и заказчика, то внесение записи {Ma Maison, Aniseed Syrup, Berglunds snabk..op} потребует также внесения второй записи {Exotic Liquids, Aniseed Syrup, Berglunds snabk..op}. В самом деле, заказчик «Berglunds snabk..op» решил, что ему не хватает продукта *Aniseed Syrup* (анисовый сироп). Он решил покупать его у поставщика *Ma Maison* (Мамаша Мейсон). Но его поставщиком уже является компания «Exotic Liquids» («Экзотические напитки»). Следовательно, согласно сформулированному принципу, придется «Экзотическим напиткам» поставлять «Berglunds snabk..op» еше и анисовый сироп.

Supplier	Product	Customer
Exotic Liquids	Aniseed Syrup	Alfreds Futterkiste
Exotic Liquids	Chef Anton's Cajun Seasoning	Berglunds snabbköp

Рис. 2-27. Данное отношение не находится в пятой нормальной форме

Декомпозиций на три различных отношения: *Supplier Product, ProductCustomer и SupplierCustomer* (Поставщик — Продукт, Продукт - Покупатель и Покупатель — Поставщик) устраняет проблему избыточности, но приводит к возникновению иных трудностей. Если вам нужно будет получить исходные данные, то придется соединить между собой все три новых отношения. Если попытаться соединить толь-

ко два отношения, в результате будут получены совершенно неверные данные.

С точки зрения системного архитектора, это сложная проблема. Вообразим, что пользователь решил получить данные о поставщиках, продуктах и заказчиках. Он может составить запрос, основанный на соединении любых двух таблиц из трех, выполнить его и получить неверные результаты (если не предпринять специальных мер, основанных на использовании встроенных механизмов безопасности СУБД). Тем не менее, полученный набор данных будет выглядеть совершенно корректным, и пользователь вряд ли обнаружит ошибку.

К счастью, такая циклическая зависимость соединения встречается столь редко, что в большинстве случаев ее можно игнорировать без каких-либо последствий. Когда нельзя избежать подобного рода проблем, лучше построить базу данных таким образом, чтобы соединения между множеством отношений не нарушали целостности данных.

Итоги

Мы рассмотрели структуру базы данных в терминах процесса нормализации. Базовым принципом, положенным в основу нормализации, является принцип декомпозиции без потерь — возможность «расщепить» отношение на несколько других, не потеряв при этом имеющуюся информацию. Формальным выражением этого принципа являются нормальные формы. Первые три нормальные формы, которые наиболее часто используются, соответствуют поговорке «Ключ, целый ключ и ничего кроме ключа, и да поможет мне Кодд». Оставшиеся гри нормальные формы используются в исключительных случаях.

В следующей главе, когда мы будем рассматривать моделирование связей между сущностями, я расскажу об операции логического соединения отношений.

Связи

В главе 2 мы подробно рассмотрели процесс нормализации модели данных. Суть этого процесса — в анализе сущностей предметной области для моделирования отношений, охватывающих все относящиеся к ним данные. Но отношения оставляют лишь определенную часть модели данных. Ее другая неотъемлемая часть — связи между отношениями и ограничения, налагаемые на эти связи. Эта глава посвящена моделированию связей между сущностями. Ее материал должен быть для вас достаточно прост и понятен, если вы знакомы с семантикой модели данных. Есть и некоторые особые случаи, которые не укладываются в рамки модели связей. Их мы рассмотрим отдельно.

Основные понятия и определения

Прежде всего, дадим определения основных терминов. Сущности, между которыми существуют связи, называются *участниками* (participants), а число участников связи — *размерностью* (degree) связи. Большинство связей между сущностями — это *двойные связи,* то есть такие, в которых участвуют две сущности. Встречаются также *унарные связи* (в которых сущность связана сама с собой) и *тройные связи* (в которых участвуют три сущности). Большинство примеров, приведенных в этой главе, иллюстрируют двойные связи. Унарные и тройные связи — это особые случаи, которые мы рассмотрим отдельно.

Участие каждой сущности в связи бывает *полным* или *частичным,* в зависимости от того, может ли эта сущность существовать, если данная связь не определена. Например, для связи двух сущностей — *Customer* (Покупатель) и *Order* (Счет), участие сущности *Customer* будет частичным, поскольку информацию о покупателе можно вводить, до того как покупателю будет выписан первый счет. Участие сущности *Order* в связи «Customer — Order» будет полным, поскольку любой счет можно выписать только какому-либо покупателю.

Тот же самый принцип часто используется при классификации сущностей как *слабых* (то есть таких, участие которых в связи является полным) или *обычных* (участие их в связи является частичным). Слабые сущности могут существовать только при наличии связей с другими сущностями, в то время как обычные сущности существуют независимо от наличия связей между ними и другими сущностями. Подобная классификация положена в основу метода составления диаграмм «сущности — связи», предложенного Ченом (Chen).

Итак, связи можно классифицировать одним из трех возможных способов: как полные или частичные, необязательные или обязательные, а также в терминах слабых и обычных сущностей. Подобная классификация всех связей не является строго обязательной при создании модели данных. И все же если при моделировании больших сложных систем указано влияние отношения на сущность, задача разработчика заметно облегчается. Чтобы указать на диаграмме «сущности — связи», является ли сущность слабой или обычной, вы можете использовать обозначения, показанные на рис. 3-1.

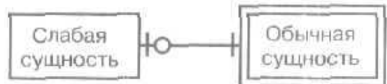


***Рис. 3-1. Различия** между слабыми **и** обычными сущностями*

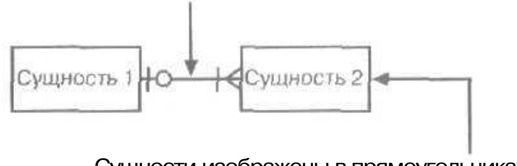
В некоторых случаях полезно разделить все связи на два типа — *IsA* («является А» или «входит в А») и *HasA*(«обладает А»). Принцип такого разделения весьма прост, мы проиллюстрируем его на примере сущностей B и C: B либо *IsA* C, либо *HasA* C. Например, для сущностей *Employee* (Сотрудник) и *BasketballTeam* (Баскетбольная команда) может быть определена связь *Employee IsA BasketballTeam* (сотрудник является членом баскетбольной команды); для сущностей *Employee* и *Address* (Адрес) может быть определена связь *Employee HasA(n) Address.* Конечно, английские термины «Is» и «Has» не всегда правильно описывают природу связи между сущностями. Например, если связь между *Employee* и *SalesOrder* (Счет-фактура) определена как *Employee HasA SalesOrder,* это отнюдь не означает, что сотрудник обладает счетом — на самом деле он выписывает счет. Но все же при семантическом анализе очевидно, что связь между *Employee* и *SalesOrder* не может быть *Employee IsA SalesOrder* — это допущение слишком далеко от реальности.

Кроме того, классификация участия в связи подразумевает указание *обязательности* данной связи: является ли для сущности участие в данной связи обязательным. Здесь возникают сложности, поскоaccess

ку реализация проектируемой системы, осуществляемая механизмом СУБД, существенно отличается от домена предметной области. Мы увидим, какие трудности могут при этом возникнуть, когда перейдем к рассмотрению методов реализации целостности данных в главе 4.

Максимальное число экземпляров одной сущности, которое может быть связано с экземпляром другой сущности, мы будем называть *мощностью* данной связи. (Следует отметить, что применительно к связям понятия «мощность» и «размерность» имеют несколько иной смысл, чем применительно к отношениям.) Существуют три основных разновидности мощности связей: «один к одному», «один ко многим» и «многие ко многим».

Для указания мощности и обязательности связей я использую условные обозначения, показанные на рис. 3-2. На мой взгляд, техника представления связей между сущностями «птичья лапа» (о которой рассказывалось в главе 1) — наиболее простой и понятный из всех способов представления, им удобно пользоваться при обсуждении проекта системы с заказчиком. Конечно, существуют и другие методы представления, имеющие свои преимущества, и вы можете выбрать тот, который вам больше нравится.



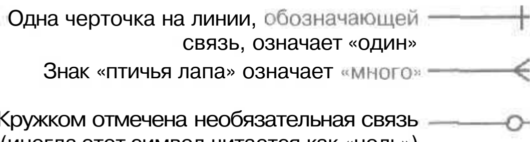


Рис. 3-2. Так обозначают обязательность и мощность связей па диаграммах

Моделирование связей

После того как вы определили, что данная связь существует в системе, приступайте к ее моделированию, включая атрибуты одного отношения (*ссылочное отношение,* primary relation) в другое (*ссылающееся отношение,* foreign relation), как показано на рис. 3-3.

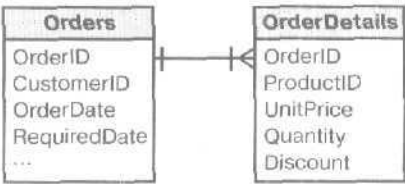


Рис. *3-3. Атрибуты* ***ссылочного отношения (Orders)*** *включаются в ссылающееся* ***отношение*** *(OrderDetails)*

Вы, несомненно, заметили некоторые различия между этой и формальной диаграммой «сущности — связи» на рис. 1-6. Во-первых, атрибуты не представлены как отдельные объекты. На этом этапе проектирования нас интересуют связи между сущностями, а не их структура. Обозначение же на диаграмме атрибутов сущностей усложняет схему, перегружает ее лишними деталями.

По тем же причинам на диаграмме не указан и тип связей. Я считаю это излишним, поскольку описание связи зависит от того, с какой стороны вы начинаете ее рассматривать (сравните: «преподаватели читают лекции студентам» и «студенты слушают лекции преподавателей»). Но хотя я никогда не обозначаю тип связей на диаграмме, все же иногда указываю атрибут, который будет использоваться для реализации связи в схеме базы данных. Это удобно, когда, например, у ссылочной сущности более одного ключа-кандидата, и вы хотите явно указать, какой из этих ключей будет использоваться.

Как я уже упоминала, данный стиль диаграмм очень удобен при обсуждении с клиентами деталей реализации системы. Такие диаграммы легко строить вручную с помощью различных средств построения диаграмм, например Visio Professional или Micrografx Flowcharter 7. В состав Microsoft Access, Microsoft SQL Server и Microsoft Visual Basic также входят средства построения диаграмм, которыми вы можете воспользоваться вместо или дополнительно к описанной здесь методике.

Использование интерактивных инструментов: окна связей между сущностями в Access (для файла .mdb механизма базы данных Microsoft Jet) или диаграмм баз данных (в том случае, если база данных реализуется средствами SQL Server). — обладает очевидными преиму-

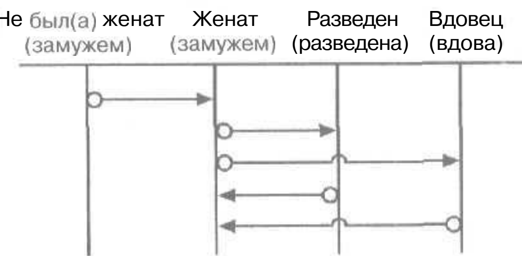


Рис. 3-5. Диаграмма состояний перехода: возможные варианты изменения семейного положения

Но если разрабатываемая модель должна отражать лишь текущее семейное положение, реализовать сущности абстрактного отношения, чтобы проследить изменения семейного положения и проверить их правильность, совершенно излишне. Если же все-таки для вас важна информация, что Джон и Мери Смит поженились в 1953 и развелись в 1972 г., после чего Мери снова вышла замуж в 1975 и овдовела в 1986 г., то потребуется использовать сущности абстрактного отношения.

Связи «один к одному»

Это, пожалуй, самый простой тип связей. Связи «один к одному» между сущностями X и Y — это такие связи, при которых каждый экземпляр сущности X может быть связан только с одним экземпляром сущности Y. Большинство связей типа *IsA* являются связями «один к одному», другие примеры весьма редки. Если при проектировании системы вы решили выбрать связь «один к одному» между некоторыми сущностями, убедитесь, что эта связь не будет изменяться на протяжении всего срока эксплуатации системы, или что подобное изменение окажется несущественным и эта связь не будет интересовать вас в дальнейшем. Предположим, вы создаете базу данных служебных помещений в некоем здании, где в каждом кабинете находится только один сотрудник. Такая *связь* будет связью «один к одному» (рис. 3-6).

Рис. 3-6. Между сущностями Office (кабинет) и Employee (сотрудник) существует связь «один к одному»

Но подобная связь между сущностями *Office* (Кабинет) и *Employee* (Сотрудник) будет правильной лишь до определенного момента времени. Очевидно, что рано или поздно в одном кабинете соберется несколько сотрудников. (Расположение кабинетов в здании также может измениться, но мы сейчас не берем это во внимание). Используя связь «один к одному», как показано на рис. 3-6, вы получите простую модель служебных помещений, имеющихся в. здании, однако проследить историю перемещения сотрудников из одного кабинета в другой в рамках этой модели не сможете. Возможно, подобная проблема вас просто не волнует. Например, если вы разрабатываете систему для рассылки корреспонденции, все. что необходимо знать - куда направлять корреспонденцию, адресованную Джейн Ду (Jane Doe), а информация о том, куда направлялась эта корреспонденция три месяца тому назад, для вас совершенно бесполезна. Но если вы проектируете систему для агентства, работающего с недвижимостью, информация о том, кто и когда занимал данное помещение, окажется очень нужной для составления различных отчетов — надо же знать, насколько часто сменяются арендаторы помещения.

Хотя связи «один к одному» в реальном мире встречаются довольно редко, в проектировании баз данных они широко используются, например, чтобы уменьшить число атрибутов в отношении, а также при моделировании подклассов сущностей. Существует ограничение числа полей в таблице: для механизма **СУБД** Microsoft Jet оно не должно превышать 255, а для SQL Server — 250. Обычно модели данных не выходят за рамки этих ограничений, поскольку таблицы с таким огромным числом полей встречаются крайне редко. Однако мне приходилось сталкиваться с системами (они были предназначены для медицинских и научных учреждений), где у одной сущности имелось более 255 уникальных атрибутов. И у разработчиков не было иного выбора, как создать новое отношение с некоторым произвольным подмножеством атрибутов и определить связи «один к одному» между этим и исходным отношением.

Другая область, где модель данных часто содержит сущность с чересчур большим числом атрибутов — это составление моделей тестов и опросных листов. Например, если нужно разработать тест с произвольным числом вопросов, велико искушение смоделировать ответы каждого тестируемого (рис. 3-7).

Такую структуру проще всего реализовать; однако она далека от идеала. Атрибуты сущности *Answer* (Ответ) представляют собой повторяющуюся группу, и поэтому нельзя считать, что отношение находится в первой нормальной форме. Модель на рис. 3-8 намного удачней.

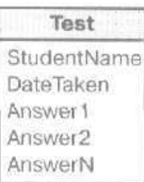


Рис. 3- 7. Данная структура, используемая для создания моделей тестов и опросных листов, далеко не идеальна

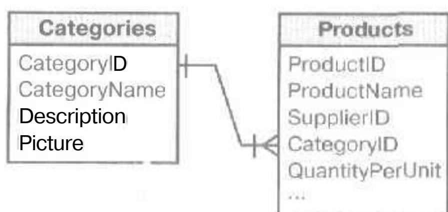


***Рис. 3-8. Эта структура, хотя сложней с точки зрения реализации,** лучше подходит для моделирования тестов и опросных листов*

Реализация сущностей как классов-наследников

Более интересный случай применения отношений «один к одному» — классы-наследники для сущностей. Эта концепция заимствована из объектно-ориентированного программирования. Чтобы наглядней представить себе преимущества классов-наследников, сначала рассмотрим наиболее простой и общеизвестный пример их применения. В базе данных *Northwind*, которая входит в комплект поставки Microsoft Access, каждый продукт связан с категорией продукта (рис. 3-9).

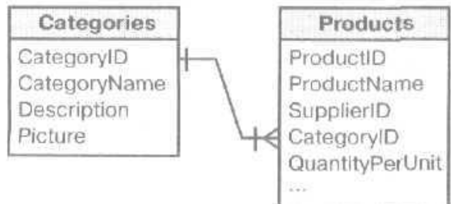


Рис. 3-9. В базе данных Northwind каждый продукт связан с категорией продукта

Наличие связи с отношением *Categories* позволяет сгруппировать продукты, например, для получения отчетов. Однако при реализации системы, представленной на рис. 3-9, вы можете иметь дело только с

продуктом как таковым, а не с сущностью его отдельной категории. Все атрибуты, определенные для отношения *Products,* хранятся совместно для всех продуктов, независимо от их типов. Этот подход не слишком подходит для описания предметной области, поскольку такие категории продуктов как напитки и специи, очевидно, должны иметь совершенно разные атрибуты.

Возможно, у вас появится искушение смоделировать список продуктов из базы данных *Northwind,* как показано на рис. 3-10. Такая модель позволяет хранить всю информацию, специфичную для отдельных видов продуктов, при этом каждый тип продукта представляет собой отдельное отношение. Но при этом затруднительно работать с продуктом как таковым, поскольку приходится иметь дело со множеством различных видов продуктов.

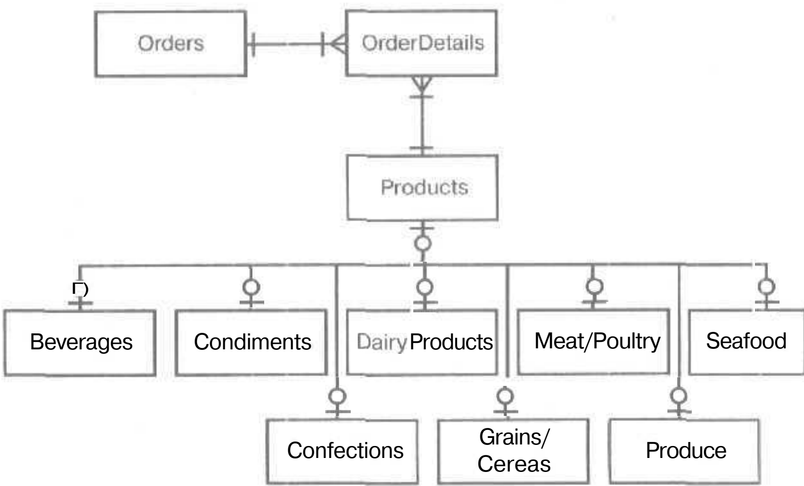


Рис. 3-Ю. Данная модель позволяет отразить все атрибуты, свойственные конкретной категории

Представьте себе, например, процесс проверки, правильно ли пользователь ввел код продукта. В данной модели он выражается следующим образом: «если данный код присутствует в отношении *x,* или в отношении *y,* или...». Это ничуть не лучше, чем запрос к повторяющимся группам данных, о которых шла речь в главе 2. Кроме того, в процессе реализации могут возникнуть нарушения целостности данных, когда некоторые атрибуты применимы только к одной категории продуктов. Например, *UnitsPerPackage*(число бутылок или другой тары е упаковке) — атрибут, применимый к категории продуктов

Beverages (Напитки), совершенно не применим к категории *Dairy-Products* (Молочные продукты) — и категория отдельного продукта изменяется. Что делать в подобной ситуации? Удалять все старые значения? А если изменение было сделано по ошибке, и пользователь хочет немедленно его отменить?

Реализация сущности *Product* (Продукт) в виде классов-наследников сочетает все положительные моменты обоих подходов. Можно выделить в отдельные структурные единицы всю информацию, относящуюся к отдельным категориям продуктов, без утраты возможности работать с продуктами как с отдельным типом там, где это необходимо. При этом удается избежать обязательного удаления данных, ставших неприменимыми, сохранив возможность удалять данные, когда вы действительно хотите это сделать, На рис. 3-11 показана модель, в которой используется реализация сущностей как классов-наследников.

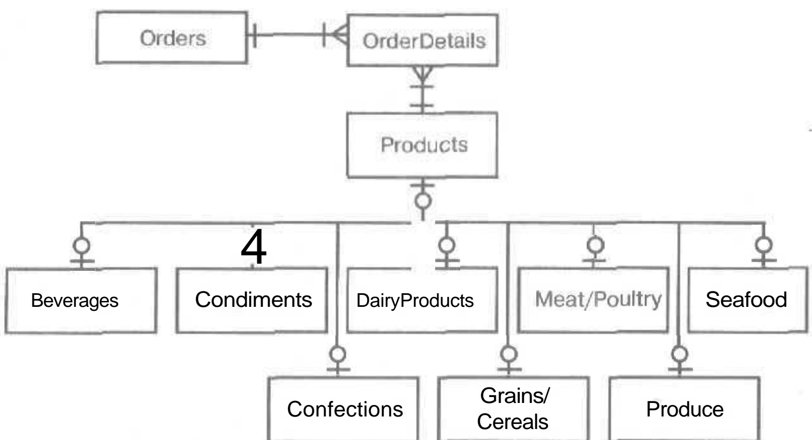


Рис. 3-11. Модель, в которой используется наследование классов, сочетает в себе все преимущества моделей, показанных на рис. 3-9 и 3-10

Бесспорно, реализация сущностей как классов-наследников — весьма изящный прием. Но ее практическая реализация может оказаться весьма непростым делом. Приведу только один пример: для отчета, который содержит сведения о продукте, требуется дополнительная условная обработка данных, чтобы отображались только поля, относящиеся к данному классу-наследнику. Задача в принципе разрешимая, но прежде чем приступать к ее реализации, нужно подумать, стоит ли вообще это делать.

В большинстве случаев я стараюсь не допускать чрезмерного упрощения модели данных лишь для того, чтобы облегчить задачу программистам. Но нельзя впадать и в другую крайность — усложнять модель, когда в этом нет никакой необходимости. Если нужно всего лишь создать возможность группировать сущности или распределять их по категориям для составления отчетов, то очевидно, излишне реализовать сущности в виде классов-наследников. В этом случае структура, представленная на рис. 3-9, будет вполне адекватной.

Часто для отношений, между которыми существуют связи «один к одному», нелегко правильно определить ссылающееся и ссылочное отношения, исходя из семантики модели данных. Если вы решили воспользоваться методом реализации сущностей как классов-наследников, то производящая сущность (generic entity) будет ссылочным отношением, а каждый из ее классов-наследников — ссылающимся.

ПРИМЕЧАНИЕ В таких случаях внешний ключ, существующий для классов-наследников, часто является ключом-кандидатом, поскольку не имеет смысла определять для классов-наследников собственные идентификаторы.

С другой стороны, если отношения «один к одному» используются с целью решить проблему с ограничениями на число полей (и следовательно, атрибутов сущности в системе), либо в предметной области между сущностями существуют уникальные отношения «один к одному», выбор ссылочного отношения зачастую произволен. Пожалуй, единственный случай, когда выбор ссылочного отношения определяется особым правилом — когда связь является необязательной. Если связь не обязательна только для одного из участников (а я еще никогда не видела такой модели, в которой она была бы необязательной для обоих участников), то именно он и будет ссылочным отношением. Другими словами, если из двух сущностей, между которыми существует связь, одна слабая, а другая — обычная, то обычная сущность будет ссылочным отношением, а слабая — ссылающимся.

Связи «один ко многим»

Наиболее распространена ситуация, когда один экземпляр сущности связан с нулевым числом, с одним или множеством экземпляров другой сущности. В большинстве случаев результатом процесса нормализации, о котором шла речь в главе 2, являются сущности, между которыми существуют связи «один ко многим».

Связи «один ко многим» — также достаточно любопытный случай, требующий внимательного отношения. Здесь легко ошибиться

при определении, для какого из участников данная связь будет необязательной. Существует довольно распространенное заблуждение, что таковой она будет для отношения, участвующего в связи со стороны «многие», однако посмотрите на рис. 3-12 и вы поймете, что это не так.

Рис. 3-12, Эта связь является необязательной в обоих направлениях

Связь между сущностями *Client* и *CustomerServiceRep* является необязательной в обоих направлениях. Это означает, что у *CustomerServiceRep* может быть произвольное число клиентов, в том числе и ни одного. Атрибут клиента *CustomerServiceRep, если таковой определен,* должен присутствовать в отношении *CustomerServiceRep.* Определение необязательного участника связи «один ко многим» важно, как для реализации, так и для эксплуатации системы. Подробнее об этом мы поговорим в главах 4 и 14, а сейчас просто запомните, что реляционная теория отнюдь не предписывает, чтобы связь была обязательной для одного из отношений, участвующих в связи «один ко многим».

Определить, какое из отношений, участвующих в связи «один ко многим», будет ссылающимся, а какое — ссылочным, очень просто. Отношение, участвующее в связи со стороны «один», всегда ссылочное, и его ключ-кандидат копируется в отношение, участвующее в связи со стороны «многие», которое является ссылающимся. Ключ-кандидат ссылочного отношения часто выступает как составная часть отношения, участвующего в связи со стороны «многие», однако не обеспечивает уникальную идентификацию кортежей ссылающегося отношения. Чтобы сформировать ключ-кандидат ссылающегося отношения, ключ ссылочного отношения следует скомбинировать с одним или несколькими другими атрибутами.

Связи «многие ко многим»

Связи «многие ко многим» часто встречаются в реальном мире, Например, студенты посещают множество лекций, причем каждый отдельный курс посещает множество студентов. Покупатели приобретают товары в разных магазинах, и в каждом магазине приобретает товары множество покупателей. Но в реляционной базе данных реализовать связь «многие ко многим» невозможно. При моделировании таких связей разработчики используют промежуточную связь «один ко многим» с каждым из отношений — участников связи «многие ко

многим» (рис. 3-13). Такое промежуточное отношение называется *промежуточной таблицей* (junction table); этот термин используется и при создании модели данных, где речь, конечно же, идет об отношениях, а не о таблицах.

Эта связь «многие ко многим»

будет представлена в модели данных так:

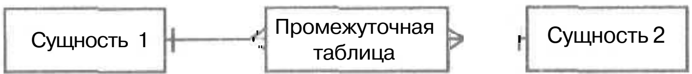


Рис. 3-13. Для моделирования связи «многие ко многим» используется промежуточная таблица

Поскольку при моделировании связи «многие ко многим» заменяются двумя связями «один ко многим», то совершенно ясно, какое из отношений будет ссылающимся, а какое — ссылочным. Как уже говорилось, отношение, участвующее в связи «один ко многим» со стороны «один», всегда является ссылочным. Это означает, что все первоначальные сущности, то есть сущности, участвующие в связи «многие ко многим», замененной двумя связями «один ко многим», в данной модели будут ссылочными отношениями, а промежуточная таблица — ссылающимся. Ключи-кандидаты промежуточной таблицы включают ключи-кандидаты первоначальных отношений, связанных с промежуточной таблицей.

Часто промежуточные таблицы состоят только из одних ключей-кандидатов отношений — участников связи «многие ко многим». На самом деле промежуточные таблицы представляют собой особый случай сущностей абстрактного отношения, которые уже упоминались ранее. Поэтому они могут содержать дополнительные атрибуты, необходимые для моделирования данной системы.

Унарные связи

Итак, мы подробно рассмотрели различные типы двойных связей, то есть связей, в которых участвуют два отношения. В унарных связях существует только один участник — отношение, связанное с самим собой. Классический пример — связь между сущностями *Employee* и *Manager*. Если сотрудники, не являющиеся менеджерами, подчиняются кому-либо из менеджеров, то менеджер, как правило, подчиняется самому себе, а также вышестоящему менеджеру.

Принципы моделирования унарных связей не отличаются от принципов моделирования связей с двумя участниками — ключ-кандидат ссылочного отношения добавляется в ссылающееся. Единственное различие в том, что и ссылочное, и ссылающееся отношения в данном случае — это одно и то же отношение. Таким образом, если для отношения *Employee* существует ключ-кандидат *EmployeeID* (Идентификатор сотрудника), определенный на домене *EmployeeID,* то вам придется добавить к отношению атрибут *ManagerID* (Идентификатор менеджера), также определенный на домене *EmployeeID,* как показано на рис. 3-14.

Рис. 3-14. Унарная связь существует, когда отношение связано с самим собой

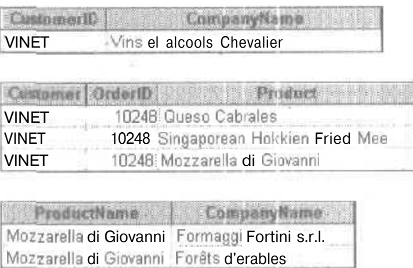
Унарные связи могут иметь любую мощность. Унарные связи «один ко многим» помогают реализовать иерархии. Пример — организационная структура компании, ясно прослеживаемая в связи *Employee-Manager.* Унарные связи «многие ко многим», как и двойные связи этого типа, реализуются с помощью промежуточных таблиц. Такие связи могут быть необязательными для одной из участвующих в них сторон (рис. 3-14). Так, в большинстве организаций у генерального директора нет вышестоящего менеджера (акционеров компании мы в расчет не принимаем).

Тройные связи

Как правило, тройные связи представляют собой связи вида *X выполняет Y для Z.* Как и связи «многие ко многим», их невозможно непос-

редственно моделировать в реляционной базе данных. Но для тройных связей не существует единых правил моделирования, и в этом их отличие от отношений «многие ко многим».

Рассмотрим пример на рис. 3-15. Продукт *Mozzarella di Giovanni*, который закупает компания «Vins et alcools Chevalier», поставляют две компании — «Formaggi Fortini s.r.l.» и «Forêts d'erables». Однако невозможно узнать, какой именно из поставщиков поставил определенную партию этого продукта компании «Vins et alcools Chevalier». Иными словами, тройная связь в модели данных утеряна. Однако при разработке моделей баз данных следует учитывать, что поставщики не просто поставляют имеющиеся у них в наличии продукты — они поставляют их конкретным клиентам.



CustomerID	CompanyName
VINET	Vins el alcools Chevalier

Customer	OrderID	Product
VINET	10248	Queso Cabrales
VINET	10248	Singaporean Hokkien Fried Mee
VINET	10248	Mozzarella di Giovanni

ProductName	CompanyName
Mozzarella di Giovanni	Formaggi Fortini s.r.l.
Mozzarella di Giovanni	Forêts d'erables

Рас. 3-15. Отношения не позволяют определить, у какого поставщика компания « Vins et alcools Chevalier» приобрела продукты

Чтобы полностью разъяснить ситуацию, рассмотрим отношения в типичной проблемной области (рис. 3-16). На этой диаграмме каждый продукт поставляется только одним поставщиком, и поддерживается тройная связь. Вы всегда можете узнать, какой именно поставщик поставил определенный продукт.

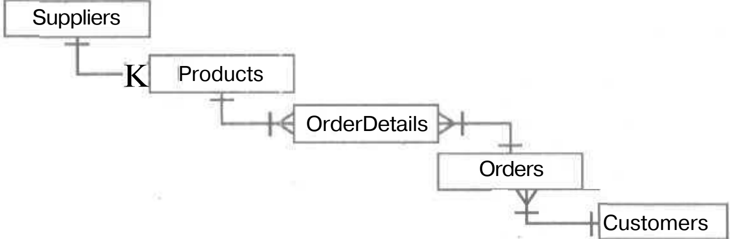
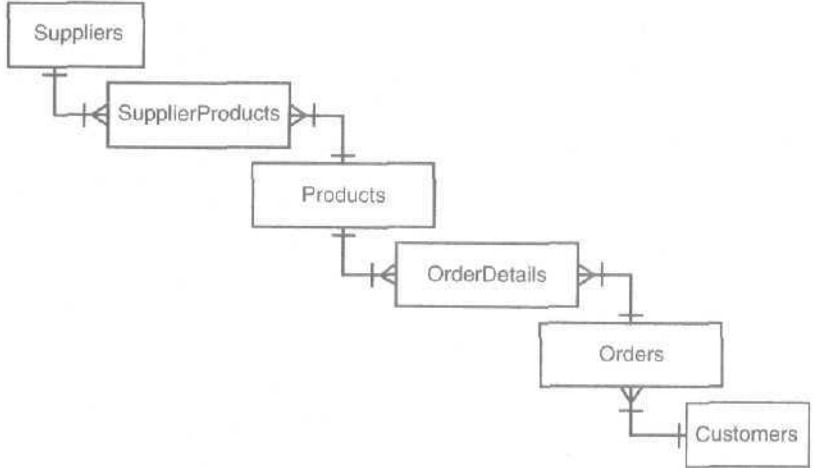


Рис. 3-16. Типичная цепочка связей между сущностями, используемыми для моделирования процесса заказа

Посмотрим теперь на рис. 3-17: каждый продукт может быть получен от разных поставщиков, и тройная связь отсутствует.



Рис, 3-17. Тройная связь отсутствует

Один из способов решить проблему — выяснить направление связи «один ко многим». Для каждой отдельной сущности, участвующей в связи со стороны «многие», можно однозначно определить соответствующую сущность, участвующую в связи со стороны «один». Так, для каждого конкретного значения атрибута *OrderDetails* (рис. 3-16) можно определить, к какому из значений атрибута *Orders* он принадлежит; зная значение *Orders,* нетрудно определить значение *Customers.* Точно так же выполнимы эти действия и в обратную сторону: зная значение *OrderDetails,* можно определить, какой именно продукт был заказан и кто являлся его поставщиком.

Однако в обратном направлении связи «один ко многим» подобный подход применить не удастся, поскольку каждой сущности, участвующей в связи со стороны «один», невозможно сопоставить единственную сущность, участвующую в связи со стороны «многие». Рассмотрим пример на рис. 3-17. По определенному значению атрибута *OrderDetails* невозможно определить, какой продукт был заказан. Если же это известно, нельзя определить, с какой именно сущностью *SupplierProducts*связан этот продукт, или другими словами, выяснить, кто его поставил.

Отсюда следует упрощенное правило: в цепочке связей нельзя менять направления связей с «один ко многим» на «многие к одному»

более одного раза. В цепочке связей на рис. 3-16 направление изменяется только один раз, на сущности *OrderDetails*. В цепочке связей на рис. 3-17 направление меняется дважды — один раз на сущности *OrderDetails*, а второй — на *SupplierProducts*.

Проблему можно решить, исключив сущность *Products* из цепочки связей (рис. 3-18).

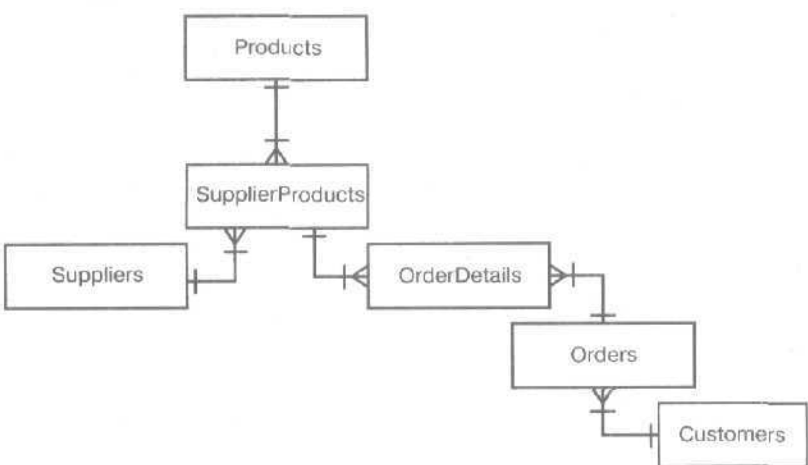


Рис. 3-18. Тройные связи сохранены

В получившейся цепочке направление связи меняется только один раз — на сущности *OrderDetails*, и таким образом связь поддерживается. При этом сущность *Products* отнюдь не исключена из модели, Скорее всего, заказы будут оформляться на продукты (сущность *Products*), а не на поставщиков (сущность *SupplierProducts*), и сущность *Products* позволит учитывать это при разработке пользовательского интерфейса.

Конечно, может быть, для вашей предметной области тройные связи не играют существенной роли. Или же тройная связь не нужна, так как вы применяете иной способ проследить необходимые связи (например, с помощью номера на коробке с расфасованным товаром). Важно, что модель на рис. 3-18 ничуть не «лучше» и не «правильнее» модели на рис. 3-17. Выбор модели определяется прежде всего семантикой предметной области.

Связи определенной мощности

Часто число кортежей отношения, участвующего в связи «один ко многим» со стороны «многие», можно оценить заранее: известно ми-

нимальное, максимальное или точное число таких записей. Например, у школьников младших классов каждый день бывает не больше пяти уроков, скелет взрослого человека состоит из 200 костей, а на шахматном турнире каждый из участников соревнований должен сыграть определенное число партий,

В подобной ситуации появляется искушение построить модель данных таким образом, чтобы включить ключ-кандидат каждого кортежа как атрибут отношения с одной стороны (рис. 3-19). Но у такого способа два серьезных недостатка, Первый — в подобной модели существуют повторяющиеся группы атрибутов, второй — она не надежна.

StudentID:StudentNumber	FirstName:Name	LastName:Name	Per	Period	Period3:ClassPeriod	Period4:ClassPeriod
1	Nancy	Davolio	Biology	French	English	History
2	Andrew	Fuller	Physical Education	Biology	French	English
3	Janet	Leverling	Physical Education	Biology	French	English
4	Margaret	Peacock	French	Physical Education	History	English
5	Steven	Buchanan	Biology	French	Physical Education	History
6	Michael	Suyama	French	Physical Education	Biology	History
7	Robert	King	History	French	English	Biology

Рис. 3-19. Модель для связи с известной мощностью (неверный подход)

Повторяющиеся атрибуты на рис. 3-19 легко выявить по именам, однако все они определены на одном домене — *ClassPeriod*. Когда в модели несколько атрибутов определяются на одном домене, весьма вероятно, что значения категории или типа будут ошибочно приняты за имена атрибутов.

К тому же, как уже говорилось, структуры, организованные по этому принципу, не надежны. Допустим, компания проводит такую кадровую политику; за каждым менеджером закрепляется не более пяти служащих, отчитывающихся непосредственно перед ним. Увы, политика и реальность — это разные вещи. Закладывая политику в модель данных, вы тем самым реализуете ее как жесткое системное ограничение, которое не может быть изменено впоследствии. Могу поспорить, что при вводе реальных данных обнаружится, что хотя бы за одним менеджером в компании закреплено шесть служащих, а не пять. Что произойдет в этом случае? Будет ли одному из служащих назначен другой менеджер? Будет ли продублирована запись, идентифицирующая менеджера в базе данных, и соответственно увеличится ли заработная плата, начисляемая менеджеру? Или же дело ограничится тем, что сотрудник отдела информационных технологий, к которому обратился озадаченный пользователь, позвонит вам и выскажет все, что он думает о таком разработчике баз данных?

Подобные ограничения, налагаемые на мощность связи, должны реализоваться как системные и никак не влиять на структуру данных в проектируемых отношениях. Кроме того, следует очень тщательно

продумать последствия, к которым приведут ограничения на мощность связи, вне зависимости от способа их реализации (подробнее — в главе 16).

Итоги

Эта глава была посвящена связям между сущностями. Мы подробно рассмотрели каждый из видов связей: «один к одному», «один ко многим» и «многие ко многим». Проследили на примерах, как эти связи реализуются в модели данных: определяется ссылочное отношение, и ее ключ-кандидат включается в другое отношение, участвующее в связи — ссылающееся отношение. Мы рассмотрели также несколько специальных случаев — унарные и тройные связи и то, как эти типы связей реализуются в модели данных.

Теперь вы познакомились со всеми основными компонентами модели данных: сущностями, атрибутами и отношениями между ними. В главе 4 я расскажу о целостности данных и механизмах, позволяющих поддерживать базу данных в целостном состоянии.

Целостность данных

Создание сущностей и связей между ними — это только часть процесса моделирования предметной области. Вы должны записать правила, которые система будет использовать для поддержания данных, физически хранящихся в базе данных, в согласованном состоянии. Другими словами, вам нужно создать модель *целостности данных.*

Шансы гарантировать реальную корректность данных ничтожно малы. Возьмем, к примеру, запись о заказе, которая говорит, что Мэри Смит купила 17 пил 15 июля 1999 г. СУБД может проверить, что Мэри Смит и вправду является заказчиком компании, что компания и вправду торгует пилами, и что она принимала заказы 15 июля 1999 г. Можно даже убедиться, что у Мэри Смит достаточно денег на счете, чтобы заплатить за 17 пил. Чего СУБД точно не может проверить, так это того, действительно ли миссис Смит заказала 17 пил, а не семь и не одну, и что она заказала именно пилы, а не отвертки. Лучшее из того, что может сделать система: предупредить пользователя, что 17 пил — слишком большой заказ для розничного покупателя, хотя выгоды от реализации такой функции могут не оправдать затраченных усилий,

Увы, это все, что может система, и средствами современной СУБД очень легко реализовать подобную проверку. Но не существует ни базы данных, ни проектировщика, которые гарантируют, что данные в системе соответствуют действительности. Можно только утверждать, что данные подчиняются определенным *ограничениям целостности.*

Ограничения целостности

Иногда ограничения целостности отождествляют с бизнес-правилами. На самом деле, понятие «бизнес-правила» имеет значительно более широкий смысл: оно включает все ограничения модели, а не толь-

ко ограничения целостности данных. В частности, безопасность системы (то есть правила, определяющие, кто, что и при каких обстоятельствах может делать с системой) — задача системного администрирования, а не поддержания целостности данных. Однако системная безопасность — это требование бизнеса, и она включает в себя ряд бизнес-правил. Мы рассмотрим системную безопасность в главе 8.

Целостность данных реализуется на нескольких уровнях. Ограничения на домены, преобразования и сущности определяют целостность на уровне отдельных отношений. Ограничения ссылочной целостности гарантируют поддержание необходимых связей между отношениями. Ограничения базы данных управляют базой данных в целом, а ограничения целостности транзакций управляют изменением данных в одной или нескольких базах одновременно,

Целостность доменов

Как мы уже говорили, домен представляет собой множество возможных значений данного атрибута (см. главу 1). Ограничение целостности домена, чаше называемое *ограничением домена* — это правило, определяющее разрешенный для хранения набор величин. Разумеется, чтобы полностью описать домен, может потребоваться ввести несколько ограничений домена.

Домен и тип данных не одно и то же, и определение домена в терминах физических типов данных является ошибкой. Опасность в том, что придется налагать на данные дополнительные ограничения -- например, если вы выбрали целый тип данных для хранения домена, содержащего значения, не превышающие 255 (поля целого типа могут хранить значения больше 255).

Тем не менее, тип данных может служить подходящим ограничением при моделировании, поэтому выбор логического типа данных часто является первым шагом при создании ограничения домена. Под *логическим типом данных* я подразумеваю типы *date* (дата), *string* (строка), *image* (изображение). Пример, наиболее полно раскрывающий преимущества такого подхода — тип *date.* Я не рекомендую присваивать домену *TransactionDate* тип *DateTime*, который отражает физический способ хранения данных. Тем не менее, определив его как *date*, вы получите возможность сформулировать для него ограничение «домен содержит значения дат, со дня начала работы компании до настоящего момента включительно» и пренебречь другими нудными правилами.

Когда вы определите логический тип данных, может потребоваться дополнительно задать размер и точность числового типа данных или максимальную длину строкового типа. Это очень похоже на спе-

цификацию физического типа данных, но вам все еще следует мыслить категориями логической модели. Очевидно, вы не будете возражать, если я скажу, что логический тип данных «строка, состоящая не более чем из 30 символов» можно коротко записать как физический тип *char(30)*. Но чем более абстрактно вы будете описывать логическую модель, тем больше пространства для маневра у вас останется в будущем, и тем меньше вероятность, что придется налагать на систему дополнительные ограничения.

Еще один аспект целостности домена, который надо учитывать — может ли домен содержать неопределенные или несуществующие величины. Мы еще не раз будем обсуждать этот вопрос в этой и следующих главах книги. А сейчас достаточно упомянуть о разнице между несуществующей и неопределенной величинами. Также следует помнить: могут ли такие величины храниться в домене, часто (но не всегда) поддается явному определению.

На логическом уровне различие между «несуществующей» и «неопределенной» величиной понять несложно. (Помните, что модель данных — логическая конструкция.) «У моего отца нет второго имени» — пример несуществующей величины. «Я не знакома со своими соседями» — пример неопределенной величины. Вопросы реализации нас пока не касаются, но логические различия необходимо понимать.

Определяя, может ли домен содержать неопределенные или несуществующие величины, нужно решить, важно ли это для вашей системы. Возвратимся к примеру с *TransactionDate*: дата транзакции может быть неопределенной, но транзакция всегда происходит в определенный момент времени, поэтому дата транзакции не может отсутствовать. Иначе говоря, дата транзакции всегда существует, другое дело, что мы порой ее не знаем.

На данный момент договоримся считать, что любая величина может быть неизвестной. Нужно определить, может ли система хранить такую величину. Может оказаться, что нецелесообразно хранить значение, пока оно не известно, или нельзя идентифицировать сущность, пока не известна какая-то величина.

В любом случае, лучше избегать появления неизвестных величин в базе данных, хоть это и не всегда возможно при определении домена. До некоторой степени решение зависит от того, какое количество сущностей предметной области можно описать в рамках домена.

Допустим, вы определили домен *Name* и объявили, что атрибуты *GivenName* (Имя), *MiddleName* (Второе имя), *SurName* (Фамилия), и *CompanyName* (Название компании) определены в этом домене. Та-

кой подход имеет свои преимущества; общие правила (или множество правил) для атрибутов определяются в одном месте.

С другой стороны, в этом случае вы не сможете определить, допускает ли домен неопределенное или несуществующее значение, это придется указать на уровне сущностей. Но конечно, все эти атрибуты можно определить и в отдельных доменах.

Наконец, вы более точно определите набор величин, разрешенных для домена. Например, наш домен *TransactionDate* — не просто набор дат, он содержит множество дат, начиная со дня начала деятельности компании до настоящего времени. Это множество еще больше сузится, если исключить выходные, праздники и любые другие дни, в которые компания не торгует.

Иногда можно просто перечислить величины, разрешенные для домена. Домен, определяющий обычные выходные дни полностью описывается множеством {Суббота, Воскресенье}. В другом случае проще сформулировать для домена ряд правил, ограничивающих множество возможных значений, как для *TransactionDate.*

Обе технологии вполне приемлемы, хотя некоторые методики проектирования могут потребовать вполне определенных способов описания ограничений. Важно сформулировать ограничение как можно более аккуратно и полно.

Целостность на уровне переходов

Ограничения целостности переходов определяют состояния, через которые может проходить кортеж. Диаграмма «состояния-переходы» показывает состояния, через которые может проходить заказ (рис. 4-1).

Ограничения переходов помогут убедиться, что статус данного заказа никогда не изменится с «Введен» на «Выполнен», если заказ не прошел через некоторые стадии, или предотвратить изменение статуса отмененного заказа.

Состояние сущности обычно определяется значением единственного атрибута. При этом ограничение целостности переходов можно рассматривать как специальный тип ограничения целостности домена.

Иногда состояния определяются совокупностью значений нескольких атрибутов отношения или даже атрибутов нескольких отношений. Так как ограничения переходов могут существовать на любом уровне детализации, удобно рассматривать их в ходе создания модели как отдельный тип ограничений.

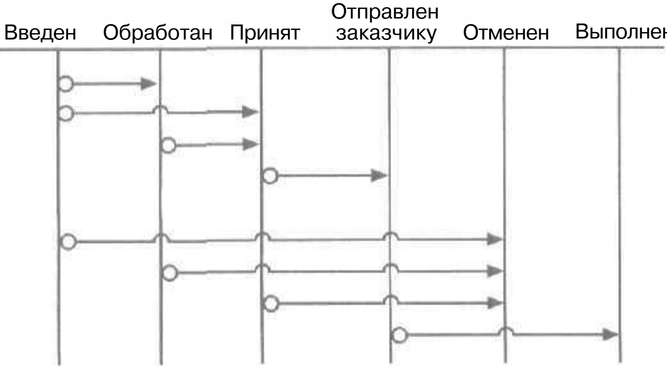


Рис. 4-1. Диаграмма состояний,** через которые может* ***проходить заказ

Например, статус заказчика может меняться только с *Normal* (Обычный) на *Preferred* (Привилегированный), если его кредит выше определенной величины и он являлся заказчиком компании не менее года. Состояние счета заказчика лучше контролировать, используя соответствующий атрибут отношения *Customers.* А длительность отношений компании с заказчиком и вовсе не нужно нигде не хранить явно, ее можно вычислять на основе даты наиболее раннего заказа, которая содержится в отношении *Orders.*

Целостность на уровне сущности

Ограничения на уровне сущности (entity constraints) направлены на обеспечение целостности моделируемой сущности. На простейшем уровне существование первичного ключа является ограничением сущности, налагающим на нее правило «каждая сущность должна быть уникальной».

В определенном смысле, это конкретное ограничение целостности на уровне сущности, все другие могут рассматриваться как ограничения целостности на уровне сущности с технической точки зрения. Ограничения, определенные на уровне сущности, могут управлять поведением отдельного атрибута, множества атрибутов, или отношения в целом.

Целостность на уровне отдельного атрибута определяется в первую очередь с помощью ограничений домена этого атрибута. Атрибут отношения наследует ограничения целостности своего домена. На уровне сущности эти наследуемые ограничения часто формулируют более строго, но менее гибко. Иными словами, ограничение на уров-

не сущности может быть подмножеством ограничений на уровне домена, но не надмножеством, Например, на атрибут *OrderDate*, определенный в домене *TransactionDate,* допустимо наложить дополнительное ограничение, чтобы он содержал только даты текущего года, в то время как домен разрешает хранить любую дату от начала работы компании до сегодняшнего дня. Но ограничение на уровне сущности расширит диапазон хранимых в *OrderDate* значений по сравнению с диапазоном, определенным на уровне домена.

Аналогично для атрибута *CompanyName*, определенного на домене *Name*, вы вправе задать запрет на хранение пустых строк, даже если сам домен *Name* допускает их существование. Здесь мы вновь имеем дело с дополнительным сужением диапазона возможных значений атрибута по сравнению с ограничениями на уровне домена.

ПРИМЕЧАНИЕ Проектировщики часто определяют правила проверки значений и хранения неопределенных величин именно на уровне сущности, а не на уровне домена. Обычно они аргументируют свое решение тем, что эти ограничения используются только на уровне сущности. Для такой позиции есть некоторые основания, но я рекомендую определять как можно больше ограничений на уровне домена. Это облегчит дальнейшую работу по спецификации ограничений.

Кроме того, в процессе сужения диапазона разрешенных значений для единственного атрибута ограничения на уровне сущности могут влиять на другие атрибуты. Пример такого ограничения — требование, чтобы дата поставки в заказе (*ShippingDate*)не предшествовала дате самого заказа *(OrderDate)*. Ограничения на уровне сущности, тем не менее, не могут ссылаться на другие отношения. Неприемлемо, например, определять ограничение на уровне сущности, которое задает значение скидки для заказчика *DiscountRate* (атрибут отношения *Customer)* на основании значения общей суммы сделанных заказов *TotalSales*(которое вычисляется с помощью суммирования данных из множества записей отношения *OrderItems*). Ограничения, зависящие от множества отношений, определяются на уровне базы данных (подробнее — далее в этой главе).

Будьте осторожны с ограничениями, связанными с несколькими атрибутами: необходимость их использования может свидетельствовать, что модель плохо нормализована. Если вы ограничиваете или вычисляете значение одного атрибута на основании другого, то вероятно, все в порядке. Ограничение на уровне сущности, которое звучит как «статус не может иметь значение *Preferred* (Привилегирован-

ный), до тех пор пока самой первой записи о заказе клиента не исполнится по крайней мере один год» выглядит прекрасно. Но если значение одного атрибута определяет значение другого: например, «Если запись о заказе была сделана более года назад, тогда значение статуса равно *Preferred* (Привилегированный)», — то имеется функциональная зависимость и отношение не находится в третьей нормальной форме.

Ссылочная целостность

В главе 3 мы рассмотрели декомпозицию отношений, выполняемую с целью минимизировать избыточность данных, и внешние ключи, используемые для организации связей между отношениями. Если эти связи будут разрушены, система станет в лучшем случае ненадежной, а в худшем — откажется работать. *Ограничения ссылочной целостности* (referential integrity constraints) поддерживают и защищают эти связи.

На самом деле существует только одно ограничение ссылочной целостности: внешние ключи не могут «осиротеть». Иначе говоря, каждой записи таблицы, содержащей внешний ключ, должна соответствовать запись в другой таблице, содержащей первичный ключ, Кортежи, содержащие внешние ключи, для которых не существует соответствующих значений ключа-кандидата в ссылочной таблице, называются осиротевшими сущностями. Причин их появления три:

- кортеж, добавленный в таблицу, содержит значение внешнего ключа, которому не соответствует ни одно значение ключа-кандидата в ссылочной таблице;
- изменилось значение ключа-кандидата в ссылочной таблице;
- запись, на которую ссылается внешний ключ, удалена из ссылочной таблицы.

Все эти случаи необходимо учесть, чтобы поддерживать ссылочную целостность. Первый из них, когда добавляется ни на что не ссылающаяся запись, обычно легко предотвратить. Но учтите, что неопределенные и несуществующие величины не принимаются во внимание, Если связь объявлена как необязательная, в отношение можно добавлять любое количество записей, содержащих в качестве значений внешнего ключа неопределенные величины.

Второй случай осиротевших сущностей — изменение значения ключа-кандидата в ссылочном отношении — наблюдается не очень часто. Я рекомендую не изменять значение ключа-кандидата везде, где это возможно, то есть использовать на уровне сущности ограничение типа: «Значения ключа-кандидата запрещено изменять». Но если модель позволяет менять значения ключей-кандидатов, вы дол-

жны быть уверены, что соответствующие изменения вносятся во внешние ключи. Такой подход известен как *каскадное обновление.* И Microsoft Jet, и Microsoft SQL Server обеспечивают простые механизмы его поддержки.

Последний случай «сиротства» внешнего ключа — удаление записи из ссылочной таблицы. Например, если кто-то удалит запись из отношения *Customer*, что произойдет при этом с заказами удаляемого покупателя? Как и в случае изменения значения ключа-кандидата, рекомендация — использовать запреты. Следует запрещать удаление записей из ссылочных таблиц везде, где это возможно. Это самое простое и ясное решение проблемы, если, конечно, ваша система позволяет его применить. Если же это невозможно, используйте каскадное удаление.

Впрочем, есть еще одна возможность, которая несколько более сложна с точки зрения реализации. В любом случае, ее нельзя реализовать автоматически. Вам может понадобиться «переместить» ссылки с одной записи на другую. Потребность в этом возникает нечасто, но иногда это необходимо. Допустим, заказчик А приобрел компанию — заказчика Б. Тогда может потребоваться удалить запись о заказчике Б и связать все записи о заказах, сделанных заказчиком Б, с заказчиком А.

Специальный случай ограничения ссылочной целостности — вопрос о максимальной мощности отношения, обсуждавшийся в главе 3. В модели данных такие правила как «Менеджеры могут иметь не более пяти подчиненных» определяются как ограничения ссылочной целостности.

Целостность на уровне базы данных

Наиболее общая форма ограничения целостности - *целостность базы данных.* Ограничения на уровне базы данных определяются для нескольких отношений: «Заказчик не может иметь статус *Preferred* (Привилегированный), если он не является заказчиком компании в течение, по крайней мере, года». Большинство ограничений на уровне базы данных имеют подобную форму.

Желательно определять ограничения как можно более полно, и ограничения на уровне баз данных не являются исключением. Тем не менее, следует проявлять осторожность, чтобы не перепутать ограничение на уровне базы со спецификацией рабочего процесса. *Рабочий процесс* - это нечто, что совершается по отношению к базе данных. Например, добавление заказа в базу. В то время как ограничение целостности - это правило, определяющее содержание базы данных.

Правила, которые определяют задания, выполняемые с использованием базы данных, являются ограничениями рабочих процессов, а не ограничениями базы данных. Рабочие процессы оказывают огромное влияние на структуру модели данных, но не могут быть ее частью (см, главу 8).

Не всегда ясно, является ли данное бизнес-правило ограничением целостности, рабочим процессом или чем-нибудь еще. Разница может быть несущественной. Реализуйте правило наиболее удобным способом: если существует возможность непосредственно превратить его в ограничение базы данных, поступите именно так. Но иногда такой подход делает модель запутанной - это может произойти, даже когда правило очевидно является ограничением целостности. Вынесите такое правило в клиентскую часть приложения, где его можно будет реализовать, используя процедурный подход.

С другой стороны, если в предметной области часто меняются «правила игры» и это приводит к частым изменениям бизнес-правил, вероятно, удобнее поддерживать систему, в которой правила являются частью схемы базы данных. Тогда изменение отдельного элемента повлечет за собой изменения (но не крах) во всех связанных с этим элементом системах.

Целостность на уровне транзакций

Последняя разновидность целостности базы данных - *целостность транзакции.* Ограничения целостности на уровне транзакций управляют способами манипулирования данными в базе. В отличие от других, это процедурные ограничения и они, таким образом, не являются частью модели данных.

Транзакции тесно связаны с рабочими процессами. Фактически, эта концепция предполагает, что данный рабочий процесс может включать в себя одну или несколько транзакций, и наоборот. Не совсем верно, но удобно представлять себе рабочий процесс как некую абстракцию («добавить заказ»), а транзакцию - как физическое действие («обновить таблицу *OrderDetails*»).

Транзакцию обычно определяют как «логическую единицу работы», что я всегда считала бесполезным и риторическим. Существенно го, что транзакция является группой действий, которые должны быть либо совместно завершены, либо совместно отменены. База данных обязана соответствовать всем определенным в ней ограничениям целостности до начала транзакции и после ее завершения, но может нарушать одно или несколько ограничений во время транзакции.

Классический пример транзакции — перевод денег с одного банковского счета на другой. Если система снимет деньги со счета А, но не сможет зачислить их на счет В, деньги будут потеряны. Очевидно, что если зачислить деньги не удалось, то и снятие денег со счета нужно отменить. На языке баз данных это называется *откатом* (rolling hack).

Транзакция может включать в себя множество записей, множество отношений, и даже множество баз данных. Точнее, все виды операций в базе данных являются транзакциями, даже обновление единственной записи. К счастью, такие транзакции низкого уровня выполняются непосредственно сервером баз данных незаметно для пользователя.

Как механизм баз данных Microsoft Jet, так и SQL Server обеспечивают средства для поддержания целостности транзакций путем использования команд BEGIN TRANSACTION, COMMIT TRANSACTION и ROLLBACK TRANSACTION. Как и можно было ожидать, реализация целостности транзакций в SQL Server более работоспособна и лучше обрабатывает ошибки, возникающие из-за сбоев оборудования и программного обеспечения. Впрочем, это уже вопрос реализации, а они не являются предметом нашего рассмотрения. Что действительно важно с точки зрения проектирования — это умение сформулировать и создать транзакционные зависимости, наполнив неконкретное выражение «логическая единица работы» практическим содержанием,

Реализация целостности данных

Вплоть до этого момента мы концентрировали внимание на концептуальном, абстрактном моделировании предметной области. А сейчас рассмотрим несколько вопросов, возникающих при создании физической модели предметной области, конкретно — при создании схемы базы. Переход от одного этапа к другому сопровождается в первую очередь изменением терминологии — отношения становятся таблицами, а атрибуты — полями. Вопросы, касающиеся целостности данных, однако, возникают на каждом из этапов.

Неопределенные и несуществующие величины

Ранее в этой же главе я заметила, что необходимо определить, могут ли домены и атрибуты быть пустыми или содержать неопределенные значения, не вдаваясь в подробности того, каким образом такие ограничения будут реализованы. Между тем, реализация — это действительно серьезный вопрос, и его никак нельзя обойти, если мы говорим о схеме базы данных.

«Проблема отсутствующей информации» возникла тогда же, когда была предложена первая реляционная модель. Как показать, что какая-то часть информации либо неизвестна (заказчик в действительности имеет фамилию, мы только не знаем, какую), либо отсутствует (у заказчика нет второго имени)? Большинство реляционных баз данных, включая как Microsoft Jet, так и и SQL Server, поддерживают возможность использовать «пустое значение» — так называемый *Null* — как способ работы с неопределенными и несуществующими значениями.

Будет, вероятно, преувеличением назвать значение *Null* решением проблемы, так как с ним связан ряд дополнительных трудностей. Некоторые эксперты в области баз данных полностью отвергают концепцию *Null*. К. Дж. Дейт заявляет, что *Null* «разрушает модель», и я уже не помню, сколько раз я слышала в адрес *Null*-концепции слово «вредная». Любые замечания о сложности обработки *Null*-значении всегда сводятся в конечном итоге к выводу: «О Боже! Вам не следует их использовать. Их нужно выкинуть».

В качестве альтернативы те, кто считают, что значения *Null* вредны, рекомендуют использовать специфические величины, описанные для соответствующего домена, чтобы показать, что величина не определена или не существует (или все вместе). Я называю это *подходом договорных значений* (conventional value approach).

На мой взгляд, этот подход имеет ряд недостатков. Во-первых, во многих случаях «договорное значение» условно. Дата 9/9/1900 на самом деле не означает, что значение даты неизвестно, мы просто согласились, что будем интерпретировать эту дату как неопределенное значение.

Я не вижу здесь никаких преимуществ по сравнению с *Null*-значениями. Разумеется, *Null* тоже является «договорным значением», но его ни с чем нельзя перепутать, оно поддерживается реляционной моделью и большинством серверов баз данных.

Второй проблемой, делающей с моей точки зрения подход договорных значений непригодным, является нарушение ссылочной целостности.

Возьмем, например, необязательную связь между заказчиком и представителями отдела обслуживания клиентов (CSR): каждый CSR, однажды обслужив клиента, должен быть занесен в таблицу CSR. Подход договорных значений требует, чтобы запись, которая добавляется в таблицу CSR, соответствовала «договорному значению», выбранному, чтобы показать, что CSR не связан ни с одним клиентом (рис. 4-2).

Customers

CustomerID	CompanyName	CSR
ALFKI	Alfreds Futterkiste	Nancy Davolio
ANATR	Ana Trujillo Emparedados y helados	Andrew Fuller
ANTON	Antonio Moreno Taquería	Anne Dodsworth
AROUT	Around the Horn	Steven Buchanan
BERGS	Berglunds snabbköp	Margaret Peacock
BLAUS	Blauer See Delikatessen	UNASSIGNED
BLONP	Blondel pere e! f\\s	Steven Buchanan
BOLID	Bólido Comidas preparadas	Nancy Davolio
BONAP	Bon app'	UNASSIGNED
BOTTM	Bottom-Dollar Markets	Robert King
BSBEV	B's Beverages	UNASSIGNED
CACTU	Cactus Comidas para llevar	Robert King
CENTC	Centro comercial Moctezuma	Laura Callahan
CHOPS	Chop-suey Chinese	Janet Leverling
COMMI	Comercio Mineiro	Michael Suyama
CONSH	Consolidated Holdings	Andrew Fuller
DRACD	Drachenblut Delikatessen	Steven Buchanan
DUMON	Du monde entier	UNASSIGNED
EASTC	Eastern Connection	Laura Callahan
ERNSH	Ernst Handel	Robert King
FAMIA	Familia Arquibaldo	Margaret Peacock
FISSA	FISSA Fabrica Inter. Salchichas S.A.	Janet Leverling
FOLIG	Folies gourmandes	UNASSIGNED
FOLKO	Folk och fä HB	Janet Leverling
FRANK	Frankenversand	Janet Leverling
FRANR	France restauration	UNASSIGNED
FRANS	Franchi S.p.A.	Janet Leverling

CSRs

EmployeeID	LastName	FirstName
1	Davolio	Nancy
2	Fuller	Andrew
3	Leverling	Janet
4	Peacock	Margaret
5	Buchanan	Steven
6	Suyama	Michael
7	King	Robert
8	Callahan	Laura
9	Dodsworth	Anne
10	UNASSIGNED	

Рис. 4-2, «Договорные значения» требуют дополнительных «пустых» записей для поддержания ссылочной целостности

Теперь подсчитаем, сколько представителей службы клиентского сервиса (CSRs) работает в компании. На одного меньше, чем содержится в таблице CSRs, так как одна из записей является «пустой». Каково среднее число клиентов, приходящееся на одного CSR? Оно равно числу записей в таблице *Customer* минус число записей, соответствующих «UNASSIGNED» CSR, деленных на число, равное числу записей в таблице CSR минус 1.

«Договорные значения», тем не менее, весьма удобны для создания отчетов. Например, вы можете подставить *Unknown* (неизвестно) вместо значений *Null* и *Not Applicable* (не определено) — вместо «пустых» величин. Разумеется, такой подход сильно отличается от хранения «договорных значений» в базе данных, где, как мы увидели, они серьезно затрудняют обработку данных.

Возможно, значения *Null* вредны, и конечно, выглядят безобразно, но это лучший способ работать с неопределенными значениями. Ищите альтернативу использованию *Null* там, где это имеет смысл, и применяйте *Null* там, где альтернативные пути неприемлемы.

Одна из проблем, связанных с *Null*, такова: для доменов, содержащих *Null* (за исключением строковых или текстовых типов данных), значения *Null* могут иметь двойной смысл. Поле, объявленное как *DateTime*, может содержать либо даты, либо *Null*. Если соответствующий атрибут содержит и неизвестные, и несуществующие величины, и обе представляются как значения *Null*, нет способа определить, означает ли конкретное значение *Null* неизвестную или несуществующую величину. Эта проблема не возникает для строковых или текстовых типов данных, так как вы можете использовать пустые строки нулевой длины для обозначения несуществующих величин, а *Null* — для представления неизвестных значений.

На практике такие ситуации не столь уж часты. Еще несколько видов доменов (не содержащих строковые данные или текст) допускают представление несуществующих величин, поэтому в таких доменах значение *Null* всегда можно интерпретировать как неизвестную величину. Для доменов, которые допускают представление несуществующих величин, часто можно подобрать подходящую альтернативу *Null*. Заметьте, я говорю о неких реальных величинах, а не о «договорных». Например, хотя отношение *Product* (Продукт) имеет атрибут *Weight* (Вес), для продукта *Service Call* (Звонок в сервисную службу), который, очевидно, не обладает весом, можно использовать О (этот способ подходит для представления отсутствующих данных в большинстве случаев).

Вторая и значительно более серьезная проблема *Null* — очень сложная обработка таких данных. Логическое сравнение величин усложняется, а результаты выполнения запросов становятся трудными для понимания. Мы будем детально рассматривать эти проблемы в главе 5.

Я не отношусь к значениям *Null* чересчур легкомысленно, и даже когда существует альтернатива, рекомендую использовать именно их. Не ломайте модель данных, чтобы облегчить жизнь программистам. Помните об этом, и если система требует использовать значения *Null* — используйте их.

Реакции на нарушения целостности

Проектируя схему базы данных, вы должны не только найти способ эффективно реализовать ограничения целостности, но и решить, какие действия предпримет механизм баз данных в ответ на нарушения

целостности. Конечно, в большинстве случаев, база данных просто отвергнет команду-нарушителя, отправив пользователю сообщение об ошибке. Но иногда СУБД выполняет самостоятельную коррекцию команды. Примеры: обеспечение величины по умолчанию для атрибутов, которые не разрешают пустые значения, или операции каскадного обновления и удаления для обеспечения ссылочной целостности. Мы будем детально обсуждать реакции системы на ошибочные действия в части 3.

Декларативная и процедурная целостность

Серверы реляционных баз данных обеспечивают поддержку целостности двумя способами: декларативно и процедурно. *Поддержка декларативной целостности* явно определяется (декларируется) как часть схемы базы данных. И механизм баз данных Microsoft Jet, и SQL Server обеспечивают определенную поддержку декларативной целостности. Декларативная целостность — предпочтительный способ для организации целостности данных. Ее следует использовать везде, где это возможно.

SQL Server поддерживает *процедурную целостность* путем использования *триггеров* — процедур, которые выполняются, когда запись вставляется, изменяется или удаляется. Механизм баз данных Microsoft Jet не поддерживает триггеры или какую-либо другую форму процедурной целостности. Когда ограничение целостности нельзя обеспечить средствами декларативной целостности, поддержку этого ограничения следует возложить на пользовательское приложение.

Мы обсудим специфические особенности переноса ограничений целостности, определенных в модели данных, в физическую базу данных в конце этой главы.

Целостность на уровне домена

SQL Server обеспечивает поддержку ограниченного ряда доменов в виде определяемых пользователем типов данных (UDDTs). Поля, типами которых являются UDDTs, наследуют объявления типов данных, также как и ограничения целостности уровня доменов UDDT.

Столь же важно, что SQL Server запрещает сравнивать между собой поля, определенные с помощью разных UDDTs, даже когда сравниваемые UDDTs основаны на одном и том же базовом типе данных. Например, хотя домены *CityName* (Название города) и *CompanyName* (Наименование компании) определены как *char(30)*, SQL Server отвергнет выражение CityName = CompanyName как неверное. Это можно преодолеть, используя функцию преобразования CityName = CONVERT(char(30), CompanyName). Но тогда вам придется задумы-

ваться об этом каждый раз, когда вы сравниваете поля, объявленные в различных доменах (хотя чаще всего такое сравнение не имеет смысла).

UDDT создается либо с помощью SQL Server Enterprise Manager, либо с помощью системной хранимой процедуры *sp_addtype*. В любом случае для UDDT указывается имя, тип данных и определяется, могут ли они содержать значения *Null*. Когда UDDT создан, вы вправе определить для него значения по умолчанию и правила проверки. *Правило* SQL Server является логическим выражением, которое определяет значения, приемлемые для UDDT (или для поля, если правило относится к полю, а не к UDDT). (*Значение по умолчанию* — это просто величина, которую система подставляет в поле, если пользователь не задал значение этого поля явно).

Привязка правила или величины по умолчанию к UDDT состоит из двух шагов. Сначала нужно создать правило или величину по умолчанию, а потом — привязать их к UDDT (или полю). Целесообразность такой сложной двухступенчатой процедуры объясняют, как правило, следующим образом: будучи однажды определенной, правило или значение по умолчанию может быть использовано повторно. Я нахожу это утомительным, так как такие объекты повторно используются редко. Когда определяется таблица, SQL Server обеспечивает возможность декларации значений по умолчанию и ограничений CHECK в самом определении таблицы. (Ограничения CHECK во многом подобны правилам, но обладают большими возможностями). К сожалению, такая простая декларация недоступна для объявления UDDT — для них нужно использовать метод «создать, потом связать». Было бы желательно, чтобы корпорация Microsoft добавила в будущие версии SQL Server поддержку значений по умолчанию и ограничений CHECK в объявления UDDTs.

Второй способ реализации ограничений целостности на уровне домена — использовать просмотровые таблицы. Этот способ применим как для Microsoft Jet, так и для SQL Server. В качестве примера возьмем домен *USStates*. Теоретически вы можете создать правило, проверяющее значения всех 50 штатов. Реально это мучительный процесс, в частности для механизма баз данных Microsoft Jet, где такое правило пришлось бы переписывать для каждого поля, определенного в этом домене. Намного легче создать таблицу *USStates* и использовать ссылочную целостность, которая будет гарантировать, что поле содержит только значения из таблицы.

Целостность на уровне сущности

В схеме базы данных ограничения на уровне сущности управляют конкретным полем, множеством полей или таблицей в целом. И ме-

ханизм баз данных Microsoft Jet, и SQL Server предоставляют средства, гарантирующие целостность на уровне сущности. SQL Server обеспечивает более богатый спектр возможностей (и это не удивительно), впрочем, разница не столь уж велика.

Самое фундаментальное ограничение целостности на уровне отдельных полей — это, конечно, тип данных. Механизм баз данных Microsoft Jet и SQL Server обеспечивают богатый спектр типов данных (табл. 4-1).

Табл. 4-1. Типы данных Microsoft Jet и SQL Server

Логический тип данных	Тип данных SQL Server	Тип данных Microsoft Jet	Хранимый диапазон значений	Размер
Integer	Int	Long integer	Целые числа от -2 147 483 648 до 2 147 483 647	4 байта
	Smallint	Не определен	Целые числа от -32 768 до 32 767	2 байта
	Tinyint	Integer	Целые числа от 0 до 255	1 байт
Точное числовое значение (exact numeric)	Decimal	Number (различные типы)	Целые или дробные числа от -10^{38-1} до 10^{38-1}	2–17 байт
Числа с плавающей точкой	Float (до 15 знаков после десятичной точки)	Double	Действительные числа от $-1{,}79E^{308}$ до $1{,}79E^{308}$; положительные числа от $2{,}23E^{-308}$ до $1{,}79E^{308}$; отрицательные числа от $-2{,}23E^{-308}$ до $-1{,}79E^{308}$	8 байт
	Real	Single	Действительные числа от $-3{,}40E^{38}$ до $3{,}40E^{38}$; положительные числа от $1{,}18E^{-38}$ до $3{,}40E^{38}$; отрицательные числа от $-1{,}18E^{-38}$ до $-3{,}40E^{38}$	4 байта

(продолжение)

Строка символов фиксированной длины	Char	Не определен	Не более 255 символов для Microsoft Jet; 8000 символов для SQL Server 7.0 (255 — в более ранних версиях)	1 байт на символ
Строка символов переменной длины	Varchar	Text	Не более 255 символов в Microsoft Jet, 8000 символов для SQL Server 7.0 (255 — в ранних версиях)	1 байт на символ
Денежный тип	Money	Currency	Числа с точностью до четырех знаков после десятичной точки в диапазоне от -922 337 208 685 477,5808 до 922 337 208 685 477,5807	8 байт
	Smallmoney	Не определен	Числа с точностью до четырех знаков после десятичной точки в диапазоне от -214 748,3648 до 214 748,3647	4 байта
Дата и время	Datetime	Date/Time	От 1 января 1753 г. до 31 декабря 9999 г. для SQL Server; от 1 января 100 г. до 31 декабря 9999 г. для Microsoft Jet	8 байт
	Smalldatetime	Не определено	От 1 января 1900 г. до 6 июня 2079 г.	4 байта
Двоичные данные фиксированной длины	Binary	Не определено	Не более 8000 байт	Количество байт плюс 4 байта
Двоичные данные переменной длины	Varbinary	Поддерживается только для присоединенных таблиц	Не более 8000 байт	Количество байт плюс 4 байта

(продолжение)

Логический тип данных	Тип данных SQL Server	Тип данных Microsoft Jet	Хранимый диапазон значений	Размер
Большие двоичные или текстовые данные	Text	Memo	Строковые данные вплоть до 2 Гб для SQL Server или 1 Гб для Microsoft Jet	Размер данных плюс 16 байт
Логический тип	Bit	Yes/No	0 или 1	1 байт, но в SQL Server битовые значения комбинируются в байт

Как я уже упоминала, SQL Server также разрешает определять поля, используя UDDT Поля типа UDDT наследуют возможность хранить *Null,* значения по умолчанию и правила, которые были определены для типа, но могут быть переопределены на уровне поля. Логически ограничение на уровне поля должно сужать ограничения для UDDT, но фактически SQL Server просто заменяет определения для UDDT на те, что определены для полей. То есть можно разрешить хранение *Null* для поля, даже если UDDT, в котором определено поле, не позволяет этого.

SQL Server и механизм баз данных Microsoft Jet позволяют контролировать, могут ли поля хранить значения *Null.* Когда определяют поле в SQL Server, его просто специфицируют как NULL или NOT NULL или щелкают мышью соответствующий элемент управления в Enterprise Manager.

Для механизма баз данных Microsoft Jet допустимость значений *Null* можно задать двумя способами — используя флаг *Null* или поле *Required.* Кроме того, в механизме баз данных Microsoft Jet существует флаг *AllowZeroLength,* который определяет, можно ли хранить в полях типа *Text* или *Memo* пустые строки («»). В SQL Server для этого используют ограничение CHECK.

Задание соответствующего свойства при определении поля позволяет определить значения по умолчанию в механизме баз данных Microsoft Jet. В SQL Server можно определить свойство *Default* при создании поля или связать некое значение, генерируемое системой, со свойством *Default,* как это описано для UDDT. Я рекомендую объявлять значение по умолчанию в определении таблицы, если вы не можете сделать этого на уровне домена.

Наконец, и Microsoft Jet, и SQL Server позволяют специфицировать ограничения на уровне сущности. Microsoft Jet обеспечивает два свойства поля — *Validation Rule и ValidationText.* SQL Server разрешает создать ограничения CHECK на уровне поля или привязать системное правило к полю позднее, но первый метод предпочтительнее.

На первый взгляд, правила проверки Microsoft Jet и ограничения CHECK сервера SQL Server выглядят одинаково, но между ними есть несколько существенных отличий. Оба имеют форму логических выражений, и никогда не позволяют сослаться на другие столбцы или таблицы. Однако правила проверки Microsoft Jet должны давать оценку *True* для проверяемой величины. Ограничения CHECK сервера SQL Server не должны давать оценку *False.* Это важно: как *True,* так и *Null* являются для CHECK приемлемыми значениями, для правил проверки Microsoft Jet приемлемо только *True.*

Кроме того, для одного поля можно определить много ограничений CHECK. Фактически, одно правило и любое количество ограничений CHECK можно применить для одного поля в SQL Server, в то время как поле в Microsoft Jet имеет единственное свойство *Validation Rule.* Кстати, использование свойства *ValidationText* в Microsoft Jet позволяет возвращать в клиентское приложение сообщение об ошибке. Microsoft Access показывает текст в окне сообщений, и этот текст доступен в Microsoft Visual Basic и других средствах программирования через обработку строковых величин в коллекции объектов *Errors.*

Ограничения на уровне сущности, которые ссылаются на множество столбцов одной таблицы, реализуются в качестве правил проверки на уровне таблицы в Microsoft Jet и табличных ограничений CHECK в SQL Server. Несмотря на то, что объявляются они в другом месте, такие ограничения на уровне таблиц функционируют точно так же, как соответствующие ограничения на уровне полей.

Самое фундаментальное ограничение на уровне сущности — требование, чтобы каждый экземпляр сущности можно было однозначно идентифицировать. Помните, что именно это и есть *правило целостности сущности;* все другие правила в основном представляют собой то, на что ссылаются как на ограничения целостности на уровне сущности. Microsoft Jet и SQL Server поддерживают ограничения уникальности, но разными способами. Оба продукта реализуют ограничение, используя индексы, но SQL Server скрывает это от пользователя. Создает ли сервер индекс явным образом (Microsoft Jet) или объявляет ограничение (SQL Server) — это технические детали.

И Microsoft Jet, и SQL Server поддерживают уникальные наборы полей. Оба также поддерживают первичные ключи, состоящие из нескольких полей, что подразумевает уникальность такого набора. Для таблицы может быть определен только один ключ, порой состоящий из нескольких полей. Ограничений уникальности может быть сколько угодно.

Еще одно важное различие между ограничениями уникальности и первичными ключами: уникальные индексы могут содержать *Null,* а первичные ключи — нет. Серверы баз данных по-разному обрабатывают *Null* в уникальных индексах. Microsoft Jet поддерживает свойство *IgnoreNulls,* который предотвращает добавление информации о записях, содержащих *Null* в индексных полях, в индексы. Записи добавляются в таблицу, но содержимое индекса не изменяется. Эта возможность не реализована в SQL Server.

Кроме того, SQL Server разрешает добавить в таблицу только одну запись, содержащую *Null* в индексируемом поле. Это логически не оправдано, так как позволяет рассматривать записи, содержащие *Null,* как одинаковые, хотя они, конечно, таковыми не являются. *Null* не эквивалент чего бы то ни было, не исключая «другого» *Null.*

Интересно, что ни Microsoft Jet, ни SQL Server не требуют обязательного определения первичного ключа для таблицы или даже существования уникального набора полей. Другими словами, можно создавать таблицы, которые не являются отношениями (кортежи в отношениях должны быть уникальны, а записи в таблицах — не обязательно). Почему разработчики остановились на этом решении, я не понимаю, но такая возможность существует.

SQL Server также обеспечивает процедурные механизмы обеспечения целостности на уровне сущности, чего не может Microsoft Jet. *Триггеры* (triggers) — это небольшие элементы кода (на языке Transact-SQL для SQL Server), которые автоматически выполняются, когда происходит определенное событие. Для каждого события INSERT, UPDATE, или DELETE можно определить множество триггеров, и один триггер может быть определен для нескольких событий.

Ссылочная целостность

Хотя Microsoft Jet и SQL Server поддерживают ссылочную целостность практически одинаковым образом, они используют для этого различные парадигмы.

SQL Server позволяет объявлять внешние ключи на уровне определения таблицы. *Ограничение внешнего ключа* задает ссылку на ключ-кандидат другой, ссылочной таблицы. Когда ссылка определена, SQL

Server предотвращает создание «осиротевших» записей, отменяя операцию вставки, если такой записи не соответствует ни одна запись в ссылочной таблице.

Значения *Null* допускаются в столбцах внешних ключей, хотя могут быть запрещены, если столбец входит в первичный ключ таблицы. SQL Server также запрещает удалять записи в ссылочной таблице, если им соответствуют значения внешнего ключа.

Microsoft Jet поддерживает ссылочную целостность, используя объект *Relation* внутри базы данных. Терминология Microsoft здесь неуместна — объект *Relation* в Microsoft Jet является физическим представлением связи между двумя сущностями. Не путайте объект *Relation* с логическими отношениями, которые определяются в модели данных.

Проще всего создать объект *Relation,* используя пользовательский интерфейс базы данных Access (с помощью команды Relationships в меню Tools), но это можно сделать и программным способом. Свойства *Table* и *ForeignTable* объекта *Relation* библиотеки Data Access Object (DAO) определяют две таблицы, участвующие в связи, в то время как набор объектов *Fields* определяет для каждой из таблиц поля, по которым устанавливается связь.

Способ, которым Microsoft Jet будет поддерживать ссылочную целостность для отношения, определяется через свойство Attributes отношения:

- **dbRelationUnique** — связь «один к одному»;
- **dbRelationDontEnforce** — связь не установлена (не поддерживается ссылочная целостность);
- **dbRelationInherited** — связь существует в другой базе данных, которая содержит две связанные таблицы;
- **dbRelationUpdateCascade** — поддерживается каскадное обновление;
- **dbRelationDeleteCascade** — поддерживается каскадное удаление.

Обратите внимание на флаги *dbRelationUpdateCascade* и *dbRelationDeleteCascade.* Если определен флаг для обновления и ссылочное поле изменяется, Microsoft Jet автоматически обновит поля в связанной таблице.

Точно так же, определение флага удаления приведет к автоматическому удалению записей из связанной таблицы в случае удаления соответствующей записи из ссылочной таблицы. У SQL Server аналогичные флаги отсутствуют, но каскадные операции легко организовать с помошью триггеров.

Другие виды целостности

В модели данных мы определяем три дополнительных вида целостности: базы данных, преобразования и транзакции. Некоторые ограничения преобразования достаточно просто объявить с помощью правил проверки. Большинство их, тем не менее, как и все ограничения базы данных и транзакции, реализуют процедурно. Для баз данных SQL Server это означает применение триггеров. Microsoft Jet не поддерживает триггеры, эти ограничения нужно реализовать в клиентском приложении.

Итоги

В этой главе мы познакомились с моделированием целостности данных. Три вида ограничений целостности: домена, преобразования, и сущности, — управляют конкретными отношениями, а ограничения ссылочной целостности обеспечивают поддержку связи между отношениями. Наконец, ограничения баз данных и ограничения транзакций управляют базой данных в целом.

Целостность данных реализуется в схеме базы данных путем сочетания декларативной и процедурной целостности. Декларативная целостность объявляется явно как часть схемы. Это предпочтительный способ реализации целостности. Однако не все ограничения можно реализовать с помощью декларативной целостности, и там, где этого сделать нельзя, применяют процедуры.

В главе 5 мы рассмотрим реляционную алгебру и операции, которые можно выполнять над отношениями в базе данных.

Реляционная алгебра

Отношения могут быть базовыми или производными. В предшествующих главах мы дали определение *базового отношения*, которому соответствует физическое представление в базе данных. *Производное отношение* (derived relation) — это отношение, определяемое через другие отношения, а не через атрибуты. Реляционная модель позволяет создавать различные виды производных отношений.

На схеме базы данных базовое отношение представлено таблицей. Производные отношения существуют в виде *представлений* (views) в Microsoft SQL Server и *запросов* (queries) в механизме СУБД Microsoft Jet. Я буду использовать термин «представления» для обозначения обоих этих понятий, поскольку этот термин широко применяется в реляционной теории, а также термин «набор записей* (в основном, когда речь пойдет о запросах или представлениях).

Представления определяются в терминах реляционных операций, которым посвящена эта глава. Microsoft Access и SQL Server Enterprise Manager предоставляют графический интерфейс для создания представлений. Однако представления можно также создавать при помощи операторов SQL SELECT.

SQL (читается как «сиквел») — это *язык структурированных запросов* (Structured Query Language), стандартизированный язык для описания реляционных операций. Microsoft Jet и SQL Server поддерживают специфичные, несколько отличающиеся Друг от друга, диалекты SQL. Впрочем во всем, что касается реляционной алгебры, различия минимальны. В этой главе мы обсудим вопросы реляционной алгебры, и там, где синтаксис версий SQL будет отличаться, я приведу примеры для обоих вариантов.

Оператор SELECT — это очень мощное средство, однако пользоваться им не так просто, как кажется на первый взгляд. Я не буду

подробно описывать этот оператор и углубляться в подробности его применения, а приведу в качестве примера его простейшую структуру. Вот синтаксис оператора SELECT:

```
SELECT <список_полей>
FROM <список_наборов_записей>
    <тип_соединения> JOIN <условие_соединения>
WHERE <критерий_выборки>
GROUP BY <список_полей_оператора_GroupBy>
HAVING <критерий_выборки>
ORDER BY <список_полей_оператора_OrderBy>
```

<Список_полей> в операторе SELECT - это список, состоящий из одного или более полей, которые будут включены в результирующий набор записей, возвращаемых оператором. Поля могут быть взяты из нижележащих наборов записей либо вычислены. <Список_наборов_записей> в выражении FROM — это, как вы, наверное, уже догадались, перечисление таблиц и представлений, па которых основывается оператор SELECT. Ключевые слова SELECT и FROM обязательны для SQL-оператора SELECT, все остальные ключевые слова — нет.

Оператор JOIN определяет связь между наборами записей, перечисленными в <списке_наборов_записей>. Более подробно об операторе JOIN я расскажу далее в этой главе. Ключевое слово WHERE определяет логическое выражение <критерий_выборки>, ограничивающее число записей, которые будут включены в результирующий набор. Об условии WHERE мы также подробно поговорим далее.

Оператор GROUP BY объединяет в одну те записи, в которых указанные поля имеют одинаковые значения. Ключевое слово HAVING используется для задания дополнительных критериев отбора записей, полученных в результате выполнения оператора GROUP BY. Оператор ORDER BY сортирует набор записей в том порядке, в котором они указаны в <списке_полей_оператора_OrderBy>.

Значения *Null* (еще раз о трехзначной логике)

Для выполнения большинства операций реляционной алгебры используются *логические операторы,* то есть операторы, возвращающие, как правило, *булевы значения — True* или *False.* Я не оговорилась, когда сказала «как правило», поскольку значения *Null,* используемые в реляционной модели, существенно усложняют дело.

Null — особые значения, используемые наряду с булевыми. Поэтому вам придется иметь дело с тремя значениями — *True, False* и

Null. Это объясняет происхождение термина *трехзначная логика.* Значения известных логических операторов для трехзначной логики приводятся в табл. 5-1.

Табл. 5-1. Значения логических операторов *And, Or* и *XOr* для трехзначной логики

AND	True	False	Null
True	True	False	Null
False	False	False	Null
Null	Null	Null	Null

OR	True	False	Null
True	True	True	Null
False	True	False	Null
Null	Null	Null	Null

XOR	True	False	Null
True	False	True	Null
False	True	False	Null
Null	Null	Null	Null

Проанализировав табл. 5-1, легко заметить, что результатом выполнения любой логической операции над *Null* и любым другим значением (*True, False* или *Null)* является *Null.* В виде формулы это можно выразить так: *Null* OP любое_значение = *Null,* где OP — логический оператор. Это же правило выполняется и для логических операторов сравнения (табл. 5-2).

Табл. 5-2. Логические значения операторов сравнения (= и ≠ ?) для трехзначной логики

=	True	False	Null
True	True	False	Null
False	False	True	Null
Null	Null	Null	Null

≠ ?	True	False	Null
True	False	True	Null
False	True	False	Null
Null	Null	Null	Null

SQL Server обрабатывает трехзначную логику неоднозначно. В дополнение к стандартному режиму обработки трехзначной логики, он

предоставляет «расширение» для стандартных логических операций. Если стандартный режим обработки трехзначной логики (ANSI_NULLS) отключен (SET ANSI_NULLS OFF), то результатом операции *Null* = *Null* будет *True,* а результатом операции *Null* = <произвольное_значение>, где <произвольное_значение> — любое значение, кроме *Null*, будет *False.* (Несомненно, это связано с тем, что для уникальных индексов SQL Server допускает присутствие в каждом индексе единственного значения *Null).*

Для обработки значений *Null* в SQL существуют два унарных оператора — IS NULL и IS NOT NULL. Результаты выполнения этих операторов над различными значениями показаны в табл. 5-3. Как и в предыдущем примере, <произвольное_значение> — это любое значение, кроме *Null.*

Табл. 5-3. Значения операторов IS NULL и IS NOT NULL

	Is Null	**Is Not Null**
<произвольное_значение>}	False	True
True	**False**	True
False	False	**True**
Null	True	False

Реляционные операторы

Мы начнем знакомство с операторами реляционной алгебры с четырех типов таких операторов: *ограничение* (restriction), *проекция* (projection), *соединение* (join) и *деление* (divide).

Первые два из них могут выполняться только над одним набором записей, хотя этот набор записей, в свою очередь, порой является результатом выполнения некоторых операций над произвольным числом наборов записей (например, представлением, основанным на множестве наборов записей).

Оператор соединения — это один из основных операторов в реляционной модели, он задает правила объединения двух наборов записей.

Последний из упомянутых, оператор деления, используется довольно редко, с его помощью удобно определять, какие записи из одного набора совпадают со всей совокупностью записей второго,

Все четыре типа реляционных операторов реализуются с помощью SQL-оператора SELECT. Вы можете использовать всевозможные комбинации этих операторов, допускаемые системными ограничениями по длине и сложности составляемых выражений.

Ограничение

Оператор ограничения возвращает только те записи, которые удовлетворяют заданному критерию выборки. Он реализуется при помощи выражения WHERE оператора SELECT, например:

```
SELECT * FROM Employees WHERE LastName = "Davolio";
```

Для базы данных *Northwind* этот оператор возвращает запись, относящуюся к сотруднику Нэнси Даволио (Nancy Davolio), поскольку однофамильцев в этой компании у нее нет (символ *, подставленный вместо списка полей оператора SELECT, означает «все поля»).

Критерий выборки, задаваемый в условии WHERE, может быть сколь угодно сложным. Допускаются операции AND и OR над логическими выражениями. Выражение проверяется для каждой отдельной записи из набора. В результирующий набор включаются все записи, для которых результатом проверки условия WHERE будет *True*. Записи, для которых результат проверки условия WHERE — значение *False* или *Null,* не включаются в результирующий набор.

Проекция

Как мы только что убедились, ограничение позволяет сделать горизонтальный разрез набора данных. *Проекция* же, напротив, делает вертикальный разрез — возвращает подмножество полей оригинального набора данных.

Эта операция выполняется в SQL простым перечислением полей в <списке_полей> оператора SELECT. В результирующий набор записей включаются только те поля, которые есть в списке. Например, чтобы получить список сотрудников компании, можно использовать следующий оператор:

```
SELECT LastName, FirstName, Extension
FROM Employees
ORDER BY LastName, FirstName;
```

Оператор ORDER BY сортирует данные. В нашем примере полученный список будет отсортирован в алфавитном порядке: сначала по полю *LastName,* затем по полю *FirstName.*

Соединение

Операции *соединения,* пожалуй, наиболее распространены из всех реляционных операций. Вряд ли я переоценю их значение, если скажу, что эти операции составляют фундамент реляционной модели — ведь без них декомпозиция данных не имела бы никакого смысла из-за

невозможности вновь соединить данные, разбитые на множество отношений. Оператор соединения как раз и выполняет эту работу: объединяет наборы записей, основываясь на сравнении значений одного или нескольких их полей.

Соединения реализуются при помощи оператора JOIN в операторе SELECT. В зависимости от типа сравнения полей, указываемых в операторе, а также от того, каким образом обрабатываются результаты сравнения, соединения можно разделить на несколько видов. Рассмотрим каждый из них.

Эквисоединения

Соединение, в основе которого лежит оператор равенства, называется *эквисоединением* (equi-join). В результате эквисоединения возвращаются только те записи, для которых значения указанных полей совпадают.

Рассмотрим типичный случай связанных таблиц, полученных в результате процесса нормализации. *OrderID*— первичный ключ таблицы *Orders* и внешний ключ таблицы *Order Details* (рис. 5-1).

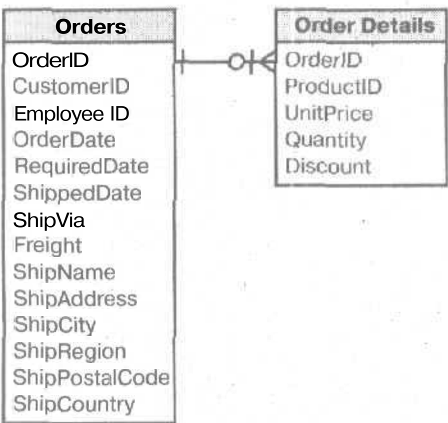


Рис. 5-1, Эти таблицы можно соединить, используя оператор JOIN

Чтобы соединить (и следовательно, денормализовать) таблицы, используйте следующий оператор SELECT:

```
SELECT Orders.OrderID, Orders.CustomerID, [Order Details].ProductID
FROM Orders
INNER JOIN [Order Details] ON Orders.OrderID = [Order Details].OrderID
WHERE (((Orders.OrderID)=10248));
```

После выполнения этого оператора вы получите набор записей, показанный на рис. 5-2.

OrderID	CustomerID	ProductID
10248	VINET	42
10248	VINET	72
10248	VINET	11

Рис.** 5-2. Этот набор **записей является результатом соединения таблиц Orders и Order Details

ПРИМЕЧАНИЕ Если вы выполните этот запрос к базе данных *Northwind* в Access 2000, то в наборе результатов увидите имя клиента, а не его идентификатор (*CustomerID*). В Access существует возможность отображать не то, что непосредственно хранится в полях таблиц, если при создании таблицы использовался элемент управления, позволяющий просматривать и преобразовывать значения (lookup control). Это, несомненно, облегчает интерактивную работу с Access, однако может доставить много дополнительных трудностей тем, кто хочет с помощью этой базы данных проиллюстрировать выполнение какого-либо оператора.

Естественные соединения

Естественные соединения (natural joins) — это частный случай эквисоединений; соединения, удовлетворяющие следующим условиям:

- основаны на операторе равенства;
- в них участвуют все общие поля;
- в результирующий набор записей включен только один набор общих полей.

Естественные соединения ничем не выделяются среди остальных видов соединений — они не представляют собой особых случаев обработки данных, и механизм СУБД не предусматривает для них специальной поддержки. Это всего лишь наиболее распространенный вид соединений, что отражено в их названии.

Один особый случай естественных соединений достаточно интересен с точки зрения его обработки механизмом СУБД Microsoft Jet. Если между двумя таблицами существует связь «один ко многим», и общие поля, включенные в представление, находятся на стороне «многие» этой связи, механизм СУБД Microsoft Jet выполнит стандартную операцию автоматического исправления записей *(Row Fix-Up* или *AutoLookup*). Если в курсоре используются поля, которые содержатся в критерии соединения, механизм СУБД Microsoft Jet автоматически подставляет общие поля — эффектный «трюк», весьма облегчающий жизнь программистам.

Тета-соединения

Формально все соединения являются тета-соединениями. Однако если соединение основано на операторе равенства, его обычно называют эквисоединением или просто соединением. Соединение, основанное на любом другом операторе сравнения, кроме равенства (<>, >, >=, <, <=) называется *тета-соединением.*

На практике тета-соединения встречаются довольно редко, однако они представляют собой очень удобный и изящный метод решения определенных типов задач. Как правило, эти задачи связаны с поиском записей, для которых значение некоторых полей превосходит среднее или суммарное значение, либо записей, для которых значение некоторых полей укладывается в определенный диапазон.

Предположим, вы создали два представления, одно из которых содержит среднее число единиц проданного товара для каждой категории товаров (рис. 5-3). Далее в этой главе я расскажу, как создать такое представление. Сейчас же просто договоримся, что оно уже существует.

ProductCategoryAverages

CategoryName	AverageSold
Beverages	678
Condiments	442
Confections	608
Dairy Products	915
Grains/Cereals	662
Meat/Poultry	700
Produce	598
Seafood	640

ProductTotals

ProductID	ProductName	TotalSold
1	Chai	828
2	Chang	1057
3	Aniseed Syrup	328
4	Chef Anton's Cajun Seasoning	453
5	Chef Anton's Gumbo Mix	298
6	Grandma's Boysenberry Spread	301
7	Uncle Bob's Organic Dried Pears	763
8	Northwoods Cranberry Sauce	372
9	Mishi Kobe Niku	56
10	Ikura	742
11	Queso Cabrales	706
12	Queso Manchego La Pastora	344
13	Konbu	891
14	Tofu	404

Рис. 5-3. Эти представления можно соединить, используя тета-соединение

Этот оператор SELECT, основанный на операторе сравнения «>», позволяет получить список продуктов, лидирующих по продажам в каждой категории:

```
SELECT DISTINCTROW ProductCategoryAverages.CategoryName, -
    ProductTotals.ProductName
FROM ProductCategoryAverages
INNER JOIN ProductTotals
ON ProductCategoryAverages.CategoryID = ProductTotals.CategoryID
AND ProductTotals.TotalSold > [ProductCategoryAverages].[AverageSold];
```

Результат выполнения этого оператора показан на рис. **5-4.**

CategoryName	ProductName
Beverages	Chai
Beverages	Chang
Beverages	Guaraná Fantástica
Beverages	Steeleye Stout
Beverages	Outback Lager
Beverages	Rhönbräu Klosterbier
Beverages	Lakkalikööri
Condiments	Chef Anton's Cajun Seasoning
Condiments	Gula Malacca
Condiments	Simp d'érable
Condiments	Vegie-spread
Condiments	Louisiana Fiery Hot Pepper Sauce
Condiments	Original Frankfurter grune Soße
Confections	Pavlova

Рис. ***5-4. Набор записей — результат тета-соединения***

В приведенном примере представление можно создать, использовав условие WHERE в операторе выборки. **Но на** самом деле Access изменит синтаксис SQL-запроса, если вы оставите его в таком виде:

```
SELECT DISTINCTROW ProductCategoryAverages.CategoryName,
    ProductTotals.ProductName
FROM      ProductCategoryAverages
INNER JOIN ProductTotals
ON ProductCategoryAverages.CategoryID = ProductTotals.CategoryID
WHERE
(((ProductTotals.TotalSold)>[ProductCategoryAverages].[AverageSold]));
```

С технической точки зрения, все соединения, в том числе эквисоединения и естественные, можно переформулировать, переписав в виде операторов SELECT с соответствующими ограничениями. **(С** точки зрения баз данных, тета-соединение не является *атомарным оператором*). В случае тета-соединений, такая формулировка запро-

сов предпочтительна, так как механизмы СУБД лучше оптимизируют выполнение запросов,

Внешние соединения

До сих пор мы рассматривали *внутренние соединения,* то есть соединения, возвращающие записи, для которых выполняется условие соединения (результат выполнения соответствующей логической операции — значение *True).* Часто внутреннее соединение определяют как операцию, возвращающую записи, у которых совпадают значения определенных полей. Однако это не так: данное определение предполагает, что соединение основывается на операторе равенства, а как мы видели, далеко не для всех соединений это утверждение справедливо.

В реляционной алгебре поддерживается также другой вид соединений — *внешние соединения* (outer joins). Внешнее соединение возвращает все записи, которые возвращает внутреннее соединение, плюс все записи из одного или обоих наборов данных, участвующих в соединении. Отсутствующие значения (те, для которых не существует соответствия) будут замещаться значениями *Null.*

Внешние соединения можно разделить на несколько групп: левые, правые и полные — в зависимости от того, какие именно дополнительные записи включены в них. *Левое внешнее соединение* (left outer join) возвращает все записи из отношения, находящегося на стороне «один» связи «один ко многим», а *правое внешнее соединение* (right outer join) — все записи из отношения, находящегося на стороне «многие» такой связи. В механизме СУБД Microsoft Jet и SQL Server порядок, в котором наборы записей перечисляются в операторе SELECT, различается для левого и правого внешнего соединения. Так, следующие два оператора возвращают все записи из таблицы X, а также те записи из таблицы Y, для которых значение логического выражения <условие> — *True:*

```
SELECT * FROM X LEFT OUTER JOIN Y ON <услозие>
SELECT - FROM Y RIGHT CUTER JOIN X ON <условие>
```

Полное внешнее соединение (full outer join) возвращает все записи из обоих наборов, комбинируя те, для которых выполняется условие данного соединения. В SQL Server для полного внешнего соединения используется следующий оператор:

```
SELECT * FROM X FULL OUTEP JOIN Y ON <условие>
```

Механизм СУБД Microsoft Jet не предоставляет прямой поддержки полных внешних соединений, однако выполнив операцию объединения результатов левого и правого внешних соединений, можно получить набор результатов, аналогичный полному внешнему соединению. Операцию объединения мы подробно рассмотрим в одном из следующих разделов.

Деление

Последний из рассматриваемых реляционных операторов — *реляционное деление*. Этот оператор (названный так для того, чтобы отличать его от математического оператора деления) возвращает те записи из первого набора, которые совпадают с записями из другого набора. Рассмотрим в качестве примера набор записей, в котором представлены категории продуктов, приобретенных у каждого из поставщиков. Реляционное деление позволит получить список поставщиков, поставляющих продукты всех категорий.

Подобные задачи не столь уж редки на практике, однако решаются они отнюдь не тривиально, поскольку SQL-оператор SELECT непосредственно не поддерживает реляционное деление. Но существует множество способов получить результаты, аналогичные результатам операции реляционного деления. Наиболее простой — переформулировать запрос.

Формулировку «список поставщиков, поставляющих продукты всех категорий», трудно реализуемую средствами стандартного SQL-запроса, можно заменить эквивалентной; «список поставщиков, для которых число поставляемых ими категорий продуктов равно числу всех категорий продуктов». Это пример операции расширения, о которой мы подробно поговорим далее. Данный метод, однако, не всегда применим, и там, где он не подходит, реализуют реляционное деление, как правило, при помощи *связанных запросов* (correlated queries). Связанные запросы мы рассматривать не будем. В библиографическом указателе в конце книги есть несколько ссылок на литературу, где связанные запросы обсуждаются достаточно подробно.

Операции над множествами

Этот раздел посвящен четырем операторам, которые основываются на традиционной теории множеств. Однако между операциями, определенными в рамках традиционной теории множеств, и рассматриваемыми нами, все же есть некоторые различия — ведь мы имеем дело с отношениями, а не абстрактными множествами.

Объединение

Реляционное объединение (relational union) — это результат выполнения операции конкатенации над двумя наборами записей. Эту операцию можно представить как реляционную «версию» сложения (разумеется, данное утверждение не совсем точно). Результат операции объединения набора записей A и набора записей B — простое добавление всех записей из набора A в набор B.

Например, нужно составить список всех адресов, хранящихся в базе данных, для массовой рассылки почтовых сообщений. В базе данных *Northwind* наборы записей *Customers* и *Employees* содержат адреса покупателей и сотрудников компании, соответственно, поэтому список всех адресов, хранимых в базе данных, можно получить, выполнив операцию объединения над двумя этими наборами записей. Для этого мы используем оператор UNION:

```
SELECT CompanyName AS Name, Address, City, PostalCode
FROM Customers
UNION SELECT [FirstName] ft " " & [LastName] AS Name,
    Address, City, PostalCode
FROM Employees
ORDER BY name;
```

Поле *CompanyName* было переименовано в *Name*, а над полями *FirstName* и *LastName* таблицы *Employees* выполнена операция конкатенации. Поле, полученное в результате конкатенации, также называется *Name*. Для запроса, выполняющего объединение, совсем не требуется, чтобы все поля в <списке_полей> каждого оператора SELECT имели одинаковые имена. Нужно только, чтобы в каждом запросе было одинаковое число полей и чтобы типы этих полей были одинаковыми (или совместимыми). Результаты выполнения объединения в Access показаны на рис. 5-5.

Name	Address	City	PostalCode
Alfreds Futterkiste	Obere Str 57	Berlin	12209
Ana Trujillo Emparedados y helad	Avda de la Constitución 2222	México D.F	05021
Andrew Fuller	908 W Capital Way	Tacoma	98401
Anne Dodsworth	7 Houndstooth Rd	London	WG2 7LT
Antonio Moreno Taqueria	Mataderas 2312	México D.F	05023
Around the Horn	120 Hanover Sq	London	WA1 1DP
Berglunds snabbköp	Berguvsvägen 8	Luleå	S-958 22
Blauer See Delikatessen	Forsterstr 57	Mannheim	68306
Blondel pere et fils	24, place Kléber	Strasbourg	67000
Bólido Comidas preparadas	C/ Araquil, 67	Madrid	28023
Bon app'	12, rue des Bouchers	Marseille	13008
Bottom-Dollar Markets	23 Tsawassen Blvd	Tsawassen	T2F 8M4
B's Beverages	Fauntleroy Circus	London	EC2 5NT

Рис. 5-5. ***Оператор UNION объединяет записи из таблиц***

Пересечение

Операция *пересечения* (intersection operator) возвращает записи, общие для двух наборов. Очевидно, что пересечение можно использовать для поиска повторяющихся записей, что и происходит чаще всего. Пересечение реализуется при помощи внешних соединений,

Рассмотрим такой пример. Есть несколько списков клиентов, полученных из разных источников — старых информационных систем (рис. 5-6). Как правило, в таких таблицах есть повторяющиеся записи.

DuplicateCustomers1

CustomerID	CompanyName
ALFKI	Alfreds Futterkiste
ANATR	Ana Trujillo Emparedados y helados
ANTON	Antonio Moreno Taquería
AROUT	Around the Horn
BERGS	Berglunds snabbköp
BLAUS	Blauer See Delikatessen
BLONP	Blondel pere et fils
BOLID	Bólido Comidas preparadas

DuplicateCustomers2

CustomerID	CompanyName
ALFKI	Alfreds Futterkiste
ANATR	Ana Trujillo Emparedados y helados
ANTON	Antonio Moreno Taqueria
AROUT	Around the Horn
FAMIA	Familia Arquibaldo
FISSA	FISSA Fabrica Inter. Salchichas S.A.
FOLIG	Folies gourmandes
FOLKO	Folk och fä HB
FRANK	Frankenversand

Рис. 5-6. Таблица с повторяющимися данными

Оператор SELECT возвратит повторяющиеся записи:

```
SELECT DuplicateCustomers1 *
FROM DuplicateCustomersl
LEFT JOIN DuplicateCustomers2
ON (DuplicateCustomers1.CustomerID = DuplicateCustomers2.CustomerID)
AND (DuplicateCustomers1.CompanyName = DuplicateCustomers2.CompanyName)
WHERE (((DuplicateCustomers2.CustomerID) IS NOT NULL));
```

Результаты выполнения этого оператора показаны на рис. 5-7.

CustomerID	CompanyName
·4!! I	Alfreds Futterkiste
ANATR	Ana Trujillo Emparedados y helados
ANTON	Antonio Moreno Taqueria
AROUT	Around the Horn

Рис.** 5-7, **Внешнее соединение в сочетании с оператором IS NOT NULL выполняет операцию пересечения

Разность

Операция пересечения используется для поиска повторяющихся записей, а операция нахождения разности — наоборот, для поиска утерянных (осиротевших) строк. *Реляционная разность* (relational difference) для двух наборов записей будет содержать все записи, которые принадлежат к одному набору записей, но не принадлежат к другому.

Например, для двух наборов записей на рис. 5-6 следующий оператор SELECT возвратит все записи, не являющиеся общими для этих двух наборов записей:

```
SELECT DuplicateCustomers1.*
FROM DuplicateCustomers1
LEFT JOIN DuplicateCustomers2
ON (DuplicateCustomers1.CustomerID = DuplicateCustomers2.CustomerID)
AND (DuplicateCustomers1.CompanyName = DuplicateCustomers2.CompanyName)
WHERE (DuplicateCustomers2.CustomerID IS NULL);
```

Здесь операция внешнего соединения возвращает все записи из двух списков. Как уже говорилось, внешнее соединение возвращает значения *Null* для полей, у которых нет соответствующих записей в другой таблице.

В условии **WHERE** используется оператор IS NULL, ограничивающий число возвращаемых записей теми, для которых нет соответствующих записей в другой таблице.

Если описанный способ кажется вам слишком сложным, попробуйте выполнить эту операцию в два шага; сначала создайте внешнее соединение как представление, а затем ограничьте число записей, задав условие **WHERE** (рис. 5-8).

Шаг 1. Создать внешнее соединение

CustomerID	CompanyName
ALFKI	Alfreds Futterkiste
ANATR	Ana Trujillo Emparedados y helados
ANTON	Antonio Moreno Taquería
AROUT	Around the Horn
BERGS	Berglunds snabbköp
BLAUS	Blauer See Delikatessen
BLONP	Blondel pere et fiis
BOLID	Bólido Comidas preparadas

```
Select DuplicateCustomers1.*
FROM DuplicateCustomers1
LEFT JOIN DuplicateCustomers2
ON (DuplicateCustomers1.CustomerID = DuplicateCustomers.CustomerID)
```

Шаг 2. Ограничить число записей теми, которые содержат *Null* в *CustomerID*

CustomerID	CompanyName
BERGS	Berglunds snabbköp
BLAUS	Blauer See Delikatessen
BLONP	Blondel père et fils
BOLID	Bólido Comidas preparadas

```
Select DuplicateCustomers1.*
FROM DuplicateCustomersl
LEFT JOIN DuplicateCustomers2
ON (DuplicateCustomers1.CustomerID = DuplicateCustomers.CustomerID)
WHERE (DuplicateCustomers2.CustomerID) IS NULL
```

Рис. 5-8. Операцию разности можно выполнить в два шага

Декартово произведение

Последняя из рассматриваемых операций над множествами — *декартово произведение.* Как и аналогичная операция в классической теории множеств, декартово произведение двух наборов записей представляет собой соединение каждой записи из одного набора записей с каждой записью из другого набора.

Декартово произведение (называемое также просто произведением) двух наборов записей получают, составив оператор SELECT без оператора JOIN. Например, объединить записи о каждом из покупателей с соответствующими записями о торговых представителях, обслуживающих покупателей, можно следующим образом:

```
SELECT CustomerName, CSRName FROM Customer, CSRs:
```

Декартово произведение широко используют при анализе данных, а также как промежуточный результат, предназначенный для дальнейшей обработки. Однако намного чаще оно получается вследствие ошибки; например, если, программируя на Access, вы забудете вклю-

чить в SQL-запрос соединение. Подобные ошибки часты, поэтому на первых порах не удивляйтесь, если вместо ожидаемого результата у вас будет получаться декартово произведение.

Дополнительные реляционные операторы

С тех пор как впервые были сформулированы принципы, положенные в основу реляционной модели, прошло достаточно много времени. Появились некоторые дополнительные операции, расширившие рамки реляционной алгебры. Мы рассмотрим три такие операции, наиболее часто встречающиеся на практике: *агрегирование* (summarize), *расширение* (extend) и *переименование* (rename). Кроме того, я опишу три дополнительных оператора, поддерживаемых продуктами Microsoft: TRANSFORM, ROLLUP и CUBE.

Агрегирование

Операция агрегирования выдает результаты, которые содержат суммарные данные, сгруппированные по указанным значениям полей. Она применима в любых ситуациях, где нужно рассматривать данные на более высоком уровне абстракции, чем просто уровень данных, хранимых в базе.

Операция агрегирования реализуется при помощи выражения GROUP BY оператора SELECT. Для каждого уникального значения указанного поля или нескольких полей возвращается одна запись. Если указано несколько полей, группы будут вложенными. Рассмотрим следующий оператор:

```
SELECT Categories.CategoryName, Products.ProductName,
    SUM([Order Details].Quantity) AS SumOfQuantity
FROM (Categories INNER JOIN Products ON Categories.CategoryID =
    Products.CategoryID)
INNER JOIN [Order Details:
ON Products.ProductID = [Order Details].ProductID
GROUP BY Categories.CategoryName, Products.ProductName;
```

Этот оператор возвращает по одной записи для каждого продукта в БД *Northwind*, сгруппированные по категориям и содержащие три поля: *CategoryName, ProductName*, и *SumOfQuantity*(число единиц для каждого продаваемого товара), как показано на рис. 5-9.

Поля, перечисленные в <списке_полей> оператора SELECT, должны либо входить в <список_полей_оператора_GroupBy>, либо являться аргументом агрегатной функции SQL. *Агрегатные функции SQL* (SQL aggregate functions) вычисляют суммарные значения для

каждой записи. Наиболее широко используются агрегатные функции AVERAGE, COUNT, SUM, MAXIMUM и MINIMUM.

ЗДЙМУОДШ	ProductName	SumOfQuantity
Beverages	Chai	828
Beverages	Chang	[illegible]
Beverages	Chartreuse verte	6
Beverages	Côte de Blaye	15
Beverages	Guaraná Fantástica	1125
Beverages	Ipoh Coffee	580
Beverages	Lakkalikööri	981
Beverages	Laughing Lumberjack Lager	184
Beverages	Outback Lager	817
Beverages	Rhönbräu Klosterbier	1155
Beverages	Sasquatch Ale	506
Beverages	Steeleye Stout	883
Condiments	Aniseed Syrup	328

Рис. 5-9. Оператор GROUP BY возвращает агрегированные данные

Агрегирование — это еще один «камень преткновения», где значения *Null* могут доставить дополнительные неудобства. Значения *Null* участвуют в вычислениях агрегированных значений — они образуют группу. Однако значения *Null* игнорируются агрегатными функциями. Как правило, это единственная проблема, возникающая, если использовать поле из <списка_полей_оператора_GroupBy> в качестве параметра агрегатной функции.

Расширение

Операция *расширения* (extend) позволяет задать виртуальные поля, значения которых вычисляются на основании констант, хранимых в базе данных, но не хранятся в базе данных как определенные значения, записываемые в файл БД на физическом уровне. Чтобы создать виртуальное поле, можно просто задать формулу для его вычисления в <списке_полей> оператора SELECT, например:

```
SELECT [UnitPrice]*[Qty] AS ExtendedPrice
FROM [Order Details];
```

Вычисления, на которых основываются определения виртуальных полей, могут быть как сложными, так и простыми. Как правило, они выполняются настолько быстро, что хранить значения вычисляемых полей в таблице бессмысленно.

Переименование

И наконец, последняя из рассматриваемых здесь дополнительных операций — *переименование* (rename). Операция переименования может выполняться как над набором записей из <списка_наборов_за-

писей>, так и над отдельными полями из <списка_полей>. В механизме баз данных Microsoft Jet используется следующий синтаксис для переименования набора записей:

```
SELECT <имя_поля> AS <псевдоним_поля>
FROM <имя_таблицы> AS <псевдоним_таблицы>

В SQL Server ключевое слово "AS" опускается:

SELECT <имя_поля> <псевдоним_поля> FROM <имя_набора записей>
<псевдоним_набора_записей>
```

Переименование полей применяется, как правило, когда создается представление или выполняется самосоединение таблицы, например;

```
SELECT Manager.Name, Employee.Name
FROM Employees AS Employee
INNER JOIN Employees AS Manager
ON Employee.EmployeeID = Manager.EmployeeID;
```

Этот синтаксис позволяет разделить на логическом уровне каждый из вариантов использования поля.

Оператор TRANSFORM

Оператор TRANSFORM — это одно из расширений, добавленных Microsoft к операциям реляционной алгебры. Оператор TRANSFORM выполняет операцию поворота на 90" над результатами операции агрегирования (GROUP BY). Этот оператор часто называют *кросс-табличным запросом* (crosstab query), он поддерживается механизмом баз данных Microsoft Jet, в SQL Server до сих пор не реализован.

Вот синтаксис оператора TRANSFORM:

```
TRANSFORM <агрегатная_функция>
SELECT <список_полей>
FROM <список_наборов_записей>
GROUP BY <список_полей_оператора_GroupBy>
PIVOT <заголовок_столбца> [IN (<список_значений>)]
```

Оператор TRANSFORM <агрегатная_функция> определяет агрегированные данные, которые войдут в набор записей. Оператор SELECT должен содержать выражение GROUP BY, но в нем недопустимо выражение HAVING. <Список_полей_оператора_GroupBy> может включать несколько полей. (В операторе TRANSFORM <спи-

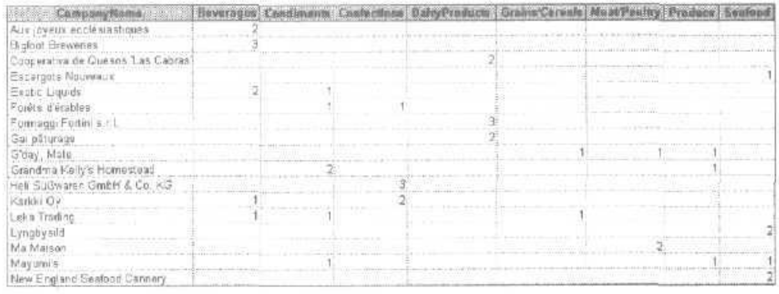
сок_полей> и <список_полей_оператора_GroupBy>, как правило, совпадают).

Выражение PIVOT задает поля, значения которых используются в качестве заголовков столбцов. По умолчанию механизм базы данных Microsoft Jet включает столбцы в набор записей в алфавитном порядке, слева направо. Необязательное ключевое слово IN позволяет задать имена столбцов, причем столбцы будут включены в <список_значений> в том порядке, в котором они были перечислены после ключевого слова IN.

Сравнив результаты выполнения оператора TRANSFORM из следующего примера с результатами выполнения операции агрегирования (рис. 5-9), можно убедиться, что они практически одинаковы.

```
TRANSFORM Count(Products.ProductID) AS CountOfProductID
SELECT Suppliers.CompanyName
FROM Suppliers
INNER JOIN (Categories INNER JOIN Products
    ON Categories.CategoryID = Products.CategoryID)
ON Suppliers.SupplierID = Products.SupplierID
GROUP BY Suppliers.CompanyName
PIVOT Categories.CategoryName;
```

Результат выполнения этого оператора показан на рис. 5-10.



CompanyName	Beverages	Condiments	Confections	DairyProducts	Grains/Cereals	Meat/Poultry	Produce	Seafood
Aux joyeux ecclésiastiques	2							
Bigfoot Breweries	3							
Cooperativa de Quesos 'Las Cabras'				2				
Escargots Nouveaux								1
Exotic Liquids	2	1						
Forêts d'érables		1	1					
Formaggi Fortini s.r.l.				3				
Gai pâturage				2				
G'day, Mate					1	1	1	
Grandma Kelly's Homestead		2					1	
Heli Süßwaren GmbH & Co. KG			3					
Karkki Oy	1		2					
Leka Trading	1	1			1			
Lyngbysild								2
Ma Maison						2		
Mayumi's		1					1	1
New England Seafood Cannery								2

Рис. 5-10. Оператор TRANSFORM выполняет операцию поворота на 90° над результатами операции агрегирования

Оператор ROLLUP

Операция агрегирования, реализованная при помощи оператора GROUP BY, генерирует записи, которые содержат агрегированные данные. Оператор ROLLUP — это логическое расширение операции

GROUP BY, позволяющее получить суммарные значения. Он поддерживается только SQL Server:

```
SELECT Categories.CategoryName, Products.ProductName,
    SUM([Order Details].Quantity) AS SumOfQuantity
FROM (Categories INNER JOIN Products
    ON Categories.CategoryID = Products.CategoryID)
INNER JOIN [Order Details]
ON Products.ProductID = [Order Details].ProductID
GROUP BY Categories.CategoryName, Products.ProductName WITH ROLLUP;
```

На рис. 5-11 показан результирующий набор записей, полученный после выполнения этого оператора.

CategoryName	ProductName	SumOfQuantity
Beverages	Chai	828
Beverages	Chang	1057
Beverages	Chartreuse verte	6
Beverages	Cote de Blaye	15
Beverages	Guaraná Fantástica	1125
Beverages	Ipoh Coffee	580
Beverages	Lakkalikööri	981
Beverages	Laughing Lumberjack Lager	184
Beverages	Outback Lager	817
Beverages	Rhönbräu Klosterbier	1155
Beverages	Sasquatch Ale	506
Beverages	Steeleye Stout	883
Beverages		8137
Condiments	Aniseed Syrup	328
Condiments	Chef Anton's Cajun Seasoning	453

Рис. 5-11. Оператор ROLLUP позволяет получить суммарные значения

Как видите, набор записей на рис. 5-11 фактически тот же, что и на рис. 5-9, к нему лишь добавлены дополнительные записи. Записи, где есть значения *Null* (одна из таких записей показана на рис. 5-11), содержат суммарные значения для групп или подгрупп. Таким образом, всего было продано 8137 единиц напитков.

Оператор CUBE

Оператор **CUBE, как и ROLLUP,** реализован только в SQL Server как расширение оператора **GROUP** BY. Оператор CUBE агрегирует каждый столбец в <списке_полей_оператора_GroupBy> по всем остальным столбцам. Концептуально оператор **CUBE** похож на **ROLLUP, но** в отличие от него, вычисляет суммарные значения для дополнительных групп, а не суммарные значения для каждого столбца, перечисленного в <списке_полей_оператора_GroupBy>.

Предположим, что <список_полей_оператора_GroupBy> состоит из трех полей — А, В, и С. Тогда оператор CUBE вернет следующие семь агрегированных значений:

- суммарное значение для всех значений столбца С;
- суммарное значение для всех значений столбца С, сгруппированных по А;
- суммарное значение для всех значений столбца С, сгруппированных по С в А;
- суммарное значение для всех значений столбца С, сгруппированных по В в А;
- суммарное значение для всех значений столбца С, сгруппированных по В;
- суммарное значение для всех значений столбца С, сгруппированных по А в В;
- суммарное значение для всех значений столбца С, сгруппированных по С в В.

Итоги

Эта глава была посвящена различным операциям над базовыми отношениями, реализованным при помощи реляционных операторов и расширений этих операторов, реализованных в языке SQL. Кроме того, мы обсудили некоторые особенности реляционных операторов и их расширений, связанные с трехзначной логикой и значениями *Null.*

Ограничение, проекция и соединение — это стандартные реляционные операции, возвращающие подмножества одного набора записей. Соединение, объединение, пересечение, разность и произведение — это реляционные операции, позволяющие объединять наборы записей разными способами. Все эти операции (за исключением разности) реализуются при помощи SQL-оператора SELECT. Разность тоже можно реализовать при помощи оператора SELECT, но иногда для этого приходится применять особые методы.

Кроме того, мы рассмотрели несколько дополнительных операторов. Агрегирование и расширение относятся к операциям, выполняющим различные вычисления с данными. Переименование позволяет ввести заголовки столбцов, отображаемые в проекции. Операторы TRANSFORM, ROLLUP и CUBE — это расширения языка SQL, реализованные Microsoft. Каждый из этих операторов представляет собой отдельный способ агрегирования и представления данных.

Итак, этим обзором реляционной алгебры мы завершаем первую часть книги. Реляционная теория достаточно сложна, и конечно, я

осветила далеко не все ее вопросы. Подробное рассмотрение реляционной теории баз данных выходит за рамки этой книги, однако все основные положения этой теории изложены в первых пяти главах.

Далее мы перейдем к практическим аспектам разработки **СУБД** и пользовательского интерфейса.

Проектирование реляционных систем баз данных

Процесс проектирования

ГЛАВА

В первой части книги мы рассмотрели основные принципы проектирования реляционных баз данных. Однако структура данных — это критически важная, но все-таки всего лишь одна из многих составляющих базы данных. Рассмотрим другие аспекты проектирования базы данных.

В этой части мы обсудим анализ и проектирование баз, включая определение параметров системы и рабочих процессов, создание концептуальной модели и схемы базы данных. Проектирование пользовательского интерфейса освещено в третьей части, поскольку это отдельная и весьма сложная тема.

Физическая реализация в книге не рассматривается, но и анализ, и проектирование не изолированы от процесса реализации системы на физическом уровне. Так что начнем с краткого обсуждения жизненного цикла проекта в целом.

Модели жизненного цикла

До последнего времени системные аналитики использовали в своей работе парадигму, известную как модель водопада. Существует несколько реализаций такой модели, один из наиболее простых изображен на рис. 6-1.

Процесс начинается с системного анализа, иногда называемого анализом требований, так как его цель — выяснить требования конечных пользователей к системе. По окончании системного анализа его результаты одобряют участники проекта. Затем производится детальное проектирование системы. Далее уточняются сроки и финансовый план, система создается, тестируется, и поставляется заказчику.

Рис. 6-1. Модель водопада

Модель водопада эстетически привлекательна. Перед началом каждой новой стадии проекта предыдущая стадия должна быть обязательно закончена, и ее результаты — утверждены. Такой подход позволяет хорошо контролировать бюджет, персонал и рабочее время. Предоставьте результаты проекта в срок, уложитесь в бюджет — и ваш заказчик, по всей вероятности, будет вполне доволен.

Проблема, разумеется, в том, что реальность резко отличается от идеала. Модель предполагает, что вся необходимая для завершения определенной задачи информация будет доступна на этапе выполнения, какие-либо дополнительные факторы не повлияют на процесс. Что, конечно, маловероятно (за исключением, быть может, небольших информационных систем).

Модель водопада не позволяет также вносить изменения в бизнес-требования на этапе создания системы. Но полагать, что система, которая удовлетворяла всем бизнес-требованиям в начале проектирования, будет все еще удовлетворять им в конце двухлетнего периода разработки — безрассудство. Заказчику совсем не понравится, если

вы поставите ему систему, которую нельзя использовать, даже если вы уложитесь в сроки и бюджет.

Тем не менее отдельные стадии, из которых состоит модель водопада, совершенно необходимы. Если вы пропустите хоть одну, то непременно провалите проект. Недостаток модели водопада в том, что она линейна: ни одна из стадий не может быть проведена повторно, и ее результаты нельзя пересмотреть.

Есть несколько альтернативных моделей. Спиральная модель предлагает осуществлять итерации, то есть каждая последующая стадия позволяет уточнить результаты предыдущей (рис. 6-2).

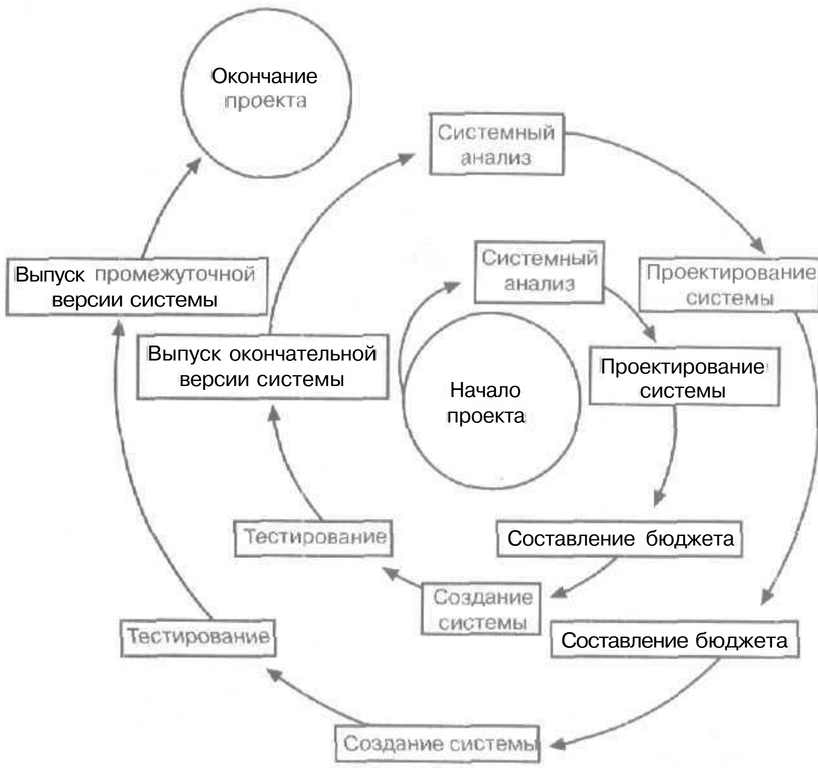


Рис. 6-2. Спиральная модель

Недостаток спиральной модели в том, что нельзя предвидеть изменения, которые возникнут на поздних стадиях разработки — а ведь они, вполне вероятно, могут сделать бесполезной всю предыдущую работу. Результат — перерасход бюджета и разочарованные разработ-

чики. Такая ситуация крайне опасна для проектов баз данных, где расширение требований может изменить семантику данных, а изменения в схеме базы данных — повлечь за собой изменения, катастрофические для системы в целом.

Модель, которую я рекомендую для проектирования больших систем, называется методом приращений или эволюционной разработкой (рис. 6-3).

Рис. 6-3. Эволюционная разработка

В этой модели, которая чаше всего является просто вариантом спиральной модели, предварительный анализ выполняется для системы в целом, а не для ее части. За этим следует проектирование архитектуры — опять же для системы в целом. Цель этого этапа — определить отдельные компоненты, которые можно реализовать более или менее независимо от остальных составляющих системы, а затем описать зависимости и способы взаимодействия этих компонентов. Детальное проектирование и реализация каждого компонента выполняются, лишь когда система в целом отвечает предъявляемым к ней требованиям. На этой стадии я и использую спиральную модель, так как она наиболее гибка.

В данном случае спираль включает дополнительное действие – интеграцию. Разумеется, интеграция компонентов присутствует неявно в спиральной модели, но задача интеграции сильно усложняется при использовании метода приращений. Это одна из причин, почему я предпочитаю спиральную модель при разработке компонентов. Детальная проработка компонента непосредственно перед тем, как он будет разработан, позволяет приспособить компоненты к любым требованиям, сформулированным во время предыдущего интеграционного процесса, и избежать проблем, возникающих при обычной интеграции.

Метод приращений предполагает, что любая большая система разделяется на ряд компонентов, но это не обязательно справедливо для всех без исключения систем. Может потребоваться написать много вспомогательного кода. Например, такая ситуация: экран ввода данных должен вызывать компонент Microsoft ActiveX, который будет осуществлять поиск кода заказчика и выполнять некоторые действия, если код не найден. Но компонент ActiveX еще не создан. Разработчики вынуждены написать дополнительный код, позволяющий осуществить вызов методов без ошибок. Сложные системы могут включать значительный объем такого вспомогательного кода.

А ведь кроме того, есть вероятность, что компонент ActiveX так никогда и не будет реализован: не выделят денег на его создание, или вы обнаружите, что он не нужен. Тогда придется создать компонент, чьей единственной функцией будет поддержка работы других компонентов. Не очень элегантное решение, и вам определенно не захочется рассказывать об этом специалисту по технической поддержке.

Так как анализ и проектирование архитектуры выполняются в начале проекта, есть риск, что полученные результаты устареют (один из главных недостатков модели водопада). А значит, важно пересмотреть результаты этих двух стадий до того, как начнется детальное проектирование компонентов (особенно это относится к анализу требований, так как требования изменяются чаше всего). Внести изменения в еще не разработанные компоненты легко, можно даже просто изменить последовательность, в которой они создаются, не разрушая при этом всего, что было сделано ранее.

У метода приращений множество преимуществ. Так как целостное представление о системе определяется в самом начале процесса, уменьшается до минимума вероятность сделать работу впустую. Крупные проекты разбивают на малые части, а отдельными частями легче управлять. Например, вы сможете поставить пользователю ядро системы на ранней стадии проекта. Система начнет окупать себя, и это будет способствовать дальнейшему финансированию проекта заказчиком.

Проектирование базы данных

Какой бы метод проектирования вы не выбрали, придется выполнить определенную работу по анализу и проектированию. Делаете ли вы это последовательно или итеративно, состоит ли задача в разработке системы в целом или ее отдельной части, является ли ваша технология формальной или неформальной, — хотя бы однажды нужно все пройти эти стадии проекта.

Определение параметров системы

В идеале, необходимо ясно представлять: чего, зачем и как вы хотите достичь. Формулирование целей проекта — самая первая стадия создания системы. Определив цель проекта, вы сможете четко очертить его границы, то есть обозначить задачи, которые нужно решить в ходе проекта. Тогда станут яснее способы реализации, то есть как вы будете создавать систему. Все этих аспекты: цели, задачи, средства, - обсуждаются в главе 7.

Проектирование рабочих процессов

Внешне база данных выглядит, как инструмент сохранения и вывода информации и поддерживает один или несколько рабочих процессов. Пользователи сохраняют данные не ради самих данных — они хотят их использовать. Понимание того, какие данные требуются для поддержки рабочих процессов, критически важно для понимания семантики модели данных. Рабочие процессы обсуждаются в главе 8.

Построение концептуальной модели данных

Концептуальная модель данных больше, чем набор структур, она определяет, как данные будут использоваться в системе. Концептуальная модель включает не только логическую модель данных, но и то, как рабочие процессы взаимодействуют с данными. Это тема главы 9.

Подготовка схемы базы данных

Схема базы данных является, по сути, переводом логической модели данных на язык физической реализации. Она включает описание таблиц, которые будут созданы в системе, а также физическую структуру данных. Физические структуры данных и схемы баз данных подробно обсуждаются в главе 10.

Проектирование пользовательского интерфейса

Независимо от того, насколько технически совершенна ваша система, если пользовательский интерфейс выполнен грубо, непонятен или неудобен, проект вряд ли будет успешен. Все-таки для большинства

пользователей именно интерфейс является системой, с которой они работают. Проектирование пользовательского интерфейса обсуждается в части 3.

Замечания о стандартах и технологиях проектирования

Я не большая поклонница четко очерченных методов проектирования компьютерных систем. По-настоящему хорошо спроектированные системы — это те, где аналитик с самого начала активно включается в работу с пользователями.

Тем не менее, некоторые общие процедуры для управления самим процессом проектирования необходимы, особенно для больших проектов, в которых занято множество аналитиков и программистов. Существуют разные методы такого управления, для большинства из которых есть автоматизированные средства поддержки. Я не намерена давать каких-либо рекомендаций. Во-первых, это вопрос веры, во-вторых, само существование таких методов намного важнее конкретного выбора.

Создание документации также надо подчинить определенным правилам, по меньшей мере, в нескольких первых ваших проектах. В главе 11 приводятся общее описание этого процесса и некоторые рекомендации. Если вам нужна более подробная информация, используйте аналитические таблицы, включенные в приложение (см. компакт-диск, прилагаемый к книге).

Определение параметров системы

«Любая система предназначена для решения *конкретных*, а отнюдь не *всех* задач». Эти слова принадлежат Роберту Холлу (Robert Hall), сотруднику известной американской компании «North American Aviation».

Говорят, что эту фразу Роберт Холл произнес в разговоре с вице-президентом компании, пытавшимся расширить границы его проекта сверх разумных пределов, и это возражение едва не стоило Холлу работы. (В системном проектировании, как и в любой другой области, следует учитывать, к какой аудитории вы обращаетесь). Как бы то ни было, я целиком разделяю мнение Холла о процессе разработки систем. Если вы все-таки намерены успешно выполнить свой проект, то должны очертить его рамки. Без четкого понимания, что именно вы хотите сделать, неизбежны серьезные трудности.

Итак, прежде чем приступать к работе, определите параметры системы, а именно:

- *цели* — не то, зачем создается данная система, а цели проекта как целого;
- *критерии проектирования* — они будут использоваться в процессе оценки компромиссов (которые неизбежны при разработке и внедрении системы) а также возможностей ее реализации;
- *границы применения* — какие функции будут реализованы в данной системе, а какие нет.

Цели и границы применения системы

Казалось бы, определить цели и границы применения системы достаточно просто. Если вам повезет, так и будет — иногда эти параметры четко ясны еще при составлении первоначального плана. Однако

чаше этот процесс чрезвычайно сложен. Формальные аналитические технологии и гибкие подходы, используемые на стадии разработки, сочетаются с дипломатическим талантом системных аналитиков.

Цель процесса разработки, как правило, и есть наиболее важный фактор в определении границ применения системы и критериев ее оценки. В конечном итоге, это набор аргументов в пользу ее создания. Очевидно, вы не сможете принять обоснованного решения по какому-либо конкретному вопросу, до тех пор пока не будет окончательно ясно, чего, собственно, вы хотите достигнуть в итоге.

Не следует смешивать понятия «цель» и «краткое описание». Краткое описание системы, как и бюджет, составляют обычно в самом начале проекта. Например, при разработке решения, позволяющего автоматизировать процесс регистрации заказов, поступающих от покупателей, краткое описание системы будет сформулировано как «автоматизация регистрации заказов, поступающих от клиентов фирмы». Но отнюдь не это является целью создания данной системы. Цель — это фактор или набор факторов, обусловливающих необходимость реализации данного проекта.

По правде говоря, само понятие «цель» в применении к системе не вполне корректно. В большинстве случаев у систем несколько целей, которые могут быть как четкими и конкретными, так и не вполне. И выявить их непросто. Для чего вообще нужно автоматизировать систему регистрации заказов? Для ускорения процесса регистрации? Для повышения точности и сохранности вводимых данных? Для снижения накладных расходов? Чтобы поднять авторитет компании среди пользователей этой системы? Чтобы менеджер чувствовал, что «идет в ногу со временем»? Вполне вероятно, что в данном случае создание системы будет преследовать все эти цели, а также множество других.

Впрочем, я отнюдь не предполагаю, что вы собираетесь ставить людей в неловкое положение: задавать им нескромные вопросы, посягать на информацию, составляющую коммерческую тайну, и т. д. Поэтому некоторые из целей, очевидно, так и останутся вам неизвестны, по той простой причине, что они вас не касаются. Например, вам вовсе не обязательно знать, что начальник отдела фирмы, для которой вы разрабатываете систему, активно пользуется Интернетом. Гораздо более важно следующее: чтобы система оправдывала себя, среднее время обработки одного заказа должно сократиться с 10 до 2 минут.

Кроме того, может потребоваться уточнить несколько моментов, которые менеджеры по продажам и сотрудники отдела маркетинга, как правило, скрывают за неопределенными выражениями, напри-

мер «позиционирование продукта» и «удовлетворение нужд и запросов клиента». К счастью, это довольно легко — вы можете просто расспросить об этом представителя фирмы — заказчика проекта.

Четко сформулировать цели очень важно — ведь потом вам придется перевести их на язык простых и понятных критериев, используемых в процессе разработки. Порой достаточно всего лишь задать несколько дополнительных вопросов. Например, цель, сформулированная как «помощь в удовлетворении нужд и запросов клиента», обычно означает, что в компании существуют проблемы с обслуживанием клиентов, которые легко перевести на язык вполне определенных критериев. Либо вовсе не учитывать, если они никак не связаны с разрабатываемой системой или не могут быть решены с ее помощью.

Допустим, у вашего заказчика трудности с соблюдением сроков поставок, потому что сотрудники отдела продаж назначают совершенно нереальные сроки, только бы заключить сделку. В этом случае автоматизированная система регистрации заказов может оказаться весьма полезной. Например, вы можете ввести ограничение на сроки поставок таким образом, чтобы существовал достаточный промежуток времени между приемом заказа и запланированной датой поставки товара, гарантирующий возможность выполнения заказа. Но если же трудности с соблюдением сроков поставок связаны с процессом контроля за качеством производимой продукции, автоматизированная система регистрации заказов вряд ли поможет, и вам обязательно нужно разъяснить это заказчику. Это отнюдь не означает, что от создания системы следует отказаться, важно лишь объяснить всем заинтересованным лицам, что такая система не поможет решить данную проблему.

Не все нечетко сформулированные цели легко перевести на язык вполне определенных критериев. Яркий пример — «позиционирование». Например, вас могут попросить создать Web-сайт, чтобы «позиционировать компанию на уровне современных требований». Подобные фразы, как правило, означают лишь то, что самого факта наличия системы достаточно для достижения цели. Это легко проверить — просто спросите заказчика, каким образом можно определить, достигнута ли цель. В большинстве случаев окажется, что цель достигнута, когда достигнуты все цели, связанные с конкретными параметрами — например, производительностью и функциональностью системы. И тогда нечетко сформулированные цели достигаются автоматически.

Часто при постановке цели используют слова «увеличить» или «снизить». Говорят, например, что целью разрабатываемой системы является «повышение эффективности» и «увеличение производительности» — согласитесь, довольно расплывчатая формулировка. Вопрос: «Как определить, что цель достигнута?» — выручит и здесь. Один клиент как-то рассказал мне забавную историю, наглядно демонстрирующую, сколь важны количественные критерии оценки выполняемой работы для достижения адекватного результата. (Критерии, которыми пользуется разработчик, и критерии оценки выполненной работы во многом схожи — ведь схожи и конечные цели, которым они служат). История такова: в начале своей карьеры, будучи торговым агентом, рассказчик получал указания от менеджера. Менеджер объяснил молодому торговому агенту, что в его служебные обязанности входит «продвижение товаров и услуг компании». Выйдя за дверь, торговый агент начал громко выкрикивать нечто вроде «Покупайте наши товары, они самые качественные!» Очевидно, инструктируя подчиненного, менеджер имел в виду совсем не это.

На самом начальном этапе анализа определите, до какой степени улучшать те или иные параметры. В противном случае вы рискуете выйти за рамки бюджета. Если цель разрабатываемой системы — увеличить эффективность, то насколько именно? Нужно повысить производительность? Отлично. Каковы нынешние показатели и какими они должны стать, когда система будет запущена в эксплуатацию?

Здесь вы снова можете столкнуться с неожиданными затруднениями. Конечно, принцип, что каждая цель должна в конечном итоге быть выражена некоторой измеряемой величиной, звучит весьма убедительно. Цель «уменьшить время, необходимое для обработки заказа, с 10 до 3 минут» гораздо более ясная, чем абстрактное «увеличение эффективности». Но в таком случае предполагается, что вам уже известно, сколько времени требуется для обработки заказа. А ведь выяснение этого обстоятельства может потребовать немало времени и сил. К тому же нередко подобные исследования весьма недешевы и их стоимость превышает риск сделать ошибку. В случае, подобном тому, о котором мы сейчас говорим (автоматизация регистрации заказов) вряд ли стоит нанимать команду аналитиков, чтобы точно выяснить, сколько времени занимает ввод заказа. А вот несколько лет назад я участвовала в проекте, где было потрачено около 50 тыс. долларов, чтобы определить, оправдано ли приобретение коробочного программного продукта по розничной цене 2,5 тыс.

Итак, чтобы превратить неясные требования к системе в четкие критерии, следует призвать на помощь здравый смысл, чувство со-

размерности и руководствоваться принципом «Лучшее — враг хорошего». Если от внедрения новой системы зависят коммерческий успех компании или чья-либо карьера, следует проявить особую осторожность, тщательно продумывать и планировать свои действия. Если же речь идет о системе, не затрагивающей главные бизнес-процессы компании, можно позволить себе меньшую осмотрительность. Вернемся к автоматизированной системе регистрации заказов: вам вполне достаточно знать, что работая вручную, один сотрудник в среднем обрабатывает и регистрирует 25 заказов в день. Скорее всего, здесь не потребуется углубленный анализ — эту цифру вам сообщит начальник отдела.

Обязательно следует выяснить, почему вообще требуется улучшать те или иные показатели. Возможно, компания столкнулась с такой ситуацией: в связи с ростом объемов продаж сотрудники не успевают зарегистрировать все поступающие заказы, и руководители стоят перед выбором — взять ли на работу еще несколько человек или попытаться ускорить процесс регистрации заказов. Зная об этом и получив цифры ожидаемого увеличения объема продаж, вы можете оценить, насколько должно сократиться среднее время регистрации заказа.

Но вдруг результат, которого вам удастся достичь, будет отличаться от того, на что ориентировался заказчик? Очевидно, если вас просят вдвое сократить среднее время обработки заказа, то придется сделать все возможное, чтобы достичь желаемого результата. Но если вы точно знаете, что на самом деле необходимо сократить время обработки лишь на 25%, то обстоятельства гораздо менее вас стесняют, и вы можете пожертвовать «чистой» производительностью, немного увеличив среднее время обработки заказа. Зато система будет удобнее в эксплуатации или повысится ее надежность.

Вряд ли заказчики предъявят вам чрезмерные или невыполнимые требования, станут сознательно вводить в заблуждение или же ставить в вину просчеты, в которых вы не виноваты. Но в любом случае вы обязаны помочь клиенту уяснить, какие проблемы разрабатываемая система решит, а какие — нет. Программисты и аналитики часто считают, что жизнь была бы прекрасна, если бы заказчики точно знали, чего хотят. На самом деле, заказчики действительно знают, чего хотят, они не знают только одного — как перевести то, чего они хотят, на язык компьютерной системы. Как раз в этом и заключается ваша задача.

Я также часто слышала о заказчиках, которые встречают вас с готовыми набросками экранных форм и отчетов. В этом случае вам сообщается уже готовое решение, а не сама проблема. Такая ситуация

требует немало такта и терпения: нужно «нащупать» проблему, ни разу при этом не намекнув, что тот, кто рисовал эти экранные формы, либо полный профан, либо изначально выбрал неверный путь.

Я рекомендую нащупать брод, прежде чем переправляться через реку. Если заказчик упорно не желает отвечать на ваши вопросы, объясните ему, что ваша работа принесет пользу только в том случае, если вы вникнете в бизнес-процессы компании. Если же и это не поможет, придется либо реализовать систему так, как того желает заказчик, либо отказаться от проекта (последнее, понятно, не всегда возможно). Самое большее, на что вы можете рассчитывать — это пересмотр готового решения, которое вам предложили. Если вы обнаружите в нем серьезные недочеты, обсудите их с заказчиком. Скажите, например: «Я не могу этого сделать; однако я могу сделать то-то или то-то. Какой вариант вам больше подойдет?»

Как правило, для систем баз данных процесс выявления целей нетипичен. Такие системы отличаются от всех прочих в первую очередь тем, что в них изначально присутствуют (хотя чаще всего как побочный продукт) данные об организации. Эти данные, будь то список подписчиков или серия форм для оформления счетов, имеют особую ценность для компании как в рамках процесса, который они непосредственно поддерживают, так и вне оных.

Безусловно, я отнюдь не предлагаю создавать централизованное хранилище данных, охватывающее все бизнес-процессы предприятия, в рамках каждого проекта. Но будет не лишним проверить, имеют ли данные, используемые в разрабатываемой системе, определенную ценность в каких-либо других областях деятельности этой организации, или в других процессах в той же самой области. (Подобные случаи встречаются гораздо реже, чем можно было бы ожидать).

Например, если в автоматизированной системе регистрации заказов ведется список клиентов, и отделу продаж нужен список рассылки для отправки информационных бюллетеней, имеет смысл предоставить сотрудникам отдела продаж доступ к списку клиентов, который они могут использовать для составления списка рассылки. Разумеется, речь отнюдь не идет о добавлении функциональных возможностей списка рассылки в систему регистрации заказов.

Я упомянула об этой проблеме вот по какой причине. Иногда чтобы существенно расширить круг пользователей системы, достаточно совсем небольших изменений в структуре данных. Подробней об этом мы поговорим в главе 9, а сейчас приведу лишь простой пример.

Когда мы обсуждали понятие «атомарные значения», я говорила, что в рамках определенной модели адрес может быть реализован в

виде большого двоичного объекта — набора символов, которые просто выводятся на печать при подготовке конвертов для массовой рассылки, Реализация адреса как одного или нескольких атрибутов зависит от семантики системы, и при создании списка адресов для местного рок-клуба разумно реализовать адрес как один атрибут. Но если эти данные предполагается использовать где-либо еще, то скорее всего, адрес будет представлен в виде нескольких атрибутов. Это усложнит систему, зато позволит избежать повторного ввода одной и той же информации.

Если вы действительно решили внести небольшие изменения, следует тщательно взвесить все «за» и «против». Этот подход приемлем, только если не потребует значительных усилий по переделке системы, и если использование одних и тех же данных для разных целей действительно возможно. Я сталкивалась с не оптимально спланированными системами, где пользователю приходилось вводить значительный объем данных, не имеющих прямого отношения к тому процессу, в котором он непосредственно участвовал. А все из-за того, что эти данные, по всей вероятности, будут использоваться еще где-то. Поэтому, планируя изменения, связанные с дальнейшим развитием компании, не усложняйте чрезмерно уже существующую систему.

Разработчики и аналитики всегда должны помнить, что цели могут изменяться в процессе реализации проекта, и быть готовыми пересмотреть их на более поздних стадиях проектирования.

Для любого проекта, рассчитанного на срок от нескольких недель и более, весьма вероятны изменения бизнес-требований в процессе реализации. Это может быть обусловлено многими объективными факторами: например, реальные объемы продаж отличаются от запланированных, в компании проводится набор сотрудников, а не сокращение штатов и т, п. Как правило, в процессе реализации долговременных крупных проектов аналитики неоднократно проверяют, не произошло ли радикальных изменений в организации бизнеса.

Даже для проектов, рассчитанных на короткие сроки, существует вероятность обнаружить, что цели, определенные в самом начале, неверны или недостижимы. Конечно, хотелось бы получать информацию именно в тот момент, когда она необходима. Однако в действительности вы лишь постепенно подходите к пониманию того, что требуется для системы. Новая, непрерывно поступающая информация, часто приводит к пересмотру целей проекта. Поэтому как можно чаще проводите пересмотр всей информации о системе и определенных для нее целей. Кстати, результатом может быть и подтверждение намеченной цели.

Определение критериев разработки

После того как обозначены все цели проекта, можно приступать к определению критериев разработки. Как правило, сохранить строгую последовательность действий, запланированную изначально, не удается и критерии разработки частично будут определяться еше на этапе определения целей. Но мы, для простоты, договоримся, что у вас есть готовый список целей проекта, и теперь необходимо составить список критериев.

Критерии разработки указывают, достигнуты ли поставленные цели. Все критерии должны соответствовать одной или нескольким целям проекта. Если вдруг обнаруживается, что это не так — вы, возможно, упустили из виду какую-нибудь важную цель.

Обнаружить такое несоответствие весьма непросто даже опытному аналитику. Всегда существует опасность, что аналитик не получает нужных сведений, потому что даже не подозревает об их существовании. Подобные случаи нередки там, где аналитики являются сторонними консультантами и мало знакомы с внутренней жизнью организации, для которой проектируют систему.

Как правило, критерии разработки формулируются в одной из следующих форм.

- **Требования, выражающиеся в измеряемых единицах,** например «Отчет должен быть создан и распечатан за время, не превышающее двух часов».
- **Критерии, определяемые внешним окружением,** например «Система должна работать в существующей компьютерной сети».
- **Основные направления разработки,** например «Предоставление пользователю контекстно-зависимой справки».

Эта классификация отнюдь не является строгой и совершенно обязательной. Но внимательно рассмотрев перечисленные виды критериев, вы сможете оценить, насколько хорошо понимаете саму систему. Большинство критериев относится к первой или второй категории. Если же у вас есть только список основных направлений разработки, боюсь, вам еще далеко до четкого понимания проблемы.

ПРИМЕЧАНИЕ В какой бы форме ни выражались определяемые вами критерии, следует остановиться, как только они будут выполнены – проект закончен, можно отдохнуть. На первый взгляд, совет тривиален. Но давайте рассмотрим конкретный пример. Представим себе, что вы оптимизируете фрагмент программного кода. Чтобы выполнялись критерии разработки, некоторая функция должна вычислять определенное значение за время, не превышающее 10 секунд. Вам уда-

лось добиться того, что значение вычисляется за время, не превышающее 9 секунд, но вы уверены, что стоит приложить еше немного усилий — и вы сократите время вычисления вдвое. **Не** делайте этого. Или же, если в этом действительно возникнет необходимость, сделайте это позже. Как только система начала удовлетворять всем определенным критериям, все работы должны прекратиться, иначе вы никогда не завершите проект.

Пожалуй, единственное исключение из этого правила — исследовательские проекты, но у них обычно другие цели и критерии. Скорее всего, критерий будет сформулирован не как «добиться, чтобы время вычисления не превышало 10 минут», а «определить оптимальный метод вычисления некоторого значения». Поскольку в любом случае вы не можете быть абсолютно уверены, что решение является оптимальным, то и требование критерия никогда не будет выполнено. Таким образом, вы будете продолжать исследования до бесконечности или, точнее, пока не выйдете за рамки бюджета.

На этой стадии проекта очень важно не привязываться к определенному решению или архитектуре. Возможно, вы уверены, что будете использовать Microsoft Transaction Server для обеспечения масштабируемости системы, но это архитектурное решение, а не критерий разработки. Критерием разработки является то, что система должна поддерживать *x* пользователей одновременно.

Если у вас возникают какие-либо сомнения в правильности определения критериев разработки спросите себя, например, можно ли считать систему полностью завершенной, если вы будете использовать Microsoft Transaction Server? Может быть и да, но о том, что система полностью завершена, вы едва ли будете судить на основании того факта, что она реализована при помощи Microsoft Transaction Server. Скорее всего, решение об окончании работ будет зависеть от ответа на другой вопрос: «Если система успешно поддерживает одновременную работу *x* пользователей, можно ли считать, что она завершена?» Положительный ответ на этот вопрос, скорее всего, будет означать, что для данной системы выполняются и остальные критерии разработки.

Критерии, выражаемые в измеряемых единицах

Я уже упоминала, насколько важно определить критерии, выражаемые в измеряемых единицах. Если вы успешно справитесь с этой задачей, многие оставшиеся критерии определятся сами собой. Например, если цель — сократить время вычисления в два раза, и на данный момент время вычисления составляет 10 минут, очевидно, что

критерий разработки будет формулироваться так: «Обеспечить время вычисления, не превышающее 5 минут».

Иногда сложно различить цели, определяемые конкретными значениями системных параметров, и критерии, выражаемые в измеряемых единицах. Однако я не помню случаев, чтобы подобная путаница приводила к сколько-нибудь серьезным последствиям для проекта. Безусловно, никто не будет возмущаться системной спецификацией, где некоторая величина будет выступать одновременно и в качестве цели, и в качестве критерия. Но если вы обнаружите, что для нескольких целей проекта, определяемых конкретными значениями системных параметров, не удается определить соответствующие критерии, выражаемые в измеряемых единицах, это должно послужить сигналом, что в проектируемой системе, очевидно, что-то не так.

При определении критериев разработки в измеряемых единицах не следует слишком углубляться в детали, относящиеся непосредственно к реализации. Например, может оказаться, что для завершения некоего процесса за время, не превышающее 1 минуты, необходимо, чтобы время выполнения одного из запросов не превышало 10 секунд. Но ограничение, налагаемое на время выполнения запроса, относится к реализации, а не к критериям разработки, и на данной стадии вы еще не настолько четко представляете себе все детали проекта, чтобы принимать решения, относящиеся к реализации.

Критерии, определяемые внешним окружением

Большинство ограничений, налагаемых внешним окружением, связаны с вычислительными средами — операционными и другими системами, с которыми придется взаимодействовать разрабатываемой системе. Случаи, когда приходится начинать проектирование с чистого листа, весьма редки. Вероятнее всего, у вашего клиента уже есть установленное и налаженное оборудование и программное обеспечение, и предполагается, что ваша система будет работать в этой среде.

Очень важно правильно оценить объем обрабатываемых данных. Однажды, работая независимым консультантом в региональном представительстве компании, занимавшейся продажей компьютеров и комплектующих к ним, я допустила серьезнейшую ошибку. Требовалось составить техническое предложение для разработки системы регистрации и учета числа продаж. Обсудив с заказчиком требования к системе, я составила предложение с описанием системы на основе Microsoft Access 2.0, позволяющей регистрировать число продаж в региональном представительстве. Но затем выяснилось, что нужно было разработать систему, позволяющую вести учет продаж в рамках всей компании, к тому времени насчитывавшей более 500 региональ-

ных представительств и имевшей оборот в несколько десятков миллионов долларов — такую систему, очевидно, было невозможно реализовать при помощи Microsoft Access 2.0. Я ошиблась, предположив, что заказчику требовалось лишь регистрировать и учитывать объемы региональных продаж. Полагаю, излишне говорить, что после этого фирма расторгла контракт со мной.

При оценке объема обрабатываемых данных следует обращать внимание на два основных фактора. Первый — это «чистый» объем данных, а второй — приращение объема данных и его динамика. Например, в библиотеке находится несколько миллионов томов, но каждый день ее фонд пополняется лишь несколькими книгами. Поэтому, хотя общий объем записей библиотечной картотеки весьма значителен, ежедневно в базу будет добавляться лишь несколько новых записей. Для системы регистрации заказов все может быть с точностью до наоборот. Каждый день в эту систему добавляется несколько сотен записей, однако архивироваться эти записи будут только после завершения фактических сделок, и таким образом «чистый» общий объем данных не превысит нескольких тысяч записей. Очевидно, что основные направления разработки для этих двух систем будут существенно различаться.

При планировании системы важно учитывать, что ресурсы, выделяемые на поддержку данных, зависят от объема данных, хранимых в системе. Здесь главное — не ошибиться в меньшую сторону. При оценке объема, который предполагается выделить для хранения данных, я рекомендую взять за основу цифру, на 10% превышающую максимум, полученный при проведении предварительного анализа. Эти 10% будут использоваться для выполнения различных служебных операций по поддержке данных. Для систем меньшего масштаба следует предусмотреть «резерв» в 20–25% от основного объема данных. Чем больше масштаб системы, тем меньшую роль, как правило, играет объем данных. Хорошо спланированная клиент-серверная система может поддерживать объемы и в 10 тыс., и в 100 тыс. записей без заметного изменения производительности. А вот распределенная многопользовательская система, реализованная на Access, рассчитана на поддержку лишь нескольких десятков тысяч записей, и, вероятнее всего, вам не удастся масштабировать ее до нескольких миллионов записей, вне зависимости от того, насколько хорошо она спланирована.

Основной фактор оценки критериев, определяемых внешним окружением — число пользователей системы. Для большинства систем важны несколько различных категорий пользователей, и вам, очевид-

но, потребуется формулировать системные требования для каждой. Например, в системе, предназначенной для регистрации и обработки заказов, одни пользователи буду? вводить данные о поступивших заказах, другие — запрашивать систему о статусах заказов и обновлять хранимые в системе данные, и наконец, третья группа пользователей будет генерировать отчеты с использованием всех данных, хранимых в базе. Для каждой из этих групп требуются различные функции поддержки со стороны системы, и соответственно, различные критерии разработки.

Число пользователей, подключенных к системе, и число пользователей, активно работающих с ней, как правило, существенно различаются, Например, механизм баз данных Microsoft Jet может поддерживать не более 255 пользователей, одновременно подключенных к базе данных, то есть база данных может быть открыта не более чем 255 пользователями одновременно. Однако это совсем не означает, что все 255 человек смогут одновременно обновлять эту базу данных.

Основные направления разработки

Некоторые цели проекта совсем не просто перевести на язык чисел и измеряемых величин. Например, такую цель как «повышение точности ввода данных», довольно трудно выразить в количественных характеристиках. В этом случае определение числа допущенных ошибок может оказаться просто бесполезным, поскольку стоимость такого исследования превзойдет получаемую от него выгоду.

Но и игнорировать такие цели нельзя, их надо просто иначе сформулировать: не как критерии, выражаемые в измеряемых единицах, а как направления разработки. Например, так: «Повысить точность ввода данных, используя везде, где только возможно, не ввод данных вручную, а списки, из которых пользователь будет выбирать значение». Или так: «Снизить вероятность отказа в кредите, реализовав соответствующие средства для проверки платежеспособности клиента перед принятием заказа».

Так же, как и при определении критериев, выражаемых в измеряемых единицах, при определении основных направлений разработки не следует слишком сильно углубляться в детали — ведь пока вы не занимаетесь собственно разработкой системы. В приведенных примерах говорится, что использовать некоторый стандартный элемент интерфейса следует «везде, где это только возможно», и надлежит «реализовать соответствующие средства проверки возможности предоставления кредита». Детали: где именно использовать стандартные элементы интерфейса и как реализовать средства проверки кредитоспо-

собности клиента, — будут определяться на более поздних стадиях проектирования системы, когда станут ясней требования к ней.

При определении основных направлений избегайте пустых фраз. Конечно, сложно спорить с утверждением, что «Система должна быть разработана с учетом требований пользователя и предоставлять дружественный интерфейс» — в конце концов, вряд ли пользователи будут рады получить систему с враждебным для них интерфейсом. Но эта фраза не содержит абсолютно никакой полезной информации. В то же время критерий «Система должна быть разработана с учетом требований, изложенных в руководстве «Windows Interface Guidelines for Software Design» («Руководство по разработке интерфейса для Windows-приложений»)» ставит перед разработчиком четкие границы, которых он должен придерживаться. Вопрос о том, предоставляет ли система дружественный интерфейс, может оказаться спорным. Цель же вашей работы по формулированию критериев — не увеличивать, а уменьшать число разногласий и спорных моментов.

Определение масштаба и границ системы

Как только стало ясно, зачем, собственно, понадобилось создавать эту систему, можно начать выяснять, какие именно функции должны быть реализованы в ней, а какие — нет. Как и критерии разработки, все функции, которые будут реализованы в системе, должны соответствовать определенным целям.

Предположим, что одна из целей проекта — повысить эффективность процесса продаж. Возможность распечатать зарегистрированные заказы отвечает этой цели и, очевидно, будет включена в число реализуемых функций. Но выпуск каталога продукции компании не отвечает этой цели, а значит не должен быть включен в число реализуемых в системе возможностей. Это верно, даже если каталог выпускают те же сотрудники, которые работают с системой регистрации заказов, и при выпуске каталога используют те же данные, что и при регистрации заказов.

В некоторых случаях приходится переопределять цели системы: например, создавать не систему регистрации заказов, а систему автоматизации процесса продаж. И если такое решение все-таки будет принято, выпуск каталога продукции станет одной из целей создания этой системы. Вот тут-то и выяснится, насколько полно вы определили цели проекта.

С другой стороны, переопределение целей — процесс, таящий множество опасностей. Намеченный масштаб системы не подходит для определения всех целей. Искушение расширить границы систе-

мы, чтобы реализовать еще несколько изящных, простых и удобных функций, слишком велико. Но если вы это сделаете, будьте готовы ответить на простой вопрос пользователя: «Почему вы думаете, что я вообще собираюсь делать то-то и то-то?».

Итак, правило, утверждающее, что в систему следует включать только те функции, которые однозначно соответствуют целям системы, определенным в процессе анализа, имеет смысл нарушать в двух случаях. Первый — когда очевидно, что эта функция нужна пользователям, сами пользователи подтверждают необходимость ее реализации, и стоимость реализации не превосходит той пользы, которую она может принести. Наглядный пример — выпуск каталога продукции, о котором говорилось выше.

В этом случае, возможно, разумней не расширять границы проектируемой системы, а проявить осторожность и включить все необходимые для реализации этой цели функции в дополнительные возможности. Тогда сразу станет ясно, что эти функции не являются жизненно важными для системы, и реализовать их необязательно. Их можно также исключить впоследствии, если окажется, что затраты на реализацию превосходят ожидаемую выгоду, или эта реализация займет слишком много времени, или приведет к существенному перерасходу бюджета.

Второй случай — когда заказчик настаивает на реализации данной функции. Например, вам может быть совершенно ясно, что составление списка телефонов сотрудников компании выходит за рамки системы регистрации заказов, но если заказчик желает включить ее в систему — пойдите ему навстречу. Конечно, вы можете обратить его внимание на то, что такая функция не соответствует ни одной из поставленных целей, но в конце концов, ваша задача — удовлетворить требования клиента, а не диктовать ему свои условия.

Стоимостный анализ

Возможно, вы решите провести стоимостный анализ функций, которые предполагаете включить а систему. Это полезно, в особенности если проектируемая система содержит несколько компонентов. Соотношение затрат и прибылей для различных системных компонентов поможет определить порядок, в котором они будут реализованы. Если вы планируете реализовать в системе дополнительные возможности, стоимостный анализ еще раз покажет, правильно ли она была спланирована.

При прочих одинаковых условиях, первыми, очевидно, следует реализовать те компоненты, для которых отношение прибылей и затрат наиболее велико. Такой подход оптимален в «битве за доллар» и

позволяет окупить затраты на создание системы в кратчайший срок. Он часто используется в долговременных проектах. Если вы находите способ быстро реализовать основные функции, то тем самым закладываете прочный фундамент для дальнейшего развития системы и снижаете вероятность, что она устареет еще на этапе разработки и окажется совершенно непригодной к использованию вследствие изменения бизнес-условий.

Если система начнет окупаться еще на ранних стадиях разработки, то у вас появится возможность реализовать некоторые интересные дополнительные функции, которые не были включены в спецификацию ранее. И наоборот, заказчик может решить, что основных функций системы вполне достаточно для удовлетворения всех его нужд, и отложить реализацию остальных функций на неопределенный срок.

Очевидно, что стоимостный анализ целесообразен, только если он окупается. Сам по себе стоимостный анализ несложен, однако занимает много времени, и вряд ли стоит отводить два — три дня на анализ системы, разработка которой займет один день. Часто специалисты, занимающиеся проектированием систем, проводят «неформальный анализ», полагаясь на интуицию.

Существует множество вариантов стоимостного анализа, но основной его принцип очень прост. Вычисляется отношение двух чисел — предполагаемой выгоды, которая будет получена от реализации некоей функции, и предполагаемых затрат на реализацию этой функции. Полученные значения можно сравнивать для различных функций или компонентов системы. Чем больше эта разница для компонента, тем выше его ценность по сравнению с другими компонентами системы.

ПРИМЕЧАНИЕ Разумеется, по результатам стоимостного анализа можно судить только о предполагаемых затратах и выгодах. Сколько в действительности затрачено на реализацию той или иной функции, выяснится только после того, как эта функция будет реализована. А реальную выгоду от ее реализации можно оценить лишь через некоторое время после того, как система будет запушена в эксплуатацию.

При стоимостном анализе сложно выбрать обшие единицы измерения. Чтобы сравнивать отношения предполагаемых выгод и затрат для различных компонентов, величины выгод и затрат должны быть выражены в одинаковых единицах для всех компонентов, хотя очевидно, что единицы измерения затрат и выгод могут не совпадать.

Например, сравнивать отношения величин предполагаемых выгод в долларах и предполагаемых затрат в человеко-днях.

Предполагаемые затраты, как правило, измеряются в единицах рабочего времени (часы рабочего времени, человеко-дни и т. п.) или денежных единицах (например, в долларах). Перевод величин из единиц времени в денежные единицы или наоборот не составляет труда, если известна стоимость единицы рабочего времени. Но стоимость единицы рабочего времени не абсолютная величина, и она может изменяться. Поэтому целесообразно выражать цифры предполагаемых затрат в некоторых производных величинах, например в «единицах работы». Таким образом, стоимостный анализ не превратится в смету проекта или в план внедрения системы.

А вот более сложная ситуация. Предположим, что по вашим оценкам, автоматизация рабочего процесса повысит производительность труда на 20% и снизит число ошибок при вводе данных на 50%. Повышение производительности труда легко выразить в сэкономленных часах рабочего времени, а их, в свою очередь, — в денежном эквиваленте. Однако экономический эффект, ожидаемый от повышения точности ввода данных, выразить в денежном эквиваленте не так легко. Но здесь есть и другая прямая выгода — уверенность пользователя, что в системе хранятся только правильные данные о зарегистрированном счете клиента. Эту выгоду точно выразить сложно, но ценность ее и так понятна.

В подобных случаях вы можете оценивать несколько различных выгод, получаемых от реализации проекта, причем для каждой из них использовать свои единицы измерения. Например, оценить эффект от внедрения системы позволяют следующие категории: «Сэкономленные средства», «Прибыль» и «Нематериальные выгоды». При этом придется вычислять три отношения прибылей и затрат для каждой из реализуемых в системе функций — по одному для каждой категории. Это, конечно, несколько усложнит стоимостный анализ, и к тому же возникнет дополнительная проблема. Придется решить, какой из функций назначить более высокий приоритет: той, для которой отношения прибылей и затрат — 3/6/2 или той, для которой соответствующие показатели составляют 6/2/3. Избежать такой ситуации можно, если привести все цифры к некоему общему знаменателю, чтобы для каждой функции получить однозначную оценку ее полезности.

Самый простой выход — просуммировать полученные значения для каждой категории выгод. Этот метод подходит, если асе категории одинаково важны. Разумеется, все эти величины выражены в раз-

ных единицах измерения, и это все равно что складывать число яблок, груш и апельсинов в школьной задаче по арифметике. Однако вас может интересовать и общий результат — если вы считаете фрукты, а не яблоки, груши и апельсины по отдельности. В другом случае, когда нужна взвешенная оценка, можно использовать в качестве критерия сравнения средние значения всех категорий выгод. Лично я пользуюсь как первым, так и вторым методом.

В большинстве случаев, однако, разные категории выгод имеют разную степень важности для организации. В этом случае следует присвоить каждой из них свой коэффициент важности и, приводя все числа к общему знаменателю, просто умножать значение, полученное для данной категории, на этот коэффициент. Вернемся к рассмотренному выше примеру. Предположим, вы решили, что категория «Сэкономленные средства» имеет самую низкую степень важности, в то время как категория «Нематериальные выгоды», напротив, весьма важна, а категория «Прибыль» — еще в два раза важнее. Следует назначить коэффициент 1 категории «Сэкономленные средства», 2 — категории «Нематериальные выгоды», и 4 — категории «Прибыль». Полученные результаты представлены в табл. 7-1.

Табл. 7-1. Ожидаемый эффект от внедрения различных функций

	Прибыль, $ (коэффициент 4)	Сэкономленные средства (коэффициент 1)	Нематериальные выгоды (коэффициент 2)	Сумма	Сумма с учетом коэффициентов	Среднее значение	Среднее значение с учетом коэффициентов
Функция 1	3	6	2	11	22	3.6	7.3
Функция 2	6	2	3	11	32	1.6	10.6

Точно так же можно сравнивать численные значения различных категорий выгод для разных функций и выражать одни числовые значения через другие. Скажем, для некоей функции X весьма затруднительно дать точную количественную оценку нематериальной выгоды. Ожидается, однако, что нематериальные выгоды от ее реализации будут вдвое больше, чем от реализации функции Y, а нематериальные выгоды от реализации функций Y и Z — приблизительно одинаковыми. Вполне естественно принять за единицу нематериальные выгоды от реализации функции Y, выразив через нее значения нематериальной выгоды для остальных функций.

Стоимостный анализ — удобный и практичный инструмент для оценки ожидаемой прибыли. Он позволяет сравнивать относительную ценность различных компонентов проектируемой системы. Од-

нако это всего лишь инструмент, позволяющий сделать соответствующие оценки, и цифры, полученные в результате подобных исследований, ни в коем случае нельзя принимать за непреложную истину и при проектировании системы руководствоваться только ими. Даже если в процессе стоимостного анализа обнаружится, что для функции X отношение прибылей и затрат окажется равным 12, для функции Y — 2, может быть принято решение реализовать функцию Y в первую очередь, а функцию X — во вторую.

Результаты стоимостного анализа нужно использовать с большой осторожностью, и учитывать не только относительную выгоду от реализации тех или иных функций или компонентов, но и множество других важных факторов, например системные зависимости. Впрочем, последние также не являются некими абсолютами, и в процессе проектирования системы вы можете вносить в них поправки. Поэтому прежде чем приступать к реализации каждого следующего компонента, заново пересмотрите сделанные для него оценки. Возможно, в процессе работы выяснится, что действительная стоимость реализации некоторых уже готовых компонентов сильно отличается от расчетной. Подобная «переоценка» на основе точных данных может сильно повлиять на вашу оценку стоимости, а в некоторых случаях — даже целесообразности реализации остальных компонентов системы.

Итоги

В этой главе мы рассмотрели методы, позволяющие оценить систему на ранних этапах разработки. В первую очередь следует определить цели системы и выразить их в точных критериях, позволяющих судить, успешно ли завершен данный проект. Кроме того, нужно определить границы применения системы. Все функции и работы, выходящие за рамки этих границ, не являются обязательными для данного проекта.

Данные действия можно назвать «нулевым этапом» подготовки. Это всего лишь шаги, предпринимаемые перед тем, как вы приступите к собственно разработке. В следующей главе мы подробно рассмотрим первый шаг процесса разработки: определение рабочих процессов, которые будут реализованы в создаваемой системе.

Определение рабочих процессов

Большинство баз данных проектируют для того, чтобы хранить и предоставлять пользователю доступ к некоторому множеству данных, Главная задача всякой базы данных — обеспечить поддержку некоей деятельности пользователя. *Рабочий процесс* — это одна или несколько отдельных задач, которые, будучи собраны вместе, образуют одну из областей работы организации. Процесс торгового заказа и поиск телефонного номера клиента совместно образуют рабочий процесс, хотя существенно различаются по степени сложности.

Задача — отдельный вид деятельности, отдельный шаг рабочего процесса. Например, процесс регистрации и выполнения торгового заказа состоит из нескольких задач: «Записать заказ», «Проверить счет покупателя», «Проверить наличие товара» и «Доставить заказ». В то же время процесс поиска телефонного номера клиента состоит из единственной задачи: «Найти запись, относящуюся к этому клиенту».

Разница между задачей и деятельностью иногда трудно различима. Это напоминает ситуацию, когда атрибут, представляющий собой скалярную величину в одной модели данных, в другой представлен в виде нескольких отдельных атрибутов. Нечто, трактуемое как деятельность в одной модели, трактуется как процесс в другой и разбивается при этом на отдельные задачи на более низком уровне детализации. Как и для процесса проектирования модели данных, решение основано на семантике предметной области.

Некоторые системы не связаны с анализом рабочих процессов. Специализированные средства построения отчетов, например, не поддерживают специфические процессы, так как предназначены для обеспечения определенных *видов деятельности*. В этих случаях наи-

более приемлемо построение пользовательских сценариев. Они обсуждаются в конце этой главы.

Выявление существующих рабочих процессов

Определение границ создаваемой системы — первый шаг в ходе анализа бизнес-процессов. Прежде чем начинать собственно анализ, нужно четко уяснить, какие процессы вы собираетесь анализировать. Порядок изучения процессов обычно не играет роли. Даже если вы проектируете или создаете какие-то элементы системы в первую очередь (метод приращений), то должны выполнить хотя бы беглый анализ всех рабочих процессов, которые будет поддерживать система, до того, как приступите к реализации, Сделав это, вы получите картину зависимостей между процессами, а это может повлиять на порядок, в котором вы будете создавать компоненты.

Беседы с пользователями

После того как вы определили процессы, которые будут включены в рамки создаваемой системы, надо понять, как эти процессы происходят. На данном этапе не следует слишком много внимания уделять вопросу, какие вообще существуют задачи, и какие процессы требуют дополнительного анализа. Просто найдите кого-нибудь, кто скажет: «Итак, мы получаем этот документ от продавца, и в первую очередь просматриваем его на предмет корректного заполнения. Если он заполнен правильно, то мы заносим его в папку покупателя, и потом...» Задавайте множество вопросов и записывайте ответы. Вам нужно также взять примеры форм и отчетов, которые либо являются входящими документами, либо создаются в процессе работы. Пока ваша цель — просто понять, что происходит.

Кстати, большинство аналитиков называют эту фазу «интервьюирование пользователей». Я предпочитаю термин «беседа». Легко недооценить страх пользователей перед компьютерами, распространенный даже среди тех, кто активно ими пользуется. Множество людей опасаются, что компьютер вытеснит их *с* рабочего места, и «интервьюирование пользователей» может привести их к ошибочному мнению, что все это затеяно с целью сократить штаты. Это особенно касается больших организаций, где невозможно переговорить с каждым сотрудником, и большинство совершенно не знают, кто вы такой и что собираетесь сделать.

Там, где это возможно (и даже там, где невозможно), постарайтесь побеседовать с сотрудниками, причем не столько с менеджерами высшего звена и руководителями, сколько с непосредственными исполнителями. Обычно топ-менеджеры имеют только самое общее

представление о процессах, которыми руководят. Те, кто ежедневно выполняет черновую работу, лучше и полнее расскажут вам о проблемах, с которыми сталкиваются. Конечно, следует поговорить и с руководителями — они лучше знают, для чего выполняются те или иные операции иг что можно считать успешным выполнением операции, а что ошибкой.

Во время этих бесед обязательно затроньте вопросы исключений из правил. Скажем, пользователь говорит: «Мы проверяем заказ на полноту заполнения формы». Так вот, вы должны точно знать, что происходит, если форма заполнена не полностью. Скорее всего, они просто отправляют ее назад, ну а если наоборот: они пытаются заполнить форму самостоятельно? Возможно, ваша система позволит облегчить этот процесс. Для любой операции, которую выполняет пользователь, следует знать, какие ошибки могут возникать, и что происходит при этом дальше.

Нужно четко представлять, что происходит с полученными данными при выполнении операции. Какие элементы информации будут использоваться? Откуда они поступают? В какой форме? Что происходит, если их нет или они представлены неверно? Из ответов складывается прелварительъная грубая картина концептуальной модели данных, которую мы будем обсуждать в следующей главе.

Большинство задач внутри процесса выполняются разными людьми. Очевидно, вам придется переговорить с каждым. Эта рекомендация относится и к людям, чья деятельность лежит вне рамок проекта. Например, возьмем описанный выше процесс создания торгового заказа. Задача «Поставка товара» может представлять собой, по сути, операцию «Отправить заказ в отдел доставки», а то, что происходит потом в отделе доставки, уже находится вне рамок проекта. Я советую вам все же поговорить с сотрудниками отдела доставки и выяснить, получают ли они всю необходимую информацию и удобна ли для них форма ее предоставления.

Другой пример: если на какой-то стадии процесса формируется отчет, вам нужно найти человека, который этот отчет получает, и выяснить, что он с ним делает. (Удивительно, как много бесполезных бумаг циркулирует в организациях!) Но чаше отчет действительно нужен, и тогда очень важно, чтобы он содержал верную информацию и был представлен в нужном формате.

Определение задач

После бесед с пользователями, чью работу призвана облегчить ваша система, у вас должно сложиться четкое представление обо всех выполняемых операциях. Проанализировав полученную информацию,

определите последовательность связанных между собой операций. Главная же цель — формулирование бизнес-правил процессов.

В начале главы я определила задачу как отдельную операцию. Теперь уточню, что значит слово «отдельная» в данном контексте: операция имеет четко определенные начало и конец, и все соответствующие бизнес-правила справедливы до и после ее выполнения. Временно нарушать эти правила допустимо лишь на этапе выполнения задачи.

Бизнес-правило не более чем ограничение, которое вытекает из особенностей предметной области, в отличие от ограничений, определенных, например, для типов данных. Так «Заказы не могут иметь датой поставки 36 апреля» не является бизнес-правилом, так как это ограничение домена «дата». (Это, конечно, наивный пример.) Но «Дата поставки товара не может предшествовать дате заказа» вытекает из особенностей бизнеса, так что это — бизнес-правило (или, по меньшей мере, может являться таковым). Кстати, термин «бизнес-правило» используется, даже если организация не занимается коммерческой деятельностью. База данных, которую вы создаете для сохранения информации о коллекции древних наконечников стрел, также содержит бизнес-правила.

Огромная важность бизнес-правил связана со способами обработки данных; например, «Почтовый код заказчика не может быть пустым» и «Дата счета-фактуры не должна предшествовать дате поставки товара». Другие правила, такие как «Требуется одобрение руководителя торгового отдела, если заказчик превысил свой кредит» не налагают на данные явных ограничений, хотя какие-то конкретные значения данных могут «запускать» это бизнес-правило в работу.

Не пугайтесь, формулирование бизнес-правил не столь уж тяжелая задача. Все, что нужно сделать — это ответить на вопрос: «Какие здесь могут произойти ошибки, и что происходит, если они случаются?» Вам не нужно слишком сильно беспокоиться на этом этапе о деталях. Детали — это часть концептуальной модели данных, которую вы построите позже. А сейчас сгруппируйте операции и определить бизнес-правила, которым эти операции подчиняются.

Давайте рассмотрим еще один пример. Допустим, вы составили следующий список задач.

1. Проверить, все ли пункты заявки заполнены.
2. Запросить данные о покупателе (если они уже введены в систему).
3. Записать информацию о поставке.
4. Внести сведения о заказе.
5. Присвоить новому покупателю порядковый номер.

6. Проверить наличие товара.
7. Проверить кредит, которым располагает покупатель.
8. Собрать заказ.
9. Упаковать товар.
10. Подготовить транспортную документацию.

Что можно сказать, бегло просмотрев этот список? Прежде всего, порядок операций записан произвольно. Но это не страшно — ведь вы просто пытаетесь понять, как выполняются те или иные задачи, а сами процессы проанализируете позже. Задачи также различаются по степени детализации, и позднее мы рассмотрим и этот вопрос. А сейчас давайте просто идентифицируем задачи.

Действие, описанное в п. 1: «Проверить, все ли пункты заявки заполнены», — имеет четко определенные начало и конец. Оно начинается, когда сотрудник получает новый заказ, и заканчивается, когда документ полностью проверен. Возможно, все бизнес-правила выполнены еще до начала работы, тогда никаких осложнений не будет, Если поступивший документ не соответствует бизнес-правилам, он будет отвергнут. Таким образом, если процесс начался, то все бизнес-правила выполнены. Так что задача 1 — это отдельная операция.

В п. 2 содержится единственное бизнес-правило — «Запросить данные о покупателе». Это означает, что данные о покупателе должны быть доступны запрашивающему их сотруднику. Предположим, что любой сотрудник, выполняющий это действие, обладает доступом к информации. Операция начинается, когда заявка проверена, но поскольку данные о покупателе будут использоваться и в дальнейшем, нельзя сказать, что действие завершено тогда-то и тем-то. Это не задача, это один из элементов задачи.

Данные о покупателе используются в п. 3 («Записать информацию о поставке») и п. 5 («Присвоить покупателю порядковый номер»). Следовательно, эти пункты относятся к одной задаче. Действительно, пп. 2–5 могут быть объединены в одну задачу — «Обработать заказ». Тогда п. 4 «Внести сведения о заказе» — часть операции по записи информации. И не удивляйтесь, что некоторые операции происходят одновременно. В системах с ручной обработкой данных часто легче заполнить одновременно две формы; а то, что эти формы относятся к логически разным операциям, не имеет значения.

Мы получили некоторую последовательность операций, разбитых на четыре группы. Она начинается, когда документ проверен, и заканчивается, когда вся информация записана. Но существует проблема. Клиент не может разместить заказ, если исчерпан его кредит, но наличие кредита не проверяется вплоть до п. 7 («Проверить кредит,

которым располагает покупатель»), а это происходит уже после того, как пользователь убедился в наличии товара на складе. Тем не менее, до тех пор, пока кредит не проверен, вы не можете гарантировать, что выполнены все бизнес-правила. Поэтому п. 7 также должен быть частью задачи «Обработка заказа».

Тем не менее, п. 6 («Проверить наличие товара») не относится по смыслу к процессу обработки заказа. Фактически, он представляет собой отдельную задачу, которая начинается, если обработка заказа завершена, и заканчивается подтверждением, что такой товар есть на складе в требуемом количестве.

То, что задачи выполняются в другом порядке, часто просто следствие сложившейся практики, а иногда — недосмотр организаторов процесса. Увы, в любом случае это может негативно повлиять на работоспособность системы.

Вернемся к нашему примеру. Если наличие товара проверяется в первую очередь просто потому, что так легче, вы можете изменить порядок выполнения операций. Но следует понять, как взаимодействуют процессы обработки заказа и процессы, происходящие на складе, и почему требуется немедленная проверка наличия товара на складе, а затем учесть это взаимодействие при создании системы.

Особо отмечу: не нужно детально определять бизнес-правила для каждой задачи, убедитесть только, что они выполняются. Предположим, что заказ не будет принят, если он не удовлетворяет бизнес-правилам, тогда на данном этапе детально изучать эти правила не нужно, Если бы мы обнаружили, что заказ не может быть принят, пока на складе нет нужного товара в нужном количестве, задача «Проверить наличие товара» стала бы частью задачи «Обработать заказ», а перегруппировка операций внутри задач и между ними достаточно проста. Критически же важно правильно определить задачи и обозначить их четкие границы.

Анализ оставшихся трех пунктов: 8, 9, и 10, — зависит от границ проекта. Если разрабатываемая система должна поддерживать поставку товара, нужно тщательно изучить их. Если же функции системы ограничены обработкой заказа и отправкой его в отдел доставки, эти пункты можно объединить в одну задачу — «Отправить заказ в отдел доставки».

Конечно, может оказаться, что впоследствии станет известно еще что-либо, и этой информацией нельзя будет пренебречь. Ясно, что каждый из этих пунктов является отдельной задачей, а последняя задача в списке — «Подготовить транспортную документацию», даже

отдельным рабочим процессом. Но эти процессы уже находятся вне рамок проекта, так что мы не будем их рассматривать.

Итак, вот переработанный список задач.

- **Задача 1.** Проверить, все ли пункты заявки заполнены,
- **Задача** 2. Обработать заказ.
  - *Этап 1.* Запросить данные о покупателе.
  - *Этап 2.* Записать информацию о поставке.
  - *Этап 3,* Внести сведения о заказе.
  - *Этап 4,* Присвоить покупателю порядковый номер.
  - *Этап 5.* Проверить кредит, которым располагает покупатель.
- **Задача** 3. Проверить наличие товара.
- **Задача 4.** Отправить заказ в отдел доставки.
  - *Этап 1.* Собрать заказ.
  - *Этап 2.* Упаковать товар.
  - *Этап 3.* Подготовить транспортную документацию.

Создавая список задач и процессов, вы обязательно обнаружите, что некоторые вещи поняли неверно, и в этом случае придется повторно беседовать с пользователем, чтобы прояснить ситуацию. В любом случае, нужно чтобы пользователи оценили результаты вашей работы. Часто это приводит к тому, что они вспоминают о некоторых важных вещах, которые были ими упущены и о которых вы забыли спросить.

Анализ рабочих процессов

Теперь, когда у вас появилось ясное представление о происходящих в организации процессах, самое время проанализировать эти процессы, чтобы понять, можно ли их улучшить. Как я уже говорила, большинство организаций чрезвычайно инертны. Это относится как к рабочим процессам, так и к работе с документами. Внедрение компьютерной системы — хорошая возможность изменить ситуацию.

Часто рабочие процессы можно существенно улучшить простой перегруппировкой операций, устранив необходимость многократно повторять одни и те же действия. Процесс типа: «Я делаю действие А и передаю результаты вам, потом вы делаете В и передаете обратно мне, и тогда я делаю С» трудно оценить, особенно если вы в него вовлечены, но все становится совершенно ясно, если составить список операций.

Чтобы провести этот вид анализа, следует ясно осознавать связь между операциями. Внутри любого процесса некоторые операции зависят от других и должны быть завершены в определенном порядке. Вы не можете, например, передать заказ в отдел доставки, прежде

чем этот заказ будет обработан. Но другие операции можно выполнять в произвольном порядке. Например, не имеет значения, присваивается номер заказчику до или после записи информации о поставке.

Важно обнаружить связь между данными. Некоторые операции отвечают за создание данных, используемых в других операциях: например, таких как номера заказчиков. В одной из компаний — моих клиентов, существовал порядок приема заказов, аналогичный описанному в общих чертах в предыдущем разделе, за исключением того, что присвоением номера заказчику и проверкой его кредита занимался не отдел продаж, а бухгалтерия. Отдел продаж отвечал за обработку заказа. Этот процесс представлялся в виде следующего списка.

- **Задача 1.** Проверить, все ли пункты заявки заполнены.
- **Задача** 2. Обработать заказ.
  - *Этап 1.* Запросить данные о покупателе.
  - *Этап 2.* Записать информацию о поставке.
  - *Этап 3.* Внести сведения о заказе.
- **Задача 3.** Проверить кредит, которым располагает покупатель.
  - *Этап 1.* Навести справки о покупателе.
  - *Этап 2.* Проверить кредит.
  - *Этап 3.* Присвоить покупателю порядковый номер.
- **Задача 4.** Завершить обработку заказа,
  - *Этап 1.* Проверить наличие товара.
  - *Этап 2.* Отправить заказ в отдел доставки.

Задачу 3 выполняет бухгалтерия, которая возвращает заказ в отдел продаж, только если покупатель располагает достаточным кредитом. Разумеется, остается проблема: первоначальные данные уже внесены. Дело не только в том, что, если у покупателя не было средств на счете, ранее выполнялась лишняя работа. Возникала также необходимость периодически удалять лишние заказы. А это вело к тому, что персонал, занятый вводом данных, постоянно «разрывался» между бухгалтерией и отделом продаж, которым требовалось заполнить заявки (и получить комиссионные). Изменение порядка выполнения операций, когда остаток средств на счете проверялся до обработки заказа отделом продаж, устранило эти проблемы.

Помимо подтверждения связей между задачами, следует проверить, нужна ли вообще та или иная операция. Такие ненужные этапы редко бросаются в глаза сразу же, но отслеживая поток данных между задачами процесса, их можно обнаружить. Маловероятно, что целые задачи или части задач являются совсем уж лишними. Чаше всего такие этапы «маскируются» под взаимодействие между процессами,

Генерация ненужных отчетов — хороший пример такой бесполезной деятельности.

Конечно, это не повод полностью ломать устоявшиеся рабочие процессы. Я бы не советовала вам объявлять тетке Гертруде, что ей нужно заново переучиваться вязать, если она попросила вас компьютеризировать процесс разработки узора. Неэффективные процессы встречаются не так уж часто. Но если ваша задача — помочь заказчику эффективно организовать работу, то пересмотр рабочих процессов — один из способов достичь этой цели.

Документирование рабочих процессов

Как и любые другие составляющие процесса проектирования, продолжительность анализа рабочих процессов и документирование результатов должны быть пропорциональны сложности системы. При проектировании простой системы для отслеживания имен и номеров телефонов потребуется не более часа на обсуждение и составление кратких заметок «для внутреннего пользования».

Тем не менее, я бы рекомендовала провести не менее двух встреч с заказчиками даже самого простого проекта. Цель второй встречи — уточнить, правильно ли вы поняли требования заказчика, подтвердить взаимопонимание и согласовать план работ.

Более сложные проекты могут потребовать нескольких недель обсуждения деталей с десятками людей, столь же сложным будет и документирование. Структурированный список задач и стадий наподобие описанного в этой главе, подходит для документирования простого проекта. Для более сложных случаев я предпочитаю создавать иллюстрации.

В индустрии программного обеспечения хорошо стандартизованы диаграммы «сущности — связи», используемые для документирования. Для диаграмм рабочих процессов столь же четких стандартов не существует. Метод диаграмм все более сращивается с технологией системного анализа. Если вы свободно владеете одним из этих методов, нет смысла переходить на что-то другое. Цель — понять и структурировать полученную информацию. Диаграммы потоков данных и диаграммы качества процессов очень удобны. Специальные методы построения диаграмм, на мой взгляд, не более чем религиозные догматы, хотя используются достаточно часто.

При отсутствии формальной технологии вы можете легко разработать свою собственную. Вам понадобятся пять символов, означающих: задачу, документ, элемент данных, точку принятия решения, и

событие (например, начало и конец задачи). Мои собственные обозначения показаны на рис. 8-1.

Рис 8-1. Символы рабочих процессов

Если для завершения задачи требуется относительно мало шагов, я перечисляю их внутри символа задачи. Если таких шагов много — рисую для такой задачи специальную диаграмму, в которой каждый шаг обозначен как отдельная задача. Иногда чтобы показать, что задача выполняется неким внешним участником (например, бухгалтерией, проверяющей кредит покупателя), я использую тень или выделение жирным шрифтом.

Символ элемента данных может обозначать либо отдельный атрибут (например, номер покупателя) или целую сущность (покупатель). Чтобы указать, что элемент создается в ходе выполнения задачи, вы можете использовать шрифтовые выделения. Некоторые аналитики любят выделять случаи, когда элемент данных задействован в ходе выполнения определенной задачи, но никогда не использовался ранее другими задачами процесса. Я, честно говоря, не вижу в этом особого смысла.

Выбрав символы, которые будете использовать (я рекомендую простые), определите способ их организации на диаграмме. Я использую отрезок, чтобы показать зависимость, и прерывистую линию, если задача может выполняться в произвольном порядке. Открытый кружок на линии показывает, что задача необязательна, так же, как это сделано в диаграммах «сущности — связи» (рис. 8-2).

Если вам кажется, что рабочий процесс слишком сложен, чтобы использовать простые технологии для его документирования, предлагаю обратиться к методам, описанным в учебниках по проектированию и системному анализу. Некоторые из них вы найдете в списке литературы в конце книги.

Рис. 8-2. Связи в рабочих процессах

Пользовательские сценарии

Создание *пользовательских сценариев* (user scenarios) — альтернатива формальному анализу. Пользовательский сценарий состоит из двух элементов: множества *профилей пользователя* (user profiles), которые определяют множество пользователей системы, и *сценариев использования* (usage scenarios) для каждого из профилей. Последние представляют собой описание того, как пользователь предполагает работать с системой — то есть действий, которые он будет выполнять.

Хотя пользовательские сценарии можно иногда применять для анализа рабочих процессов, они, скорее, предназначены для описания интерфейса взаимодействия пользователя и системы. Сложно создать пользовательский сценарий, который не описывал бы внешний интерфейс системы.

Но все-таки цель проекта — удовлетворить требования пользователя, и пользовательские сценарии очень удобны при построении систем, которые должны автоматизировать уже существующие процессы. Они позволяют аналитику сосредоточиться на том, какие процессы осуществляются в организации, не вдаваясь в подробности их внутреннего содержания.

Например, даже простой сценарий: «Продавец будет использовать систему для отслеживания статуса покупателей», — выполняющийся на всех стадиях прохождения заказа, от его получения до поставки товара, выставления счета, и вероятно, платежа, объясняет, как группа сотрудников будет использовать систему. При этом подробности организации пользовательского интерфейса не обсуждаются.

Разумеется, разработка пользовательских сценариев и анализ рабочих процессов не являются совершенно взаимоисключающими. Анализ — способ понять процессы как они есть, а пользовательские сценарии описывают то, как пользователи будут взаимодействовать с

системой. Для большинства систем эти вопросы одинаково важны, Если проект требует значительных усилий, стоит выполнить оба вида анализа, при этом пользовательские сценарии часто вытекают из анализа рабочих процессов и не требуют дополнительного обсуждения.

Итоги

Я рассказала, как определить и понять процессы, которые должна поддерживать система. Для этого существует несколько способов — можно проводить анализ рабочих процессов различной степени детализации, создавать пользовательские сценарии, а также сочетать оба этих метода.

В главе 9 мы рассмотрим концептуальную модель — логическую организацию Данных, структур и утилит.

Концептуальная модель данных

ГЛАВА

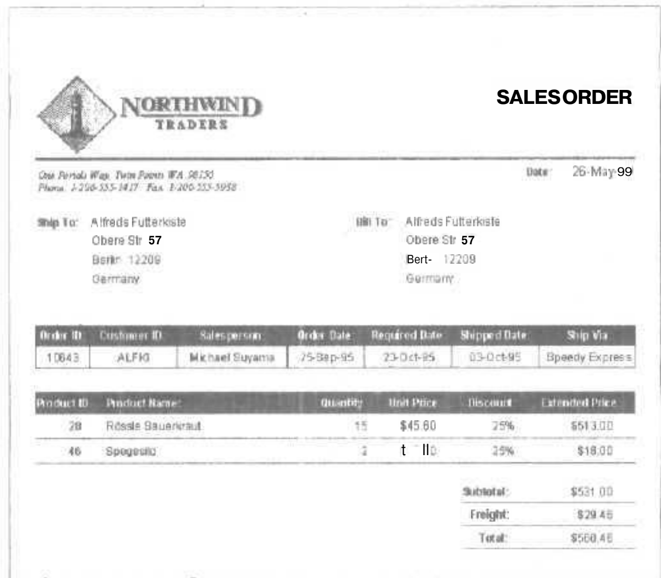
На этой стадии проектирования вы должны уже ясно представлять, чего хотите достичь в результате. Определены границы системы, сформулированы принципы проектирования, проанализированы рабочие процессы. Пора создавать модель данных.

Помните, что концептуальная модель описывает сущности, атрибуты и связи между ними. Это не схема базы данных, которая содержит описание физической реализации таблиц. Для такой реализации у вас все еще мало информации. Сначала нужно спроектировать пользовательский интерфейс и выбрать архитектуру, которую вы будете использовать при создании системы.

Определение объектов базы данных

На ранних стадиях анализа руководствуйтесь уже имеющимися документами или создайте их самостоятельно. Это и документы заказчика (формы заявок, отчетов и т. п.), и подготовленная вами документация по рабочим процессам. Первый шаг при создании модели данных — получение этих документов и определение на их основании данных, которые будут использованы в системе.

Начнем с рабочего процесса. Разумеется, он не один, но я обычно выбираю какой-либо из наиболее важных, определяющих большую часть создаваемых сущностей. Большинство рабочих процессов начинаются после создания определенных документов, например, после того как секретарь передает в отдел продаж заполненную форму заказа (мы возвращаемся к примеру из главы 8). Иногда процессы начинаются, когда происходит некое событие, в этом случае обычно одна из задач — это заполнение формы документа (рис. 9-1).



NORTHWIND TRADERS

SALES ORDER

One Portals Way, Twin Points WA 98156
Phone: 1-206-555-1417 Fax: 1-206-555-5958

Date: 26-May-99

Ship To: Alfreds Futterkiste, Obere Str. 57, Berlin 12209, Germany

Bill To: Alfreds Futterkiste, Obere Str. 57, Berlin 12209, Germany

Order ID	Customer ID	Salesperson	Order Date	Required Date	Shipped Date	Ship Via
10643	ALFKI	Michael Suyama	25-Sep-95	23-Oct-95	03-Oct-95	Speedy Express

Product ID	Product Name	Quantity	Unit Price	Discount	Extended Price
28	Rössle Sauerkraut	15	$45.60	25%	$513.00
46	Spegesild	2	[illegible]	25%	$18.00

Subtotal:	$531.00
Freight:	$29.46
Total:	$560.46

Рис. *9-1.* ***Форма торгового заказа, инициирующая рабочий процесс***

Найдите эту форму и запишите все элементы информации, которые она содержит. Не думайте пока о том, являются ли эти элементы сущностями или атрибутами, просто составьте список. Отметьте все повторяющиеся группы. Опираясь на свой анализ рабочих процессов, отметьте также элементы данных, которых, по вашему мнению, не хватает. У вас должен получиться примерно такой список элементов данных (рис. 9-2).

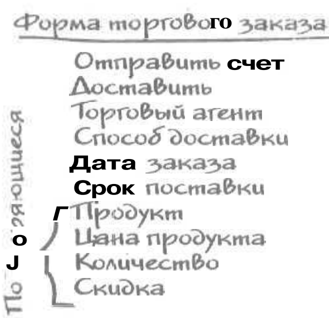


Рис. 9-2. Первоначальный список элементов данных для формы торгового заказа

Список составлен — можете выделять сущности, атрибуты и связи. Для каждого элемента списка нужно определить, является ли он самостоятельным объектом или свойством объекта. Объекты становятся сущностями, а свойства — атрибутами сущностей (рис. 9-3).

***Рас. 9-3. Предварительное разделение элементов формы заказа на сущности** и атрибуты*

Элементы *Bill To* (Отправить счет) и *Ship To* (Доставить) определены как элементы сущности *Customer* (Покупатель). Тем не менее, не очевидно, относятся ли два адреса только к сущности *Customer*, только к сущности *Sales Order* (Форма торгового заказа) или к обеим сущностям. Принцип, который используется здесь, аналогичен тому, что применялся при отнесении атрибута *Unit Price* (Цена продукта) к сущности *Order Detail* (Информация о заказе). Как текущая цена отличается от цены, по которой товар был продан, так же и адреса заказа и поставки покупателя логически отличаются от адресов, на которые были отправлены счет-фактура и заказ.

Являются ли адреса заказа и поставки также и атрибутами сущности *Customer,* зависит от того, как организована работа в данной компании и каковы требования пользователей. Отнесение этих адресов к сущности *Customer* позволяет задать значения по умолчанию в процессе обработки заказа, что уменьшит время обработки и снизит вероятность ошибок при вводе данных. Но для поддержки этих атрибутов в сущности *Customer* требуются дополнительные ресурсы. Если компания поставляет товары по нескольким адресам (например, в случае цепочки продаж), выделение этих ресурсов может стать весьма обременительным, и лучше вносить данные в процессе обработки заказа,

Иногда имеет смысл пойти на компромисс. Вы можете добавить множество адресов поставки к сущности *Customer,* но не требовать, чтобы пользователи обязательно их вводили. Если адрес только один, можно использовать его по умолчанию. Если адресов много, пусть пользователи выбирают их из списка. Или можно спроектировать систему так, что в качестве значения по умолчанию будет предлагаться наиболее часто используемый адрес, но пользователь будет вправе выбрать другой адрес из списка.

Предоставьте пользователям возможность обновлять данные, относящиеся к сущности *Customer,* в процессе обработки заказа. Если адреса нет или пользователь вводит новый адрес, система может выводить на экран запрос о необходимости добавить этот адрес в данные о клиенте, Здесь есть некоторые проблемы, связанные не с моделью данных, а с пользовательским интерфейсом.

Из факта существования элемента данных *Salesperson* (Торговый агент) вытекает, что в модели может присутствовать сущность *Employee* (Сотрудник). Но мы пока не располагаем необходимой для ее создания информацией, поэтому отметим это как вопрос, требующий разрешения. Существует вероятность, что эту сущность используют другие процессы. Если это так, добавьте необходимые атрибуты. Если нет, оставьте *Salesperson* в качестве атрибута сущности *Sales Order*. Помните: решение зависит от семантики системы. Если эти данные используются только в сущности *Sales Order,* нет никаких причин создавать сложную по структуре сущность *Employee.*

Элемент *Product* (Продукт) определен как сущность, и несколько элементов объединены в повторяющуюся группу: *Product* (Продукт), *Unit Price* (Цена продукта), *Quantity* (Количество) и *Discount* (Скидка). Эта группа определена как сущность *Order Details* (Информация о заказе). Для сущности *Product* определено первоначальное множество атрибутов, предположительно на основании ряда других документов, и на основании этого же списка выделены сущности *Supplier* (Постав-

щик) и *Category* (Категория продукта), хотя подробности их структуры еще не известны.

Так как сущности определяются на концептуальном уровне, нас не волнует, являются ли атрибут *Extended Price* (Пересчитанная цена) сущности *Order Detail* или атрибут *Units In Stock* (Количество упаковок на складе) сущности *Product* (Продукт) хранимыми величинами или они вычисляются по мере необходимости. Мы еще не знаем, можно ли их вычислить и решим этот вопрос позднее.

Интересен атрибут *Ship Via* (Способ доставки продукта). Большинство форм заказов предоставляют возможность выбрать способ доставки: например, Parcel Post (посылкой) или 2nd Day Air (авиапочтой). Это атрибуты или сущности? Ответ зависит от семантики системы (вы этого ожидали, не так ли?). Как много вариантов существует? Если более двух, введите специальную сущность для этого элемента данных.

Насколько постоянны используемые при заполнении формы значения? Существует вероятность, что вы имеете дело с внешними поставщиками услуг. Может ли организация сменить поставщика или привлечь еще одного? Насколько зависит деятельность организации от конкретного метода поставки? Возвратят ли транспортные компании заказ назад, если им нужно будет нанять альпинистов и забросить товар на вершину Эвереста? Ответы на эти вопросы также нужно учесть при создании модели.

Создание для метода доставки отдельной сущности позволяет изменять эти данные, но влечет за собой усложнение модели и пользовательского интерфейса. Конечно, пользователю не так уж трудно выбрать значение из списка, нажав пару клавиш. Но если не принять специальных мер, это может довольно существенно замедлить работу.

Если компания, выполняя заказы, применяет нестандартные методы доставки товара, нужно учесть и это. Вы должны пройти по лезвию ножа между гибкостью и эффективностью. Для нашего примера наилучшее решение — ввести дополнительный атрибут *Special Instruction* (Специальная инструкция), но это должно быть обязательно учтено в модели данных и при анализе рабочих процессов.

Такое решение может неожиданным образом повлиять на системные ограничения. Хотя организации нужно точно знать, как доставить товар заказчику, трудно будет детально определить специфический метод доставки. Система должна следить за тем, чтобы атрибуты *Shipping Method* (Способ доставки) и *Special Instructions* (Специальные инструкции) не оказались пустыми одновременно, поэтому в модель придется добавить достаточно сложные правила проверки.

Если в рамках проекта автоматизируется процесс реальной поставки товара, трактовка *Shipping Method* как отдельной сущности полезна, но потребует значительно усложнить систему. Как определить сведения о метоле доставки, который может появиться в будущем? Либо создать общую сущность, содержащую типичные для методов поставки атрибуты (например, номер дока и время погрузки), либо определить только известные методы поставки и завершить на этом работу — исключительные случаи не будут обрабатываться системой.

Здесь есть опасность чрезмерно усложнить систему и перегрузить пользователя избыточной информацией. Стоит слишком увлечься наращиванием функциональности — и система станет непригодной к использованию. Да, значения по умолчанию легко поддерживать и регулярно обновлять (лучше всего при выполнении некоторых задач). Все это очень удобно для персонала, отвечающего на вопросы заказчиков о состоянии поставок, но стоит ли тратить усилия на ввод информации о способе доставки для тысяч заказов, если это интересует только пятерых клиентов фирмы?

Нужно учитывать последствия решений, которые вы принимаете, Легко не заметить недостатки решения, когда вас озаряет какая-то идея: «Разве это не хорошо? Это сэкономит много времени». Допустим, вы обрабатываете какую-либо часть информации. Подумайте, не используется ли она еще где-то, либо как значение по умолчанию, либо как ограничение. Если информация о способе доставки вводится в любом случае, почему бы не сделать ее доступной для секретаря?

Но где бы вы ни использовали данные, помните о том, как они будут создаваться и поддерживаться. Лучше предоставить пользователю список значений с возможностью выбрать нужное, чем неструктурированную текстовую информацию. Но список нужно создать и поддерживать, а для этого — разработать интерфейс. А значит, при моделировании структуры данных неизбежны компромиссы.

Разумеется, следует устранить повторный ввод данных. Однако не стоит принуждать пользователя обращаться к кому-то для того, чтобы этот кто-то ввел необходимые данные в систему. Мы обсудим этот серьезный вопрос в третьей части книги, а сейчас важно определить, где вводятся и используются данные.

Определение связей

Поработав со всеми имеющимися в вашем распоряжении документами, вы получите черновик структуры сущностей и атрибутов предметной области, Остается определить связи между сущностями и еще раз пересмотреть атрибуты и ограничения сущностей.

Теоретически, вы можете в первую очередь заняться именно атрибутами. Но мне кажется, что лучше начать со связей, так как при этом часто приходится вносить новые сущности и атрибуты в уже существующую модель.

Если вы используете примерно те же приемы работы, что и я, то сейчас перед вами пачка рукописных заметок со стрелочками и каракулями вроде «см. стр. 12», которые никто кроме вас не может расшифровать. Ваш первый шаг — структурировать всю эту информацию, привести ее к ясной и понятной форме.

Начните с построения черновика диаграммы «сущности — связи» модели данных. Если ваши записи беспорядочны и вы не уверены, что разберетесь в них и за три недели, попробуйте составить списки атрибутов для каждой сущности.

Сначала выберем одну из центральных сущностей и добавим сущности, которые с ней связаны. Определите характер связи («один к одному», «один ко многим», «многие ко многим») или просто начертите прямую линию в качестве наглядного символа связи, а проанализировать эту связь можно позже. Я обычно провожу анализ сразу, но некоторые мои коллеги предпочитают сначала обозначить все сущности, а потом повторно проанализировать схему.

Первый черновой вариант диаграммы «сущности — связи» для процесса обработки заказа приведен на рис. 9-4. Это простой пример, и диаграмма легко читается. Допустим, вы решили, что *Salesperson* является атрибутом только сущности *Sales Order* и для него не существует специальных сущностей.

Если вы работаете над сложной системой, то можете создать множество диаграмм, каждая из которых описывает только часть данных. В этом случае для построения диаграмм лучше использовать автоматизированные средства, Иначе их синхронизация станет утомительным занятием.

Черновик диаграммы позволит более детально проанализировать связи между сущностями. Для каждой сущности нужно определить следующие свойства связи:

- мощность;
- обязательность присутствия каждого участника;
- связываемые атрибуты;
- налагаемые ограничения.

Проанализировав диаграмму процесса обработки заказа, вы получите результат (рис. 9-5).

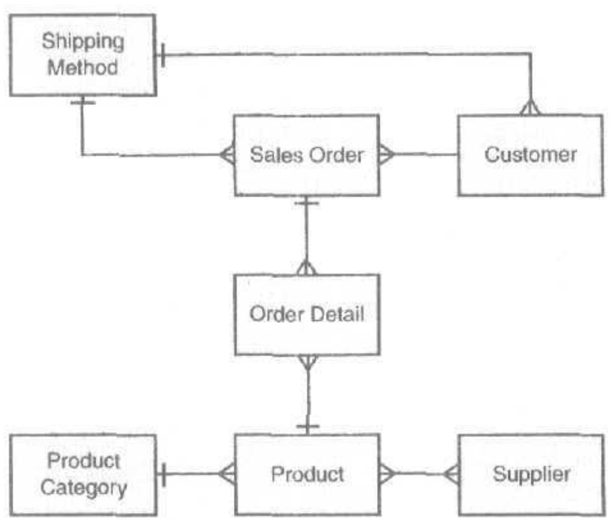


Рис. 9-4. Первый черновик диаграммы «сущности — связи» процесса обработки заказа

Рис. 9-5. Диаграмма процесса обработки заказа после анализа связей между сущностями

Мощность связи

Итак, вы обозначили связи между сущностями на диаграмме (рис. 9-4). Теперь еще раз проанализируйте диаграмму и уточните параметры связей.

Там, где вы обнаружили связь «многие ко многим», введите промежуточную сущность и создайте связи «один ко многим» между каждой из участвующих в исходной связи сущностей и этой промежуточной сущностью (промежуточная сущность — со стороны «многие»). Связь между *Supplier*(Поставщик) и *Product* (Продукт) в нашей модели имеет тип «многие ко многим», и чтобы ее реализовать, нужно добавить сущность *Product-Supplier*. Связь между *Sales Order* (Форма торгового заказа) и *Product* (Продукт) также имеет тип «многие ко многим», но в этом случае в качестве промежуточной сущности выступает *Order Detail* (Информация о заказе).

Обязательность связи

Определив тип связи между двумя сущностями, разберитесь, является ли связь обязательной для каждого из участников. В нашем примере связь между сущностями *Customer* (Покупатель) и *Shipping Method* (Метод доставки) необязательна для обеих сторон — то есть, заказчик не обязательно предпочитает определенный способ доставки (способ по умолчанию), а способ доставки не обязательно связан с конкретнымзаказчиком.

Связь между *Product Category* (Категория продукта) и *Product* (Продукт) необязательна только для одной из сторон. Категория продукта не обязательно связана с конкретным продуктом, но любой продукт должен относиться к определенной категории товара.

Связь между *Sales Order* (Форма торгового заказа) и *Shipping Method* (Метод доставки) более сложна. Метод доставки может существовать независимо от заказа, так что *Sales Order*— необязательный участник связи. *Shipping Method*, тем не менее, не обязан участвовать в связи, только если заказ предусматривает специальный метод доставки товара. Это важное ограничение, и его нужно отметить на диаграмме.

Атрибуты связи

В большинстве случаев все, что нужно знать о связи — то, что она существует, например, что заказчик разместил конкретный заказ. Но иногда требуется дополнительная информация: например, когда образовалась и сколь долго продолжалась связь. Эти свойства определяют атрибуты связи, а не ее участников.

Когда связь сама по себе имеет атрибуты, ее можно представить в виде сущности. В случае с обработкой заказа мы могли бы присвоить определенному поставщику статус «привилегированный» (Preferred Supplier). Так как у нас уже есть промежуточная сущность между сущностями *Product* (Продукт) и *Supplier* (Поставщик), атрибут *Preferred Supplier можно* просто добавить к сущности. В ином случае следует создать новую сущность, чтобы отразить с ее помощью атрибуты связи.

Дополнительные ограничения

Наконец, нужно определить ограничения, которые налагаются на связи. Каково минимальное и максимальное число записей для сущности на стороне «многие» связи «один ко многим»? Есть ли условия, которым должна удовлетворять связь, чтобы иметь право на существование? Или условие, при котором связь обязана существовать?

В нашем примере таким ограничением является условие, согласно которому связь между *Sales Order* (Форма торгового заказа) и *Shipping Method* (Метод доставки) необязательна, если только существуют специальные инструкции для поставки данного заказа. Правило, что заказчик не может разместить заказ, если его кредит не проверен — другое ограничение. Это правило представлено на диаграмме в виде аннотации. Если ограничений много или ограничение слишком сложно, чтобы представить его в виде аннотации, документируйте его иным способом. Но нужно, по крайней мере, указать на диаграмме, что такое ограничение существует.

Повторный анализ сущностей

Теперь, когда у нас есть полная картина сущностей, пора проанализировать каждую подробно. Для каждой сущности нужно выяснить:

- как она связана с предметной областью;
- какие рабочие процессы ее создают, изменяют, используют и уничтожают;
- какие сущности взаимодействуют с этой сущностью, а какие от нее зависят;
- какие у нее существуют ограничения и бизнес-правила;
- каковы ее атрибуты.

Связь между сущностью и предметной областью

Обычно такую связь очень легко определить. «Сущность *Customer* (Покупатель) моделирует организации и частных лиц, которые покупают наши товары». Наибольшая трудность, с которой я сталкивалась — сформулировать определение этой связи так, чтобы избежать тавтологии. «Сущность *Employees* (Сотрудники) моделирует сотрул-

ников организации» — типичный пример такого не слишком удачного определения.

Если связь представляется в виде сущности, дать определение труднее, так как связь не отражает напрямую конкретные объекты предметной области. Вот пример: «Поставщик может поставлять множество товаров, и один и тот же товар может поставляться несколькими поставщиками, Сущность *Product-Supplier*(Продукт-Поставщик) моделирует эту связь, также как и свойство «привилегированный», присваиваемое любому конкретному поставщику данного товара».

Некоторые понятия предметной области (вероятно лучший пример здесь — торговый заказ) моделируются с помощью нескольких сущностей. Я называю эти понятия ***составными сущностями.*** Форма торгового заказа представляется в виде комбинации сущностей *Sales Order* (Форма торгового заказа) и *Order Detail* (Информация о заказе).

Я предпочитаю описывать составные сущности как единый объект. Например: «Сущности *Sales Order* и *Order Detail* образуют вместе единый заказ, размещаемый заказчиком. Сущность *Sales Order* моделирует сам заказ, а элемент *Order Detail* описывает заказываемые товары».

Рабочие процессы, влияющие на сущности

Информацию, в каких рабочих процессах используются конкретные данные, лучше всего включить в модель данных. Если потребуется изменить структуру сущности, например, добавить атрибут, вы легко найдете в модели список процессов, на ход которых повлияет изменение сущности.

Определить процессы, непосредственно взаимодействующие с сущностью, также не составляет труда, а вот выявить процессы, работающие с сущностью опосредовано — более тяжелая задача. Например, что процесс обработки заказа может изменять приписанный заказчику стандартный метод доставки товара, или что *Special Bonus* (свойство категории товара) может повлиять на величину скидки, а вследствие этого — на стоимость заказа, не является очевидным фактом. Такие особенности рабочих процессов, не документированные надлежащим образом, могут создать дополнительные сложности для обслуживающих систему программистов.

Большинство аналитиков заносят такую информацию в документацию по анализу рабочих процессов, что весьма удобно, если все изменения касаются только рабочих процессов. Но иногда изменения затрагивают и саму модель — непосредственно (изменяется организация дела в компании) либо косвенно (изменения в рабочих процессах требуют адекватного изменения модели), В этом случае легче просмотреть документацию, описывающую изменяемую сущность,

чтобы определить, на какие процессы повлияет изменение модели данных.

Связь между сущностями и процессами — это перекрестная ссылка. Как и все такие ссылки, ее трудно создать и поддерживать, но в перспективе она может быть очень полезна.

Взаимодействие между сущностями

Диаграммы «сущности — связи» — весьма удобны, но не позволяют отразить слишком большой объем информации. Если картина взаимодействия между сущностями очень сложна, ее можно задокументировать на уровне описания сущностей. Вам нужно будет создать для этой цели специальные аннотации.

Когда модель состоит из множества диаграмм, и одна сущность присутствует в нескольких диаграммах, следует внести в описание сущности список элементов, с которыми она связана. Это очень полезно, если сущность используется в нескольких рабочих процессах. Например, сущность, в которой хранятся заголовки обращения типа «Mr.», «Mrs.», «Dr.», и «Ms.», может использоваться в нескольких различных местах клиентского приложения. Если в описании сущности вы явно перечислите, где она используется, то сэкономите время и силы, когда вам нужно будет изменить сущность.

Бизнес-правила и ограничения

Еще один вид документации, необходимый для описания сущностей — описание ограничений. Все сущности должны быть уникально идентифицированы, и из этого вытекает необходимость выбрать для каждой сущности атрибуты, которые будут служить первичным ключом.

Любые ограничения, относящиеся к атрибутам: например, «Атрибуты *Shipping Method* (Метод доставки) и *Special Instructions* (Специальные инструкции) не могут одновременно содержать пустые значения», должны быть документированы.

Атрибуты

Наконец, нужно создать списки доменов и атрибутов. При составлении списка вы будете отталкиваться от исходного описания сущности. Вам придется при этом создать все внешние ключи, чтобы удовлетворялись требования ссылочной целостности.

Придется также определить для каждой из сущностей по меньшей мере один ключ-кандидат. Он станет первичным ключом соответствующего отношения, когда вы будете реализовать схему базы данных. Помните, что первичные ключи не могут содержать значения *Null*. Из-за этого для их формирования не всегда можно использовать

существующие атрибуты, придется вводить дополнительный искусственный атрибут, значения которого генерируются самой системой.

Для сущности *Customer* (Покупатель) из нашего примера, по-видимому, придется вводить такой искусственный атрибут. Если заказчиком может быть как физическое, так и юридическое лицо, то список атрибутов этой сущности будет выглядеть, как на рис. 9-6.

Рис. 9-6. Список атрибутов сущности Customer (Покупатель)

Даже если оставить в стороне вопрос уникальности имен, у нас все равно будут проблемы. Если покупатель — физическое лицо, атрибут *Company Name* (Название компании) будет содержать *Null.* Если покупатель — организация, значение *Null* будет содержать атрибут *Individual Name* (Имя физического лица). Один из этих атрибутов всегда будет содержать *Null.* Поэтому такие поля не могут являться ключами-кандидатами или входить в их состав, даже если бы они уникально идентифицировали запись (хотя это не так).

Здесь мы вплотную подходим к следующей проблеме с сущностью *Customer* (Покупатель). Имена не уникальны. В нашем примере даже целый список атрибутов не гарантирует уникальности значений, поскольку всегда существует вероятность, что два человека с одинаковыми именами имеют одинаковые адреса. Например, система регистрации заказов не дает уверенности, что вы не перепутаете Джона Смита-младшего, который живет на Бейкер-стрит, 18, с его отцом, Джоном Смитом-старшим, который проживает в том же доме.

Конечно, Джон Смит-старший (John Smith Sr.) и Джон Смит-младший (John Smith Jr.) — разные люди, и к ним относятся разные записи. Но те данные, которые позволяют их уникально идентифицировать, вряд ли пригодны для использования в качестве первичной информации о клиенте. Можете ли вы вообразить, что вас начнут расспрашивать об обстоятельствах вашей личной жизни, когда вы

заказываете что-то в торговой компании? «Извините, сэр, но имеются ли у вас родственники с такими же именами, проживающие вместе с вами? Эта информация требуется только для учета в нашей компьютерной системе...» Не очень-то приятно, не правда ли?

К счастью, есть простой способ — ввести атрибут *Customer Number* (Индивидуальный номер покупателя). И если в организации до сих пор не существовало определенных правил присвоения такого номера, то это можно сделать средствами Microsoft Jet или Microsoft SQL Server, которые обеспечивают механизмы автоматической генерации таких номеров (типы данных *AutoNumber* и *Identity*, соответственно).

Если вы используете произвольный идентификатор, будьте готовы предусмотреть альтернативную форму идентификации. Вы же не хотите поставить в неудобное положение клиента, забывшего свой номер? Спросить «Мистер Джон Смит, вы проживаете в Окридже или вы в Цинцинатти?» — это гораздо лучше, чем просить Джона Смита найти старую квитанцию с номером, а потом перезвонить.

Сущность *Customer* (Покупатель)— еще один пример показывающий, как надо использовать уникальный системный идентификатор. Даже если предположить, что комбинация адреса и имени даст подходящий уникальный идентификатор, для такого идентификатора потребуется слишком большое число полей. Помните, что каждый первичный ключ имеет двойника — внешний ключ сущности, которая на него ссылается. Намного эффективнее связать по одному атрибуту в каждой сущности, чем по пять или по шесть.

Анализ доменов

Список атрибутов на рис. 9-6 использует формат представления *Имя:Домен.* Большинство аналитиков игнорируют существование доменов и описывают атрибуты напрямую, с помощью типов данных и ограничений. Это не очень правильно, но вряд ли повлияет на успех всей вашей работы.

Доводы в пользу анализа доменов — экономия усилий и возможность собрать дополнительную информацию. Все, что может облегчить работу и сделать ее более эффективной, следует использовать в модели. Анализ доменов позволяет провести моделирование более технологично.

Давайте рассмотрим только один пример: атрибуты *CompanyName* (Наименование компании) и *Individual Name* (Имяфизического лица) на рис. 9-6 определены в домене *Name* (Имя). Мы можем описать домен *Name* следующим образом: «Этот домен — строка из слов, набранных в соответствующем регистре, которая имеет максимальную

длину 75 символов. Строка может содержать только символы алфавита, символы «точка» (.) и «запятая» (,)».

Домен нужно определить только один раз, а использовать его в системе можно сколько угодно. Конечно, вы вправе определить такое ограничение для каждого соответствующего атрибута, но зачем? Более того, так как эти атрибуты определены в одном домене, **мы** можем логически сравнивать их значения. При определении каждого атрибута в отдельности эта возможность не столь очевидна.

Поиск заказчиков с одинаковыми атрибутами *CompanyName* или *IndividualName* — также один из возможных способов облегчить жизнь пользователя. А вот поиск компании по ее номеру весьма неудобен.

Формальное определение домена — набор значений, который может содержать атрибут. Концептуально это просто, но как определить домен на практике? Вам следует задать три параметра, а именно: тип данных; все ограничения, налагаемые на диапазон хранимых данных; формат хранения и представления данных в домене (это необязательно).

Выбор типа данных

Первый шаг — определить тип данных, который будет использоваться для представления домена в схеме базы данных. Это разрушает представление о разделении модели данных на концептуальную схему и ее физическую реализацию.

Тип данных определяет диапазон хранимых в домене значений. Просто «целое» не является доменом в «чистой» математике, а тип данных домена *Quantity* — «целое». Тем не менее, я бы не рекомендовала увлекаться на этом этапе глубоким изучением конкретных реализаций типов данных в серверах баз данных. Вопрос о выборе сервера еще впереди.

Тип данных домена также может быть другим доменом. Например, у вас уже есть общий домен типа «дата», который определяет, что система не допускает дат более ранних, чем 1 января 1900 г., а формат предусматривает четырехзначное представление года. Вполне приемлемым будет определить домен *Event Date* (Дата события) как «Даты, насупившие после 23 октября 1982 г.» (именно тогда компания начала свою деятельность).

Ограничения на диапазон данных

Теперь определите диапазон данных, которые может содержать домен. Порой самый простой путь — задать правило, например: «*Quantity* (Количество) — это целое положительное число».

Иногда можно просто перечислить хранящиеся в домене величины. «Район может быть одним из следующих: Северо-Западный. Северо-Восточный, Центральный, Южный». В этом случае вам определенно придется включить домен в модель в виде сущности. Это значительно легче, чем каждый раз вводить данные вручную, а кроме того, позволяет легко изменять содержимое домена.

Единственное исключение — это когда данных в домене очень мало и они не изменяются. Скажем, вы моделируете список вопросов экзамена, и у вас есть домен *Answer* (Ответ), который содержит значения *True* или *False.* Нет смысла создавать для такого домена отдельную сущность. Так как других возможных величин не существует, то создание отдельной таблицы для хранения двух величин усложнит модель, и только.

Домен, определяемый несколькими атрибутами, также можно представить с помощью сущности, например домен *State* (Штат). Если вам нужно учитывать в системе множество стран, то придется оперировать множеством параметров.

Если заказчик живет Австралии, то *New South Wales* (Новый Южный Уэльс) является верным значением для штата, *a Alabama* (Алабама) — нет. В этом случае, домен будет включать в себя атрибуты *Country* (Страна) и *State* (Штат). Этот пример не является точным определением домена и моделируется с помощью связи в диаграмме «сущности — связи». Тем не менее, такой домен можно рассматривать как составной.

В конце концов, наша цель — облегчить реализацию ограничений базы данных, а определение часто встречающихся доменов уменьшает время разработки и вероятность ошибок.

Спецификация домена должна включать также описание, может ли домен содержать неопределенные значения, строки нулевой длины или оба этих вида величин. Стоит прямо указать на такую особенность в определении домена, даже если способность содержать неопределенные значения может быть определена с помощью связи между сущностями.

Анализ доменов и определение списка атрибутов для любой сущности — тесно связанные между собой итеративные процессы. На практике удобней определять домены одновременно с атрибутами. Если подходящий для атрибута домен уже существует, вы можете просто связать этот атрибут с доменом. Если нет — определите домен на этапе составления списка атрибутов.

Некоторые атрибуты имеют дополнительные ограничения, помимо определенных в доменах. Это совершенно нормально. Вы можете

определить домен *Event Dale* (Дата события), который содержит даты наступления определенных событий. С одной стороны диапазон этих ограничен датой начала деятельности компании. Так, даты торгового заказа: *Order Dale* (Дата заказа) и *Shipping Date* (Дата поставки), — могут быть определены в домене *Event Date*. Атрибут *Shipping Date* должен содержать дату более позднюю, чем *Order Date*. Это ограничение на уровне сущности, и оно должно быть отражено в ее описании.

При определении ограничений на уровне домена (и дополнительных ограничений на уровне атрибута) следует попытаться сделать это как можно более подробно, не принимая во внимание сложности реализации и использования. Мы еще обсудим этот вопрос в третьей части книги, но чем более точно вы определите домен на данном этапе, тем легче будет дальнейшее проектирование. Но если случайно удалить какие-либо значения, система не сможет работать.

Определение формата

Определение формата представления данных не всегда обязательно, но часто полезно. Если однажды определить, что дата представляется в формате DD-MM-YYYY, то не придется делать это повторно.

Нормализация

Вы удивлены, что я нигде не упомянула здесь о нормализации данных? Если вы правильно организовали сущности, устранив повторяющиеся группы данных и отношения «многие ко многим», то эти сущности, скорее всего, находятся в третьей нормальной форме.

Конечно, новичку полезно еще раз проанализировать модель данных. Помните, что любая сущность в модели должна подчиняться правилу «ключ, полный ключ и ничего кроме ключа»,

Итоги

Мы рассмотрели построение концептуальной модели данных. Процесс начинается с определения необходимых для деятельности заказчика данных, которые организуются в виде множества сущностей. Затем анализу подвергаются связи между сущностями, сами сущности и их атрибуты.

В следующей главе мы рассмотрим реализацию концептуальной схемы в виде физической схемы для конкретного сервера баз данных.

Схема базы данных

В предыдущей главе мы рассмотрели концептуальную модель данных, определяющую их логическую структуру. А сейчас познакомимся со схемами баз данных, которые описывают физическую структуру данных. Схема базы данных — это логическая конструкция. При ее построении для описания физической структуры базы данных используются абстрактные термины. Но непосредственное представление данных на физическом уровне осуществляется при помощи механизма баз данных, без вмешательства пользователя или разработчика системы.

Системная архитектура

Перед тем как приступить к описанию схемы базы данных, определимся с архитектурой системы. К сожалению, в научной литературе под термином «архитектура» понимают две разных, хотя и связанных друг с другом, модели. Чтобы различать эти понятия, я введу два термина: *программная архитектура* (code architecture) и *архитектура данных* (data architecture). Подчеркиваю, что это моя собственная терминология и в другой специальной литературе она не встречается.

Программная архитектура

То, что я подразумеваю под термином «программная архитектура», часто называют по-разному: модель приложений, многоуровневая архитектура, модель служб. Программная архитектура содержит описание построения логической структуры программы. Структура прикладной программы, как правило, зависит от конкретной реализации, и подробно обсуждать ее мы не будем. Тем не менее, именно особенности программной архитектуры определяют, будут ли ограничения целостности реализованы в схеме базы данных, и как имен-

но. Поэтому мы все же рассмотрим некоторые аспекты программной архитектуры.

На заре развития СУБД типичная системная архитектура представляла собой сплошной монолит: огромные фрагменты программного кода, имеющие простейшую структуру. Системный аналитик того времени, поставивший себе задачу изменить (или даже просто понять) такую систему, был похож на человека, пытающегося разделить на отдельные макаронины целое блюдо спагетти. Чтобы навести хоть какой-то порядок в этом первобытном хаосе, программисты начали структурировать программный код, выделяя отдельные компоненты: процедуры, модули или объекты.

Теперь вместо бесчисленных строчек запутанного программного кода программисты-аналитики имели дело с фрагментами кода гораздо меньшего объема. Эти фрагменты как-то взаимодействовали друг с другом, однако догадаться, как именно, было весьма затруднительно.

Современным программистам управление программными компонентами облегчает их организация в виде служб, иначе говоря, выделение различных уровней в модели. Эти службы выполняют разные функции на разных логических уровнях. Существует множество способов организации многоуровневой архитектуры. Мы подробно рассмотрим два наиболее распространенных: трехуровневую и четырехуровневую архитектуру.

Трехуровневая архитектура

В трехуровневой архитектуре выделяют три уровня программных компонентов: *пользовательский* (User Services), промежуточный или *бизнес-уровень* (Business Services) и *уровень данных* (Data Services). Microsoft Visual Modeler — интерактивный инструмент для моделирования данных, интегрированный с Microsoft Visual Studio 6.0, поддерживает эту модель (рис. 10-1).

В трехуровневой модели пользовательский уровень, как правило, включает компоненты, отображающие данные и определенным образом реагирующие на действия пользователя. Весь пользовательский интерфейс инкапсулирован внутри этого уровня. На промежуточном уровне реализованы бизнес-правила и функции проверки правильности вводимых данных. Компоненты промежуточного уровня взаимодействуют с компонентами пользовательского уровня и уровня данных. Программные компоненты уровня данных, взаимодействующие только с компонентами промежуточного уровня, осуществляют все операции манипулирования данными.

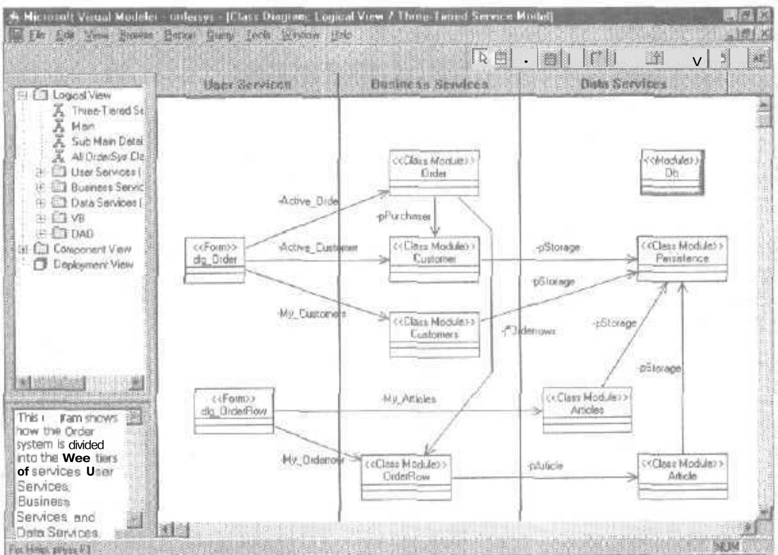


Рис. 10-1. Visual Modeler поддерживает трехуровневую архитектуру

Трехуровневая модель проста и понятна, и Visual Modeler — практичное, широко распространенное средство моделирования данных. Однако я почти не пользуюсь им для разработки реальных систем. Дело в том, что каждая система, как правило, включает в себя функции, не принадлежащие ни к одному из уровней. Рассмотрим пример: некий фрагмент данных требуется отформатировать, прежде чем выводить пользователю. Скажем, индивидуальный номер, присваиваемый при выдаче свидетельства социального страхования, может храниться в базе данных как девятиразрядное десятичное целое число, но отображаться на экране пользовательского компьютера он должен в виде 999-99-9999. К какому уровню должно относиться форматирование данных — к пользовательскому или к уровню данных? Можно привести аргументы в пользу и того, и другого. Еще вопрос: к какому уровню отнести управление транзакциями — к бизнес-уровню или уровню данных? При моделировании сложных систем с использованием иерархических представлений и преобразования данных на эти вопросы ответить не так-то легко.

Если действовать последовательно, вопрос, к какому из уровней отнести подобные функции, не принципиален; однако именно в этом месте модель дает сбой, При создании модели невозможно обойтись без искусственно введенных ограничений и соглашений: например,

«форматирование следует отнести к пользовательскому уровню, а построение иерархической структуры данных — к бизнес-уровню», И в конце концов, может наступить момент, когда сложность модели сведет на нет ее положительные качества.

Четырехуровневая архитектура

Выделение четырех уровней в архитектуре программного кода позволяет избежать многих проблем, свойственных трехуровневой архитектуре. В четырехуровневой архитектуре (рис. IO-2J, часто называемой также парадигмой уровней, выделяют, соответственно: *уровень интерфейса пользователя* (User Interface layer), *уровень интерфейса данных* (Data Interface layer), *уровень интерфейса транзакций* (Transaction Interface layer) и *уровень интерфейса внешнего доступа* (External Access Interface layer).

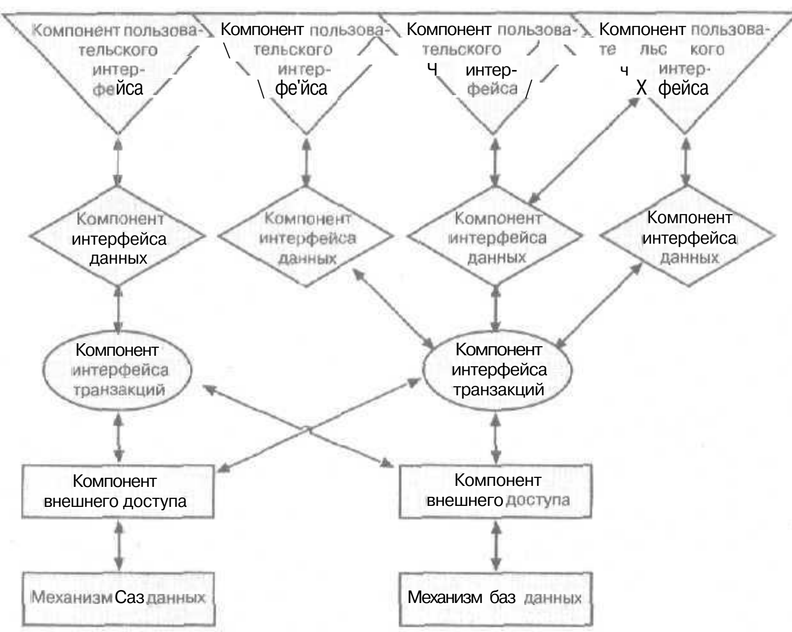


Рис. 19-2. Четырехуровневая модель

Уровень интерфейса пользователя соответствует пользовательскому уровню трехуровневой модели. Программные компоненты уровня интерфейса пользователя осуществляют взаимодействие с пользователем, в том числе: представляют данные посредством объектов диалоговых окон, отвечают на изменения пользователем состояния объек-

тов диалоговых окон (например, изменение размеров экранной формы), а также вводят пользовательские запросы в систему.

На уровне интерфейса данных осуществляется поддержка данных в физической памяти (на постоянных носителях она выполняется средствами программных компонентов уровня интерфейса данных и механизма СУБД). Уровень интерфейса данных содержит большинство функций, принадлежность которых к тому или иному уровню в трехуровневой архитектуре весьма затруднительно определить, не прибегая к искусственно введенным правилам. Это, например, функции, позволяющие определенным образом форматировать данные и создавать ***виртуальные наборы записей.*** (Виртуальные наборы записей существуют только в физической памяти, они не размешаются на жестком диске или любом другом устройстве, предназначенном для постоянного хранения данных.)

Как правило, каждый программный компонент уровня интерфейса данных тесно связан с одним из компонентов уровня интерфейса пользователя. При этом один программный компонент интерфейса данных может поддерживать несколько компонентов интерфейса пользователя (рис. 10-2). Допустим, в системе присутствует форма *Customer Maintenance,* где отображается информация об одном покупателе и форма *Customer Summary,* которая выводит информацию о нескольких покупателях. Поскольку обе этих формы представляют сущность *Customer* (Покупатель), они могут иметь общие модули **для** форматирования и проверки параметра *CustomerNumber* (Номер покупателя) — функции, принадлежащей уровню интерфейса данных.

На физическом уровне компоненты интерфейса пользователя, как правило, соответствуют формам в Microsoft Visual Basic или Microsoft Access. а связанные с ними компоненты интерфейса данных — реализуются в виде модулей форм. Некоторые процедуры задействованы в нескольких модулях одновременно, как правило, такие процедуры реализуются в общем модуле, используемом в различных формах.

На уровне интерфейса данных реализуются правила проверки данных, однако логика бизнес-процессов не входит в этот уровень. Так, фрагмент кода или программный модуль, выполняющий проверку правильности ввода значения *CustomerNumber* ~ компонент уровня интерфейса данных. Но программные средства, определяющие порядок наступления каких-либо событий или выполнения каких-либо операций в системе (например, запрещающие отправку заказанных товаров, пока не проверена платежеспособность заказчика) относятся к уровню интерфейса транзакций.

Уровень интерфейса транзакций управляет использованием данных в приложении. Программные компоненты этого уровня создают и инициируют запросы, получают данные от компонентов уровня интерфейса внешнего доступа, поддерживают бизнес-процессы, а также обрабатывают ошибки и исключения, полученные от элементов уровня интерфейса внешнего доступа.

Компоненты уровня интерфейса транзакций гораздо чаше используются повторно, чем компоненты уровня интерфейса данных. Для физической реализации первых удобнее всего применять Visual Basic и Access. Например, объект *Customer* задействует метод *Update*, вызываемый еще несколькими компонентами уровня интерфейса данных. Важно следующее: поскольку уровень интерфейса данных должен быть независим от уровня интерфейса пользователя (по крайней мере, е теории), обновляемые величины следует перечислить явно. Другими словами, вызов метода может иметь следующий вид:

```
MyCustomer.Update CustomerNumber. CustomerName ..
```

При этом метод *Update* составляет запрос с использованием оператора UPDATE, затем передает этот запрос на выполнение в уровень интерфейса внешнего доступа, и при этом обрабатывает все возможные ошибки: либо непосредственно устраняет состояние ошибки, либо передает соответствующее сообщение обратно по той же цепочке для вывода на уровне интерфейса пользователя.

На уровне интерфейса внешнего доступа реализуется обмен данными между приложением и внешними источниками данных. В системах баз данных программные компоненты уровня интерфейса внешнего доступа обрабатывают события обмена данными с механизмом СУБД. Они выполняют запросы (в том числе и сообщения об ошибках) и передают результаты обратно по цепочке компонентов на другие уровни.

В идеале все процедуры на этом уровне должны быть реализованы так, чтобы полностью изолировать транзакции от специфики механизма СУБД. Теоретически, при этом для переноса приложения, разработанного для работы с механизмом СУБД Microsoft Jet, на платформу SQL Server достаточно всего лишь изменить программные компоненты уровня интерфейса внешнего доступа. Однако воплотить эту идею на практике весьма непросто.

Как уже упоминалось, на уровне слоя интерфейса транзакций запросы формируются, а на уровне слоя интерфейса внешнего доступа — только выполняются. Но учитывая различия между разными диалектами языка SQL, редко удается реализовать систему так, чтобы

полностью разделить эти два уровня. Рассмотрим простой пример: генерацию набора результатов, полученного при выполнении запроса одним оператором TRANSFORM в диалекте **SQL,** используемом механизмом СУБД Microsoft Jet. Чтобы выполнить запрос, потребуется написать несколько операторов на диалекте SQL, используемом SQL Server.

Если заранее известно, какие запросы будет выполнять разрабатываемое приложение, можно включить их непосредственно в схему базы данных и тем самым избежать проблем, связанных с особенностями синтаксиса. Здесь очень полезны параметрические запросы. Допустим, нужно найти в базе данных записи, относящиеся к клиенту по фамилии Джонс (Jones) или Смит (Smith). Вам не придется всякий раз заново составлять запросы — достаточно передать значение Jones или Smith в качестве параметра заранее составленного запроса.

К сожалению, не всегда удается точно предугадать, какие именно запросы придется составлять. А в системах, где пользователю надо предоставить возможность создавать собственные произвольные запросы к данным, такие прогнозы практически невозможны: не удается полностью изолировать уровень интерфейса транзакций. (Во всяком случае, я не могу сформулировать правил, позволяющих решать эту проблему в общем случае. И если вам, читатель, все-таки это удастся, прошу поделиться секретом.) А пока общее решение не найдено, вам, скорее всего, придется включить в слой интерфейса данных условный код, например:

```
theEngine = myData.EngineName
Select Case theEngine
    Case "SQLServer"
        ' build a SOL Server flavored query
    Case "Jet"
        ' build a Microsoft Jet query
    Case Else
        ' return an "unknown engine" error to the Data Interface layer
End Select
```

При проектировании приложения, предназначенного для работы только с одним механизмом СУБД, может возникнуть соблазн объединить уровни интерфейса транзакций и интерфейса внешнего доступа. Я не рекомендую делать этого. Хотя как всякая отдельная задача, создание уровня внешнего доступа требует некоторого времени, сам процесс не настолько уж и сложен, к тому же реализация этого уровня существенно сэкономит ваши время и силы в дальнейшем.

Например, создав компонент, осуществляющий обмен данными с SQL Server 7.0 при помощи ADO 2.0, вы сможете использовать его в других системах, которые будете разрабатывать в дальнейшем. При этом вам не потребуется вносить никаких изменений, если только вы не решите использовать другой механизм СУБД или изменить объектную модель.

Программная архитектура и схема базы данных

Программная архитектура влияет на схему базы данных: и в плане изоляции уровня интерфейса внешнего доступа (или уровня данных. если вы используете трехуровневую модель), и в плане проверки правильности данных. Как я уже говорила, изоляция уровня интерфейса внешнего доступа обеспечивает независимость программ этого уровня от изменений механизма баз данных. Это достигается за счет составления стандартных запросов к базе данных и включения этих запросов в схему базы. Данный метод к тому же позволяет увеличить, и зачастую весьма существенно, производительность системы.

Однако реализация проверки правильности ввода данных — гораздо более сложная задача, чем изоляция уровня интерфейса внешнего доступа. Подробно методы и способы проверки правильности введенных значений будут обсуждаться в главе]6. Сейчас же поговорим о том, в каком случае имеет смысл реализовать подобную проверку и где именно следует разместить фрагменты программного кода, выполняющие ее.

Некоторые разработчики предпочитают реализовать все правила проверки правильности введенных данных в самом механизме СУБД. Таким образом и все ограничения, обеспечивающие целостность данных, и все бизнес-правила сосредоточены в одном месте, где практически исключена возможность их случайного изменения вследствие ошибки.

Но этот метод имеет ряд серьезных недостатков. Во-первых, далеко не все правила проверки удается реализовать на уровне механизма баз данных. Например, в Microsoft Jet правило, запрещающее изменять первичный ключ при обновлении записей в базе данных, невозможно задать без помощи триггеров. Даже широкие возможности, предоставляемые SQL Server, не позволяют реализовать все правила непосредственно в механизме СУБД.

Во-вторых, ожидание, пока данные отправляются для проверки их механизмом СУБД, затем выполняется проверка и выдастся ответ, может существенно снизить производительность системы. Как правило, проверка правильности ввода данных выполняется практически сразу же, после того как пользователь введет очередное значение.

Иногда (например, когда вводимое значение должно содержать только цифры, а не буквы) проверка выполняется сразу же, как только пользователь введет очередной символ с клавиатуры. В других случаях — лишь после того как данные введены в поле или в несколько полей: например, если нужно проверить, что дата, введенная в поле *DesiredDelivery Date* (Планируемая дата доставки), не предшествует дате, введенной в поле *OrderDate* (Дата заказа).

Даже при работе с приложением, установленным на выделенном компьютере (на том же самом, где работает СУБД), отправка набора данных механизму СУБД после ввода каждого символа или заполнения очередного поля приведет к значительному снижению производительности. Если же приложение подключается к базе данных через локальную сеть (не говоря уже об удаленном доступе к базе данных через глобальную сеть или Интернет), весьма вероятно, что время отклика системы окажется столь длительным, что система не будет оправдывать себя.

Единственный выход в подобной ситуации — отправлять данные механизму СУБД для проверки, только после того как будет завершен ввод всей записи. Однако вероятнее всего, в это время пользователь уже начнет выполнять какие-то другие действия. Вряд ли кто-либо сочтет удобной систему, сообщающую об ошибке ввода данных через десять минут после того, как ввод данных завершен.

Чтобы максимально сократить время отклика, я советую реализовать функции, выполняющие проверку данных, непосредственно в пользовательском приложении. Если база данных используется только одним приложением, и критерии проверки правильности ввода данных относительно неизменны, вы, в принципе, можете реализовать проверку данных только на уровне приложения, полностью исключив проверки на уровне механизма СУБД. Это предотвратит двойную работу и сократит время разработки. Но все же данный способ крайне не рационален и по ряду причин я не рекомендую его применять.

Допустим, впоследствии для работы с той же базой данных будет создано другое приложение — тогда база лишится механизмов, гарантирующих целостность данных. И конечно, всегда остается вероятность, что пользователи будут работать с базой данных не только при помощи созданного вами клиентского приложения, а выполнять произвольные запросы посредством Microsoft Access или SQL Server Enterprise Manager. Безусловно, можно попытаться создать строгую модель защиты данных, запретив любые изменения данных иным способом, кроме как из созданного вами приложения. Но тогда будут существенно ограничены возможности доступа к данным.

Итак, реализуйте средства проверки данных и в клиентском приложении, и в схеме базы данных. Microsoft Access делает это автоматически. Если определить правило проверки правильности ввода данных на уровне поля и затем перетащить мышью это поле в связанную форму, то форма наследует правило проверки правильности ввода данных, определенное на уровне схемы базы данных.

В версиях Microsoft Access более ранних, чем Access 2000, существует проблема синхронизации правил проверки в клиентской части и схеме базы данных. Включив поле в связанную форму, а затем изменив правило проверки правильности ввода данных для этого поля на уровне таблицы, вы обнаружите, что соответствующие изменения не распространились на связанную форму автоматически. В Microsoft Access 2000 эта проблема устранена, а вот в Visual Basic она все еще существует, даже в последних версиях. Допустим, имеются ссылки на одно из полей таблицы из нескольких форм (или, вернее, из нескольких компонентов уровня интерфейса данных, которые используются в разных формах). При каждом изменении правил проверки для этого поля придется обновлять соответствующие правила во всех формах или компонентах уровня интерфейса данных, которые содержат ссылки на это поле.

Чтобы решить эту проблему, отправляйте запросы на подтверждение к базе данных на этапе исполнения клиентского приложения. Это сильно увеличит нагрузку на вычислительные ресурсы, однако при частом изменении бизнес-правил может оказаться выгодным, поскольку все правила обновляются в одном месте.

Загрузка правил проверки из механизма СУБД выполняется или при первом запуске приложения, или при загрузке формы, или перед обновлением каждой записи. Я считаю оптимальным второй вариант. Если выполнять проверку при запуске приложения, то одновременно с нужной информацией будет загружаться и много лишней, относящейся к формам, которые пользователю никогда не понадобятся. Загрузка правил проверки перед обновлением каждой записи, конечно, полностью гарантирует, что используемые правила проверки не устарели. Но при этом приложение обращается к схеме базы данных столько раз, сколько к самим записям. Лично я не сталкивалась на практике с системами, в которых правила проверки изменялись бы настолько часто, чтобы приходилось загружать их перед обновлением каждой записи.

В заключение хочу обратить ваше внимание еще на один момент. Маловероятно, что правила проверки изменятся за тот промежуток времени, пока пользователь вводит данные в открытую форму (вве-

денные данные будут удовлетворять старым критериям проверки, но не новым). Но даже в этом маловероятном случае целостность данных не нарушится, поскольку подобные конфликты разрешаются на уровне механизма базы данных. Тем не менее, меня пугает одна лишь мысль о том, что схема базы данных может изменяться, в то время как система активно используется.

Архитектура данных

Для разработки приложения мало определить структуру программного кода проектируемой системы, нужно еще выбрать подходящую архитектуру данных. В главе 1 упоминалось, что **СУБД** состоит из отдельных компонентов: самого приложения, механизма СУБД и непосредственно базы данных (рис. 1-1). Основываясь на четырехуровневой модели программных компонентов, мы можем теперь детализировать эту структуру (рис. 10-3).

Рис. 10-3. СУБД состоит из шести отдельных уровней

Определяя архитектуру данных приложения, вы должны решить, где разместить уровни. Теоретически, каждый из них (или даже каждый отдельный компонент) может находиться на своем компьютере, подключенном к сети. Возможна и противоположная ситуация: все компоненты установлены на одном компьютере, не включенном в сеть. В конечном счете, все сводится к нескольким стандартным конфигурациям, которые следует использовать в различных ситуациях. Рассмотрим их подробней.

Одноуровневая архитектура

Каждое логическое объединение компонентов в архитектуре данных называется слоем. Наиболее простая архитектура — одноуровневая, то есть такая, где все компоненты объединены в один логический уровень. Наипростейший вариант такой архитектуры — система, не подключенная к сети (изолированная система).

В изолированной системе все компоненты установлены на одном компьютере и доступны только пользователю, работающему с этим компьютером. Даже если компьютер физически подключен к сети или Интернету, СУБД недоступна остальным пользователям. При этом и хранение данных, и все вычисления осуществляются локально, производительность ограничивается лишь физическими параметрами системы — мощностью процессора и объемом физической памяти. Изолированные системы требуют большого размера памяти, и поэтому гораздо менее эффективны, чем другие конфигурации.

В большинстве изолированных систем используется механизм баз данных Microsoft Jet. Конечно, на отдельном компьютере можно реализовать и систему на основе SQL Server, но строго говоря, такую систему нельзя считать одноуровневой. Исключение составляет Microsoft Data Engine (MSDE), поставляемый вместе с Access 2000. Это своего рода «упрощенный вариант SQL Server», работающий в одноуровневой архитектуре.

Распространенный вариант одноуровневой архитектуры — база данных, к которой есть доступ по сети. В такой модели база данных или ее часть находится на компьютере, подключенном к сети, однако все вычисления выполняются локально.

ПРИМЕЧАНИЕ Размещать приложение, работающее с базой, на сетевом диске крайне неразумно. Теоретически это возможно, однако такая конфигурация приведет к существенному возрастанию нагрузки на сеть. Гораздо удобнее разместить приложение на локальном компьютере, а для доступа к данным, находящимся на сетевых ресурсах, использовать связанные таблицы.

База данных, размещенная на сетевом диске, позволяет работать с данными нескольким пользователям одновременно. (В основном, конечно, это относится к механизму баз данных Microsoft Jet). Теоретически число пользователей, одновременно работающих с такой базой, не может превышать 255. Однако на практике максимальное число пользователей зависит от того, какие действия они выполняют с базой данных, а также от производительности системы. Очевидно, что 20 пользователей, одновременно вводящих данные с максимально возможной скоростью, больше загрузят систему, чем 50 человек, просматривающие данные о продажах и определяющие маркетинговую стратегию продвижения товара.

Снижение нагрузки на сеть непосредственно влияет на схему базы данных. Все вычисления выполняются на локальном компьютере; таким образом, компьютер, на котором размещается база данных, с точки зрения пользователя практически ничем не отличается от удаленного жесткого диска. Различие лишь во времени доступа к данным: для сети оно существенно выше, чем для локального диска. Кроме того, существует еще один фактор, ограничивающий производительность системы — это пропускная способность сети. С ростом числа пользователей этот фактор начинает сказываться все сильнее. Поэтому при использовании сетевого доступа разработчики, как правило, стараются свести к минимуму передаваемый объем данных, Здесь решение в большой степени зависит от внешних условий, и каждый раз приходится искать компромисс, оптимизируя взаимосвязанные параметры. Более подробно различные варианты реализации одноуровневых систем раскрываются в литературе, перечисленной в библиографическом указателе, а также в материалах на компакт-диске, прилагаемом к книге.

На производительность сети влияют и другие параметры схемы базы данных: расположение объектов и использование индексов. Как я уже упоминала, очень важно, чтобы объекты пользовательского интерфейса размещались на локальных компьютерах. Кроме того, можно разместить на пользовательских компьютерах локальные копии данных, которые редко изменяются.

Например, список продуктов, которыми торгует компания — к этим данным пользователи обращаются достаточно часто. Если таблица *Products* (Продукты) не занимает слишком много места на диске (не более нескольких мегабайт, даже 1 Гб ~ уже слишком много), то копии такой таблицы можно хранить на компьютерах всех пользователей системы. Это часто позволяет уменьшить обмен данными по сети и улучшить производительность Конечно, при таком варианте

придется предусмотреть механизм обновления данных, но это не так уж трудно.

Можно привести и другие примеры данных, которые допустимо хранить локально: списки почтовых индексов, штатов, стран и региональных отделений компании. Эти списки, как правило, достаточно компактны и обновляются относительно редко. Кроме того, пользователи часто обращаются к этим данным. Решая вопрос, будет ли таблица храниться локально или на сетевом диске, руководствуются обычно именно этими соображениями; компактность таблицы, частота обновления данных, а также частота обращения и число пользователей, обращающихся к данным. Очевидно, не имеет смысла размещать на локальных дисках компьютеров всех пользователей системы таблицу, к которой изредка обращается лишь системный администратор.

А списки клиентов компании или списки студентов, напротив, не должны храниться локально. Данные в таких списках обновляются часто, и размещение базы данных на сетевом ресурсе позволяет предоставить пользователям возможность работать с последней версией базы.

Еще один фактор, определяющий влияние схемы базы данных на производительность сети — использование индексов. Упрощенно, индексы можно рассматривать как специальную небольшую таблицу, где все записи располагаются в определенном порядке. Эта таблица содержит только поля, записи в которых нужно упорядочить определенным образом, и указатели на записи, хранящиеся в реальной таблице (рис. 10-4).

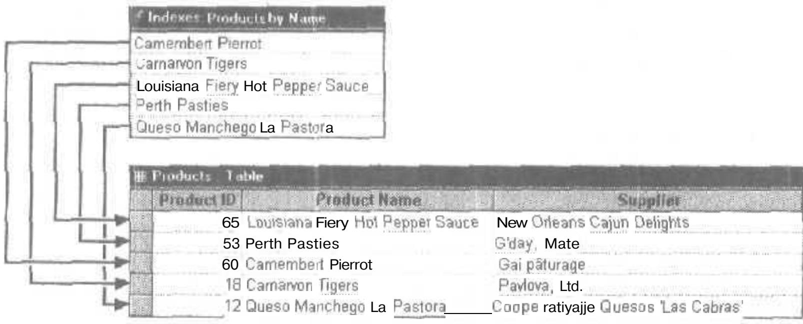


Рис. 10-4. Индекс представляет собой специальную компактную таблицу, записи в которой отсортированы в определенном порядке

Следует пояснить, что подразумевается под термином «указатель», если речь идет об индексе. Указатель в индексе не имеет ничего общего с указателем на объект или указателем на область памяти (эти термины широко используются в программировании). На самом деле. физическая реализация индексов гораздо сложнее, и соответствует приведенной мною модели лишь приближенно, но эту модель вполне можно считать рабочей. Более подробно об этом рассказывается в «Microsoft Jet Database Engine Programmer's Guide» — руководстве про программированию Microsoft Jet.

Изменение физического порядка записей в базовой таблице при определении или изменении порядка сортировки, в большинстве случаев, задача слишком трудоемкая, требующая больших затрат времени. Поэтому Microsoft Jet сортирует только файл индекса. Кроме существенного выигрыша в производительности, этот метод обеспечивает быстрый доступ к таблице с использованием различных порядков сортировки, поскольку для каждой таблицы можно создать несколько индексов.

В SQL Server существует *кластерный индекс*, определяющий физический порядок записей в таблице. Для каждой таблицы может существовать не более одного кластерного индекса.

Использование индексов позволяет существенно увеличить производительность системы при обработке данных. Часто Microsoft Jet выполняет сложные операции, не обращаясь к базе данных непосредственно, а применяя только индекс. Даже для изолированных систем, не использующих сеть для доступа к данным, индексы существенно снижают время обработки данных, сокращая число операций чтения с локального диска. Для распределенных систем, где данные передаются по сети, правильное использование индексов может стать критическим фактором, определяющим производительность.

Приведу конкретный пример. В таблице *Customers* (Покупатели) содержится 100 000 записей, длина каждой — 1500 байтов. Требуется найти в базе данных запись, удовлетворяющую определенным условиям — например, запись, относящуюся к покупателю Джонс Констракшн (Jones Construction), для которого известно значение идентификатора клиента *CustomerID*, JONSCON. Для этого нужно выполнить такой запрос:

```
SELECT * FROM Customers WHERE CustomerID = "JONSCON"
```

Если в таблице отсутствует индекс или первичный ключ, запрос выполняется следующим образом: механизм СУБД Microsoft Jet считает каждую из хранимых в базе данных записей, чтобы выбрать удов-

летворяющие критерию запроса. Это означает, что по сети передается около 150 Мб данных. Но если поле *CustomerID* проиндексировано, явно или через объявление первичного ключа, механизм баз данных Microsoft Jet считывает только записи индекса, что означает передачу всего лишь нескольких килобайтов. При этом нужная запись в базовой таблице определяется довольно быстро.

Индексы могут увеличить производительность поистине ошеломляюще, но следует помнить, что этот выигрыш отнюдь не дается даром. При использовании индекса возникает дополнительная нагрузка на систему, поскольку при каждом добавлении или изменении записей в базовой таблице Microsoft Jet обновляет индексы в ней. Как правило, эта дополнительная нагрузка невелика, и ее можно не принимать во внимание при проектировании системы. Однако чем больше индексов в одной таблице, тем существеннее влияние, оказываемое на производительность системы при обновлении данных. В конце концов, непродуманное использование индексов может привести к ситуации, когда время, необходимое на поддержку индексов, превысит время, сэкономленное за счет их использования.

Двухуровневая архитектура

В двухуровневой архитектуре база данных и механизм баз данных расположены на удаленном компьютере. Каждый из них может находиться на одном компьютере или на разных. Одна база данных не обязательно должна размещаться на одном компьютере; физически она может быть распределена между несколькими компьютерами, но логически такая система все равно останется двухуровневой. Такую архитектуру можно реализовать только на основе SQL Server или другого сервера баз данных, например Oracle. При помощи механизма баз данных Microsoft Jet сделать ото невозможно.

На первый взгляд, различие между базой данных, с которой пользователи работают по сети, и двухуровневой системой не так уж велико. Какая в сущности, разница, что использовать — Microsoft Jet на локальном или SQL Server на удаленном компьютере? Дело в том, что при работе с базой данных по сети все вычисления выполняются на локальном компьютере, а в двухуровневой системе часть вычислений выполняется локально, а часть — на сервере баз данных. На рабочей станции обрабатываются события, связанные с действиями пользователя, а удаленный компьютер управляет доступом к данным. SQL Server выполняет все операции манипулирования с данными и отправляет результат на клиентскую рабочую станцию. Поэтому двухуровневые СУБД называют также ***клиент-серверными системами*** (client/server systems).

Чтобы наглядно проиллюстрировать различие между базой данных, где используется сетевой доступ, и двухуровневой системой, сравним, как выполняется в каждой из систем SQL-запрос, который мы приводили в качестве примера, в разделе, посвященном индексам;

```
SELECT * FROM Customers WHERE CustomerID = "JONSCON"
```

В первом случае (пользователи работают с базой данных по сети) механизм баз данных Microsoft Jet прочитает индекс (разумеется, если он существует), определит запись, удовлетворяющую критерию запроса, и найдет эту запись в базе данных. Во втором случае (клиент-серверная система) с клиентской рабочей станции запрос будет отправлен на SQL Server, после чего сервер вернет обратно на клиентский компьютер результат его выполнения — соответствующую запись, (Безусловно, это упрощенная модель, в действительности все гораздо сложнее.)

Когда речь идет о выполнении таких простых запросов, различие в производительности между базой данных, использующей сетевой доступ, и клиент-серверной системой весьма незначительны. Может даже оказаться, что клиент-серверная система работает медленнее. Но производительность клиент-серверных систем, поддерживающих работу многих пользователей, при выполнении сложных запросов, несомненно, гораздо выше, а время отклика значительно меньше, чем систем с одноуровневой архитектурой.

Использование клиент-серверной системы позволяет существенно улучшить время отклика по сравнению с одноуровневой архитектурой за счет того, что вычисления распределяются между клиентской рабочей станцией и сервером. В то время как сервер обрабатывает запрос, поступивший от клиента, на рабочей станции могут выполняться другие операции, например обрабатываться дополнительные запросы пользователей. И наоборот, пока рабочая станция выдает пользователю ответ на какие-либо его действия (или просто ожидает ввода данных), сервер баз данных может обрабатывать другие запросы. SQL Server — несравненно более сложная и ресурсоемкая система, но он способен обеспечить значительно меньшее время отклика, чем Microsoft Jet. А значит, может обслуживать гораздо больше пользователей, чем база данных с сетевым доступом к данным.

Чтобы использовать клиент-серверную систему максимально эффективно, следует распределить вычисления между клиентской рабочей станцией и сервером так, чтобы сервер принял на себя основную нагрузку по обработке данных. Если для базы данных, с которой пользователи работают по сети, имеет смысл хранить запросы на ло-

кальном компьютере, то в двухуровневой клиент-серверной системе они должны храниться на сервере.

Если доступ к данным на SQL Server осуществляется при помощи Access, следует проявлять осторожность при использовании SQL-запросов, которые успешно работали на локальном компьютере. Например, оператор SELECT, который содержит пользовательскую функцию, будет выполняться на локальном компьютере в Access, а не на SQL Server, поскольку SQL Server не поддерживает пользовательские функции в операторах SELECT. (Их не поддерживает и Microsoft Jet; вот почему запрос, который содержит пользовательскую функцию, невозможно выполнить из Visual Basic, даже если он сохранен в файле .mdb.)

При проектировании баз данных для клиент-серверных систем возникает множество сложных вопросов. В библиографическом указателе вы найдете ряд книг, где они освещены достаточно ясно и полно.

Многоуровневая архитектура

В хорошо спроектированной двухуровневой системе можно добиться существенного увеличения производительности и снижения времени отклика по сравнению с одноуровневой. Такой эффект достигается за счет распределения вычислений между клиентской рабочей станцией и сервером. Использование не двух, а большего числа уровней еще более улучшит эти параметры. В четырехуровневой архитектуре (рис. 10-3), как правило, компоненты уровня интерфейса транзакций и уровня внешнего доступа к данным выделяются в отдельные модули и распределяются между дополнительными промежуточными вычислительными системами, объединенными в систему с четырехуровневой архитектурой.

К сожалению, с увеличением числа уровней в системе сложность реализации возрастает экспоненциально. При переходе от двухуровневой архитектуры к трех- и четырехуровневой существенно усложняются доступ к данным, защита данных, управление процессами. Во многих случаях эти процессы становятся настолько сложными и ресурсоемкими, что требуют выделения дополнительных серверов (например, Microsoft Transaction Server). Возможно, именно поэтому их и назвали многоуровневыми. (Видимо, здесь сказалась примитивная житейская логика: «много» — это все, что больше трех).

К счастью, все эти сложности связаны только с реализацией. Вы можете выбрать ту или иную архитектуру — на процесс проектирования базы данных это почти не повлияет. Особо тщательно следите, чтобы логические уровни программной архитектуры были изолированы. Правильно спроектированная схема, работающая в системе с

двухуровневой архитектурой, легко масштабируется и может использоваться в системе с многоуровневой архитектурой без всяких изменений.

Интернет и интранет-архитектура

База данных в Интернет- или интранет-среде — особая разновидность многоуровневой архитектуры. Здесь применяются свои технологии — для передачи данных используется протокол HTTP, а пользователи работают с системой не с помощью Access, а посредством Internet Explorer или другого браузера. Но концептуально архитектуры этих систем очень похожи на обычные многоуровневые системы.

Наиболее существенное различие между системой баз данных, создаваемой для работы в Интернете, и системой для локальных сетей — для Интернет-среды отсутствует понятие *состояния* (state). В обычной клиент-серверной среде приложение запрашивает имя пользователя и пароль при запуске и затем использует эти данные при подключении к SQL Server. После того как соединение с сервером установлено, а имя пользователя и пароль успешно прошли проверку, сервер поддерживает соединение на протяжении всей сессии, Пока поддерживается соединение, сервер хранит информацию о пользователе, и если пользователь отправляет серверу запрос, то сервер отвечает на него. Эта информация о пользователе, поддерживаемая сервером баз данных на протяжении сессии, и называется состоянием,

Если же система проектируется для работы в Интернете, сервер баз данных не может постоянно поддерживать данные о состоянии. Каждый раз, когда приложение обращается к серверу баз данных, соединение устанавливается заново, и выполняется идентификация пользователя. После того как сервер баз данных выполнит запрос, соединение разрывается.

В большинстве случаев дополнительная нагрузка на систему, связанная с установкой соединения при каждом обращении к серверу, мало влияет на СУБД, а на схему базы данных не влияет вообще. Но Интернет, где отсутствует понятие состояния, налагает свою специфику на пользовательское приложение, что в свою очередь, может отразиться на схеме базы данных.

Для большинства приложений, разрабатываемых для Интернета, характерна архитектура, называемая *тонким клиентом.* Это означает, что вычислительная нагрузка распределяется между клиентом и сервером так, что клиентский компьютер выполняет минимум функций; как правило, на нем размещается лишь пользовательский интерфейс.

Особый интерес представляют случаи, когда в результате выполнения запроса клиентскому приложению возвращается очень много

записей, и одновременный вывод всей этой информации пользователю не имеет смысла. В обычных системах возвращаемый набор результатов помешается в кэш либо на клиенте, либо на сервере. Но для приложения, работающего в Интернет-среде, результаты не могут кэшироваться на сервере, поскольку сервер разрывает соединение после окончания выполнения запроса, и после отправки первого пакета данных уже не обладает информацией, куда отправлять следующий пакет. Если же использовать кэширование данных на клиенте, компоненты, выполняющие обработку данных (то есть компоненты уровня интерфейса транзакций и уровня интерфейса внешнего доступа) также должны размещаться на клиентском компьютере. А подобную конфигурацию уже никак нельзя назвать тонким клиентом.

В технологии ActiveX Data Objects (ADO) для таких ситуаций используется механизм, который называется *пейджингом.* Перед тем как отправить запрос серверу приложение может задать число записей, возвращаемых за один раз. Для этого используется свойство *PageSize* объекта *Recordset.* Если задать число записей равным 15, то сервер будет возвращать записи партиями (страницами) по 15 записей за один раз. Свойство *AbsolutePage* позволяет указать, какую именно страницу следует отправить, а свойство *PageCount* возвращает общее число страниц. Это похоже на возможность выбрать первые N записей из отсортированного списка, предоставляемую выражением TOP N стандартного SQL-оператора **SELECT.** Различие в следующем — в технологии ActiveX можно выбрать и отправить пользователю не только самые верхние N записей, но и N записей из середины результирующего набора.

Пейджинг означает, что сервер заново выполняет запрос каждый раз, как только пользователь запрашивает новую страницу. Для запросов, которые выполняются относительно быстро, повторное выполнение практически не влияет на производительность системы. Но повторная обработка сложных запросов, требующая значительного времени может снизить производительность до критического уровня. Приложение, заставляющее пользователя ждать ответа около двух минут, еще может оказаться пригодным, но вообразите себе систему, где несколько сотен пользователей ждут несколько минут, чтобы просмотреть следующие пять записей!

Проблему сложного запроса в приложении, предназначенном для работы в Интернете, можно решить двумя способами. Первый, предпочтительный для большинства подобных случаев, — оптимизировать запрос, чтобы время его выполнения не выходило за разумные пределы. Создайте временные таблицы, денормализуйте данные — сделайте все, чтобы сократить время отклика системы.

Если же оптимизировать запрос невозможно, то скорее всего, останется только прибегнуть к решению, известному пол названием толстый клиент; разместить все компоненты, выполняющие обработку данных, на клиентской рабочей станции. Такая архитектура позволяет кэшировать результаты выполнения запроса на клиентской рабочей станции, воспроизводя таким образом среду, являющуюся аналогом обычной базы данных, доступ к которой осуществляется по сети.

Однако, несмотря на все преимущества толстого клиента, эта архитектура гораздо чаще используется в интранет-приложениях, чем в Интернет-среде, не исключая и Microsoft Distributed Networked Architecture. Причина здесь одна; большинство пользователей не хотят загружать на свой компьютер компоненты кода.

Но когда приложение доступно всем пользователям одновременно, сопротивление возникает гораздо реже. К тому же, скорее всего, информация, доступная пользователям при помощи этого приложения, настолько ценна для них, что они согласятся терпеть некоторые неудобства.

Компоненты схемы базы данных

После того как вы создали концептуальную модель данных и выбрали системную архитектуру, у вас есть вся необходимая информация для создания схемы базы данных.

Схема базы данных представляет собой описание объектов данных, которые содержит база. На схеме указывается также, где будет размещен каждый из объектов, если только база данных и пользовательское приложение не размещаются на одном компьютере.

Для системы, реализуемой средствами Access, схема базы данных содержит определения всех таблиц, запросов и связей, но не описания форм, отчетов и программных компонентов, даже если последние хранятся в файле .mdb.

Если же система создается на основе SQL Server, схема базы данных содержит описания всех таблиц, представлений, хранимых процедур и триггеров.

Таблицы и связи

Определение физических таблиц в схеме базы данных можно получить непосредственно из концептуальной модели данных. Сущности в концептуальной модели становятся таблицами, а атрибуты сущностей — полями таблиц. Такие прямые соответствия позволяют создать основу схемы базы данных. Особого внимания требуют лишь ограничения, связи и индексы.

Ограничения

При построении концептуальной модели данных вы определили ограничения для сущностей, атрибутов и доменов. Будут ли эти ограничения реализованы в схеме базы данных, зависит от того, какую системную архитектуру вы выбрали. Как я уже упоминала, некоторые разработчики реализуют все ограничения на уровнях интерфейса данных и интерфейса транзакций в четырехуровневой модели или на бизнес-уровне — в трехуровневой,

Для большинства случаев я рекомендую реализовать ограничения на обоих уровнях. Если вы согласились с моей точкой зрения и решили включить ограничения непосредственно в базу данных, вам нужно определить их на уровне схемы базы данных. Вопросы реализации целостности данных уже обсуждались в главе 4, и все же вернемся к ним еше раз.

Большинство ограничений, определенных на уровне домена и атрибутов, станут ограничениями на уровне полей в схеме базы данных (в Access это, как правило, правила проверки правильности данных). Если вы решили создавать базу данных при помощи операторов SQL, а не DAO или пользовательского интерфейса Access, воспользуйтесь ограничением CHECK, которое поддерживают и Access, и SQL Server.

Ограничения на уровне сущностей, определенные при создании концептуальной модели, на физическом уровне обычно реализуются как ограничения на уровне таблиц — правила проверки правильности данных или ограничения CHECK языка SQL. Ограничения целостности сущности, позволяющие уникально идентифицировать каждый экземпляр сущности, реализуют, определяя первичный ключ для каждой таблицы.

Если вы реализуете базу данных средствами SQL Server или механизма СУБД Microsoft Jet, может оказаться, что некоторые ограничения, используемые в концептуальной модели данных, невозможно реализовать в определении таблицы. SQL Server позволяет задать дополнительные ограничения при помощи триггеров. Но механизм СУБД Microsoft Jet не поддерживает триггеры, и поэтому такие ограничения следует реализовать в приложении.

Связи

Вопросы моделирования связей между сущностями в реляционной базе данных мы обсудили в главах 3 и 9. Первый шаг в этом процессе — включить уникальный идентификатор из ссылающегося отношения в ссылочное. На уровне схемы базы данных это означает, что поля первичного ключа из ссылающейся таблицы должны быть включены в ссылочную.

Многие разработчики ограничиваются этим, предпочитая реализовать ссылочную целостность только в приложении, а не с использованием средств, предоставляемых механизмом баз данных. Но лично я проповедую комплексный подход: реализую проверку правильности данных и в самом приложении, и непосредственно в базе данных. Таким образом, средства поддержки ссылочной целостности находятся в пользовательском приложении, что обеспечивает удобство работы пользователей с системой. А средства, предоставляемые механизмом баз данных, гарантируют поддержку ссылочной целостности независимо от того, какие компоненты будут использоваться в дальнейшем для работы с этой базой.

Индексы

Как я уже говорила, индексы повышают производительность системы. В каждой таблице должен существовать по крайней мере один индекс; база данных создает его автоматически, когда вы определяете первичный ключ в этой таблице. Следует создать еще один индекс, в который будет включено одно или несколько полей, используемых при соединении таблиц. Как правило, для таблицы, представляющей ссылочное отношение, определить, какие поля следует включить в индекс, очень легко, поскольку поля, используемые при соединении таблиц, составляют первичный ключ. Но для таблицы, представляющей ссылающееся отношение, могут понадобиться дополнительные индексы, поскольку поле или поля, участвующие в соединении, не всегда полностью составляют первичный ключ ссылающегося отношения.

Если поле или поля, составляющие внешний ключ, входят также в первичный ключ, но не составляют его полностью, я создаю в таблице дополнительный индекс, включая в него все поля внешнего ключа. Например, в таблице *OrderItems* (Заказанные продукты) существует первичный ключ, в который входят поля *OrderID, ItemID*. И хотя индекс, определенный на полях первичного ключа, можно использовать для соединения этой таблицы с главной таблицей *Orders* (я считаю, что так можно получить большинство результатов, нужных пользователям), на всякий случай лучше создать в таблице *OrderItems* дополнительный индекс на поле *OrderID*.

Также имеет смысл включить в индекс все поля, по которым будет выполняться сортировка данных в таблице. Списки клиентов в базах данных чаше всего сортируют по фамилиям клиентов, а списки заказов — по дате приема заказа, даже если ни одно из этих полей не входит в первичный ключ и не используется при соединении таблиц. Индексирование таблиц по этим полям существенно ускоряет сортировку данных.

Все же не следует выходить за разумные рамки и создавать индексы, которые не будут использоваться в системе. Поддержка каждого индекса приводит к дополнительной нагрузке на систему. Эта нагрузка не столь велика, если речь идет об одном индексе, но если база данных состоит из множества таблиц, в каждой из которых — несколько индексов, то суммарная дополнительная нагрузка по поддержке всех индексов может оказаться отнюдь не пренебрежимо малой. Если операция сортировки выполняется часто, то каждое поле, по которому сортируются данные, следует включить в индекс. В противном случае данные можно отсортировать при помощи SQL-оператора ORDER BY, не используя индекс.

Максимальное число индексов для таблицы определяется тем, насколько часто обновляется эта таблица — ведь дополнительная нагрузка, связанная с поддержкой индекса, возникает только при добавлении и обновлении данных или обновлении полей индекса Например, для такой таблицы как *Orders* (Заказы), обновляемой достаточно часто, я не рекомендую создавать более 10 или 15 индексов, включая индексы, используемые для соединения таблиц, и первичный ключ. А вот в таблице *Products* (Продукты) индексов может быть больше, поскольку она обновляется сравнительно редко, но интенсивно используется в системе. Как видите, решая вопрос о числе индексов, следует исходить из того, как используются данные в таблице.

Представления и запросы

В Access и SQL Server можно сохранять SQL-операторы SELECT. Эти сохраненные операторы называются ***представлениями*** в SQL Server и ***запросами*** в Access. Я буду называть их запросами, независимо от того, какой механизм баз данных используется, поскольку этот термин шире распространен. В большинстве случаев производительность системы при использовании сохраненных запросов окажется гораздо более высокой, чем при выполнении произвольных запросов на выборку данных. Разумеется, это отнюдь не непреложное правило, для которого не существует исключений, но при проектировании баз данных им вполне можно руководствоваться.

Чтобы определить, какие запросы включать в схему базы данных, следует обратиться к концептуальной модели данных и выделить все составные сущности (напоминаю, что это сущности, представленные в модели данных несколькими таблицами для повышения производительности системы, но на логическом уровне являющиеся отдельными сущностями). Вам потребуется включить в схему базы данных запрос, денормализуюшии все составные сущности, существующие в модели. Большинство составных сущностей — таблицы, участвующие

в связях «один ко многим», как например *Orders и OrderItems.* Но в модели могут присутствовать и составные сущности, участвующие в связях «один к одному»; для их поддержки нужно включить в схему базы данных соответствующие запросы.

Часто нужно найти в базовой таблице запись, удовлетворяющую определенным условиям: например, относящуюся к указанному покупателю, или к заказу, сделанному определенным покупателем, или к заказу, датируемому определенным днем. Это еще один случай, где используются запросы, сохраненные в базе данных и включенные в схему. Как правило, такой поиск может быть реализован при помощи параметрического запроса, позволяющего выполнять поиск в базе данных, изменяя критерии при каждом следующем выполнении запроса.

Иногда для одной сущности нужно использовать несколько разных запросов. Например, чтобы найти заказ, относящийся к определенной дате (используется атрибут *OrderDate*), или заказ, сделанный определенным клиентом (атрибут *CustomerID*), или заказ по регистрационному номеру (атрибут *OrderID*). Для поиска заказов по каждому из этих параметров нужно составлять отдельный параметрический запрос.

С другой стороны, выполняя поиск данных, пользователи не будут обращаться ко всем таблицам. Если, например, в базе данных есть таблица, которая содержит список всех штатов США, маловероятно, что пользователям потребуется ее просматривать, чтобы найти название штата. Такие справочные таблицы чрезвычайно полезны, но к некоторым из них пользователи практически никогда не выполняют запросы.

Нужно также выделить все запросы, используемые в формах и отчетах, которые содержит приложение. Для связывания полей и поддержки некоторых элементов пользовательского интерфейса (например, списков) часто создают отдельные запросы к базе данных. Если в системе существуют зависимости между формами, для поддержки этих зависимостей также могут использоваться параметрические запросы. Пример такой зависимости — диалоговое окно, вызываемое из формы ввода информации о заказе, где отображается вся информация о пользователе.

В зависимости от того, как устроены рабочие процессы в проектируемой системе, вам может понадобиться включить в схему базы данных запросы (и хранимые процедуры SQL Server), которые выполняют определенные действия. Если в системе регулярно будут архивироваться записи о сделанных заказах или обновляться цены на продаваемые продукты, то сохраненные запросы или хранимые процедуры, несомненно, более эффективны, чем составление запросов пользователями.

На этапе реализации, вероятно, в базу данных будут добавлены дополнительные запросы, поддерживающие некоторые действия пользователя. В отличие от индексов, использование запросов и хранимых процедур не приводит к существенной дополнительной нагрузке на систему, и их можно применять без всяких ограничений.

Процесс разработки системы нелинейный. Изменение таблиц на этапе разработки может привести к серьезным проблемам, которые в дальнейшем усугубятся. Но добавление запросов в схему базы данных таких негативных последствий иметь не будет. Поэтому я рекомендую использовать запросы везде, где возможно.

Защита данных

Итак, вы составили представление о рабочих процессах в системе и создали концептуальную модель данных. Теперь нужно заняться вопросами администрирования системы и определить соответствующие требования к ней. Эти требования могут не оказать непосредственного влияния на схему базы данных, но все они представляют собой бизнес-правила, которые должны быть реализованы в окончательной версии системы.

Требования, связанные с администрированием системы — это некие общие требования или метатребования, поскольку они относятся ко всей системе в целом, а не к предметной области, которую система моделирует. Эти требования можно разделить на две группы: определяющие права доступа пользователей к системе и регулирующие доступ к данным. Ко второй группе относится, например, ответ на такой вопрос: должна ли система постоянно находиться в рабочем режиме и обеспечивать непрерывный доступ к данным в течение 24 часов в сутки все семь дней в неделю, или время ее работы можно ограничить теми часами, когда большинство сотрудников находятся на своих рабочих местах. Другой пример требований второй группы — насколько часто пользователи будут создавать резервные копии данных. Поскольку вопросы доступа к данным практически целиком относятся к реализации системы, мы обсудим только требования, определяющие права доступа.

Реализовать политику зашиты данных зачастую весьма непросто. Вам поможет техническая документация к Access, механизму баз данных Micrisoft Jet и SQL Server, где эти вопросы подробно освещены. Кроме того, разработка баз данных и реализация системы — это разные процессы, и при проектировании базы данных вам нужно всего лишь разработать логическую схему защиты данных. А на логическом уровне эти вопросы совсем не сложны.

Уровни защиты данных

Прежде всего, нужно определить уровень зашиты данных, обеспечиваемый проектируемой системой. Здесь мы обсуждаем только вопросы зашиты данных, а не программного кода системы — защита программного кода относится к реализации системы. Access и Visual Basic предоставляют возможность зашиты программного кода от случайного или намеренного повреждения.

На самом низком уровне защиты данных в системе отсутствуют какие-либо механизмы ограничения доступа. Все пользователи имеют неограниченный доступ к данным. Разумеется, этот уровень проще всего реализовать, и такую систему легко администрировать.

И все же, если данные, хранимые в системе, представляют для вас какую-либо ценность, эта политика вряд ли оправдана. Впрочем она допустима, если в вычислительной сети клиента администратор реализовал механизмы ограничения доступа, полностью удовлетворяющие требованиям безопасности. Дублировать ограничения доступа не имеет смысла.

Следующий уровень защиты данных — *общего доступа к данным (share-level security)*. На этом уровне зашиты данных для всей базы назначается единый пароль, и все, кто его знают, получают полный доступ к системе. Этот уровень также легко реализовать, и администрирование такой системы не будет слишком сложным делом: все, что требуется от администратора — регулярно изменять пароль. Во многих случаях защита на уровне общего доступа к данным вполне приемлема.

Защита на уровне пользователя (User-level security), хотя и гораздо более сложна с точки зрения реализации и администрирования, позволяет дифференцировать права доступа пользователей к базе данных. Этот уровень защиты позволяет администратору системы задавать права доступа для каждого объекта в отдельности. Право доступа к определенному объекту может быть предоставлено только нескольким пользователям, причем каждый из них вправе совершать с объектом только те действия, на которые он получит от администратора соответствующие права. Например: пользователь Joe (Джо) может добавлять и изменять данные для сущности *Customers,* однако к данным сущности *Orders* он имеет доступ «только для чтения». Пользователь Магу (Мери) может добавлять и изменять данные для сущностей *Customers* и *Orders.* Однако ни Joe, ни Магу не имеют права удалять записи ни для одной из сущностей.

Название этого уровня — «зашита на уровне пользователя», не совсем точно отражает его специфику. Права на совершение определенных действий с объектами могут быть предоставлены не только отдельным пользователям системы, но и встроенным пользовательским ролям, которые, в свою очередь, назначаются отдельным пользователям. Подобный метод, где права определяются не для каждого пользователя в отдельности, а для определенных пользовательских ролей, более эффективен, поскольку упрощает администрирование системы.

При использовании такой модели в первую очередь следует определить, какие группы пользователей существуют в системе: системные администраторы, сотрудники, вводящие в систему данные о заказах, сотрудники отдела продаж и т. д. После того как станет ясно, сколько групп пользователей будет в системе, нужно определить, какие права на каждый из объектов в системе получит каждая из них. Определяя права для каждой группы, помните, что вовсе не обязательно, и более того, нерационально предоставлять пользователям права доступа к объектам данных. Например, вы решили, что сотрудники отдела продаж должны иметь право добавлять, изменять и удалять записи в таблице *Customers,* но не хотите, чтобы они обращались к этой таблице напрямую. Предоставьте им права доступа к специальной форме, позволяющей выполнять все необходимые действия с записями, но не давайте прав доступа к самой таблице. Такая политика гарантирует, что пользователи не смогут выполнить какие-либо действия в обход специальных правил обработки данных, реализованных в этой форме.

Часто нужно предоставить отдельным группам пользователям доступ только к некоторым фрагментам записей, хранимых в таблицах, а не ко всей записи целиком. Например, право просмотра данных, хранящихся в полях *Name* (Фамилия) и *Extension* (Внутренний номер телефона) таблицы *Employees* (Сотрудники) нужно предоставить всем пользователям системы, но право просматривать информацию, хранящуюся в поле *Salary* (Заработная плата) — только менеджерам. Или требуется ограничить права доступа торговых агентов к данным о заказах, относящимся к деятельности всей компании в целом, предоставив каждому торговому агенту право просматривать информацию только о тех заказах, которые сделаны его клиентами, но не клиентами его коллег. Чтобы реализовать все эти требования, вы можете предоставить пользователям права на выполнение запросов, но запретить доступ к нижележащим таблицам.

Отслеживание и регистрация системных событий

В большинстве случаев возможности управлять доступом к данным недостаточно, нужно также знать, какие действия с данными пользователи выполняли, работая с системой. Объем и степень подробности информации о действиях пользователей зависит от политики компании и может существенно различаться для разных систем. Например, для одной организации вполне достаточно вести журнал учета входа пользователей в систему с указанием времени регистрации. Для другой необходима подробная информация о том, кто и какие изменения данных выполнил за время работы с системой. Третья организация выбирает промежуточный вариант между этими двумя.

Модель, которую вы выберете для учета и регистрации необходимой информации, зависит от конкретных требований организации. Если нужно всего лишь знать, кто и когда работал с системой, то скорее всего, одной сущности с атрибутами *UserName*(Фамилия пользователя), *LogOn* (Время регистрации в системе) и *LogOff*(Время завершения работы с системой) вполне достаточно. При регистрации пользователя в системе будет создана новая запись, которая затем обновится при завершении работы пользователя с системой.

Если требуется также информация о том, кто создал запись в базе данных, то вы можете использовать первичную сущность с двумя дополнительными атрибутами: *CreatedBy*(Пользователь, создавший запись) и *CreatedOn* (Дата создания записи).

Отслеживать операции удаления сложнее, чем изменения в базе данных. Здесь я могу предложить два разных метода. Первый — запретить пользователям удалять записи, а операцию удаления записей заменить установкой флага *Deleted* (Удалена) для соответствующей записи. Информацию о том, кто и когда удалил запись, можно сохранить, добавив атрибуты *DeletedBy*(Пользователь, удаливший запись) и *DeletedOn* (Дата удаления записи). Этот метод подойдет, и если нужно скопировать записи в архивный файл перед удалением их из базы данных,

Другой способ — разрешить пользователям удалять записи, однако записывать все изменения не в базу данных, а в файл журнала. Этот журнал практически ничем не отличается от журнала, который вы ведете при регистрации начала и завершения работы пользователей с системой. При этом вам, скорее всего, потребуется создать дополнительные атрибуты. Но вряд ли информация о том, кто удалил запись, окажется полезной без возможности восстановить эту запись.

Если вы решили хранить подробную информацию о том, кто, когда и какие изменения выполнил в базе данных, то придется добавить в модель дополнительные таблицы регистрации изменений, по одной для каждой сущности. Это позволит регистрировать, кто и когда выполнил изменения, а также новое и старое значения данных.

Реализуя любую из описанных выше возможностей отслеживания событий при помоши механизма баз данных Microsoft Jet, запретите пользователям прямой доступ к таблицам, поскольку он позволяет обойти все механизмы зашиты данных. Для SQL Server запрещать прямой доступ к таблицам не требуется, поскольку этот сервер поддерживает триггеры, которые невозможно обойти.

Определяя требования к модулю регистрации и отслеживания системных событий, помните, как, кто и в каких ситуациях будет использовать эту информацию. Очевидно, доступ к таблицам, где хранится информация о событиях в системе, должен быть ограничен. Может потребоваться определить в системе специальные рабочие процессы регистрации и отслеживания событий. Решая эти задачи, ответьте на несколько главных вопросов. Нужно ли предусмотреть для системных администраторов возможность отменить сделанные изменения и восстановить первоначальные данные? Нужно ли создавать отчеты об использовании системы?

Мой опыт подсказывает, что в большинстве случаев регистрация системных событий — это одна из форм страховки от потери и порчи данных, и информация, записанная в соответствующие файлы, будет использована только в исключительных ситуациях. А если так, то добавлять в систему специальные рабочие процессы вообще не нужно, Системные администраторы могут использовать интерфейс Access или SQL Server Enterprise Manager для доступа к данным и выполнения необходимых действий.

Итоги

В этой главе мы обсудили создание физической схемы базы данных на основе концептуальной модели. Начало главы было посвящено различным архитектурам — трехуровневой и четырехуровневой модели, позволяющим определенным образом структурировать программный код системы. Основное внимание уделялось тому, как выбранная архитектура влияет на схему базы данных.

Мы рассмотрели несколько вариантов архитектуры данных. В одноуровневой системе и приложение, предназначенное для работы с данными, и сами данные размещаются на одном компьютере. Приложение с одноуровневой архитектурой может работать в отсутствие

сети. Или же данные размещены на сетевом ресурсе, но доступ к ним имеет только один пользователь. Приложения, реализованные на основе механизма баз данных Microsoft Jet, в которых пользователи обращаются к данным через сеть, также являются одноуровневыми с логической точки зрения, однако доступ к данным в этом случае имеют несколько пользователей, которые могут работать с базой данных одновременно.

Двухуровневые, или клиент-серверные, приложения можно реализовать на основе SQL Server. Для этой архитектуры характерно распределение операций по обработке и поддержке данных между клиентской рабочей станцией и сервером. Сервер выполняет операции манипулирования с данными, а клиентский компьютер — все операции взаимодействия с пользователем. Принципы, положенные в основу двухуровневой архитектуры, можно развить, увеличивая число уровней и, соответственно, число компьютеров, на которых размещаются компоненты этих уровней.

Далее мы обсудили, как на основе концептуальной модели данных создать схему базы данных. Как правило, такой процесс не вызывает существенных затруднений и не занимает много времени, Единственное, что остается добавить к уже созданным элементам концептуальной модели, чтобы получить рабочую схему базы данных — это индексы и запросы, которые создаются в проектируемой базе данных.

И наконец, я подробно рассказала, какое влияние на схему базы данных оказывают требования к защите данных, и вкратце описала процесс проектирования схемы защиты данных на логическом уровне.

Следующая глава будет посвящена документированию разработки системы и представлению результатов проектирования заказчику и разработчикам, которые будут работать над ее созданием.

Сотрудничество при проектировании

Если вы создаете систему, чтобы с ней работали другие люди, вам придется общаться с ними во время разработки. Заметьте: я сказала *«общаться»*, а не «писать для них документацию». Ни один документ не стоит и ломаного гроша, если его невозможно понять. Вы же не учите иностранный язык, читая словарь? Точно так же нельзя понять суть проекта, ознакомившись лишь с описанием структуры таблиц.

Умение общаться с конечным пользователем — важнейшее качество, которым должен обладать проектировщик. Так что если вы не уверены в своих способностях грамотно работать с конечным пользователем или не сильны в грамматике, почитайте об этом или прослушайте соответствующий курс лекций.

Общение с заказчиком

Всегда нужно иметь четкое представление, к какой аудитории вы обращаетесь. Кто будет читать ваши документы? Для чего? Какую информацию в них следует включить? Ответы на эти вопросы особенно важны на этапе системного проектирования.

Если документ он предназначен для разработчиков, те потребуют включить в него подробности, для обычного человека не более понятные, чем китайские иероглифы. Заказчики же всего лишь хотят убедиться, что вы правильно поняли их требования, и получить подтверждение, что создаваемая система поможет им в работе. Подробное описание реализации системы им не нужно.

Лучше подготовить отдельные документы для заказчиков и разработчиков, особенно если вы применяете итеративную модель разработки.

- **Каталог требований** предназначен непосредственно для заказчиков. В нем содержится общее понимание требований к системе, по возможности, без технической терминологии.
- **Спецификация архитектуры системы** предназначена прежде всего для разработчиков, но с ней нелишне ознакомиться и заказчикам. В спецификации подробно описывают элементы системы и способы взаимодействия между ними.
- **Отдельные технические спецификации для каждого компонента** будут использовать разработчики.

Для небольших систем в принципе достаточно одного единого документа. Просто составьте его так, чтобы у читателей не возникало лишних вопросов.

Структура документа

Если организация, с которой вы работаете, не придерживается жестких стандартов оформления документации, конечная структура вашего документа зависит только от его объема и вашего личного мнения, что следует туда включать. Документ может быть простым или сложным — здесь нет четких правил. Я, разумеется, дам несколько советов, но вам все равно придется действовать по обстоятельствам.

Если же вы последовали моим рекомендациям в части системного анализа, го скорее всего, обнаружите, что созданная вами документация естественным образом разбивается на несколько частей, и ее структура более или менее соответствует структуре второй части этой книги. Разумеется, подобное совпадение не случайно.

Скорее всего, первый раздел вашего документа — введение.

Введение

В больших организациях проекты чаще всего курируют уполномоченные представители заказчика. Даже если проект небольшой, обязательно найдется менеджер, который не вовлечен непосредственно в проект, но тем не менее, курирует его, хотя бы в части финансов.

Именно для таких людей и предназначено введение. Обычно их вовсе не интересуют подробности. Они задают несколько стандартных вопросов, и в ваших интересах ответить им как можно более убедительно. Вот эти вопросы.

- Какие проблемы должна разрешить создаваемая система?
- Является ли выбранное решение оптимальным с точки зрения соотношения «цена — качество»?
- Какие альтернативные варианты рассматривались?
- Каков планируемый срок выполнения проекта?

- Сколько будет стоить система?
- Как оценить риски для данного проекта?

Если вы определили цели и границы применения системы, дать пояснения по первому вопросу будет несложно. Ответ нужно сформулировать четко и ясно. Я обычно очень подробно объясняю, какие функции предполагалось включить в проект, от чего отказались и по каким причинам. После этого уже никто не сможет сказать, что я что-то скрыла от начальства.

Если вы — сторонний консультант, то можете ничего не знать о рассматривавшихся альтернативных вариантах. Тем не менее, советую описать решения, которые могли бы рассматриваться в качестве альтернативных, а также причины, по которым от них лучше отказаться. Если вы проанализировали существующие стандартные системы и сделали выбор в пользу разработки своей собственной, то скорее всего, владеете подробной информацией о ценах, функциональности, технической поддержке и прочих параметрах стандартных решений. Поместите эти данные в приложение к основному документу — объем введения не должен превышать нескольких страниц.

Вопрос о цене и времени разработки довольно каверзный, обычно на него боятся отвечать прямо. Ведь вы, по сути, предлагаете руководству купить свою систему. Если разрабатываемая система невелика, можете оценить ее стоимость на основании того объема кода, который предстоит написать. Для сложных, больших систем такой метод не подходит, но не пугайтесь — пока вам нужно лишь добиться одобрения следующего этапа разработки.

Скажите уверенно что-то вроде: «Сейчас невозможно назвать точные сроки и стоимость разработки, но проект будет завершен примерно в срок от *a* до *b*. Следующая фаза разработки потребует *c* денег и *d* времени, а в результате мы получим то-то и то-то....». Скорее всего, вы получите «добро» на продолжение работ. Но заказчик должен быть серьезно заинтересован в результатах, которые вы ему обещаете по завершении следующей стадии разработки. Люди не склонны выделять средства просто «на дополнительные исследования».

Я также рекомендую честно отвечать на вопрос о рисках. Подумайте, какие опасности могут грозить вам в процессе работы над проектом: технические проблемы, срыв сроков? В каком случае такая опасность больше? Почему? Не бойтесь, что вас сочтут параноиком, лучше перестраховаться.

А теперь спуститесь на грешную землю. Так ли уж велика вероятность, что то иди иное досадное событие действительно случится и приведет к задержке или невозможности выполнять работу? Конеч-

но, никто не может дать стопроцентной гарантии, что проектная группа не умрет в результате эпидемии какой-либо опасной болезни или что на ваш офис не обрушится «Боинг». Но вероятность таких событий ничтожно мала. Разумеется, кое-что в проекте будет иногда идти не так, и вас обязательно спросят, есть ли у вас на этот случай план действий. Не нужно красочно расписывать все возможные неприятности, но очень важно заранее оповестить руководство о возможных проблемах.

Если вы — сторонний консультант, не пытайтесь обойти молчанием вопрос о своей компетентности. Конечно, первое, о чем спрашивают человека, нанимая на работу — сможет ли он ее выполнить. Краткое введение не лучшее место для разговора на эту тему, но упомянуть о рисках, связанных с невозможностью выполнения каких-либо работ, обязательно надо.

Обзор системы

Этот раздел обязательно должен быть первым в основном документе. С ним должны ознакомиться все — и разработчики, и клиенты.

Если вы не включили в начало документа формальное краткое введение, поместите общую информацию в обзор системы, осветив некоторые моменты даже более подробно. Кроме того, иногда требуется посвятить отдельный раздел описанию альтернативных решений или вопросам управления рисками.

Главная задача этого раздела — ознакомившись с ним, участники проекта должны ясно представлять себе общую архитектуру будущей системы. Эту информацию дополнит разъяснение целей проекта и реализуемых функций. Цели проекта не нужно описывать очень подробно — достаточно, чтобы заказчик их понимал и был с ними согласен.

Важно убедиться, что и вы, и ваш клиент представляете будущую систему одинаково. Поэтому я вновь рекомендую как можно более ясно рассказать о функциях системы, от которых отказались на этапе анализа (включая те, которые можно будет реализовать позднее). Опишите и те функции, которые вы не обсуждали с пользователями.

Если вы провели анализ «цена — качество», включите в документ его результаты (не обязательно в раздел «Обзор системы»). Я стараюсь не перегружать основной документ множеством таблиц. Поместите все подробности в приложения к документу — его станет значительно приятнее читать. Но это, конечно, вопрос вкуса.

То же самое можно сказать о целях документа и его объеме. Если вы подготовите подробный анализ функций системы, свяжете их с целями проекта и расположите эту информацию в порядке убывания

приоритетов целей, это поможет вам в процессе реализации. Но учтите, что таблицу, занимающую более страницы, почти никто не станет изучать.

Рабочие процессы

Способ описания рабочих процессов зависит от того, каким образом вы их исследовали. Если вы строили диаграммы рабочих процессов, включите их в документ. Объясните значения символов, которые используете. В любом случае, расскажите, что вы подразумеваете под терминами «процесс», «задача», «деятельность» или любыми другими.

Независимо от того, в какой форме вы представите рабочие процессы, обязательно кратко опишите их. Кстати, это отличный способ проверить правильность своих диаграмм. Уверена, что вы найдете в них множество мелких ошибок.

Кроме того, пользователи чаше всего просто просматривают диаграммы, не пытаясь их понять. Снабдив диаграмму поясняющим текстом, вы облегчите читателю работу,

Если вы предлагаете что-либо изменить в рабочих процессах, включите в документ обе версии процесса — старую и новую, и выделите в тексте описание изменений. И конечно, нужно объяснить, почему необходимы предлагаемые изменения. Не исключено, что в ходе анализа вы что-то упустили, и пользователи, ознакомившись с вашими соображениями, укажут вам на это.

Описание рабочих процессов — потенциальное поле битвы различных групп читателей вашего документа. Разработчики при обсуждении рабочих процессов будут оперировать техническими терминами, вроде: «транзакции были завершены, и данные изменились». С другой стороны, заказчики хотят увидеть описание процесса в привычных им терминах. Эти термины порой совпадают с техническими, а порой — сильно отличаются от них. Когда такая ситуация возникает, то правда всегда на стороне заказчика. Разработчики могут принять терминологию клиента, но не наоборот. Если в документе приходится использовать чисто технические термины, обязательно приведите тут же их подробное разъяснение.

К сожалению, это легче сказать, чем сделать. Когда все время имеешь дело с компьютерными системами, легко забыть, что многие из часто используемых тобой терминов не являются общеупотребительными. Я веду специальный список терминов типа «транзакция» или даже «файл» и просматриваю окончательный вариант документа на предмет наличия таких терминов перед отправкой его заказчику. Это довольно легко сделать с помощью команды Find текстового процес-

сора. Помните: если клиенту будет хоть что-нибудь непонятно, то скорее всего, ваш документ вызовет у него лишь раздражение.

Концептуальная модель данных

По завершении предварительного анализа у вас появился ряд диаграмм «сущность — связь», список доменов и описания правил ограничения целостности. Наилучший способ представления информации о модели данных — диаграммы «сущность — связь». Обычно я создаю отдельный словарь доменов и ссылаюсь на него в диаграммах.

В этой части документации никак нельзя избежать множества таблиц. И вашим читателям, хотят они этого или нет, придется с ними ознакомиться. Единственное, чем вы можете облегчить их долю – попытайтесь сделать этот процесс менее утомительным.

Во-первых, используйте минимум технической терминологии. «Таблица», «поле» и «запись» — по-видимому, неизбежное зло. А терминов «сущность», «отношение» и «атрибут» лучше избегать, хотя, конечно, они более точны с технической точки зрения. Убедитесь, что объяснили значение технических терминов (только не пытайтесь прочесть краткий курс по базам данных!). Не так уж это и сложно: скажите, что таблица представляет в базе данных нечто из предметной области (например, счет покупателя), а поле содержит значение свойства этого «нечто» (например, номер счета), — этого вполне достаточно, чтобы двигаться дальше. Разумеется, возможны дополнительные вопросы, но я рекомендую отвечать на них по мере возникновения.

Если документ должен содержать еще и чисто техническую спецификацию, предназначенную для разработчиков, опишите эти детали отдельно, в качестве приложения к спецификации сущностей. Советую специально указать заказчику, что ему следует ознакомиться с описанием сущностей и доменов, но пропустить все остальное. Теоретически, ему следует ознакомиться со структурой связей между сущностями. Но, поверьте, это не принесет никакой пользы — заказчик просто не в состоянии обнаружить ошибку, даже если она там есть.

Зато он обязательно должен просмотреть типы и размеры полей, причем сделать это лучше при личной встрече. Это довольно утомительно, но совершенно необходимо. В крайнем случае, выделите в тексте места, на которые хотите обратить особое внимание заказчика.

Схема базы данных

Так как сама по себе схема базы данных является расширением концептуальной модели данных, ее документирование основано на тех же принципах.

Схема базы данных представляет интерес для разработчиков, о ней совершенно незачем знать заказчику. Если вы не описываете схему базы данных в отдельном документе, поместите ее в раздел приложений к основному тексту, А вот саму архитектуру системы и концепцию безопасности заказчик обязательно должен одобрить. Чтобы лучше донести до него эту информацию, используйте диаграммы и текстовое описание архитектуры — эти два способа представления информации дополняют друг друга. Концепцию безопасности лучше описать в виде текста, но иногда рисунки и графики более наглядны, особенно если система достаточно сложна.

Интерфейс пользователя

Лучше всего заранее обсудить с заказчиком, как должен выглядеть интерфейс пользователя (хотя для небольших систем это необязательно).

Даже если вы не собираетесь разрабатывать интерфейс отдельно от остальной части проекта, неплохо все же показать заказчику несколько примеров пользовательских форм, назвав их «черновиками». Такие примеры помогут ему лучше представить себе внешний вид системы. Но будьте начеку: после этого от вас будут ожидать именно таких экранных форм в конце разработки (неважно, что вы все время твердили заказчику: то, что он видел — только примеры). Если конечный вид системы будет сильно отличаться от показанных пользователю «черновиков», все ваши благие намерения обернутся против вас.

Обычно проблемы возникают, если из проекта «ушла» часть первоначально предполагавшихся функций вместе с соответствующими формами. Заказчик будет ожидать от вас всего набора форм, который ему показывали ранее, даже если он сам впоследствии и одобрил отказ от упомянутых функций и знал об изменениях в интерфейсе. Я, как правило, помещаю «исчезнувшие» формы в специальный раздел: «Функции, от которых отказались». В этом же разделе следует дать объяснение причин отказа.

Как только вы спроектировали интерфейс, покажите его будущим пользователям системы. Для этого есть два способа: создание прототипа системы и спецификации интерфейса. Обычно я готовлю обе презентации, если конечно, создание прототипа не обеспечивает явных преимуществ.

Прототип интерфейса

Я считаю прототип интерфейса лучшим способом продемонстрировать заказчику, что будет представлять собой система (большинству пользователей в силу отсутствия опыта сложно понять это только на основании рисунков).

Прототипы помогут решить множество задач. Самое простое - создать внешний вид меню и форм, не связывая их с рабочими процессами. В таких прототипах присутствуют все необходимые элементы управления, но к ним не привязывается никакая реальная функциональность. При выборе тех или иных пунктов меню на экран выдаются стандартные сообщения, например: «Эта команда будет выполнять то-то и то-то. Она не реализована в прототипе».

Реальная функциональность появляется в прототипе, только когда внешний вид пользовательского интерфейса тесно связан с данными. Например, вы проектируете интерфейс для ввода заказа, однако его внешний вид определяется тем, кто является заказчиком - физическое или юридическое лицо. Тогда придется написать код. позволяющий пользователю выбрать тип заказчика и отобразить соответствующую экранную форму в прототипе.

Для создания прототипов подойдут те же инструменты, что и для реальной разработки. Это очень удобно, но и у такого метода есть свои недостатки. Вы ведь создаете прототип, а не саму систему, и потому можете не обратить внимания на ряд ограничений, которые в дальнейшем станут препятствием на пути создания системы.

Очень трудно не поддаться соблазну и не использовать прототип в качестве скелета системы. В конце концов, если все меню и экранные формы уже созданы, не переписывать же их заново? Но экранные формы и меню в прототипе не более чем примеры. Если вы будете использовать прототип как основу системы, то столкнетесь с теми ограничениями инструментальных средств, которые благополучно обошли, создавая примитивное приложение-прототип.

Поэтому многие архитекторы рекомендуют создавать образцы экранных форм с помощью дизайнерских систем, а не средств разработки типа Microsoft Access или Visual Basic. Используйте средства, наиболее удобное для вас. Для меня таковыми являются инструментальные средства разработки. Вы можете применить дизайнерские системы или даже системы создания презентаций наподобие Microsoft PowerPoint. Просто помните о том, что создаете не саму систему, а пример, демонстрирующий ее внешний вид.

Спецификации интерфейса

Итак, создание прототипа — замечательный способ согласовать с пользователем внешний вид системы. Но возможности прототипа в части функциональности весьма ограничены. По этой причине он не способен заменить спецификации интерфейса. А вот обратное неверно: тщательно подготовленные спецификации могут заменить прототип.

Как и документация, описывающая модель данных, спецификация пользовательского интерфейса всегда включает в себя некоторый объем чисто технической информации. Здесь я могу только повторить свои рекомендации: используйте как можно меньше сложных терминов и всегда объясняйте пользователю их значение, ограждая его от информации технического характера везде, где это возможно.

Если вы создали прототип, подготовить спецификации интерфейса несложно. Я обычно включаю в документ изображение каждого из диалоговых окон, краткое описание его назначения, список элементов управления, описание источников данных (если они есть) и реализуемой функциональности. Если у вас нет прототипа, придется рисовать формы вручную, все остальное остается в силе.

Для большинства систем в документ следует включить описание процесса обработки данных, представив последовательность вызовов экранных форм для каждого из рабочих процессов. Это легко сделать с помощью диаграмм рабочих процессов.

Контроль за изменениями

Ваш документ наверняка будет неоднократно изменяться в ходе работы, и этот процесс нужно контролировать. Заметьте: «контроль за изменениями» не то же самое, что «постоянное сохранение предыдущих версий» (я не думаю, что такой подход вообще можно реализовать). Но если вы учтете возможность внезапных изменений, то сильно облегчите себе жизнь.

Для этого есть несколько способов. Первый — не исправлять документ непосредственно, а создать список изменений. Разумеется, это подойдет, если изменения не слишком значительны, в противном случае лучше исправить текст самого документа.

Если можно задать определенное место хранения последней версии документа, то это серьезно поможет вам. Отслеживайте изменения средствами текстового процессора, например, Microsoft Word, сохраняя документ на файловом сервере или в интрасети. Единственная проблема — заставить людей работать именно с электронной версией документа, а не с ее бумажными копиями, которые могут устареть к тому времени, как пользователь начнет читать документ.

Но в большинстве случаев способ контроля не столь важен, сколь сам его факт.

Специальные средства

Работать над документацией вам помогут два средства; Microsoft Visual SourceSafe и Microsoft Visual Component Manager. Visual SourceSafe создавался прежде всего как средство контроля за изменениями исход-

ного кода программ, но его можно использовать и для контроля за версиями проектной документации.

Visual SourceSafe очень полезен, если над созданием системы работают несколько проектировщиков. Он исключает возможность уничтожения одним человеком изменений, внесенных остальными членами группы, и гарантирует, что все работают только с последней версией документа.

Visual Component Manager — клиентская часть Microsoft Repository. Он тесно связан с Enterprise Edition Microsoft Visual Studio и предназначен для администрирования документации и компонентов, открытых для общего пользования (а не для помоши в процессе разработки). Visual Component Manager позволяет создавать индивидуальные хранилища информации для каждого участника проекта. Это очень удобно, чтобы связать несколько документов на различных пользовательских компьютерах в одно целое.

Если проект разрабатывается с помощью Visual Studio, та же самая база данных может быть использована для контроля за версиями разрабатываемых компонентов. К сожалению Visual Component Manager не интегрирован с Microsoft Access, хотя теоретически можно ввести в Repository такие возможности.

Итоги

В этой главе я рассказала об основах коллективной разработки и попыталась дать несколько полезных рекомендаций. Вы можете самостоятельно решить, подходят ли они вам.

Эта глава завершает вторую часть книги, но мы еще не закончили изучение процесса проектирования базы данных. В третьей части мы вернемся к наиболее ответственному его этапу — проектированию пользовательского интерфейса.

Проектирование пользовательского интерфейса

Интерфейс как посредник между пользователем и системой

ГЛАВА

По завершении стадии системного анализа, о которой шла речь в предыдущей части этой книги, у вас должно сложиться четкое представление о том, какие функции должна выполнять разрабатываемая система. Третья часть книги посвящена вопросам разработки пользовательского интерфейса.

В этой главе мы рассмотрим основные подходы к проектированию интерфейса и уделим внимание различным моделям, используемым при этом. Для детального освещения всех вопросов потребовалось бы несколько объемистых томов, поэтому я ограничусь лишь некоторыми общими моментами. Тех, кто хочет подробнее ознакомиться с проблемой, я адресую к литературе, перечисленной в библиографическом указателе.

Роль пользовательского интерфейса в системе

Множество пользователей отождествляют пользовательский интерфейс и саму систему. Это неудивительно, поскольку большинство из них взаимодействует с системой только через интерфейс, все остальные компоненты от конечного пользователя скрыты. Поэтому от разработки пользовательского интерфейса в значительной степени зависит успех всего проекта. Правильно спроектированный интерфейс обеспечит положительное отношение пользователей к системе, возможно, они даже простят вам некоторые промахи в реализации, без которых не обходится ни один проект. Если же интерфейс спроекти-

рован плохо, то ни мастерство разработчиков, ни оптимизация программного кода с целью повышения производительности не принесут проекту успеха.

Увы, если интерфейс спроектирован хорошо, этого, как правило, никто не замечает. Изящные решения в области пользовательского интерфейса естественны и потому не бросаются в глаза. Но и если вы допустите ошибку, весьма вероятно, что она так и останется незамеченной. Интерфейс большинства компьютерных систем, с которыми мне приходилось сталкиваться (в особенности, использующих базы данных), настолько не удобен, что вряд ли тот, что создан вами, окажется самым плохим.

Но если никто не заметит ни ваших успехов, ни ваших промахов, стоит ли вообще уделять этой проблеме внимание? Разумеется, стоит. В конце концов, в этом заключается ваша работа, и ее нужно сделать как можно лучше. Спроектировать рациональный пользовательский интерфейс несравненно сложнее, чем просто предоставить пользователю более-менее пригодные средства доступа к данным. Часто реализация удобного пользовательского интерфейса отнимает у разработчиков достаточно много времени и сил. И все же в конце концов затраченные усилия окупаются, приносят не только моральное удовлетворение, но и весьма ощутимую реальную выгоду. Хорошо спроектированный интерфейс позволяет существенно снизить время, необходимое пользователям, чтобы научиться работать с системой, а следовательно, существенно облегчает процесс ее внедрения. Кроме того, рациональный пользовательский интерфейс способствует повышению производительности, поскольку работать с ним удобно и легко.

Тщательно продуманный интерфейс, отвечающий требованиям пользователей и разработанный с учетом специфики рабочих процессов, снижает затраты на создание сопроводительной документации. И даже если пользователи не дадут созданному вами интерфейсу столь высокой оценки, какой он, безусловно, заслуживает, может быть, они все же скажут, что ваша система гораздо более эффективна, чем та, что разработана компанией «Тяп-ляп-системз, инкорпорейтед». А лучшай реклама для разработчика — выполненные им проекты.

Итак, каковы же основные критерии, позволяющие отличить хорошо спроектированный интерфейс от плохо спроектированного? Я считаю, что хороший интерфейс прежде всего эффективно помогает пользователям решать стоящие перед ними задачи, не создавая дополнительных трудностей в работе. Такой интерфейс не навязывает пользователям свои собственные «правила игры», совсем напротив — он настолько органично вписывается в рабочий процесс, что почти

не заметен. Он прост и удобен; пользователям не нужно держать в голове множество запутанных правил. И, конечно, хорошо спроектированный интерфейс никогда не преподносит пользователю «сюрпризов» — система всегда работает стабильно и предсказуемо.

Обо всех этих критериях оценки полезности и удобства пользовательского интерфейса мы еще поговорим. Но прежде рассмотрим три модели, которые помогут вам создать концепцию разработки пользовательского интерфейса.

Модели интерфейса

В книге «About Face: The Essentials of User Interface Design» («Лицо системы: основные принципы разработки пользовательского интерфейса») Алан Купер *(Alan* Cooper) выделил три аспекта восприятия пользователями различных компьютерных систем (и наоборот, реализации определенного подхода компьютерных систем к пользователям). Это ментальная модель, декларируемая модель и модель реализации. Все они — мощные инструменты для разработки концепции пользовательского интерфейса приложения.

Ментальная модель (mental model) описывает представление пользователя о том, что происходит в системе. Как правило, представления пользователей основаны на упрошенных схемах и не совсем точных аналогиях, тем не менее, пользователь имеет вполне адекватное представление о системе. (Все это почти не влияет на сам процесс проектирования интерфейса.)

Например, с точки зрения пользователя текстового процессора, работающего с различными документами, нажатие на клавишу клавиатуры приводит к появлению символа на экране монитора — это ментальная модель. На самом деле в системе происходят значительно более сложные процессы, но они выполняются независимо от пользователя и потому не относятся к ментальной модели.

Упомянутые процессы являются частью *модели реализации* (implementation model), К ней относится все, что касается «механики», приводящей в действие систему, или проблем, связанных с программным кодом. Пользователей эти вопросы не интересуют и не должны интересовать.

Пользовательский интерфейс представляет собой *декларируемую модель* (manifest model), которая находится между ментальной моделью пользователя и моделью реализации, создаваемой и используемой разработчиками. Это модель процесса, которую система представляет (декларирует) пользователю. При проектировании интерфейса цель разработчика — сделать незаметными для пользователя как можно больше элементов модели реализации. В идеальном слу-

чае модель реализации полностью изолирована от пользователя, и поэтому идеальная система практически полностью соответствует ментальной модели. Конечно, на практике редко удается добиться полного соответствия между поведением системы и ментальной моделью, но следует к этому стремиться.

Большинство читателей этой книги, очевидно, обладает достаточно глубокими знаниями в области информационных технологий, и поэтому их ментальная модель отличается от ментальной модели рядового пользователя. Например, моя ментальная модель при пользовании текстовым процессором такова: когда я нажимаю клавишу на клавиатуре, то соответствующий код ASCII сохраняется в оперативной памяти компьютера. Разумеется, и это представление далеко от модели реализации, однако оно не похоже и на представление среднего клерка.

При проектировании пользовательского интерфейса, однако, сложно отвлечься от аспектов реализации. Ведь даже если вы не принимаете непосредственного участия в разработке, то все равно посвящены в некоторые детали этого процесса. А значит, нужно либо изобрести способ, позволяющий на некоторое время полностью забыть о модели реализации системы, либо попросить нескольких пользователей описать эту систему, чтобы воспользоваться их ментальной моделью.

Наилучший вариант — формальное тестирование прототипа системы на удобство пользовательского интерфейса. К сожалению, этот подход редко используется на практике, но мой опыт показывает, что подобные эксперименты оправданы. Создав прототип системы, предоставьте пользователям возможность поработать с ним некоторое время. Затем попросите их рассказать, что, по их мнению, происходит, когда они выполняют те или иные действия. Ручаюсь, их ответы будут сильно отличаться от того, что вы ожидали услышать. В отсутствие прототипа системы вы можете, например, расспросить пользователей, как они представляют себе работу с системой. Или нарисуйте основные элементы пользовательского интерфейса на бумаге и попросите пользователя их прокомментировать. Эти методы хоть и не столь результативны, как работа с прототипом, но все же полезны.

Затем проанализируйте собранную информацию. Определите, где модель реализации (то есть то, что реально происходит в системе) расходится с ментальной моделью пользователей и выявите противоречия. Подумайте, в чем их причина? Может быть, ваша терминология просто непривычна для пользователей? Мой совет — всегда использовать те же термины, что и пользователь. Например, некое действие у вас может называться «редактирование записей», а пользователи говорят «изменение адреса». Скорее всего, вы просто по-разному назы-

ваете одно и то же, но возможны и ошибки в структуре системы. Подробней об этом — в следующей главе.

Уровни подготовки пользователей

Возможно, удобство системы с точки зрения пользователя не было заложено в основу всею проекта. И все же вряд ли кто-нибудь из разработчиков сознательно ставит перед собой цель создать систему с «недружественным пользовательским интерфейсом». Проблема в том, что за словами «дружественный интерфейс», как правило, не кроется практически никакого конкретного смысла, и вам придется поломать голову в поисках более точных критериев. Обычно под этими словами подразумевают «интерфейс, с которым легко работать» и «интерфейс, обеспечивающий удобство и простоту использования».

Оставив пока в стороне вопрос, что означает «легко», попробуем выяснить, для кого это должно быть легко. Может оказаться, что новичок легко научится работать с вашей системой, но опытному пользователю, уже работавшему с аналогичными системами, будет трудно сориентироваться. Лучше всего учесть при проектировании интерфейса требования пользователей всех уровней, разработав специальные средства для каждого.

Начинающий пользователь

«Все когда-нибудь приходится делать в первый раз», гласит поговорка, и каждый пользователь когда-то был новичком. Однако лишь очень небольшое число пользователей остаются на этом уровне всю жизнь — большинство из них довольно быстро превращаются в опытных пользователей, а затем в экспертов. При проектировании любой системы следует учитывать, что пользователи либо сравнительно быстро научатся работать с ней, либо предпочтут продукт, разработанный другой фирмой. Поэтому нужно помнить одно простое правило: не увлекайтесь средствами, которые помогут новичкам, но помешают работе более опытных пользователей.

Начинающим пользователям следует знать, для чего предназначена система и что они могут сделать с ее помощью. Ваша задача — объяснить им это, по возможности не заставляя повторно знакомиться с системой тех, кто более опытен. Разместите краткую информацию о системе и ее основных возможностях отдельно. Для простой системы достаточно одного или нескольких диалоговых окон, которые содержат ее описание. (Разумеется, следует предоставить пользователям возможность отключить вывод этого окна). Для более сложных систем, вероятно, потребуется специальный учебный курс.

Интерактивная справочная система — не самый лучший вариант для новичков. Они могут не знать о ее существовании или не уметь ею пользоваться. Чтобы решить эту проблему, разместите ссылки на интерактивное руководство пользователя в диалоговом окне, содержащем краткую информацию о системе и открывающемся при первом ее запуске, а также в меню Help.

Интерактивные справочные системы и пособия, рассчитанные на начинающих пользователей, должны быть ориентированы на решение типичных для новичков проблем. Таким пользователям нужны не столько толкования различных терминов: например, «пункт раскрывающегося меню» или «настраиваемая панель инструментов», — сколько четкие указания, как выписать счет-фактуру.

Опытный пользователь

К этой категории относится большинство пользователей различных компьютерных систем. Опытные пользователи, как правило, знают, для чего предназначена система и как она работает, но часто забывают, какие именно действия нужно выполнить, чтобы получить тот или иной результат. При проектировании пользовательского интерфейса вы должны ориентироваться именно на эту группу. В Microsoft Windows есть множество различных средств, чтобы оказать помощь этим пользователям.

В первую очередь, это хорошо продуманная организация команд меню — своего рода подсказка, напоминающая опытным пользователям о возможностях системы и позволяющая быстро выбрать нужное.

Второй уровень поддержки опытных пользователей — интерактивная справочная система. Детальное описание процесса разработки интерактивных справочных систем выходит за рамки этой книги, ограничусь лишь одним замечанием. Поскольку большинство опытных пользователей будут активно использовать индекс при работе с интерактивной справочной системой, он должен быть как можно более полным.

Эксперт

Пользователи, относящиеся к этой группе, отлично знают, как работает данная система, а также хорошо помнят, какие действия нужно выполнить для решения той или иной задачи. Большинство экспертов хотят выполнять определенные действия как можно быстрее и эффективней. Поэтому чем больше средств, позволяющих ускорить выполнение часто используемых операций, вы предусмотрите, тем более удобной окажется ваша система для пользователей-экспертов. Назначьте для каждой команды или другого элемента определенное

сочетание «горячих клавиш». Предоставьте пользователям возможность переопределять стандартные сочетания клавиш. Пользователи-эксперты очень любят эти средства, и вам следует обеспечить им максимально быстрые и рациональные способы выполнения тех или иных действий.

Эксперты высоко ценят возможность пользовательской настройки приложений, с которыми работают. Однако реализация подобных возможностей зачастую весьма трудоемка, и я советую семь раз подумать, прежде чем отважиться на такой шаг. Если же вы все-таки решили предоставить пользователям большую свободу и включили в систему функции пользовательской настройки интерфейса, уделите особое внимание тому, как будут сохраняться индивидуальные пользовательские параметры по завершении сеанса работы с системой. Ничто так не раздражает пользователей, как необходимость снова настраивать приложение при каждом входе в систему.

Возложите на пользователя ответственность за его действия

Повторюсь: выражение «дружественный пользовательский интерфейс», часто встречающееся в спецификациях и других важных документах — расплывчатое, а зачастую просто бессодержательное. Пожалуй, следующий по частоте употребления штамп — «система, ориентированная на нужды пользователя». На мой взгляд, в последнем выражении все-таки больше смысла, чем в термине «дружественный» — под ним подразумевается система, которая во всем отвечает требованиям пользователя и не навязывает ему собственных методов работы.

Проиллюстрирую свои слова конкретным примером. Один разработчик как-то описал мне придуманный им метод, гарантирующий, что пользователи будут вводить данные в том порядке, который наиболее подходит для разрабатываемой системы. Суть проста: сначала пользователю доступен только один элемент, позволяющий ввести данные. Затем, когда ввод закончен, становится доступным второй элемент, потом — третий и т. д. Я считаю, что этот метод не только не позволяет реализовать систему, ориентированную на нужды пользователя — он вообще не позволяет реализовать мало-мальски работоспособную систему!

На первый взгляд может показаться, что пользователи не придерживаются строго определенного порядка ввода информации лишь вследствие рассеянности, неаккуратности или потому, что никто не принуждает их соблюдать этот порядок. Однако это не так. Как правило, есть достаточно веские причины, побуждающие пользователя

ввести одни данные, затем пропустить несколько полей или списков: например, соответствующая информация отсутствует или ее не следует вводить в данный момент. Затем пользователь намерен продолжить ввод информации, которой уже располагает. Если же система будет принуждать его непременно ввести какое-либо значение в поле *A*, прежде чем он сможет ввести информацию в поле *B* — что же, он это сделает! Он может ввести недостоверную информацию или устаревшие данные. И система сохранит введенные данные, которые не будут соответствовать действительности. В результате борьба с «недисциплинированными» пользователями приведет лишь к созданию системы, ценность которой весьма сомнительна.

К этой проблеме мы еще вернемся в главе 16, где будем обсуждать вопросы целостности данных. Искусственное усиление целостности данных — основной способ, с помощью которого СУБД ограничивают свободу пользователей. Второй способ — использовать *режимы* (modes), то есть состояния системы, одна из функций которых — ограничивать взаимодействие с ней пользователей. Стандартные режимы для систем баз данных — добавление, изменение и просмотр. Однако система, в которой пользователь должен выбрать одну из команд меню или щелкнуть кнопку, прежде чем сможет изменить запись, которую только что просмотрел, крайне неэффективна.

К сожалению, многие разработчики выбирают именно такой путь — очевидно, за незнанием лучшего способа предотвратить порчу данных вследствие ошибки или невнимательности пользователя. Лично я вижу здесь слепое копирование традиционной модели интерфейса, использовавшейся 20 лет назад, с ее непременными атрибутами — командами Add, Edit и View в главном меню. Эту парадигму некоторые разработчики пытаются перенести в среду Windows, где она совершенно неуместна. Чтобы избежать подобных ошибок, рекомендую избавиться от идеи излишней опеки над пользователями. Если пользователь выполняет какие-либо действия — значит, он знает, что делает. Если пользователь хочет изменить запись, он должен иметь возможность сделать это, не спрашивая ни у кого позволения.

Но предоставляя пользователям свободу, позаботьтесь и о том, чтобы случайные действия не привели к порче или потере данных. Достичь этой цели помогут команды отмены и повтора нескольких последних действий, легко реализуемые средствами Microsoft Access и Microsoft Visual Basic. Кроме этого, включите в меню команду, позволяющую вернуться к последней сохраненной версии, отменив все изменения, сделанные в текущей записи.

Есть еще один прием — я использую его редко, хотя в некоторых ситуациях он вполне оправдан: перед сохранением измененной записи система запрашивает у пользователя подтверждение этого действия. На мой взгляд, сама идея сохранения чужда большинству пользователей. Вспомните ментальную модель пользователя, которую я приводила в качестве примера. Введя или отредактировав данные, он полагает, что уже изменил запись, и если теперь спросить его, хочет ли он изменить ее, пользователь может вас неправильно понять. В любом случае дополнительное окно запроса, возникающее на экране перед сохранением записи — не слишком эффективный способ заставить пользователя лишний раз проверить, правильно ли он ввел данные. Многие щелкают OK всякий раз, когда видят это окно, просто по привычке.

Впрочем, всегда можно придти к компромиссу. Однажды настойчивый клиент просто вынудил меня реализовать запросы на подтверждение действий в проектируемой системе. Но одновременно я предоставила пользователям возможность отключить вывод этих сообщений.

К сожалению, нередки и иные ситуации — когда некоторые из выполненных изменений невозможно или очень сложно отменить, Например, легко отменить изменения записи или данных в одном из полей таблицы, но удалив псе записи из таблицы, вы вряд ли сможете их потом восстановить, Поэтому, предоставив пользователю возможность выполнять такие «опасные» действия, позаботьтесь, чтобы их результаты было легко отменить, восстановив данные в их первоначальное состояние. Если же это невозможно, прибегните к методу, излишнему при выполнении обычных операций — попросите пользователя подтвердить, действительно ли он хочет выполнить эти действия.

Избегайте неинформативных системных сообщений. Четко объясните пользователю, в чем заключается опасность, и к чему могут привести те действия, которые они собираются выполнить. Например, сообщение на рис. 12-1 только встревожит пользователя, не дав никакой полезной информации.

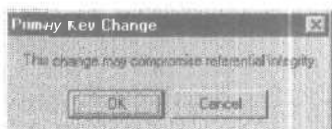


Рис. 12-1. «В результате изменений может нарушиться ссылочная целостность» — пример неинформативного сообщения

Сообщение на рис. 12-2 составлено значительно лучше. Оно не только дает объяснение тому, что произошло, на понятном пользователю языке, но и предоставляет возможность предпринять определенным образом спланированные действия вместо стандартного выбора между кнопками OK и Cancel.

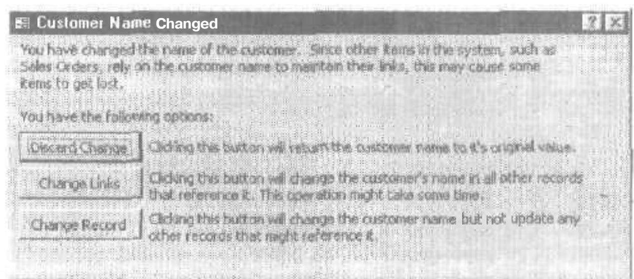


*Рис. 12-2. **Такое сообщение объясняет пользователю ситуацию и предоставляет возможность спланировать** свои действия*

Не перегружайте память пользователя!

Память любого человека не беспредельна, и это одна из причин, по которой люди пользуются компьютерами. Хорошо спроектированный интерфейс не заставляет пользователя усваивать больший объем информации, чем тот, который действительно необходим для работы с системой. Очевидно, что пользователи должны узнать (и запомнить), что вообще делает ваша система. И, конечно же, им придется выучить последовательности действий, которые нужно выполнить, чтобы получить определенные результаты. Но вы должны стремиться уменьшить объем информации, который пользователи будут держать в памяти.

Нагрузка на память пользователя снизится, если вы будете соблюдать стандарты разработки интерфейса приложений для Windows, содержащиеся в руководстве «Windows Interface Guidelines for Software Design» («Руководство по разработке интерфейса приложений для Windows»). Вам вряд ли придется часто отступать от них. Продукты Microsoft Office (в частности, Access) де-факто определяют и используют другие стандарты. Но прежде чем изменять существующие стандарты, убедитесь в наличии для этого серьезных оснований. Иначе вы; вероятнее всего, окажете пользователям медвежью услугу.

Пользу стандартных решений можно проиллюстрировать следующим примером. Предположим, вы разрабатываете систему для пользователей, привыкших работать с приложениями Microsoft Office, и в частности, с Access. При этом интерфейс Access кажется вам, разра-

ботчику, идущему в ногу со временем, весьма бесцветным и не богатым графическими элементами. И вот вы решаете подобрать собственную коллекцию пиктограмм для кнопок панели управления. Но придут ли пользователи в восторг от такой красоты? Ведь они уже привыкли к тому, что при нажатии кнопки Access выполняет переход к следующей записи в таблице. А теперь им придется запоминать, что для выполнения аналогичного действия в разработанной вами системе нужно щелкнуть кнопку, на которой красуется фотография вашей любимой собаки или загадочный символ, позаимствованный из абстрактной живописи.

Разумеется, рекомендации по разработке интерфейса приложений для Windows, приведенные в упомянутом руководстве, содержат некоторые общие правила, а отнюдь не сборник готовых рецептов на все случаи жизни. В большинстве реальных ситуаций вам придется принимать решение самостоятельно, основываясь на этих правилах. Но определив правило, строго его придерживайтесь, иначе вы рискуете испортить собственную работу. О согласованности пользовательского интерфейса мы поговорим в следующем разделе.

Чтобы уменьшить нагрузку на пользователя, исключите ситуации, когда от него будет требоваться повторный ввод данных. Здесь часто помогают значения по умолчанию. Если пользователи будут вводить информацию, используя несколько последовательно открывающихся и связанных друг с другом форм, позаботьтесь, чтобы все связанные Друг с другом данные автоматически переносились из одной формы в другую.

Избегайте ввода данных вручную, пусть пользователь, везде, где это имеет смысл, выбирает их из списка. Например, чтобы получить сведения о покупателе Джоне Ду (John Doe), пользователь не должен ломать голову над тем, как именно в базу данных были введены эти имя и фамилия — John Doe, J. Doe или J.C. Doe. Но обратите внимание: я сказала «там, где это имеет смысл». Вряд ли разумно заставлять пользователя ждать, пока система создаст список из 65 тыс. записей, и выведет его на экран. Очевидно, в подобном случае пользователь должен иметь возможность отфильтровать список и из ограниченного набора значений выбрать нужное.

Чтобы списки были содержательными, включите в них как можно больше дополнительной информации. Пользователи вашей системы не обязаны помнить, например, что Джон Смит (John Smith) — это клиент их фирмы, живущий в Мадриде, а Джонни Смит (Johnny Smith) — покупатель из Милана. Стандартные элементы управления Microsoft Access — списки и комбинированные окна, позволяют ото-

бражать содержимое нескольких полей. В Visual Basic можно использовать конкатенацию данных, хранящихся в соответствующих полях.

Возможно, вам покажется интересным и такой вариант, легко реализуемый как средствами Microsoft Access, так и Visual Basic: чтобы получить дополнительную информацию, пользователь должен выбрать соответствующую команду контекстного меню. Технически это можно реализовать следующим образом: при выборе соответствующей команды контекстного меню открывается окно с дополнительной информацией. Если объем дополнительной информации, отображаемой пользователю, невелик, можно вывести ее непосредственно в контекстном меню. Это сэкономит несколько секунд времени.

И наконец, не забывайте, что пользователи отнюдь не обязаны заучивать наизусть правила нумерации документов и записей в системе. Для обеспечения уникальности записей вы можете использовать последовательность автоматически генерируемых системой чисел. Но пользователей вряд ли заинтересует такая техническая подробность, и уникальный номер записи, идентифицирующий ее в системе, не нужно выводить на экран пользовательского компьютера. Вообще пользователя не следует перегружать цифрами — если какая-то информация не имеет отношения к предметной области (скажем, не является номером счета, выписываемого покупателю), ее лучше не выводить на экран.

Многие организации разработали список стандартных сокращений для обозначения категорий продуктов, регионов продаж и т. п. Следует ли использовать эти сокращения в разрабатываемой вами системе — однозначного ответа нет. Лично я принимаю решение на основании одного простого правила: если сотрудники часто используют эти сокращения не только в документах, но и в разговорах друг с другом, я использую эти сокращения в системе. Если пользователи думают и говорят «восточный регион», требование вводить сокращение «В/Р» создает дополнительные трудности. Но даже если я использую принятые в организации сокращения, то не ограничиваю множество допустимых значений только ими, а оставляю пользователям возможность ввести в соответствующее поле и значение «восточный регион», и сокращение «В/Р».

Будьте последовательны!

Под термином «последовательный интерфейс» мы будем понимать нечто большее, чем правильное размещение команд меню File и Edit, Последовательным должно быть также и поведение системы, и ее взаимодействие с пользователями. Обратите особое внимание на следую-

шие моменты: как выполняется переход от одной записи к другой, как представлены составные сущности и как пользователи редактируют и добавляют записи в базу данных.

В большинстве систем есть формы, соответствующие основным сущностям этой системы: например, формы *Customers* (Покупатели), *Products* (Продукты) и *Sales Orders* (Счета-фактуры). Вам нужно определить, каким образом пользователи будут выбирать нужную им запись из набора. Я рекомендую разработать единые правила выбора записей и перехода от одной записи к другой, использовать их во всех формах и без крайней необходимости не нарушать.

По умолчанию в связанных формах Access и Visual Basic выбирается первая запись из нижележащего набора. Форма также содержит кнопки, позволяющие перемещаться от одной записи набора к другой. Однако по разным причинам вам может понадобиться отказаться от этих стандартных свойств интерфейса. Например, чтобы в форме отображалась всего одна запись, и были предусмотрены механизмы, позволяющие выбрать записи, которые следует отобразить. Таким механизмом может быть отдельная форма для поиска записи или комбинированное окно, позволяющее выбрать нужную запись. На рис. 12-3 показан пример, заимствованный из стандартной базы данных Access Developer Solutions и иллюстрирующий этот подход. В нем возможность выбрать нужную запись реализована при помощи комбинированного окна Edit Products.

Рис. 12-3. Пользователи могут выбрать запись о клиенте

Итак, ваше решение отказаться от стандартного механизма перемещения между записями вполне оправдано. Но, сделав однажды

выбор, вы должны его придерживаться, и использовать тот же самый механизм во всех формах системы. Согласитесь, неудобно работать с системой, где перемещение между записями в форме Customers (Покупатели) реализовано при помоши стандартных кнопок; в форме Sales Order (Счет-фактура) для выбора нужной записи требуется щелкнуть кнопку Find Order (Найти заказ); а в форме Products (Товары) — данные представлены в табличном виде.

Единственное разумное исключение из этого правила — если в системе несколько различных видов форм. В таком случае вы можете использовать разные механизмы работы с данными в формах, предназначенных для поддержки справочников, и в формах, предназначенных для ввода данных об основных сущностях. Ваш интерфейс будет согласованным для каждой категории форм, имеющихся в системе.

Другая область, где также нужна последовательность — представление составных сущностей. Если в вашей модели некие сущности представлены несколькими таблицами, участвующими в связи «один ко многим» (классический пример — сущность *Sales Order),* то вы должны соблюдать определенные правила относительно того, как эти сущности будут отображаться пользователям. Когда те элементы бланка заказа, которые, будучи напечатанными на бумаге, «превращаются» в строки, представлены в виде табличном виде, а адрес и телефон покупателя в виде списка — это непоследовательно.

К сожалению, последовательный подход в этой области реализовать гораздо сложнее, особенно если нужно работать с несколькими наборами записей одновременно. Форма, в которой одна таблица содержит данные о клиенте, вторая — адреса, а третья — сведения о заказанных товарах, выглядит ужасно. Проявите изобретательность — разместите эти таблицы на отдельных вкладках формы или используйте отдельные всплывающие формы, в которых содержатся таблицы. Можно выбрать два метода отображения данных и придерживаться их максимально последовательно.

И наконец, третья область, где необходима строгая последовательность — это механизм создания и редактирования записей, который должен быть реализован одинаково в масштабах всей системы. Если в одной форме допускается непосредственное редактирование текущей записи, а в другой пользователь должен явно задать режим редактирования (например, щелкнув кнопку Edit или выбрав соответствующую команду меню) — это не самое лучшее решение.

В некоторых случаях совершенно оправдана блокировка сохраненных записей — то есть механизм, запрещающий непосредственное редактирование таких записей. Например, блокировку записей при-

меняют, если нужно, чтобы данные в форме заказа клиента можно было изменять только до тех пор, пока заказанный товар не будет отправлен покупателю. Затем заказ изменяет свой статус и становится архивным документом, не подлежащим изменению; данные, введенные в эту форму, архивируются и хранятся как справочная информация. Некоторые разработчики предусматривают дополнительную проверку, был ли отправлен покупателю данный заказ, когда пользователь хочет редактировать записи в этой форме. Подобная тактика приемлема, только если для всех форм в системе выполняется такая же предварительная проверка.

Я могу предложить лучшее решение: разрешите непосредственное редактирование данных в полях этой формы, но блокируйте запись (то есть сделайте все поля недоступными для редактирования), если пользователь открывает форму уже отправленного клиенту заказа. Этот способ, хотя не столь прост в реализации, весьма эффективен. Кроме того, он позволяет реализовать непосредственное редактирование данных в полях всех остальных форм в системе, где не применяется блокировка архивных записей.

Итоги

В этой главе мы познакомились с основными принципами, лежащими в основе проектирования пользовательского интерфейса. Мы начали обзор методов проектирования с рассказа о моделях пользовательского интерфейса: ментальной модели пользователя (представлений пользователя о системе и о том, как она работает), модели реализации (того, что реально происходит в системе) и декларируемой модели (того, что система сообщает пользователю о происходящем процессе).

Затем были рассмотрены различные уровни подготовки пользователей: начинающий пользователь, опытный пользователь и эксперт. Мы кратко обсудили специфические требования к пользовательскому интерфейсу, предъявляемые каждой из этих групп пользователей и то, каким образом эти требования можно удовлетворить.

Заключительная часть главы была посвящена принципам, лежащим в основе проектирования пользовательского интерфейса: ответственности пользователя за его действия, уменьшение объема информации, которую пользователь должен запоминать; согласованность интерфейса.

В следующей главе мы рассмотрим архитектуру всей системы в целом и обсудим конкретные детали, связанные с проектированием пользовательского интерфейса.

Архитектура пользовательского интерфейса

При проектировании пользовательского интерфейса следует в самом начале принять решение об общей структуре интерфейса системы — другими словами, об *архитектуре пользовательского интерфейса*. В этой главе мы рассмотрим несколько стандартных архитектур, полное описание которых содержится в руководстве «The Windows Interface Guidelines for Software Design» («Руководство по разработке интерфейса для Windows-приложений»). Все эти варианты стандартных архитектур реализованы в распространенных программных продуктах.

Разумеется, вы можете разработать свою собственную архитектуру интерфейса, но я советую воздержаться от такого шага. Если пользовательский интерфейс последователен не только в пределах одной системы, но в масштабах всего программного обеспечения, с которым работают пользователи, это существенно облегчает их работу.

Поддержка рабочих процессов

Выбор архитектуры интерфейса должен быть обусловлен рабочими процессами, которые поддерживает разрабатываемая система, а не структурой данных — вот основной принцип, лежащий в основе структуры интерфейса. Вы должны знать, какие задачи будут решать пользователи системы и какие действия при этом выполнять. Создайте такую структуру системы, чтобы максимально помочь пользователям.

Вот пример одной из наиболее распространенных ошибок разработчиков. Увидев на диаграмме «сущности — связи» сущность *Customers* (Покупатели), они создают форму с тем же названием, предназначенную для ввода и редактирования записей о пользователях.

Затем создается форма *Orders* (Заказы), использующая данные из таблицы *Customers,* доступной из этой формы только для чтения. Чтобы сделать систему «дружественной пользователю», разработчики предоставляют пользователю возможность выбрать фамилию покупателя из списка. Затем пользователь должен ввести информацию в остальные поля формы, основываясь на данных таблицы *Customers.* То что получается, напоминает форму из базы данных *Northwind* (рис. 13-1).

Рис. 13-1. Форма Orders из базы данных Northwind

Эта форма сама по себе не плоха и не хороша, но с точки зрения поддержки рабочих процессов она не оптимальна. Подумайте о гом, что делает пользователь, который вводит информацию в бланк заказа. Он открывает форму *Orders,* только чтобы убедиться, что покупатель, информацию о котором он будет вводить, еще не зарегистрирован в системе. Затем ему нужно закрыть эту форму, чтобы открыть какую-то другую и ввести данные о новом покупателе, и только потом он сможет ввести информацию о заказе. Если форма *Orders* еще не закрыта, пользователю понадобится обновить список *Customers* (Покупатели), нажав клавиши Shift+F9 (или какое-либо другое сочетание), чтобы в этом списке отображались данные о новом покупателе. Неудобно.

Некоторые разработчики решают эту проблему, включив событие (покупатель отсутствует в списке) или создав комбинированное окно, позволяющее ввести имя нового покупателя. При возникновении события *NotInList* пользователю выдается запрос — следует ли ввести информацию о новом пользователе. Если пользователь подтверждает это, открывается форма *Customers.*

Такое решение проблемы более рационально — пользователю приходится выполнять гораздо меньше разнообразных действий, чем в первом случае. Однако оно все еще не оптимально: пользователь

вынужден прерывать выполнение одной операции (ввод данных о заказе) и выполнять другие действия (обновление списка покупателей). Значительно лучше позволить пользователю сначала завершить выполнение одной операции, а потом уже перейти к другой. Например, так: если пользователь введет имя покупателя, отсутствующее в списке, все поля формы остаются незаполненными; если же такое имя уже есть в списке, в полях отображается соответствующая информация, относящаяся к этому покупателю. Особо отмечу: если запись о покупателе отсутствует в базе данных, то не нужно выдавать пользователям соответствующие звуковые сигналы и системные сообщения. После того как пользователь заполнит все поля формы, система просто должна добавить новую запись в базу данных, не требуя от пользователя дополнительных действий,

Если форма *Orders* позволяет ввести всю информацию, которую содержит отдельная запись о покупателе, не нужно заставлять пользователя делать что-то еще, после того как он заполнил бланк заказа. Однако чаще всего, чтобы все поля записи о пользователе в базе данных были заполнены, приходится ввести еще какие-либо данные о покупателе. Можно напомнить пользователю о необходимости ввести эту информацию и предоставить выбор — сделать это сейчас или немного позднее. Но такой запрос должен выдаваться только после того, как пользователь закончит ввод всей информации о заказе — прерывать или отвлекать его, пока он занят другой работой, недопустимо. Конкретное же решение должно быть основано на том, как именно осуществляется ввод данных о заказе — иными словами, зависеть от рабочих процессов.

Например, если данные о заказе вводит сотрудник отдела продаж во время личного общения с покупателем, вполне резонно запросить дополнительную информацию и ввести оставшиеся данные, после того как заказ уже принят. Поля, содержащие дополнительные сведения, целесообразно разместить на дополнительных вкладках или в нижней части формы *Orders,* чтобым сотрудник всегда имел перед глазами список вопросов, которые он должен задать покупателю,

Ну а если пользователь обрабатывает заполненные бланки заказов, присланные по факсу? Или для получения дополнительных сведений о клиенте, сделавшем заказ, ему нужно куда-то звонить или отправлять ответный факс? Тогда напоминание, что в систему введены еще не все данные о покупателе, выдаваемое сразу же после того, как сотрудник заполнит форму заказа, заставит этого сотрудника выполнить лишнее действие — закрыть окно системного сообщения. Такие напоминания должны выдаваться, только когда они действи-

тельно необходимы, и не стоит постоянно сообщать пользователю, что он еще не ввел такие-то данные. Лучше просто предусмотреть специальный системный флаг для тех записей в базе данных, в которых имеются незаполненные поля, чтобы потом, когда пользователи будут располагать необходимой информацией, они смогли легко найти эти записи и ввести недостающие данные.

После того как пользователь ввел информацию о заказе, система может выдать запрос, следует ли открыть форму *Customers*, чтобы ввести оставшиеся сведения о пользователе, или закрыть форму *Orders*. Но подобный запрос уместен, только если эту информацию вводят те же сотрудники, что и данные о заказе, и если поиск и ввод этих данных в большинстве случаев целесообразен после заполнения формы *Orders*.

Однодокументный и многодокументный интерфейс

Существует две разновидности архитектуры пользовательского интерфейса — *однодокументный интерфейс* (single document interface, SDI), когда пользователь работает только с одним окном, содержащим отдельный документ, *и многодокументный интерфейс* (multiple document interface, MDI), когда пользователь может открыть одновременно несколько документов в отдельных окнах главного окна приложения.

На вопрос, какая из этих разновидностей архитектуры удобней для пользователей, не существует однозначного ответа. У каждой из них свои преимущества и недостатки. Выбор архитектуры пользовательского интерфейса должен основываться на рабочих процессах, поддерживаемых системой.

Однодокументная архитектура

Итак, архитектура SDI характеризуется тем, что пользователи работают с одним главным окном приложения. Для вывода вспомогательной информации используются дополнительные диалоговые окна. Архитектуру SDI имеет смысл использовать в системах, поддерживающих одну логическую сущность (которая может быть реализована в виде нескольких таблиц в базе данных). Например, архитектура SDI подходит для простой системы, предназначенной для ввода и редактирования информации о сотрудниках компании.

Архитектура SDI обладает рядом преимуществ. С одним окном пользователям легче работать. Такая архитектура соответствует стандартному подходу к разработке интерфейса, реализованному в мастерах создания пользовательских документов, широко используемых Microsoft.

Системы SDI очень просто реализовать при помощи Microsoft Visual Basic. С помощью Microsoft Access сделать это не удастся, поскольку все формы содержатся в главном окне Access. Однако можно имитировать архитектуру SDI в приложении, созданном при помощи Access, развернув главную форму системы, открываемую при запуске приложения, во весь экран рабочей станции пользователя, и убрав с панели окна кнопки, позволяющие изменять размер формы на экране. Access 2000 также позволяет разместить значок окна этой формы на панели задач. Итак, если использовать некоторые особые приемы, система, созданная с помощью Access, может вести себя как приложение, использующее архитектуру SDI.

Рабочая книга

Рабочая книга — это одна из разновидностей архитектуры SDI, где различные представления данных отображаются не в разных окнах, а на разных вкладках одного окна. Типичный пример приложения с такой архитектурой — Microsoft Excel.

Явное преимущество архитектуры SDI — она предоставляет пользователям вполне надежную рабочую среду, не ограничивая их рамками одной формы. Но реализовать такую систему, обеспечив приемлемое время отклика, часто не так просто. И все же рабочая книга — очень удобный способ отображать различные представления объекта или набора тесно связанных между собой объектов, если не нужно сравнивать эти объекты между собой.

Например, можно использовать рабочую книгу для представления различной информации о статистике продаж: разместить сводный отчет по продажам за месяц на одном листе, диаграмму распределения продаж по категориям продуктов — на другом, и диаграмму, показывающую динамику объемов продаж по годам, — на третьем. Эти три вида статистических данных тесно связаны между собой, и вполне вероятно, что пользователи будут обращаться к ним при составлении различных отчетов, поэтому их желательно сгруппировать. Однако вряд ли кому-то потребуется просматривать все три вида данных одновременно — ведь их нельзя непосредственно сравнивать.

Страницы рабочей книги следуют друг за другом в определенном порядке, и это может сыграть определенную положительную роль при разработке пользовательского интерфейса. Например, если создаваемый интерфейс поддерживает рабочий процесс, состоящий из отдельных задач, которые часто, но не всегда, приходится выполнять в определенном порядке, можно организовать структуру интерфейса в виде рабочей книги, выделив для каждой задачи отдельный лист. Та-

кой подход позволит выполнять действия в определенном порядке, в то же время не обязывая пользователя строго соблюдать этот порядок.

С другой стороны, рабочая книга не самое лучшее решение, если нужно разделять рабочие процессы, представляющие собой совершенно разные виды деятельности, или выводить на экран пользовательского компьютера данные, которые впоследствии будут сравниваться между собой. Поскольку невозможно открыть несколько страниц рабочей книги одновременно, пользователю придется вспоминать столбцы цифр, размещенные на одном листе, чтобы сравнить их с цифрами, размещенными на другом.

Интерфейс, использующий стиль приложения Microsoft Outlook

Еще одна разновидность архитектуры пользовательского интерфейса — это особый стиль, который я называю «интерфейс, использующий стиль Outlook», в честь первого приложения, где я увидела подобную архитектуру. Ее характерная особенность — окно приложения делится на две области, на одной из которых в определенном порядке расположены значки, а на другой — документы (рис. 13-2),

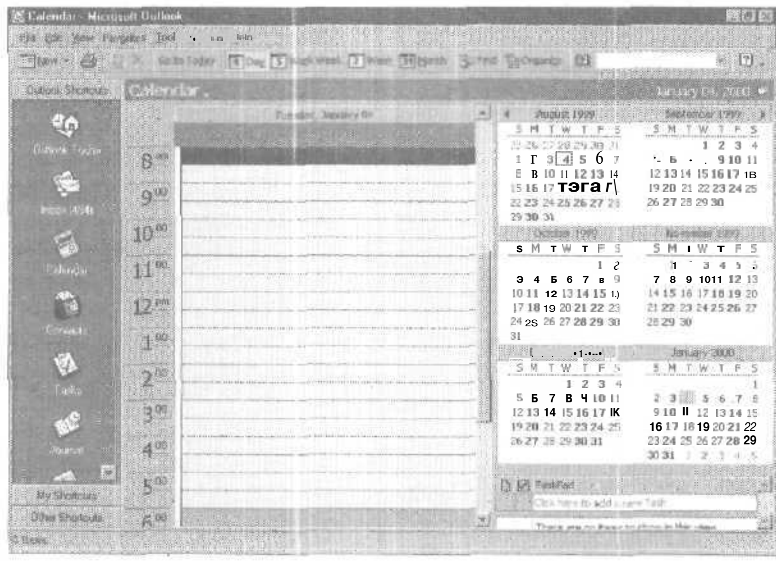


***Рис. 13-2.** Интерфейс, использующий **стиль** Outlook, **разделяет окно приложения на две рабочие** области*

Интерфейс, реализованный в стиле Microsoft Outlook, удобен для приложений, поддерживающих несколько рабочих процессов. Левая

область окна, где размещены значки, обеспечивает прямой доступ пользователей к соответствующим рабочим областям. Эти значки можно сгруппировать по различным функциям, разместив в рабочей области несколько панелей, на каждой из которых расположена отдельная группа значков.

К сожалению, ни Access, ни Visual Basic не предоставляют встроенных элементов управления для реализации такого интерфейса, хотя именно эта архитектура использована в Microsoft Access 2000 при реализации главного диалогового окна Database, предоставляющего пользователю доступ ко всем объектам баз данных (рис. 13-3). Однако многие компании, занимающиеся разработкой коммерческого программного обеспечения, предоставляют широкий набор компонентов ActiveX, использующих этот тип архитектуры. Эти компоненты легко интегрируются в среду разрабатываемого приложения.

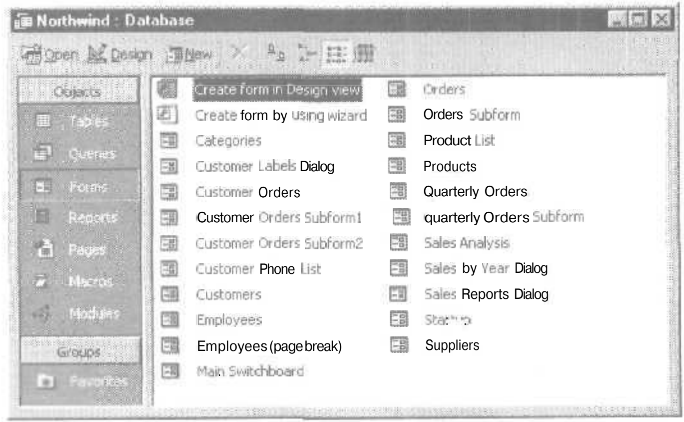


*Рис. **13-3.** **В Microsoft** Access 2000 **главное** диалоговое окно **Database** использует стиль **Outlook***

Если вы используете этот стиль интерфейса, разумно дать пользователям возможность скрыть ту область окна, в которой размещаются значки, как это сделано в Outlook. Эта область предоставляет удобные средства навигации, однако после того как нужный документ найден и открыт, инструменты навигации не нужны и только занимают место на экране.

Многодокументная архитектура

Большинство систем баз данных используют различные разновидности многодокументной архитектуры интерфейса. MDI-архитектура

позволяет открывать несколько дочерних окон из главного окна приложения.

Каждое из дочерних окон содержит различные виды данных: например, форма, где собрана информация о покупателях или заказах фирмы. Кроме того, дочерние окна могут содержать различные представления одних и тех же данных: например, форму, где представлены сведения о покупателе и отчет о покупках, сделанных этим покупателем. Или же различные наборы сходных данных: форму, которая в одном случае содержит информацию о первом покупателе, а в другом случае — о втором.

Как и для SDI-архитектуры, существует несколько вариантов структуры MDI-приложений, каждый из которых удовлетворяет определенным требованиям. Далее мы рассмотрим основные из этих вариантов. Конечно, весь спектр возможных вариантов отнюдь не исчерпывается рассматриваемыми конфигурациями, и сами эти конфигурации не обязательно должны быть взаимоисключающими.

«Классическая» архитектура MDI

Классическая структура MDI-приложения — это основное окно, из которого открываются однотипные или разные дочерние окна. Стандартный пример такой архитектуры — Microsoft Word, *в* окне этого приложения может быть открыто одновременно несколько документов. Число документов, открытых в дочернем окне, ограничивается только объемом оперативной памяти компьютера.

Приложения с MDI-архитектурой широко применяются, когда нужно сравнивать несколько различных фрагментов данных или данные, представленные в различных форматах. Но чтобы научиться работать с MDI-приложениями, пользователям, как правило, требуется больше времени, чем для освоения SDI-приложений: выбрав в меню File команду New, они видят перед собой пустое окно, что часто сбивает их с толку. Кроме того, начинающие пользователи часто пугаются в многочисленных открытых окнах.

В Word имеется специальная команда, выполняемая из командной строки — /n. Она позволяет включить или отключить режим открытия нового окна документа при запуске приложения. Вы можете предусмотреть аналогичную команду в своем MDI-приложении — пусть, например, пользователи с ее помощью открывают форму *Orders* (Заказы) и вводят данные.

Если вы решили реализовать такую команду в разрабатываемой системе, не забудьте предусмотреть возможность для тех, кто ею не пользуется, отключать ее. Кроме того, на этапе разработки позаботьтесь, чтобы неадекватные сообщения об ошибках не выдавались пользовате-

лям, просто закрывающим диалоговое окно или добавляющим пустые записи в базу данных.

Основная проблема, связанная с MDI-архитектурой интерфейса — сложность и противоречивость модели ее внутренней структуры. С точки зрения графической среды, главное окно приложения содержит все дочерние окна, открываемые из него; но далеко не всегда объекты, отображаемые в дочерних окнах, принадлежат этому же приложению. Так, например, документы Microsoft Word — это отдельные объекты файловой системы. Зачастую подобная непоследовательность серьезно затрудняет реализацию системы, а пользователей приводит в замешательство. В системах, использующих базы данных, клиентские приложения, как правило, изолированы от сложностей, связанных с архитектурой СУБД. Но даже в самых простых приложениях, работающих с базами данных, далеко не всегда удается полностью решить все эти проблемы.

Вот пример такой системы: пользователь открыл несколько окон и переходит от одного к другому, причем в каждом из этих окон выполняет какие-то изменения, не сохраняя их (рис. 13-4).

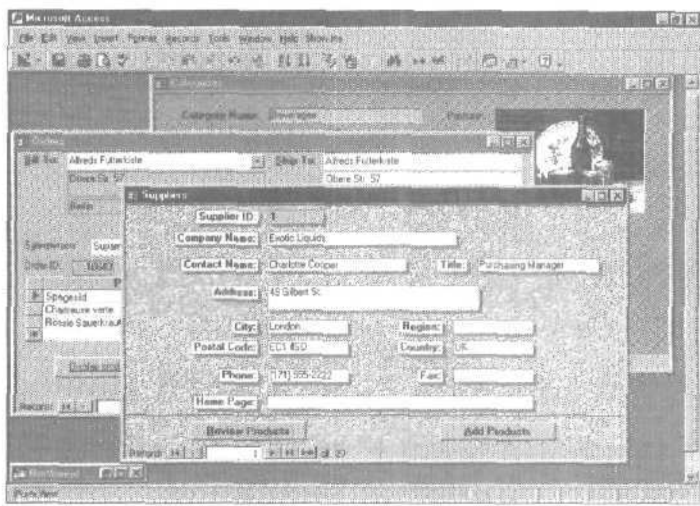


Рис. ***13-4. Модель внутренней структуры MDI-приложения: главное и дочерние окна***

Что произойдет, когда пользователь выберет в меню File (Файл) команду Save (Сохранить)? Будут сохранены только изменения, сделанные в окне *Suppliers* (Поставщики). Для вас это очевидно, поскольку вы знаете, что команды меню выполняются только для окна, кото-

рое активно в данный момент. Но известно ли это всем пользователям? Например, пользователи не работавшие ранее с Access, могут вполне логично предположить, что если команда «сохранить» выполняется в приложении *Northwind*, то будут сохранены все изменения в базе данных *Northwind*, независимо от того, в каком окне они были выполнены.

Простой выход из этой ситуации предлагает руководство «The Windows Interface Guidelines for Software Design». В меню File можно ввести дополнительную команду Save All (Сохранить все) для сохранения всех изменений, выполненных во всех открытых окнах приложения. Разумеется, это вполне разумное решение, но и оно — всего лишь компромисс. От пользователя такой системы все равно требуется четко понимать разницу между MDI-приложением и объектами, над которыми оно выполняет различные действия.

Это один из нюансов модели реализации, а не ментальной модели пользователя. И он, как правило, вызывает у пользователей большие затруднения. Даже весьма искушенные могут перепутать, забыть или просто не знать, где что хранится в этом приложении. Лично я до сих пор путаю, какие параметры форматирования хранится в списке стилей в Word, а какие — в самом документе, хотя уже не один год работаю с этим приложением.

Но несмотря на все это, «классические» MDI-приложения широко распространены. Они были и остаются наиболее удобными, если необходимо открывать одновременно несколько окон.

Диалоговая панель управления

Интерфейс многих приложений разрабатывается в стиле диалоговой панели управления. При запуске такого приложения открывается основная форма (рис. 13-5). Большинство ее кнопок предназначены для вызова различных форм или создания отчетов.

Эта структура интерфейса реализована в мастере создания новых форм Access. Очевидно, она знакома многим разработчикам, создававшим базы данных с помощью этого приложения. Такой интерфейс несложно реализовать и средствами Visual Basic.

Должна признаться, что я несколько предубеждена против данного решения. Мне все это напоминает неуклюжие программы, написанные для DOS: возрождается старая концепция структуры команд меню *Поиск записи/Редактирование записи/Печать записи*, которая только утомляет и раздражает пользователей.

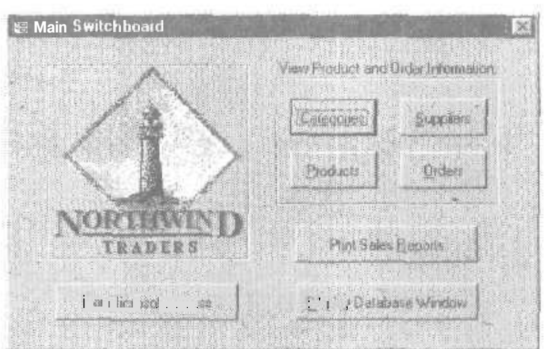


Рис. 13-5. *При запуске приложения на экране появляется диалоговая панель управления*

Использование интерфейса в виде диалоговой панели управления, однако, вполне оправдано там, где одно приложение поддерживает несколько различных рабочих процессов, и вы не хотите тратить время и силы на разработку интерфейса в стиле Outlook. Оно также полезно, если нужна возможность открывать одновременно несколько окон. Диалоговая панель верхнего уровня, на которой размешены кнопки, соответствующие рабочим процессам, позволяет пользователям быстро освоиться с приложением.

При разработке интерфейса в виде диалоговой панели управления, как и при использовании любой другой архитектуры, важно организовать структуру интерфейса таким образом, чтобы она была ориентирована на рабочие процессы, поддерживаемые системой, а не на структуру данных. Кнопки на диалоговой панели управления должны соответствовать различным категориям действий, выполняемым пользователями; вовсе не обязательно размешать на ней несколько десятков кнопок для каждой формы и каждого отчета в системе.

Проект

Я рассматриваю организацию интерфейса в виде проекта как одну из разновидностей диалоговой панели управления. В этой архитектуре нет главного окна, из которого открываются дочерние, Окно проекта позволяет открывать другие окна непосредственно на рабочем столе. SDI-интерфейс Visual Basic — типичный пример этой разновидности интерфейса (рис. 13-6).

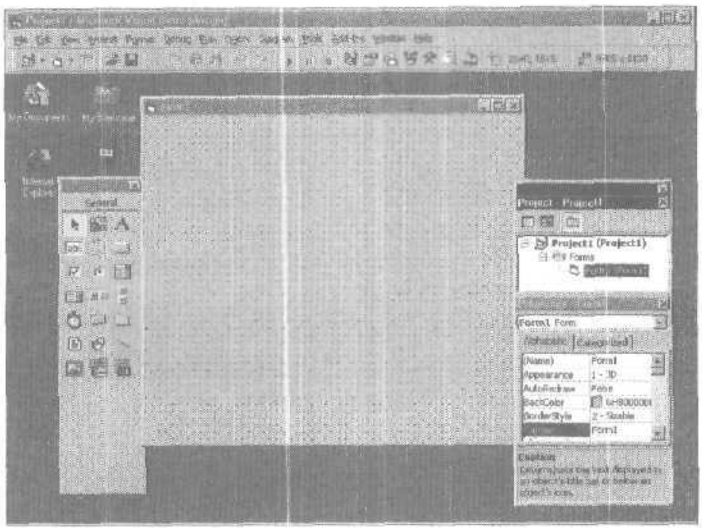


Рис. 13-6. Visual Basic в режиме SDI — это интерфейс, организованный в виде проекта

Открытые окна проекта отображаются на панели задач. Пользователи могут переключаться с одного окна на другое и управлять каждым окном в отдельности. Кроме того, механизмы управления окнами существуют и в окне проекта. Если окно проекта свернуто или закрыто, все прочие окна также будут свернуты или закрыты.

Хотя интерфейс, организованный в виде проекта, лишен недостатков классической модели MDI, в нем существует другое противоречие — связь между окном проекта и относительно независимыми окнами, управляемыми из него.

Представьте себе такую ситуацию: пользователь открывает новое окно, дважды щелкнув мышью в окне проекта; затем, экономя место на экране, сворачивает окно проекта, поскольку оно ему уже не нужно. Но при этом открытое окно, с которым он собирался работать, также сворачивается. Конечно, его всегда можно развернуть из панели задач, и все же это не совсем удобно.

Если для реализации пользовательского приложения вы выбрали архитектуру проекта, рекомендую использовать модель окна *Database* в Access. Она сочетает легкость навигации, которую предоставляет пользователям окно проекта, и отсутствие неудобств, возникающих вследствие связей между основным окном приложения и окном, с которым работает пользователь. Ведь окно *Database* — часть приложения с классической MDI-архитектурой.

Мастер

Мастер — это еше одна разновидность архитектуры интерфейса, широко применяемая в приложениях, работающих с базами данных. Как правило, мастер состоит из последовательности диалоговых окон, открываемых в определенном порядке (рис. 13-7).

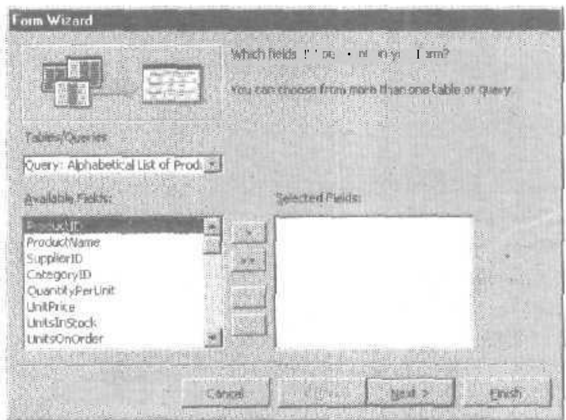


Рис. 13~ 7. Мастеры широко используются в Microsoft Access 2000

Чаше всего мастер используют, чтобы облегчить пользователям выполнение повторяющихся задач — например, при установке и конфигурировании новых устройств в системе. Но, конечно, их можно применять и для поддержки рабочих процессов в приложениях, работающих с базами данных.

Мастеры также полезны для поддержки сложных рабочих процессов. Если рабочий процесс состоит из нескольких различных задач, выполняемых в определенном порядке, то структуру пользовательского интерфейса можно организовать в виде мастера. Мастеры незаменимы, если рабочий процесс состоит из множества условных задач, то есть задач, выполнение которых зависит от выполнения некоторого условия (например: «Если выполняется условие А, выполнить задачу 3, а затем задачу 4; если выполняется условие В, перейти к выполнению задачи 6»).

Мастеры могут поддерживать сложные рабочие процессы, только если отдельные задачи или последовательности действий выполняются в том порядке, в котором их предлагает выполнять мастер. Подобные ситуации на практике не столь часты, как может показаться. Если же порядок выполнения задач не имеет значения, организация пользовательского интерфейса в виде мастера — не самое лучшее ре-

шение. По крайней мере, предусмотрите альтернативный способ выполнения этой задачи.

Если мастер используется в приложении, работающем с базой данных, обратите особое внимание на то, как будут сохраняться введенные пользователями данные. В обычной модели при вводе данных пользователь в любой момент может отменить все выполненные изменения, например, щелкнув кнопку Cancel: введенная информация не будет сохранена, и система вернется к тому же состоянию, в котором находилась перед тем, как пользователь начал вводить данные.

Вы можете спроектировать пользовательское приложение так. чтобы пользователь сохранял введенную информацию при каждом переходе к следующей странице, или же подтверждал сохранение уже введенных данных по завершении работы с мастером или отказе ог изменений, выполненных в одном из его окон. Я не рекомендую ни один из этих способов как предпочтительный, но если пользователи вводят значительный объем данных при помощи мастера, следует предусмотреть удобные механизмы подтверждения ввода.

Не будет лишней возможность приостановить работу с мастером, чтобы продолжить ее позднее. Это практичное решение, однако реализовать его довольно сложно, поскольку в этом случае придется предусмотреть дополнительные механизмы: для временного хранения данных, определения шага, на котором пользователь прервал работу с мастером и возможности продолжать работу с того шага, на котором она была прервана.

Итоги

В этой главе мы познакомились с различными вариантами структуры форм. На самом высоком уровне архитектура интерфейса системы разделяется на два класса: SDI-системы, в которых пользователи работают с одним окном, и MDI-системы, позволяющие открывать несколько окон.

Существует две разновидности SDI-систем: рабочая книга и интерфейс, использующий стиль Microsoft Outlook. Каждая имеет свои преимущества и недостатки при использовании в конкретных условиях. MDI-приложения более разнообразны: это и классические MDI-системы, и диалоговые панели управления, реализованные в мастере баз данных Access, и интерфейс в виде проекта, примером которого служит Visual Basic. И наконец, интерфейс, организованный в виде мастера чаще всего используется, чтобы облегчить пользователям выполнение повторяющихся задач.

В следующей главе мы рассмотрим внутреннюю структуру форм, определяемую структурой сущностей в модели данных.

Связь между сущностями и Формами системы

ГЛАВА

В предыдущих главах этого раздела мы концентрировали свое внимание в основном на рабочих процессах, а модель данных оставалась за рамками обсуждения. В этой главе мы рассмотрим взаимосвязь между внутренней структурой отдельных форм и способами представления сущностей в модели данных системы. Решение, какие именно данные отображать в каждой отдельной форме, зависит от структуры рабочих процессов. Но после того как это решение принято, внутренняя структура формы и выбор элементов управления, присутствующих в ней, определяются структурами данных, представленными в этой форме.

Прежде всего, следует определиться, каким образом отразить в форме модель «сущности — связи». Будет ли в ней представлена одна сущность; или две, между которыми существует связь «один к одному»; или две, между которыми существует связь «один ко многим»; или три и более сущностей? Как связана композиция формы с моделью «сущности — связи»? Ведь между структурами данных и внутренней структурой разрабатываемой формы существуют определенные зависимости. Эти зависимости, как и различные варианты архитектуры интерфейса, о которых шла речь в главе 13, отнюдь не представляют собой свод жестко определенных правил — это всего лишь общие принципы композиции форм пользовательского интерфейса, Я изложу их, основываясь на типичных случаях и наиболее распространенных вариантах структур данных.

Простые сущности

Сначала рассмотрим ситуацию, когда в форме должна быть представлена одна простая сущность, то есть сущность, которой соответствует единственная таблица базы данных. Если эта сущность связана с другими сущностями, то она либо находится на стороне «многие» этих связей, либо остальные участники связей никак не представлены в создаваемой форме.

Сущность *Customers* (Покупатели), показанная на рис. 14-1, находится на стороне «многие» всех изображенных на диаграмме связей, за исключением связи между ней и сущностью *Orders* (Заказы). Скорее всего, где-то в пользовательском приложении существует форма, предназначенная для ввода и обновления информации о покупателях. (Такая форма существует, даже если все данные о покупателях, хранимые в системе, были изначально получены из других источников или при вводе информации о других сущностях — например, во время заполнения формы при приеме заказа от покупателя). Возможно, вы не сочтете нужным включать сущность *Orders* в форму *Customers.* Ведь на логическом уровне это абсолютно разные сущности, и для ввода данных, представленных в модели сущностью *Orders,* в системе, скорее всего, будет отдельная форма.

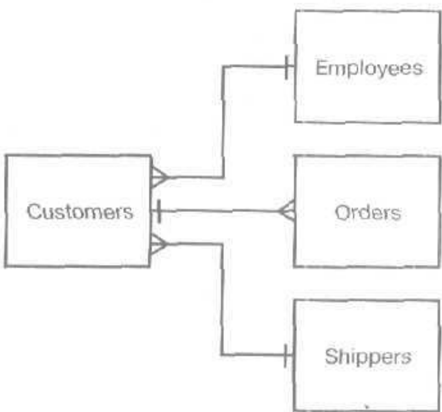


Рис. 14-1. Фрагмент диаграммы «сущности — связи»

Итак, сущность *Customers* (Покупатели) находится на стороне «многие» всех связей с сущностями, представленными в форме, предназначенной для ввода и обновления информации о покупателях, Поэтому в данном случае ее можно рассматривать как простую сущность. Это существенно облегчает процесс проектирования такой

формы: не нужно использовать подчиненные формы и представления данных в виде *формируемых таблиц* (grids). Следует всего лишь выбрать элементы управления, наиболее подходящие для каждого из полей таблицы *Customers* (о том, как это сделать, рассказывается в следующей главе) и разместить их на проектируемой форме.

Разумеется, размещать элементы управления следует не как попало, а по возможности, соблюдая все правила, перечисленные в руководстве «The Windows Interface Guidelines for Software Design» («Руководство по разработке интерфейса для Windows-приложений»). Между внешними краями поля формы и элементами управления необходима граница шириной в 7 единиц координатной сетки; расстояние между двумя соседними элементами управления должно составлять не менее 4 единиц координатной сетки; самые главные элементы управления следует поместить в левом верхнем углу формы. В этом нет ничего сложного, если размеры проектируемой формы не слишком жестко ограничены и в ней не очень много отдельных элементов управления (в противном случае придется поломать голову, чтобы уместить все эти кнопки, поля и списки на небольшом пространстве).

Однако случаи, когда в одной форме нужно разместить множество различных элементов управления, не так уж редки. В Microsoft Access и Microsoft Visual Basic число элементов управления, которые можно разместить в одной форме, ограничено. Для формы или отчета Microsoft Access оно не может быть больше 754, причем элементы, включенные в форму и впоследствии удаленные, также входят в это число. Для формы Microsoft Visual Basic максимальное число элементов управления — 254 (при этом двойные «стрелки», позволяющие увеличивать или уменьшать числовые значения в поле, рассматриваются как один элемент управления).

На практике же, однако, редко встречаются формы, которые содержат более 25–30 отдельных элементов управления или групп. (Заметьте: я сказала «отдельных элементов или групп», поскольку группа, состоящая из трех переключателей, очевидно, воспринимается как отдельный логический элемент). Итак, что же делать, если в вашей модели данных есть простая сущность с 75 атрибутами?

Советую отображать данные по частям, а не все одновременно. Самый простой выход — сгруппировать поля по какому-либо признаку или разбить их на отдельные категории. Я сначала выбираю атрибуты, позволяющие однозначно определить эту сущность. Но учтите: список таких сущностей отнюдь не ограничивается атрибутами, входящими в ключ-кандидат, а включает также описательные атрибуты, позволяющие пользователям проверить, с каким экземпляром сущности они работают.

Для сущности *Customers* (Покупатели) в такую группу атрибутов, по которым можно однозначно определить сущность, войдут атрибуты, представляющие адрес покупателя, его имя, а также, возможно, атрибут, представляющий сотрудника отдела продаж, обслуживающего этого покупателя. Для сущности *Products* (Продукты) такую группу составят атрибуты *ProductCategory*(Категория продукта), *Name* (Наименование) и *Description* (Описание). Эту группу следует поместить в верхней части формы: она должна отображаться всегда, когда форма открыта.

Оставшиеся атрибуты можно разделить на отдельные группы, все элементы которых логически связаны между собой и должны отображаться одновременно. Для сущности *Customers* можно сгруппировать атрибуты так; атрибуты, относящиеся к условиям заключения контракта на поставку продукции (размеры скидок, сроки и способ оплаты) объединить в одну группу; а дополнительную информацию о частном лице или фирме, заказавшей товар (фамилия и имя лица, сделавшего заказ, фамилия и имя менеджера по продажам и т. д.) — в другую. Для сущности *Products* — объединить атрибуты, относящиеся к технической спецификации продукции, в одну группу, а сведения об упаковке и расфасовке товара — в другую.

Разделив атрибуты на несколько групп, определитесь, как эти группы будут представлены в разрабатываемой форме. Можно разместить несколько вкладок в главной форме — по одной на каждую группу атрибутов. Лично я предпочитаю это решение всем остальным: такая структура главной формы очень удобна и наглядна — пользователи сразу видят, какая дополнительная информация доступна из главной формы. Но этот подход целесообразно использовать, только если у вас не больше пяти-шести отдельных групп атрибутов.

Если отдельных групп атрибутов множество, разместите все группы или некоторые из них на *вспомогательных формах* (subsidiary forms). Вспомогательные формы можно использовать, и если атрибуты в отдельных группах не умещаются на одной вкладке. Поместите на основной форме кнопки, щелкая которые, пользователи будут открывать вспомогательные формы. Такая композиция главной формы напоминает диалоговую панель управления. Однако и здесь не следует выходить за рамки разумных пределов. Расположение на главной панели великого множества кнопок, открывающих вспомогательные формы, ничуть не лучше, чем главная форма с 33 вкладками. Лучше вызывать вспомогательные формы из меню.

Если вы решили использовать вспомогательные формы, открываемые из главных, позаботьтесь о поддержке контекста. Пользователи

должны знать, к какой основной форме или ее фрагменту относится открытая вспомогательная форма. Чтобы напоминать им об этом, включите во вспомогательную форму некоторые элементы основного диалогового окна, из которого была открыта эта форма. Не обязательно помещать во вспомогательную форму всю группу элементов, к которой она относится — достаточно лишь информации, необходимой для привязки вспомогательной формы к основной.

Существует и другое решение: сделать вспомогательные формы модальными. Пожалуй, это один из тех редких случаев, когда модальность вполне уместна, поскольку помогает поддерживать неразрывность контекста пользовательского интерфейса. Но не забывайте, что модальные формы сильно стесняют пользователей, ограничивая свободу их действий. Поэтому я советую использовать эти формы только там, где нет никакого иного выхода, например, для поддержки контекста нельзя иначе добавить во вспомогательную форму несколько элементов из главной.

Если вы не хотите использовать множество вспомогательных окон, разместите всю дополнительную информацию в нескольких *подчиненных формах* (subforms) главной формы (в Visual Basic это специальные элементы управления — фреймы). Мне не очень нравится такой подход, поскольку так довольно сложно поддерживать пользовательский контекст. Необходимо предусмотреть механизм, позволяющий пользователю в любой момент определить, какую именно из подчиненных форм он видит на экране своего монитора. Если отображение форм на экране управляется при помоши меню, то не так уж просто подсказать пользователю, что для доступа к дополнительным данным из этой формы он должен воспользоваться командами меню. Кнопки или переключатели тоже не спасут — пользователи могут не понять, почему после щелчка этой кнопки экран принимает совершенно другой вид. Я все-таки считаю оптимальным решением форму с расположенными на ней вкладками и рекомендую применять этот известный и испытанный метод как можно чаще. Зачем прибегать к нестандартным решениям там, где можно обойтись привычными средствами?

Связи «один к одному»

В большинстве случаев процесс разработки формы, представляющей две сущности, между которыми существует связь «один к одному», принципиально ничем не отличается от разработки формы, представляющей простую сущность. Можно написать запрос, объединяющий поля двух таблиц, и затем работать с полученным результатом, как с одной простой сущностью. Если этот результат содержит множество

атрибутов, которые сложно отобразить в одной форме, используйте те же методы, что и при проектировании формы, представляющей одну простую сущность.

Когда основная сущность является участником нескольких связей «один к одному», наилучший вариант композиции формы определяется самой природой связей. Если связи между сущностями могут существовать все одновременно, рассматривайте их как единый набор записей, используя одну или несколько форм для отображения данных пользователю. Этот случай опять-таки принципиально ничем не отличается от проектирования формы, представляющей одну простую сущность.

Однако во многих случаях связи между сущностями являются взаимоисключающими: например, связь между сущностями *Product* (Продукт), *Beverage* (Напиток) и *Cheese* (Сыр). Каждый продукт может относиться к одной из категорий, но не к обеим одновременно — это может быть либо напиток, либо сорт сыра. В таких ситуациях используйте подчиненные формы или фреймы — разумеется, если вы не слишком стеснены ограничениями на размеры основной формы. Здесь вам не нужно заботиться, чтобы пользователь понял, что в данном контексте есть данные, непосредственно связанные с основной формой — таких данных просто не существует. Более того, вкладки, размещенные на основной форме, только запутают пользователя.

Если же вам все-таки не удается разместить всю необходимую информацию в одной форме, примените метод, который мы уже обсуждали, когда говорили о моделировании простых сущностей. Разбейте все элементы формы на группы и расположите эти группы на вкладках основной формы или во вспомогательных формах. Разместив элементы управления, представляющие атрибуты подчиненной сущности, на вкладке формы, придумайте общее наименование для этих атрибутов и используйте его в качестве заголовка вкладки. Если заголовок вкладки (или название кнопки) будет меняться каждый раз по мере выборки новых записей из таблицы, вы рискуете окончательно запутать пользователя.

По той же самой причине я стараюсь не размещать атрибуты подчиненной сущности на первой вкладке в форме, если пользователи могут перебирать записи первичного набора, переходя от одной записи к другой. Поскольку элементы управления для всех подчиненных сущностей различны, вид отображаемой на экране формы будет меняться всякий раз, когда выбрана следующая запись. Если на первой вкладке нет атрибутов подчиненной сущности, внешний вид фор-

мы не меняется, и это позволит несколько увеличить производительность, так как отпадет необходимость частого обновления экрана.

Связи «один ко многим»

Во многих формах представлены сущности, связанные отношениями «один ко многим». При компоновке таких форм, как правило, не возникает особых трудностей, поскольку существует правило: в форме должны быть представлены связи «один ко многим», а не «многие к одному».

При моделировании сложных связей гораздо проще рассматривать отдельные записи в таблицах, а не экземпляры сущностей. В этом случае форму следует проектировать так, чтобы записи, участвующие в связи со стороны «один», определяли, какие из записей, участвующих в связи со стороны «многие», будут отображаться пользователю.

Ни в коем случае нельзя допускать обратной ситуации: когда запись, участвующая в связи со стороны «многие», является определяющей — это лишь приведет к ненужному усложнению или противоречиям в логике работы системы и запутает пользователя.

Итак, вы убедились, что запись, участвующая в связи со стороны «один», управляет отображением данных в форме. Теперь решите, как именно будут отображаться пользователю записи, участвующие в связи со стороны «многие». У вас два варианта для выбора: выводить все записи сразу или отображать их по одной. При этом многое зависит от объема информации, выводимой пользователю: для отображения в данной форме из каждой записи могут выбираться или несколько полей, или все.

В первом случае вы можете спроектировать форму так, чтобы в ней отображались сразу все записи. На рис. 14-2 для отображения записей, участвующих в связи со стороны «многие», используется подчиненная форма, размещенная в окне с непрерывным режимом просмотра. В подчиненной форме отображаются четыре поля записей, участвующих в связи со стороны «многие». Выборка данных выполняется из базы данных *Northwind.*

Полоса прокрутки справа на подчиненной форме означает, что не все записи, выбранные из базы данных, помешаются в окне просмотра. Однако с точки зрения разработчика это такой режим отображения, при котором в форме отображаются сразу все записи. В следующей главе мы рассмотрим несколько других способов отображать в форме множество записей.

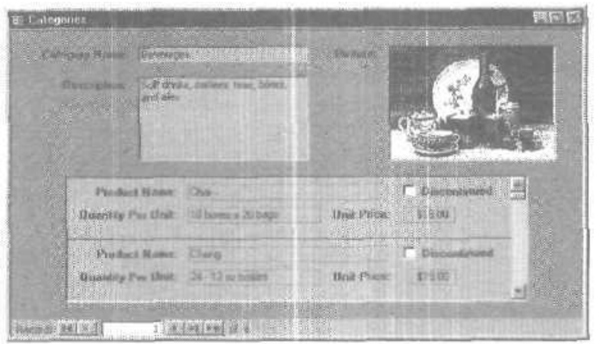


Рис. 14-2. Записи, участвующие в связи со стороны «многие», выводятся одновременно

Я предпочитаю режим, при котором в форме отображаются сразу все записи, поскольку он лучше обеспечивает поддержку контекста. Если вас сильно стесняют ограничения на размеры формы, используйте вспомогательные формы для отображения дополнительной информации, предусмотрев специальное средство вызова вспомогательной формы из главной.

Но такой способ не всегда подходит — может понадобиться выводить записи, участвующие в связи со стороны «многие», по одной (рис. 14-3). В таком случае придется решить две проблемы: дать пользователю понять, что набор не ограничивается одной записью, отображаемой в форме, и обеспечить механизм перехода от одной записи к другой.

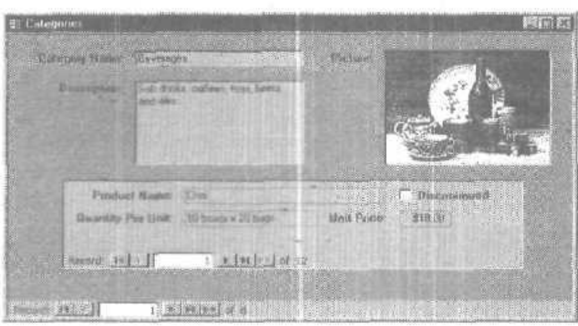


Рис. 14-3. Записи, участвующие в связи со стороны «многие», выводятся по одной

Кнопки в нижней части окна подчиненной формы, позволяющие пользователю переходить к предыдущей или следующей записи, а так-

же перемешаться в начало и конец набора записей — это стандартный механизм навигации Microsoft Access. В данном случае такой подход имеет очевидный недостаток: два совершенно одинаковых элемента управления расположены рядом, причем только места их размещения позволяют догадаться, что один относится к подчиненной форме и обеспечивает перемещение между записями подчиненного набора, а другой — к основной форме и управляет перемещением между записями основного набора. Такое решение, конечно, не является грубой ошибкой, но и изящным его назвать нельзя.

Иногда механизм навигации можно убрать из основной формы, заменив его другими элементами управления. Но при этом пользовательский интерфейс должен быть тщательно продуман. Например, в форме на рис. 14-3 можно заменить механизм навигации, позволяющий перемещаться между записями основного набора, эффективным средством поиска нужных категорий продуктов и выборки соответствующих записей. Вряд ли пользователям потребуется просматривать одну за другой все записи основной формы, содержащие служебную информацию. Зато удобное средство выборки записей по заданным параметрам наверняка окажется весьма полезным.

Конечно, существует и иной выход: заменить стандартный механизм навигации другим средством, обеспечивающим перемещение между записями. Это могут быть специальные кнопки, полоса прокрутки или другой нестандартный механизм, разработанный лично вами. Однако ради сохранения согласованности интерфейса следует по возможности избегать использовать различные способы навигации. Если же это невозможно, выберите парадигму пользовательского интерфейса и придерживайтесь ее.

Например, определите следующее правило: для перемещения между записями, участвующими в связи со стороны «один», будут использоваться кнопки, похожие на кнопки Forward и Back панели инструментов Microsoft Internet Explorer, а для перемещения между записями, участвующими в связи со стороны «многие», — стандартный механизм навигации Microsoft Access. Придерживаясь этого правила во всех формах, имеющихся в системе, вы обеспечите согласованность интерфейса. В этом случае во всех формах, представляющих простые сущности, и следовательно, не участвующих в связях со стороны «многие», для перемещения между записями следует использовать кнопки Forward и Back.

Иногда форма содержит данные из первичной таблицы, участвующей в нескольких связях «один ко многим», и нужно в этой же форме отображать записи из нескольких таблиц, участвующих в связи со стороны «многие». Советую, по возможности, избегать подобных ситуа-

ций. Разместить в одной форме данные из нескольких таблиц, участвующих в связях со стороны «многие», не так просто — трудности будут и на этапе разработки формы, и на этапе реализации. Но если у вас все же нет иного выхода, разместите данные из разных таблиц, участвующих в связях со стороны «многие», на отдельных вкладках основной формы. Такое решение обеспечит простоту и ясность контекста и наилучшую производительность, поскольку данные, размещенные на отдельной вкладке, загружаются, только когда пользователь открывает эту вкладку.

Позаботьтесь особо, чтобы несколько элементов, размещенных в одной форме и содержащих множество записей, не выводились на экран одновременно. Во-первых, такая форма смотрится не слишком хорошо, во-вторых, это может сильно снизить производительность системы. Наконец, несколько окон, списков и тому подобных элементов, содержащих множество записей, запутывают пользователей. Легко ошибиться, предположив, что между записями, участвующими в связи со стороны «многие», существуют дополнительные связи, хотя на самом деле это не так. На рис. 14-4 показана форма с двумя списками. При этом совершенно непонятно, принадлежат ли данные телефонные номера компании, название которой указано в поле *Company Name*, или это номера телефонов сотрудников, чьи фамилии перечислены в поле *Contacts*.

***Рис. 14-4.* Плохо организованная форма затрудняет восприятие пользователя**

Иерархические структуры

Теоретически, любое отношение «один ко многим» представляет собой иерархическую структуру, но обычно этот термин используют для характеристики связей, в которых участвуют три и более наборов данных, Каждый из элементов, расположенных на более высоком уровне иерархии, участвует в связи «один ко многим» с элементами более низкого уровня. Иерархические структуры часто применяют в моделях данных, но отображать их в пользовательских формах приходится значительно реже (рис. 14-5).

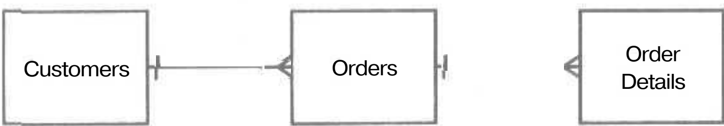


Рис. 14-5. Отношение между сущностями Customers (Покупатели), Orders (Заказы) и Order Details (Информация о заказе) представляет трехуровневую иерархическую структуру

Если в системе используется такая структура, то в большинстве приложений будет следующая архитектура: в форме *Customers* (Покупатели) данные о заказе представлены в виде сводной таблицы (если в этой форме вообще есть данные о заказе). Данные, относящиеся к сущностям *Orders* и *Order Details,* представлены в отдельной форме *Orders* (Заказы). Форма *Orders* ссылается на сущность *Customers,* однако в этой форме представлена только связь «один ко многим» между сущностями *Orders* и *Order Details.* И лишь в очень небольшом числе случаев может потребоваться создать форму, где данные из всех трех таблиц: *Customers, Orders* и *Order Details,* — отображаются одновременно.

Сложно сказать, почему возникла это стремление — при проектировании пользовательского интерфейса избегать в формах иерархических структур. Возможно, потому, что случаи, когда их действительно необходимо использовать, достаточно редки; кроме того, подобные структуры довольно трудно представлять в удобном для восприятия виде. Предположим, что заказчик настаивает на том, чтобы включить в форму, предназначенную для ввода и редактирования данных о покупателях, возможность просматривать информацию о заказах. Эту функциональность в системе легко реализовать, отображая записи из таблицы *Orders* (средний уровень в иерархии) по одной, а записи из таблицы *Order Details* -~ все сразу. При этом придется в форме *Customers,* позволяющей вводить и редактировать данные о покупателях, использовать в качестве подчиненной форму, показанную на рис. 14-2.

Вероятнее всего, в форме *Customers* нельзя отображать все записи, относящиеся к сущности *Orders,* одновременно. Ведь подобная возможность нужна пользователям, чтобы узнать, сколько заказов поступило от данного покупателя, или на какую сумму в среднем он делает заказ и т. п. Вряд ли, открывая форму *Customers,* пользователи заинтересуются данными, относящимися к конкретным продуктам. Но им будет весьма сложно получить ответ на эти вопросы, если все записи, относящиеся к таблице *Orders,* отображаются в форме *Customers* одновременно.

Можно отображать в форме *Customers* только агрегированные данные о заказах, позволив пользователям открывать вспомогательную форму *Orders,* если им будут нужны более подробные сведения. К сожалению, такой подход имеет ряд серьезных недостатков.

Если подробная информация о заказе нужна пользователю, чтобы определить, какие продукты данный покупатель заказывает чаще всего, то очевидно, ему придется открывать дополнительную форму, что само по себе неудобно. А поскольку во вспомогательной форме *Orders* главной формы *Customers* информация о каждом заказе представлена в виде отдельной формы, сравнивать данные о продуктах, которые покупатель заказывал несколько раз, будет нелегко.

Чтобы избежать всех этих сложностей, до недавнего времени иерархическую структуру чаще всего отображали в виде дерева. Элементы управления, позволяющие это сделать — очень действенный и удобный инструмент и, пожалуй, его единственный недостаток — объем данных, отображаемых пользователю на каждом уровне иерархии, очень ограничен, поскольку все данные должны умещаться в одной строке. Из-за такого ограничения нам, скорее всего, не удастся использовать дерево для представления иерархической структуры: легко ли разместить в строке всю информацию о покупателе?

К счастью, в версии Access 2000 и Visual Basic 6 включены новые средства, позволяющие отображать дополнительную информацию для элементов основной формы в виде отдельного набора записей. *Подчиненные таблицы* (subdatasheets) Access 2000 поддерживают представление иерархических структур в виде схем, позволяющих выбирать разные уровни детализации при просмотре (рис. 14-6).

Хотя по внешнему виду подчиненные таблицы уступают многим другим способам представления данных (например, деревьям), они практичны и удобны. Подчиненные таблицы позволяют реализовать иерархические структуры с числом уровней не более семи, но на каждом уровне можно реализовать не более одного вложенного набора записей с дополнительной информацией. Так, невозможно создать

подчиненную таблицу, в которой данные из таблиц *Addresses* (Адреса) и *Orders* (Заказы) представлены как элементы одного и того же уровня. Настраиваемый табличный элемент управления Hierarchical Flexgrid в Visual Basic версии 6 позволяет отображать иерархическую структуру в том же виде, что и подчиненные таблицы в Microsoft Access. Различие одно — число вложенных наборов записей на каждом уровне иерархии для этого элемента управления не ограничено.

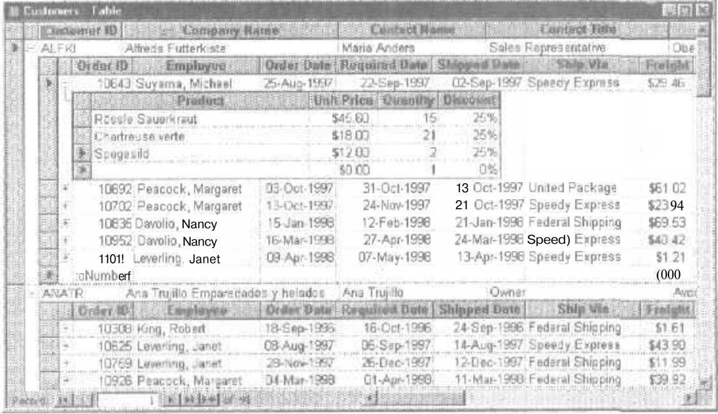


Рис. 14-6. Подчиненные таблицы Microsoft Access 2000 позволяют отображать данные иерархических структур

К сожалению, элемент управления Hierarchical Flexgrid имеет один недостаток; он позволяет только отображать данные пользователю, но не изменять их. Чтобы пользователи смогли редактировать данные, вам придется разработать дополнительные элементы управления, разместив их в главной или дополнительной форме. Разумеется, в результате пользовательский интерфейс станет неудобным, куда лучше предоставить пользователю возможность редактировать данные в той же форме, в которой он их просматривает.

Тем не менее, форма, которая содержит нередактируемые данные, вполне уместна в пользовательском интерфейсе: как правило, данные, представленные в виде иерархических структур, используются для просмотра и анализа, а их ввод и редактирование относятся к другому рабочему процессу.

Связи «многие ко многим»

И, наконец, последний тип связей — это связи «многие ко многим», представленные в базе данных тремя или более таблицами.

В большинстве случаев представление связей «многие ко многим» в пользовательских формах почти ничем не отличается от представления связей «один ко многим» и при отображении данных в пользовательской форме связь «многие ко многим» удается заменить связью «один ко многим» (рис. 14-7).

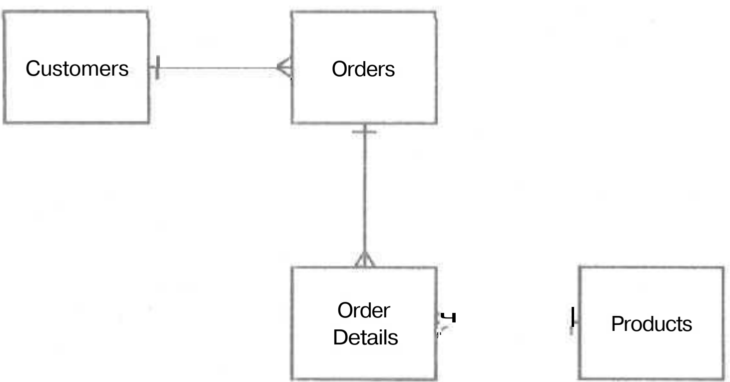


Рис. 14-7. Между сущностями Customers и Products существует связь «многие ко многим», таблицы Orders и Order Details используются как промежуточные

Весьма практично отображать пользователю информацию обо всех продуктах, заказанных покупателем (таблицу *Customers* можно рассматривать как находящуюся на стороне связи «один») или обо всех пользователях, заказавших данный продукт (таблица *Products* находится на стороне связи «один»). При этом допустимы те же самые приемы, что при представлении в форме связей «один ко многим». Вам остается только определить, где разместить данные из промежуточной таблицы и как обрабатывать повторяющиеся записи на стороне связи «многие».

Большинство промежуточных таблид содержит только первичные ключи сущностей, участвующих в связи «многие ко многим». Однако, как уже упоминалось в главе 3, сама связь иногда имеет атрибуты, которые обычно включают в промежуточную таблицу при моделировании данных (рис. 14-7). Если эти атрибуты необходимо включить в разрабатываемую форму, лучше поместить их на стороне «многие».

Если в форме должны отображаться данные о продуктах, заказанных каждым из покупателей (при этом таблица *Customers* находится на стороне «один» связи «один ко многим»), то дата заказа, хранящаяся в одном из полей таблицы *Orders*, очевидно, относится к данным

о продукте, а не о покупателе. Данные, отображаемые в этой форме, интерпретируются так: «Покупатель X приобрел продукт Y 15 числа этого месяца, а продукт Z — 18 числа этого месяца». Если же интерпретировать эту связь с другого конца, то таблица *Products* будет находиться на стороне «один» связи «один ко многим». В этом случае в форме представлены сведения о покупателях, заказавших конкретные продукты, например: «Продукт X был приобретен покупателем У 15 числа этого месяца, а покупателем Z — 18 числа этого месяца». Данные таблицы *Customers* представлены в форме так, как если бы эта таблица являлась участником связи «один ко многим» со стороны «многие».

Весьма вероятно, что в той части формы, где размещена информация из таблицы, участвующей в связи на стороне «многие», будут встречаться повторяющиеся (или по крайней мере, частично повторяющиеся) данные. Вы можете выбрать, отображать ли каждую запись отдельно или выводить пользователю агрегированные данные. Например, составляя список покупателей, купивших отдельный продукт, для каждого из продуктов, которыми торгует компания, вы можете выбрать один из следующих вариантов: каждый раз включать данные о покупателе в разные списки, составляемые для всех продуктов, которые он приобрел; или включить данные о покупателе в список только один раз, указав, сколько раз он заказывал данный продукт, а также суммарное (а возможно, и среднее) число приобретенных этим покупателем единиц каждого вида товара.

Выбирая способ представления данных пользователю, особо позаботьтесь, чтобы в отображаемых таблицах или списках не было полностью повторяющихся записей. Не имеет смысла в списке на экране пользовательской формы перечислять имя одного покупателя 27 раз, если этот список не содержит больше никакой дополнительной информации (например, даты заказа). Такая информация может пригодиться, только если необходимо подсчитать общее число заказов, сделанных данным покупателем; но в этом случае все вычисления средних и суммарных величин должно выполнять разрабатываемое вами приложение, а не пользователь.

В большинстве случаев для представления в пользовательских формах связей «многие ко многим» удается применять методы, которые используются для представления связей «один ко многим». Но в некоторых ситуациях этот способ неприемлем, и необходимо отображать в пользовательской форме связь «многие ко многим» между сущностями. Например, если менеджер по работе с клиентами, просматривая данные о покупателях, купивших какой-либо товар, захотел уз-

нать, какие еще товары заказали эти покупатели, чтобы составить комплексный заказ или стандартный набор продуктов.

Одно из наиболее простых решений — интерпретировать связи как иерархии и использовать для их отображения подчиненные таблицы или настраиваемые табличные элементы управления. Его единственный, но серьезный недостаток — сложность поддержки пользовательского контекста. Тщательно продумайте компоновку интерфейса, поскольку пользователи не всегда могут догадаться, как интерпретировать дополнительные данные, размешенные в подчиненных таблицах. Например, включенная в список проданных продуктов подчиненная таблица, где указаны только имена и фамилии покупателей, заказавших эти продукты, может озадачить пользователя. Что за информацию содержит эта вспомогательная таблица; может быть, это имена и фамилии покупателей, которые приобрели эти продукты, а может быть, поставщиков, продавших эти продукты компании?

Если же отображение данных в виде иерархической структуры кажется вам чересчур неудобным и громоздким или не нужным пользователям, то лучше использовать вспомогательное окно для вывода дополнительной информации, Так легче объяснить пользователю, какие данные содержит это окно. Кроме того, во вспомогательных окнах можно не выводить данные, непосредственно хранимые в таблицах базы данных, заменив их агрегированными данными или другой информацией.

Допустим, менеджер по работе с клиентами, просматривая список продуктов, приобретенных покупателями, захочет структурировать эту информацию как-то иначе: например, в виде списка продуктов, с указанием количества и доли продаж каждого продукта от общего объема. Или ему потребуется информация, сколько покупателей из тех, что приобрели товар А, купили также и сопутствующий товар В, и каков объем продаж товаров А и В по отношению к наиболее продаваемому продукту С. Конечно, такую информацию можно поместить в отдельном поле главной формы, но вдруг это существенно снизит производительность системы? Ведь для вычисления относительных объемов продаж придется выполнять достаточно сложные вычисления. Поэтому, если эти данные не слишком часто требуются, их лучше поместить в дополнительной форме, которую менеджер будет открывать только при необходимости,

Оба описанных способа отображения данных для связи «многие ко многим» не являются взаимоисключающими. Например, можно разместить список покупателей, которые приобрели каждый товар, в виде иерархической структуры (подчиненных таблиц) в основной

форме. И кроме того, вывести информацию о десяти покупателях, сделавших самые крупные заказы, во вспомогательной форме, которую пользователи откроют при необходимости. Как и в любом другом случае, примите решение, исходя из конкретной ситуации.

Итоги

В главе 13 мы рассмотрели связь между компоновкой пользовательского интерфейса и задачами, которые выполняют пользователи. Б этой главе я детально рассказала о структуре элементов управления, размещаемых в формах пользовательского интерфейса, и о том, каким образом структура сущностей, представленных в этих формах, влияет на выбор элементов управления.

При выборе размещаемых в форме элементов управления определяющий фактор — множество сущностей, которые должны быть представлены в этой форме, а также наличие и вид связей между этими сущностями (если в форме представлено несколько сущностей). Второй фактор — число атрибутов сущностей, которые будут отображаться в этой форме. Как правило, число разных элементов или групп элементов, одновременно отображаемых пользователю, не превышает 25-30.

В главе 15 мы перейдем к следующему уровню проектирования пользовательского интерфейса — выбору элементов управления для представления различных типов данных.

Выбор элементов управления пользовательского интерфейса

В главе 34 мы обсудили, как различными способами организовать структуры форм пользовательского интерфейса. Эти структуры зависят от связей между сущностями, представленными в форме. А теперь посмотрим, как выбор элементов управления зависит от логических типов данных, связанных с этими элементами.

Выбор типа элемента управления определяется двумя факторами. Первый и главный — это соответствие типа элемента управления ментальной модели пользователя. Иными словами, вы должны выбрать элемент управления, соответствующий представлениям пользователей о данных, с которыми этот элемент связан, и с которым пользователю наиболее удобно работать. Второе требование, которое необходимо учесть — минимизация объема данных, которые пользователю придется вводить с клавиатуры. Везде, где только возможно, старайтесь использовать окна и списки, позволяющие выбирать нужные значения, а не вводить их вручную.

Обычно пользователи компьютерных систем, работающих с базами данных, представляют себе вводимые данные как текстовую информацию. Это вполне объяснимо. Предположим, для доступа к некоторому набору данных вы используете Microsoft Access. Таблица Microsoft Access представляет этот набор данных в виде текстовых полей (рис. 15-1). Однако на самом деле только поле *CustomerName* в этой таблице имеет тип данных *text.* Поле *Customer Number* имеет тип

AutoNumber, поле *DateOfFirstOrder* — тип *date/time*, поле *CreditLimit* — тип *currency*, а поле *PreferredCustomer* содержит булевы константы (Yes/No).

CustomerStatus : Table

CustomerName	CustomerNumber	DateOfFirstOrder	CreditLimit	PreferredCustomer
Alfreds Futterkiste	10060	9/4/98	$5,000.00	No
Ana Trujillo Emparedados y helados	10061	3/4/98	$2,000.00	No
Antonio Moreno Taquería	10062	1/26/98	$10,000.00	Yes
Around the Horn	10063	4/10/98	$15,000.00	Yes
Berglunds snabbköp	10064	3/4/98	$30,000.00	Yes
Blauer See Delikatessen	10066	4/29/98	$5,000.00	No
Blondel père и fils	10066	1/12/98	$20,000.00	Yes
Bólido Comidas preparadas	10067	3/24/98	$10,000.00	Yes
Bon app'	10068	5/6/98	$25,000.00	Yes
Bottom-Dollar Markets	10069	4/24/98	$10,000.00	Yes

Рис. 15-1. В таблице значения полей различных типов представлены как текстовые величины

Когда разработчики выполняют операции манипулирования данными, они прежде всего принимают но внимание домены, на которых определены эти данные. Например, операция соединения, выполняемая над двумя полями, в одном из которых хранятся даты, а в другом — булевы значения, даст бессмысленный результат. Тем не менее, существует немало систем, где для представления пользователю данных различных типов применяют один и тот же метод и одни и те же элементы интерфейса: как правило, *текстовые поля* (text boxes). Формально такой подход не является ошибочным: в конце концов, в текстовых полях можно представлять данные любых типов. Однако он нисколько не облегчит работу пользователей с данными. Текстовое поле следует использовать, только когда никакие другие, более подходящие для этого типа данных элементы управления применить невозможно,

Данные, определенные на домене *Date/Time,* представляют собой строку символов определенного формата. Но чтобы пользователи могли работать с ними, данные должны быть представлены в виде реальной даты. Однако представления пользователей о структуре и организации данных даты, существенно отличаются от представлений о структуре и организации текстовых данных. Например, если пользователь допустит ошибку при вводе имени покупателя, он проанализирует и исправит ее таким образом: «О! Похоже, имя Mary Smith (Мери Смит) введено неверно: Jary Smith. Нужно исправить ошибку, заменив букву J на букву M». Но если пользователь увидит ошибку при вводе даты в текстовое поле, то скорее всего, подумает: «Наверняка эта буква M означает месяц». Выбрав для работы с датами подходящий элемент управления (например, позволяющий выби-

рать значения дат, автоматически вычисляя год, месяц и день недели), вы обеспечите большее соответствие интерфейса ментальной модели пользователя и существенно облегчите ему работу с системой.

Ограничения на диапазон вводимых данных также позволяют заметно улучшить производительность системы и точность ввода. Эти ограничения не следует путать с методами проверки правильности введенных данных (о них речь пойдет в следующей главе). Проверка правильности данных выполняется системой уже после того, как данные введены, и преследует цель выяснить; осмысленны ли эти данные, и не противоречат ли они бизнес-правилам. Ограничения же допустимых значений данных при вводе позволяют автоматически отфильтровать недопустимые значения данных (например, данные в неверном формате) еще при вводе их в систему.

Первое и наиболее жесткое ограничение налагается на тип вводимых данных. С точки зрения выбора подходящих элементов управления, все вводимые данные можно разделить на четыре типа: логические данные, множества значений, числовые данные и даты, текстовые данные. В этой главе мы уделим внимание каждому типу.

ПРИМЕЧАНИЕ Я не буду подробно останавливаться на разнообразных дополнительных элементах интерфейса: графических фрагментах и пиктограммах, звуковых и графических клипах. Эти элементы, как правило, разработаны фирмами, занимающимися выпуском коммерческого программного обеспечения, в виде встраиваемых компонентов ActiveX. Наш обзор будет ограничен стандартными компонентами Microsoft Visual Basic и Access. Тем, кто хочет узнать, какие еше дополнительные элементы управления можно использовать при разработке пользовательского интерфейса, я рекомендую обратиться на Web-узел корпорации Microsoft, а также на Web-узлы других компаний — разработчиков программного обеспечения. Вполне возможно, что вы найдете там уже готовое решение.

Логические значения

Как и многие другие типы данных, логические значения можно представлять в пользовательском интерфейсе как текстовые поля, но я не рекомендую это делать. Рассмотрим пример: в таблице *Customer* есть поле *CreditApproved*, определенное как булево значение (в Microsoft Access ему соответствует тип данных *Yes/No),* где хранятся данные о подтверждении кредита. Для представления данных, хранимых в поле *CreditApproved* таблицы *Customer*, можно использовать текстовое поле, значение которого пользователи будут изменять. Затем проверяется,

действительно ли пользователь ввел одно из допустимых значений: *Yes* или *No* — для Access, *True* или *False* — для других механизмов баз данных. Но если при вводе данных не задано никаких дополнительных ограничений, весьма вероятно, что пользователи будут вводить в текстовое поле произвольные значения, например «Условно» или «Нерегулярно». Конечно, есть особые случаи, когда это целесообразно, и произвольные значения определенным образом обрабатываются и интерпретируются системой. Но чаше такая интерпретация логических величин в пользовательском интерфейсе приведет к снижению производительности. Гораздо лучше элемент интерфейса, предоставляющий пользователю возможность выбрать одно из двух возможных значений. Для этого весьма подходят два элемента управления, которые имеются и в Access, и в Visual Basic — флажки и кнопки-выключатели (рис. 15-2).

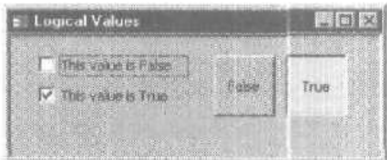


Рис. 15-2, Для представления логических величин используются флажки и кнопки-выключатели

Большинству пользователей хорошо знакомы флажки, и поэтому их чаще используют для представления логических величин в пользовательском интерфейсе. Кнопки-выключатели применяют реже. Они больше подходят для представления булевых значений, моделирующих переключение между процессами (например, при выборе режима) в пользовательском интерфейсе, чем для ситуаций, когда надо выбрать один из вариантов — «истина» или «ложь». Однако кнопки-выключатели имеют один существенный недостаток: когда они находятся в состоянии «выключено», их трудно отличить от обычных кнопок, при щелчке которых система выполняет определенные действия. Пользователей, привыкших к обычным кнопкам, может ввести в заблуждение кнопка-выключатель *Credit Approved.* Они скорее всего подумают, что щелчок этой кнопки инициирует процесс проверки кредита, тогда как на самом деле положение «включено» этой кнопки означает, что кредит подтвержден, а положение «выключено» — отказ в кредите. Поэтому я использую и кнопки-выключатели в группах, и *кнопки с зависимой фиксацией* (radio buttons), которые также называют переключателями.

Еще несколько важных замечаний. Не используйте переключатели для моделирования отдельных логических значений. В Microsoft Windows нет механизмов, которые запрещали бы делать это, однако в силу сложившихся правил для моделирования отдельных логических значений используют другие элементы управления — флажки. Переключатели же применяют там, где пользователю предлагается выбрать один из нескольких взаимоисключающих параметров.

Более того, используя переключатели (в особенности, расположенные рядом) для представления нескольких отдельных логических значений, вы опять-таки можете ввести пользователя в заблуждение: он подумает, что эти параметры связаны между собой, и из всего множества параметров нельзя выбрать более одного за один раз, причем при включении одного переключателя все остальные автоматически будут выключены. Поскольку всякое расхождение между ожидаемым и реальным поведением системы раздражает пользователей, лучше придерживаться общепринятых правил.

Наборы значений

Существует достаточно многочисленная группа элементов управления, позволяющая отображать наборы значений в формах. Какой элемент управления предпочесть — зависит от того, нужно ли пользователям выбирать только одно из значений в наборе (возможно, одну величину из некоего диапазона) или несколько значений одновременно,

Выбор одного значения из диапазона

Ситуация, когда допустимость какого-либо значения определяется его присутствием в списке значений, встречается достаточно часто. Например, определенное значение *CustomerNumber* (Номер покупателя) в записи в форме *Orders* (Заказы) допустимо тогда и только тогда, когда такой номер имеется в таблице *Customers* (Покупатели). Если вы решили свести к минимуму операции ввода пользователями данных с клавиатуры, то поместите в разрабатываемую форму список покупателей и предоставьте пользователям возможность выбрать нужное значение из этого списка.

Наиболее часто при этом используют два вида элементов управления — комбинированное окно и окно списка Их функции довольно сильно различаются, в первую очередь тем, что в комбинированное окно пользователи могут вводить значения, которых еще нет в списке. Как я уже упоминала, комбинированное окно лучше использовать для ввода записей, относящихся к связанной сущности. Даже если в разрабатываемой форме не предусмотрен ввод новых данных, комбинированное окно послужит для вывода списка значений — ведь

многим пользователям понравится возможность искать значения по нескольким первым символам, введенным с клавиатуры.

В Visual Basic существует три вида комбинированных списков (рис. 15-3), которые позволяют отображать пользователю весь список постоянно или только по требованию (последние называют раскрывающимися списками). В Access же — только раскрывающиеся списки. Кроме того, в списках Visual Basic можно размещать флажки рядом с каждым значением из списка, а в Access — нет.

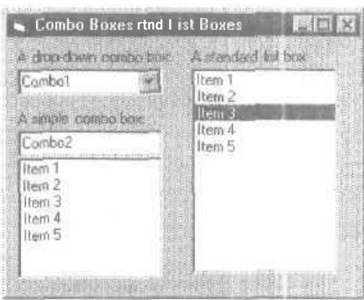


Рис. 15-3. В Visual Basic существует три вида списков

Руководство «The Windows Interface Guidelines for Software Design» («Руководство по разработке интерфейса для Windows-приложений») рекомендует задавать размер списка так, чтобы на экране одновременно отображалось от трех до восьми его элементов. Если размеры формы сильно ограничены, и весь список разместить на ней невозможно, используйте раскрывающиеся списки.

Раскрывающийся список следует использовать, даже если он умещается в окне формы, но пользователю не нужно постоянно иметь его перед глазами. Например, из списка покупателей в форме *Orders* пользователь выбирает имя интересующего его покупателя, а информация о том, какие еше покупатели зарегистрированы в системе, ему не нужна.

Однако иногда полезно предоставить пользователям возможность просмотреть все возможные значения величин, представленных в списке: например, если список или какое-либо из содержащихся в нем значений часто обновляется. Так, в библиотеках различные книги сортируют по тематике. В автоматизированной системе регистрации и учета, предназначенной для библиотеки, для каждого отдельного наименования или тома определяется тематическая категория. При вводе информации о новой книге пользователь выбирает соответствующую запись из списка тем. Этот список постоянно пополня-

ется и обновляется, поэтому пользователи, скорее всего, будут часто просматривать обновленный список тематических категорий, чтобы подобрать подходящую тему для данной книги или ввести новую, если ни одна из уже имеющихся не подходит.

И в Visual Basic, и в Microsoft Access есть интереснейшая возможность — отображать пользователю не ту информацию, которая непосредственно хранится в таблице, а лишь ее часть или данные из других таблиц. Например, если в качестве первичного ключа в таблице *Customers* используется поле с автоматически увеличивающимися значениями или поле *Identify,* вам придется хранить это значение как внешний ключ в таблице *Orders,* но отображать пользователям данные, хранящиеся в этом поле, не имеет смысла. Вместо значений первичного ключа таблицы *Orders* в форму *Orders* можно поместить список фамилий и имен покупателей (данные из таблицы *Customers*), из которого пользователи будут выбирать нужные значения.

В Visual Basic это можно сделать, указав различные значения свойств *DataField* и *ListField.* В Access используют несколько полей: одно — для хранения фамилий и имен, а другое — для хранения значений номеров покупателей. Затем следует привязать соответствующий элемент управления к полю таблицы *Orders,* в котором хранится номер покупателя, и отключить отображение номера пользователя, задав нулевую ширину соответствующего поля.

Возможность отображать значения нескольких полей таблиц в комбинированных окнах, поддерживаемая Microsoft Access, весьма удобна и часто применяется, Жаль, но в Visual Basic она отсутствует. Например, если в форме выводится список фамилий покупателей, может понадобиться предоставить пользователю дополнительную информацию: скажем, название города, где живет покупатель. В Visual Basic это можно сделать только при помощи вычисляемого поля, где выполняется конкатенация значений, хранимых в соответствующих полях таблиц; данную операцию определяют, задав соответствующее значение свойства *ListField.* Это, конечно, не так удобно. в особенности если некоторые поля таблиц содержат пустые значения.

Списки и комбинированные окна очень удобны, но их невозможно использовать, если список содержит слишком много значений, например, несколько сотен. В этих случаях приходится ограничивать число значений, которые пользователь будет видеть на экране, для чего обычно применяют фильтры. Можно предложить пользователю выбрать одну из букв алфавита (или регион, где расположены филиалы компании, или штат, где проживают покупатели), а затем — из

списка, содержащего несколько сотен записей, те, которые удовлетворяют заданному критерию.

Если же список содержит не более пяти-шести записей, и все значения, перечисленные в нем, постоянны, вместо списка или комбинированного окна используют кнопки-выключатели или, еще чаще, группу кнопок с зависимой фиксацией (рис. 15-4).

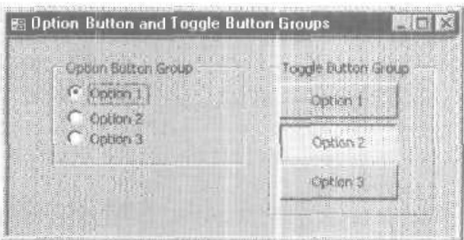


Рис. 15-4. Кнопки-выключатели и кнопки с зависимой фиксацией

Хотя многие средства разработки, например Visual Basic, позволяют генерировать группу кнопок на этапе исполнения, я не советую делать это. Пользователям гораздо удобнее работать с формами, в которых все элементы управления расположены на привычных местах, чем с постоянно изменяющимся интерфейсом. Поэтому группы кнопок-выключателей или кнопок с зависимой фиксацией следует использовать, только когда число и названия элементов в этих группах постоянны (или по крайней мере, не изменятся до выхода следующей версии программы). Во всех остальных случаях лучше применять списки или комбинированные окна. При изменении числа или значений элементов списков размер элементов управления не меняются, и не причиняют пользователям неудобств.

Выбор нескольких значений из диапазона

Если вам предстоит выбрать несколько значений из некоторого множества, то скорее всего, вы имеете дело с сущностью, находящейся на стороне «многие» связи «один ко многим». Как уже говорилось в главе 14, первое, что следует определить в такой ситуации — нужно ли отображать пользователю все записи сразу или по одной, и должен ли пользователь выбирать все нужные значения сразу или по одному.

В Access, чтобы отображать пользователю записи одну за другой, удобнее всего использовать подчиненную форму в режиме отображения одной записи. Задайте свойства *LinkChildFields* и *LinkMasterFields* в разрабатываемой форме — и Access выполнит за вас основную часть

работы. Чтобы создать такую подчиненную форму в Visual Basic, придется потрудиться, но взамен вы получите больше возможностей управлять данными. В любом случае, этот метод вполне оправдан, если нужно одновременно выбирать несколько значений из списка, и в особенности, если вы собираетесь использовать в разрабатываемой форме множество разных элементов управления.

Иногда нужно выбирать несколько значений из каждой записи, причем пользователю будет отображаться множество записей одновременно. Используйте таблицу с непрерывным режимом просмотра, если вы работаете с Access (рис. 14-2), и настраиваемый табличный элемент управления или иерархическую таблицу Microsoft Hierarchical FlexGrid, если используете Visual Basic. Таблицы Microsoft Access — чрезвычайно мощное средство, иногда даже слишком мощное. Помимо текстовых полей, в них несколько видов дополнительных элементов управления. *Настраиваемые табличные элементы* (grids) в Visual Basic порой осложняют создание пользовательских приложений, а иерархические таблицы позволяют только отображать данные в виде иерархически организованных структур, но не редактировать эти данные.

К счастью, в Access и Visual Basic арсенал средств, обеспечивающих управление данными, достаточно широк. Например, подчиненные формы в Access позволяют отображать записи в режиме непрерывного просмотра, и таким образом, пользователь видит на экране сразу несколько записей. В версии 6 Visual Basic появился новый объект *DataRepeater*, предоставляющий практически те же возможности, что и подчиненная форма Access, хотя реализованы эти два элемента управления по-разному. Каждый из них применим там, где нужно отображать пользователю множество записей и при этом использовать тип элементов управления, не поддерживаемый в подчиненных таблицах и настраиваемых табличных элементах управления (например, группу связанных параметров). Оба эти элемента управления: и подчиненные формы, и объект Data Repeater, — позволяют разбивать каждую запись на несколько строк при отображении ее в форме пользовательского интерфейса.

Еще один элемент управления, часто применяемый для отображения множества записей в форме — дерево. Как правило, дерево используют для отображения иерархических структур с различной степенью детализации, но с его помощью можно также реализовать просмотр дополнительной информации для каждой из записей, хотя редактировать данные текущей записи с помощью дерева не очень удобно. Приняв концепцию, подобную той, что используется в Microsoft

Windows Explorer, можно отображать пользователю дополнительные данные, относящиеся к выбранной записи, в другой части формы.

И наконец, в тех случаях, когда пользователям нужно выбирать несколько значений из списка, подойдет связанная пара списков. Подобная структура часто используется в мастерах. Вот пример из мастера таблиц Microsoft Access 2000: списки *Sample Fields* и *Fields in my new table* образуют связанную пару (рис. 15-5). Как правило, пользователи легко осваиваются с подобными элементами управления, однако такие списки создают немало проблем для разработчиков.

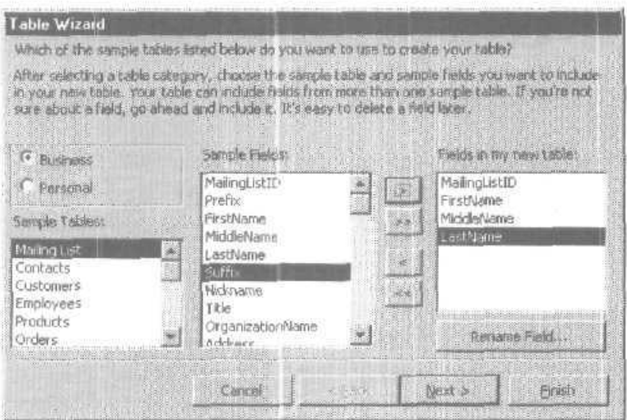


Рис. 15-5. Связанные списки в мастере таблиц Access 2000

Связанные списки применяют для первоначального ввода данных, но они не предоставляют пользователям возможности просматривать и изменять введенные данные. Кроме того, связанные списки занимают слишком много места. Вот почему такие структуры, как правило, используются в мастерах, где каждая отдельная операция ввода выполняется из нового диалогового окна. Если же пользователь, выбрав несколько элементов из списка, уже не будет впоследствии просматривать весь список, то использовать связанные списки не имеет смысла. Лучше остановиться на одном из уже обсуждавшихся нами элементов управления, или на списке, позволяющем выбрать несколько элементов.

Списки, позволяющие выбрать несколько элементов, очень удобны, но применять их следует с осторожностью. Просматривая такие списки, пользователи могут по ошибке отменить уже сделанный выбор элементов. Особенно часты такие ошибки, когда для выбора нового элемента требуется щелкнуть его мышью, а не поставить фла-

жок или переключатель в соответствующее положение. Когда данные приходится выбирать из длинного списка, отсортировать добавленные в список, удаленные и повторно выбранные записи нелегко. Возможно, лучше применить обычный список, где выбранные элементы отображаются в сочетании с текстовым полем или комбинированным окном, позволяющими добавлять записи.

Числовые данные и даты

Для отображения числовых данных и дат чаще всего используют текстовые поля. Для числовых данных целесообразно применять ограничения на вводимые величины. Если диапазон значений, вводимых в поле, достаточно широк, то задать такие ограничения не просто.

Контролировать числовые значения, вводимые в текстовые поля, позволяют маска ввода в Microsoft Access и элемент управления *MaskedData* в Visual Basic — по крайней мере, вы можете запретить вводить в поля символы алфавита и специальные символы. В Access также есть множество средств, позволяющих интерпретировать введенные данные. Но проверять правильность данных, когда они уже введены, — не самый лучший вариант, и если есть возможность ограничить диапазон допустимых значений, то нужно использовать именно этот метод.

В Visual Basic 6 появились два новых элемента управления для ввода календарных дат — *MonthView* и *DateTimePicker* (рис. 15-6). В Access 2000 имеется элемент управления «календарь», аналогичный *Month View* в Visual Basic.

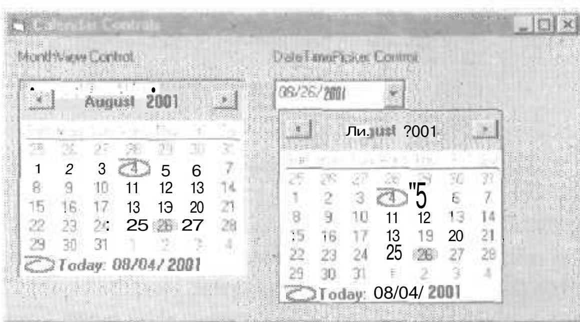


Рис. 15-6. Элементы управления MonthView и DateTime Picker

Элементы управления, обеспечивающие пользователям возможность ввода даты, по своей функциональности напоминают окна со списками и комбинированные списки: *DateTimePicker* выводит на

экран окно календаря, только когда пользователь щелкает мышью стрелку рядом с полем, в котором отображается дата, а элемент управления *MonthView* отображает календарь постоянно. Особо отмечу, что и *MonthView,* и *DateTimePicker* работают только со значениями дат, а не с датой и временем одновременно. Поэтому если их связать с полями таблиц баз данных, для которых выбран тип данных *Date/Time,* то придется предусмотреть дополнительные средства преобразования формата. Уделите особое внимание средствам работы с данными, представляющими даты.

И в Visual Basic, и в Microsoft Access присутствует элемент управления *UpDown,* также называемый циклическим счетчиком. Он используется для ввода числовых значений и данных в формате *Date/Time* в операционной системе Windows, и поэтому знаком большинству пользователей (рис. 15-7). Элементы управления *UpDown* применяют там, где вводимые значения изменяются циклически: например, введенная дата — один из семи дней недели. Или если вводимые данные нужно округлить до некоего числа: например, числовое значение — до ближайшего целого десятка.

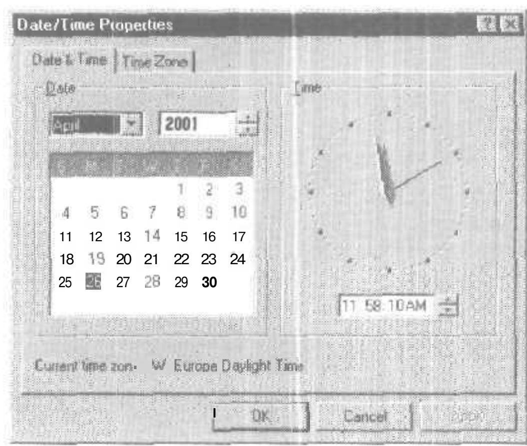


Рис. 15-7. Для настройки времени в Windows используются элементы управления UpDown

И, наконец, еще один вид элементов управления Visual Basic, которые используются для ввода числовых величин — это *бегунки* (slider controls) *и полосы прокрутки* (scroll bars), показанные на рис. 15-8.

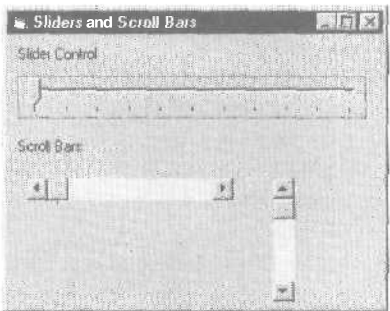


*Рис. 15-8. **Бегунки и полосы прокрутки используют для ввода числовых величин***

Эти элементы управления чрезвычайно редко применяют в приложениях, работающих с базами данных. Они — всего лишь графические средства, позволяющие задавать числовые значения некоторых параметров, и поэтому в принципе их можно использовать для ввода числовых данных.

ПРИМЕЧАНИЕ Возможно, вы и не предполагали, что полосы прокрутки можно использовать как элементы, работающие с числовыми данными — их практически нигде не упоминают в таком качестве. Действуют они так: когда пользователь ставит рычажок полосы прокрутки в определенное положение, то она передает приложению числовое значение параметра, регулируемого при помощи этой полосы (разумеется, все происходит незаметно для пользователя).

Я советую использовать эти элементы управления там, где пользователю нужно сравнивать между собой несколько значений, причем коэффициент отношения между сравниваемыми величинами гораздо важнее, нежели точное значение самих этих величин. Так, однажды я применила бегунок, чтобы пользователи могли задавать значение степени соответствия при поиске похожих записей в базе данных: например, определить 100-процентное совпадение (записи совпадают полностью) или 50-процентное (совпадает только половина символов).

Текстовые данные

И вот, наконец, мы добрались до тех элементов управления, которые я поставила в самый конец очереди — до текстовых полей. Предлагая использовать их только там, где не подходят другие элементы управ-

ления, я вовсе не хотела сказать, что делать это ни в коем случае не нужно. Просто слишком часто текстовые поля применяют там, где разумнее задействовать другой элемент управления. Большинство данных, хранящихся в базах, представляют собой текст, и поэтому текстовые поля можно успешно применять для отображения пользователю этих величин. Исключение составляет только поле внешнего ключа — в этом случае лучше подойдет комбинированный список или окно списка.

Общий принцип, определяющий выбор элементов управления, применим и к текстовым полям: их нужно использовать там, где они соответствуют ментальной модели пользователя. При этом следует задействовать все возможные способы ограничения допустимых значений.

В Microsoft Access допустимые значения вводимых данных можно ограничить при помощи маски ввода (свойство *Input Mask* текстового поля), а в Visual Basic — при помоши элемента управления *MaskedData*. Оба этих элемента управления позволяют задать шаблон, которому должны удовлетворять вводимые данные. Например, для индивидуального номера системы социального страхования в США маска ввода будет ###-##-#### (три цифры, за которыми следует дефис, затем две цифры, снова дефис, еще четыре цифры).

В отдельных случаях маска ввода очень полезна, но к сожалению, задать ее удается не часто. Кроме того, при этом есть риск чрезмерно ограничить диапазон допустимых значений. Самый простой пример — номера телефонов, хранящиеся в базе данных. Очевидно, соблазн использовать маску ввода для поля, моделирующего телефонный номер, велик. Однако прежде чем делать это, выясните, будут ли в базе данных храниться номера телефонов частных лиц или организаций, находящихся на территории одной страны, или же нужно будет вводить и иностранные номера. Очевидно, что число цифр и формат телефонных номеров в разных странах не совпадают. Но даже если база данных будет содержать телефонные номера, относящиеся к одной стране, все равно трижды подумайте: ведь при междугородних звонках кроме телефонного номера абонента часто требуется набирать различные коды городов. А в крупных организациях, насчитывающих множество отделов, иногда применяется добавочный внутренний номер, чтобы соединиться с конкретным сотрудником.

Никогда не забывайте, что ваша главная цель — обеспечить пользователям максимум удобств при работе с системой. Если же система запрещает ввод абсолютно правильных данных, просто потому что подобная ситуация не была предусмотрена при проектировании и

формат вводимых данных ограничен слишком жестко, то это серьезный недостаток.

Помимо стандартных текстовых полей в Access и Visual Basic имеется элемент управления *Microsoft Rich Textbox*, позволяющий вводить текст, отформатированный с помощью различных стилей и гарнитур шрифтов. Однако реализация этого элемента управления, как правило, требует много времени и усилий, поскольку кроме поля, позволяющего вводить текст, нужен дополнительный интерфейс, позволяющий задавать свойства текста.

Rich Textbox дополнительно усложняет и реализацию операций манипулирования данными, поскольку форматирование текста инкапсулировано в сам элемент управления. Поэтому я рекомендую использовать его, только когда те дополнительные возможности, которые он предоставляет по сравнению с обычными полями ввода, действительно нужны пользователям. И только для тех величин, которые представляют собой отдельные фрагменты текста и используются системой для вывода на экран в различных формах, но не участвуют в операциях поиска или соединения с другими значениями.

Например, форматированные элементы текста, вводимые при помощи Microsoft Rich Textbox, можно использовать для создания шаблонов стандартных писем, которые компания рассылает своим клиентам. Текст каждого из таких стандартных писем содержится в базе данных вместе с форматированием, а для его поиска используют категории или иную описательную информацию, присваиваемую каждому такому фрагменту текста. Эта информация хранится в отдельном поле таблицы, содержащем простой неформатированный текст.

Итоги

В этой главе мы рассмотрели различные элементы управления, которые используются для ввода информации в базу данных и отображения ее пользователю. При выборе элементов управления для пользовательского интерфейса прежде всего следует учитывать ментальную модель пользователя. Немаловажную роль играют ограничения допустимых значений вводимых данных.

В следующей главе мы подробно обсудим случаи, когда ограничить допустимые значения при вводе невозможно, а также вопросы, касающиеся программной реализации проверки правильности введенных данных.

Поддержка целостности базы данных

Представьте себе такую ситуацию: вы, владелец небольшой компании, потратили немало сил в борьбе за перспективного клиента. В результате клиент согласился разместить у вас выгодный заказ, однако поставил условие: заказ должен быть выполнен в течение 48 часов. Зная, что это вам вполне по силам, вы начинаете праздновать победу, Но тут «на сцену выходит» главный бухгалтер и сообщает, что не подпишет договор, не проверив, располагает ли клиент достаточным кредитом, а такая проверка займет не меньше недели. Что делать в такой ситуации? Скорее всего, вы объясните главному бухгалтеру, что в исключительных случаях можно пойти на некоторые нарушения сложившихся правил. Бухгалтер — обычный человек, а большинство людей, как правило, предпочитают не спорить с начальством и делают (или, по крайней мере, стараются сделать) то, что им приказано.

Компьютерные системы не обладают столь же покладистым характером и с ними гораздо сложнее «договориться», чем с человеком. Они часто «бастуют», отказываясь выполнять абсолютно разумные и логичные с точки зрения пользователя действия, и при этом приводят абстрактный аргумент — «поддержка целостности данных». В предыдущих главах мы уделили много внимания вопросам целостности данных, ее отражению в модели данных и реализации на программном уровне. И вот теперь, объяснив вам, как это делается, я выскажу мысль, которая разом перечеркнет все мои предшествующие замечания: поддержка целостности данных — не самая важная функция системы.

Предвидя удивленные возгласы и протесты, спешу уточнить: я отнюдь не призываю отказаться от проверки правильности данных. Я просто пытаюсь объяснить, что целостность базы данных гораздо менее важна, чем способность системы служить пользователю и помогать ему в каждодневной работе. И вы должны соответственно спроектировать вашу систему. Данные, хранящиеся в памяти компьютеров, должны быть правильными, но это отнюдь не означает, что они должны быть абсолютно верными в момент их ввода.

Остановитесь на минуту и подумайте: для чего вообще нужна разрабатываемая вами система? Чтобы помогать людям выполнять их работу. Поэтому такая проверка правильности данных, которая помогает пользователям выполнять их работу, полезна, поскольку она соответствует целям компьютерной системы. А если проверка правильности данных мешает пользователям выполнять отдельные действия в том порядке, который для них наиболее удобен или который они считают целесообразным, либо не позволяет выполнять те действия, которые пользователи считают разумными и полезными, но которые не были предусмотрены на начальных этапах проектирования системы, то такая проверка, безусловно, вредна.

Правила, которых здесь следует придерживаться, достаточно просты. Система не должна запрещать пользователю вводить информацию, если он пропустил какие-либо данные, или если вы не предвидели некоторые его действия заранее. Конечно, придерживаться этих правил гораздо сложнее, чем сформулировать их. Как всегда, разработчикам баз данных придется идти на разумные компромиссы. В этой главе мы познакомимся с различными видами ограничений, поддерживающих целостность данных, и посмотрим, как спроектировать систему, чтобы избежать случайных ошибок при вводе данных и обеспечить пользователям удобство и динамичность в работе.

Классы ограничений целостности

В главе 4 мы определили шесть различных видов ограничений целостности, относящихся к различным логическим уровням в реляционной модели. В этой главе мы рассмотрим вопросы реализации целостности данных под другим углом зрения и воспользуемся иной классификацией, Разделим все ограничения целостности на два класса: *внутренние ограничения* (intrinsic constraints) и ограничения, определяемые особенностями веления бизнеса, называемые также *бизнес-правилами* (business rules).

Внутренние ограничения регулируют физическую структуру данных и диктуются реляционной моделью. Например, ограничение,

запрещающее удалять запись из таблицы *Customers* (Покупатели), если в таблице *Orders* (Заказы) имеются связанные с ней записи, — это внутреннее ограничение, поскольку ссылочная целостность — это часть реляционной модели. При удалении из базы данных записи о покупателе, но без удаления всех записей о заказах, сделанных данным покупателем, база данных будет находиться в несогласованном состоянии. Если впоследствии добавить в нее запись о покупателе, значение первичного ключа которой совпадет со значением первичного ключа удаленной записи, то осиротевшие записи о заказах, сделанных покупателем, информация о котором удалена из базы, могут быть ошибочно связаны с новой записью. Такая ситуация вероятна, если определить значение первичного ключа на поле, которое содержит фамилию покупателя.

Но и когда возможность повторного использования значения первичного ключа полностью исключена, осиротевшие записи могут стать причиной большой путаницы и причинить немало неприятностей. Например, запросы, в которых вычисляется общее количество продуктов, проданных компанией за определенный период времени, могут выдавать разные результаты в зависимости от того, используется ли в процессе выполнения запроса соединение таблиц *Orders* и *Customers*. Чтобы получить полный список проданных продуктов, содержащий информацию о том, какие продукты и в каком количестве приобрел каждый из покупателей, используется естественное соединение таблиц *Customers* и *Orders*. В этот список войдут только те записи из таблицы *Orders*, для которых имеются соответствующие записи в таблице *Customers*, и, следовательно, только эти данные будут использованы при вычислении общих объемов продаж. Осиротевшие записи в таблице *Orders* отсутствуют в наборе результатов, получившемся в результате соединения таблиц *Customers* и *Orders*, и, следовательно, не будут учтены в вычислениях. Однако если при вычислении объемов продаж использовано соединение таблиц *Orders* и *Products*, осиротевшие записи будут присутствовать в наборе результатов и учитываться в вычислениях. Таким образом, на вопрос «Какое количество товаров было продано в июне?» система даст два разных ответа, в зависимости от способа выполнения запроса. Что, разумеется, недопустимо.

В отличие от внутренних ограничений, бизнес-правила основаны на предметной области. Например, правило, запрещающее выполнять каскадное удаление из таблицы *Orders* всех записей, относящихся к покупателю, данные о котором были удалены из базы, до тех пор пока все заказы, сделанные этим покупателем, не будут закрыты или отме-

нены, — это бизнес-правило. Бизнес-правила определяют, что обычно не делают в конкретной ситуации, а внутренние ограничения – что вообще нельзя сделать. На практике, однако, провести четкие различия между внутренними ограничениями и бизнес-правилами зачастую невозможно, да и не нужно. Важно лишь, что один из классов ограничений целостности определяется предметной областью. Вы реализуете эти ограничения лишь потому, что «так хочет пользователь» или «так пользователю удобнее». Если же на самом деле пользователь совсем этого не хочет или ему это неудобно, можете опустить эти правила и не включать их в разрабатываемое приложение.

Удаление записи о покупателе, если в базе данных имеются записи о еще не выполненных или не отмененных заказах, может привести к тому, что пользователи получат неправильные результаты при выполнении запросов к данным. Это вряд ли их устроит, и поэтому правило, запрещающее такое удаление, обязательно нужно реализовать, и оно должно выполняться всегда. Но правило, ограничивающее число сотрудников, непосредственно подчиняющихся одному менеджеру, питью, не относится к тем строжайшим правилам, которые не допускают исключений. Если какому-либо менеджеру целесообразно подчинить шесть или семь сотрудников, оно нарушается. Если же системой будет предусматривать жесткое ограничение числа подчиненных, то оно будет мешать пользователям, когда возникнет исключительная ситуация.

Нарушение бизнес-правил не приведет к нарушению целостности или дестабилизации работы базы данных. Если пользователь удалит запись об одном из клиентов компании, одновременно с этим удалив записи обо всех сделанных этим пользователем заказах, то возможно, будет потеряна важная информация, однако это никак не повлияет на правильность остальных данных. Соответствующие записи в таблицах *Customers* и *Orders* можно восстановить или ввести повторно, и база данных снова будет содержать правильную информацию.

Внутренние ограничения и бизнес-правила должны по-разному интерпретироваться системой. При нарушениях внутренней целостности нарушается согласованность базы данных, после чего она содержит неправильную информацию. Соблюдение же бизнес-правил часто зависит от желания пользователей. В следующих разделах мы подробно рассмотрим оба этих класса ограничений целостности.

Внутренние ограничения

Правила проверки правильности введенных значений, называемые внутренними ограничениями, управляют внутренней структурой базы

данных. В этот класс входят правила, регулирующие тип, формат, длину данных, допустимость значений *Null,* а также ряд ограничений: диапазона вводимых значений, на уровне сущностей и гарантирующие ссылочную целостность.

Ограничения, налагаемые на тип данных

Если при проектировании пользовательского интерфейса правильно выбран тип элементов управления, используемых для ввода и изменения данных, пользователи редко нарушают правила, диктуемые системой. Вряд ли пользователь будет вводить дату или время в поле *Amount Due* (Число единиц отправляемого товара), допускающее целочисленные значения, или пытаться ввести текст, работая с переключателем или флажком — такие действия возможны только по ошибке. Но пользователь может попытаться набрать в текстовом поле слово, а не число (например, ввести в поле *Amount Due* слово «одиннадцать» вместо числа 11). Поэтому правильности выбора элементов интерфейса следует уделять особое внимание.

Ограничения, налагаемые на формат данных

Как правило, такие нарушения не слишком заботят пользователей и разработчиков, поскольку формат данных можно изменить, сразу после того, как они введены (в Microsoft Access есть средства, позволяющие это сделать). Можно также задать маску ввода, гарантирующую правильный формат вводимых данных. Однако не следует налагать слишком строгие ограничения на формат вводимых данных там, где этого не требуется. Если введенные данные не соответствуют формату, определенному ограничениями, то возможно, лучше оставить окончательное решение за пользователем. Предложите ему вариант форматирования, близкий к правильному, и предоставьте возможность повторно ввести данные в том формате, в котором он сочтет нужным. Конечно, не исключено, что введенные данные будут просто неверны, но все же это бывает редко. Например, пользователь, который ввел в базу данных телефонный номер 9-9999-99999-99, возможно, просто пользовался новой системой офисной связи.

Ограничения, налагаемые на длину данных

Ограничения, налагаемые на длину данных (в особенности на поля, допускающие ввод букв алфавита и других нецифровых символов) вызывают больше всего трудностей. Сколь бы щедры вы ни были при задании максимальной длины вводимой строки, все равно окажется, что этого мало. Иногда, впрочем, проблем, связанных с ограничением длины данных, удается избежать, используя поля типа *Character.*

Максимальная длина записи, которую можно ввести в такое поле, составляет 255 символов, Используя тип данных переменной длины, вы, несомненно, решите большинство проблем, связанных с такими ограничениями. В SQL Server можно явно задать тип данных VARCHAR для поля таблицы базы данных. И SQL Server, и Microsoft Jet. сохраняя данные этих типов, выделяют место на диске только для символов, содержащихся в этих полях. Таким образом, хранение данных переменной длины не приводит к дополнительному расходу дискового пространства.

Но поля переменной длины можно использовать не везде и не всегда. Во-первых, они не применимы, когда вводимое значение должно содержать строго определенное число символов. Например, индивидуальные регистрационные номера системы социального страхования в США всегда состоят из девяти символов. Если пользователь введет индивидуальный номер, состоящий из десяти символов, то проблема будет не в ограничении на длину данных, а в том, что введенные данные неверны. Допускать ввод таких значений в базу данных нельзя.

Кроме того, SQL Server обновляет данные переменной длины гораздо медленнее, что может снизить производительность системы, впрочем, в большинстве случаев незначительно. Но в системах, где время обработки данных играет решающую роль, и особенно в тех, где данные интенсивно обновляются, рекомендуется использовать данные, имеющие постоянную длину. (Особо отмечу, что добавление новых данных постоянной и переменной длины сказывается на производительности одинаково, разница есть только при обновлении данных.)

Хотя использование полей, допускающих очень большую длину, весьма удобно с точки зрения пользователей, вводящих данные, в других ситуациях оно может принести вред. Данные будут отображаться на экранах форм и в отчетах в неудобном для восприятия формате; кроме того, могут возникнуть серьезные трудности при поиске в базе данных конкретной записи. Поэтому везде, где невозможно использовать длинные данные, следует предусмотреть методы обработки данных, не соответствующих заданному формату. Например, вы ограничили число символов в поле, где хранится фамилия клиента либо название компании. Пользователь пытается ввести в это поле длинное название компании, но оно не умещается полностью. И тогда, разумеется, пользователь решит название сократить. Он может опустить артикли, использовать заранее оговоренные сокращения: «Co.». означающее «Company» или «Inc.», означающее «Incorporated»,

Определяя правила, старайтесь исключить использование разных вариантов сокращений. Например, определите такое правило: если в названии компании встречается слово «Company», оно сокращается до «Co.», а все остальные символы, следующие за этим словом, отбрасываются. Тем самым вы исключите вероятность, что название компании «The Really, Really Long Name Company, Incorporated» будет введено разными пользователями один раз как «Really, Really Long Name Co.», а другой — как «Really, Really Long, Inc.». Ведь подобные ошибки могут привести к тому, что одна и та же компания будет учтена в базе дважды.

Значения *Null*

Отсутствующие значения — еще одна проблема нарушения внутренних ограничений, с которой часто сталкиваются пользователи. В предшествующих главах я уже говорила, что применение значений *Null* оправдано, если значение какой-либо величины может быть не определено или отсутствовать. Разумеется, отсутствие того или иного параметра не должно привести к тому, что данные окажутся совершенно непригодными. Если же вы отказываетесь использовать значения *Null* в системе, тщательно продумайте, как будет действовать пользователь, если ему нужно добавить новую запись в базу данных, а некоторые данные, которые требуется ввести, ему неизвестны.

Ключ к разрешению этой проблемы — определить, когда именно будут использоваться вводимые данные. Из анализа рабочих процессов системы вам должно быть известно, что какая-то часть вводимой информации может потребоваться прежде, чем определенная задача будет завершена. Но это отнюдь не означает, что все данные, используемые во всех системных процессах, непременно нужно ввести за один раз, когда в базу данных добавляется новая запись.

Рассмотрим пример. Чтобы выплатить зарплату новым сотрудникам, бухгалтер должен знать наименования банков, клиентами которых являются эти сотрудники, и номера их банковских счетов или кредитных карточек. Значит, в поля таблиц базы данных, которые содержат эти сведения, соответствующая информация должна быть введена до того, как наступит срок выплаты первой зарплаты. Ну, а если новый сотрудник не сразу сообщит полную информацию о себе и своем банковском счете пользователю, вводящему в систему эти сведения? Несовершенная система заблокирует ввод информации, пользователь не сможет создать новую запись в базе данных и ввести ту информацию о новом сотруднике, которой он располагает на текущий момент, или продолжить выполнять какую-то другую работу.

Помимо неудобств для вводящего информацию пользователя, отсутствие в системе первичных данных о новом сотруднике может привести к задержке других формальных процедур: например, незарегистрированный сотрудник не получит пропуск на вход в здание. Очевидно, что для регистрации нового сотрудника и выдачи ему пропуска совсем не требуется информация о его банковском счете. Эти данные должны быть введены в систему к моменту наступления некоего события (даты выплаты зарплаты), а в первый день выхода сотрудника на работу они, скорее всего, не понадобятся.

Итак, не нужно требовать невозможного — чтобы в систему вводились сразу все данные, относящиеся к некоторой сущности. Всему свое время, и ввод отсутствующей информации в большинстве случаев можно отложить.

Ну, а если вы принципиально против значений *Null,* даже допустимых временно? Тогда предложите пользователю, когда он не может ввести необходимую информацию, использовать значение по умолчанию.

Существует несколько способов реализации значений по умолчанию в системе. Один из них — определить значение по умолчанию на уровне схемы базы данных. Тогда выбранное значение по умолчанию будет использоваться везде, где пользователь пропустит данные при вводе.

Другой способ — предоставить пользователю возможность выбрать один из нескольких вариантов значений, например *Unknown* (Неизвестно), *Not Applicable* (Не используется) и *Yef To Come* (Будет введено позже). В некоторых случаях система может сгенерировать подходящее значение по умолчанию самостоятельно.

Ограничения, налагаемые на диапазон возможных значений

Задание диапазона возможных значений — еще один из видов ограничений на уровне атрибутов. Как и в случае со значениями *Null,* сталкиваться с нарушениями этих ограничений приходится весьма часто. При этом пользователей и разработчиков подстерегают множество подводных камней.

Некоторые ограничения диапазона значений задаются в явном виде при определении типа данных — например, короткое целое число не может быть больше 255. Если ограничение диапазона возможных значений определяется типом данных, то единственное, что можно сделать, когда введенное значение выходит за заданный интервал — выдать пользователю сообщение, разъясняющее ситуацию.

Если же подобные ограничения неприемлемы, и реальные значения, используемые в системе, превышают предельные, обусловленные типом данных, измените схему базы, выбрав другой тип данных, с более широким диапазоном допустимых значений. Но ни в коем случае не следует выполнять такие изменения, когда пользователи активно работают с системой — необходимо на время приостановить работу.

Если же ограничение диапазона возможных значений определено как правило проверки или ограничение CHECK в схеме базы данных, или же реализовано в пользовательском приложении, а не непосредственно в схеме базы данных, оно, скорее всего, является не внутренним ограничением, а бизнес-правилом. И здесь у разработчиков появляется гораздо большая свобода действий. О бизнес-правилах и их реализации мы подробно поговорим далее в этой главе.

Ограничения на уровне сущностей и ссылочная целостность

Как я уже говорила, ограничения на уровне сущностей гарантируют возможность уникальной идентификации каждой записи в таблице, а ограничения, диктуемые ссылочной целостностью, исключают возможность ссылки на несуществующие записи в той же или другой таблицы базы данных.

Эти ограничения играют важную роль в системе, но поскольку они относятся к внутренним механизмам поддержки целостности данных, их следует реализовать максимально незаметно для пользователя. Как правило, это делают, предоставляя пользователям возможность выбрать нужное значение из списка, а не вводить его вручную. К сожалению, такой подход не всегда рационален, поскольку некоторые списки могут содержать слишком много элементов.

Если использовать списки невозможно, постарайтесь спроектировать систему так, чтобы проверка выполнялась сразу же после ввода данных пользователем, и в случае нарушения ограничений соответствующее сообщение выдавалось незамедлительно.

Наиболее частый пример нарушения ограничений на уровне сущности — ввод записи, повторяющей уже имеющуюся в базе данных. В этом случае разумно вывести пользователю запись, уже имеющуюся в базе данных, или соответствующие поля, которые содержат повторяющиеся данные. Пусть он сам решит, действительно ли новая запись дублирует имеющуюся (рис. 16-1).

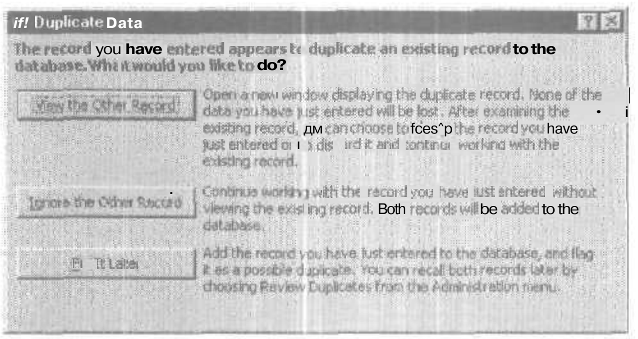


Рис. 16-1. Диалоговое окно, сообщающее что новая запись в базе, возможно, повторяет уже имеющуюся

Реализуя эту возможность, обратите особое внимание на ясность интерфейса. Он должен содержать все необходимые комментарии, разъясняющие пользователю возможные варианты действий. Позаботьтесь, чтобы данные, введенные пользователем, сохранились до окончательного разрешения вопроса с повторяющейся записью, Нельзя допустить, чтобы введенные пользователем данные пропали или были заменены другими без его согласия. Возможно такое решение: вывести уже существующую в базе данных запись в отдельном окне и позволить пользователю выбрать: заменить ли введенную им запись уже имеющейся или оставить новую запись без изменений. Обратите внимание на следующую особенность; окно на рис. 16-1 позволяет пользователю продолжать ввод данных, не просматривая запись, которая предположительно является «двойником» вводимой. Возможно, пользователь знает, что такая запись уже существует в базе, и лишний раз напоминать ему об этом не нужно.

Еще один распространенный случай нарушения ограничений на уровне сущности и ссылочной целостности — пользователь при вводе данных пропускает или оставляет пустым поле первичного ключа. Чтобы этого избежать, часто используют уникальные значения первичного ключа, сгенерированные системой. Поля *AutoNumber* в Access и *Identity* в SQL Server позволяют определить естественный внешний ключ таким образом, что пустые или повторяющиеся значения, введенные пользователями, не приведут к нарушению целостности на уровне сущности или ссылочной целостности. Если по какой-либо причине вы не можете использовать поля *AutoNumber* или *Identity* в системе, позаботьтесь, чтобы при вводе данных пользователи обязательно заполняли все поля первичного ключа. Обычные значения по

умолчанию здесь использовать нельзя, поскольку они определяются для каждой таблицы отдельно. Конечно, можно применить значения, сгенерированные системой на этапе исполнения. Но если вы собираетесь использовать произвольные значения для первичного ключа, почему не применить поле *AutoNumber*? Единственный альтернативный вариант — всякий раз обращаться к пользователю, чтобы он ввел нужное значение.

При нарушении ссылочной целостности пользователь пытается обратиться к записи, которой на самом деле не существует. Такая ситуация может возникнуть случайно — например, когда пользователь неправильно вводит имя. Ну, а если пользователь просто не знает, что записи, к которой он обращается, не существует, или же еще не ввел соответствующие данные в систему? Диалоговое окно на рис. 16-2 иллюстрирует способ разрешения подобной ситуации. Пользователь может выбрать один из четырех вариантов:

- создать новую запись в базе данных и сразу ввести всю необходимую информацию;
- создать новую запись и ввести лишь часть данных, а впоследствии обновить запись;
- обратиться к другой записи;
- исправить несуществующую ссылку позднее.

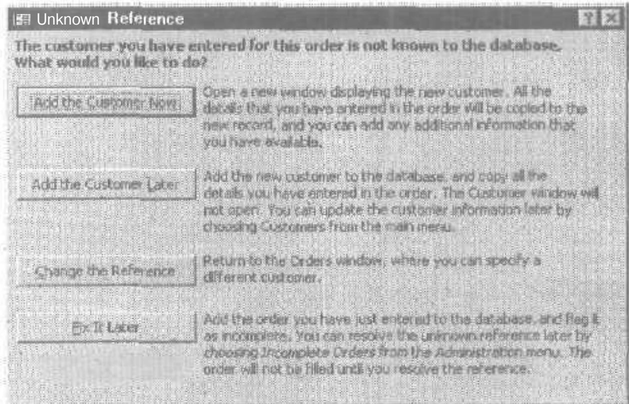


Рис. 16-2. Диалоговое окно, позволяющее разрешить ситуацию, когда пользователь обращается к несуществующим данным

Отмечу особо, что предоставить пользователю возможность на время отменить или игнорировать ограничение нельзя: тогда база данных будет содержать неверную информацию.

Если нельзя предоставить пользователю возможность выбрать нужное значение из списка потому, что такой список будет слишком велик, выведите на экран список значений, в большей или меньшей степени похожих на введенные данные. Для создания такого списка применяют разные алгоритмы — от простого SQL-оператора LIKE до поиска в базе данных соответствующих величин с использованием критерия SOUNDEX.

В некоторых случаях совершенно не нужно отображать пользователю диалоговое окно, в котором он может выбрать несколько вариантов действий. Поддержку ограничений на уровне сущности и ссылочной целостности лучше реализовать так, чтобы как можно меньше отвлекать пользователя от работы. Например, заранее известно, что при отсутствии в базе соответствующей записи пользователь, скорее всего, добавит новую и введет недостающие данные позднее и при этом в системе реализованы механизмы, позволяющие отменить ошибочные действия. Тогда надо разрешить пользователю делать это, не требуя лишних подтверждений. После того как пользователь создаст новую запись в базе данных, пусть он увидит сообщение о том, что в базу данных была добавлена новая запись. Такое сообщение можно поместить в командной строке или, чтобы привлечь внимание пользователя, в окне сообщения. Но какой бы способ вы ни выбрали, помните, что чем меньше вы отвлекаете пользователя во время ввода данных, тем более удобной он сочтет вашу систему.

Другой случай, когда нарушаются ограничения ссылочной целостности — попытка удалить или изменить запись, на которую в базе данных имеется ссылка. Например, пользователь может попытаться удалить запись о покупателе, сделавшем заказ, который еще не выполнен, или изменить значение уникального номера *Product ID* для продукта, на который имеется ссылка в таблице *OrderItems*.

Microsoft Jet поддерживает механизм каскадных обновлений и удалений данных, что позволяет обновлять и удалять записи, на которые имеются ссылки, без дополнительных запросов системы к пользователю. В SQL Server можно реализовать те же возможности при помощи триггеров. Каскадные обновления и удаления данных — очень удобный способ автоматического разрешения проблем при нарушении ссылочной целостности. Я рекомендую использовать его везде, где это возможно (разумеется, решение принимают системные аналитики и разработчики). Однако если вы предоставляете пользователю возможность выбирать, что надлежит делать при нарушении ссылочной целостности, не забудьте подробно разъяснить ему, к чему приведет каждое из предложенных действий.

Диалоговое окно на рис. 16-3 предоставляет пользователю возможность отменить операцию удаления, удалить текущую запись и все связанные с ней записи (то есть выполнить каскадное удаление) или переопределить ссылки таким образом, чтобы связать все открытые заказы с другим покупателем. Обратите внимание; окно содержит предупреждение, что информацию об открытых заказах нельзя удалить из базы данных (это бизнес-правило). Пользователю предоставляется возможность отменить все текущие заказы покупателя (фактически, нарушить бизнес-правило) или просмотреть их.

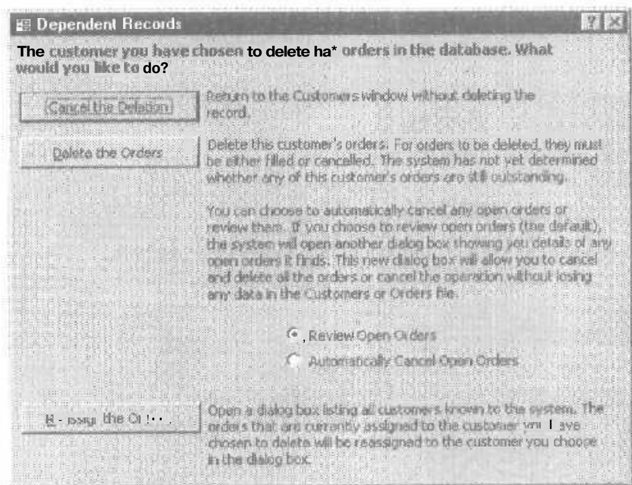


Рис. ***16-3. Пользователь получает подробные разъяснения и может выбрать вариант действий***

Если такой вариант реализации вас не устраивает, можете вывести на экран пользовательского компьютера диалоговое окно с запросом на подтверждение выполняемой операции удаления, в котором будут показаны все текущие заказы или все заказы этого покупателя. Ваша цель — дать пользователю всю информацию для принятия осознанного, обоснованного решения.

Бизнес-правила

Итак, мы рассмотрели внутренние ограничения, защищающие структуру баз данных и целостность данных. Когда действия пользователя приводят к нарушению внутренних ограничений, число вариантов действий, предпринимаемых для устранения этой ситуации, весьма ограничено. Данные должны, пусть даже не сразу, а при наступлении

определенного события, удовлетворять требованиям реляционной модели. Но нарушение ограничений, налагаемых бизнес-правилами — совсем другой случай.

Как я уже упоминала, бизнес-правила чаще определяются предметной областью, а не вытекают непосредственно из реляционной модели данных. Напомню, что модель данных представляет собой упрошенную модель некоей области реального мира, и задача системного архитектора — как можно более полно отразить в создаваемой модели все относящиеся к этой области аспекты. Но несмотря на все усилия, вам вряд ли удастся создать идеальную модель. Даже если на начальной стадии реализации она была безупречна, бизнес-правила впоследствии могут измениться. Недостаточно добиться согласованного поведения системы — вы должны предусмотреть простые и изящные механизмы разрешения нестандартных ситуаций.

Чаще всего пользователи вводят данные, которые система не может корректно обработать, либо случайно, либо если реальные условия не соответствуют системной модели.

ПРИМЕЧАНИЕ Третий случай — это когда пользователи намеренно вводят неправильные данные с целью дестабилизировать работу системы. Самые строгие бизнес-правила едва ли помогут защитить систему от умышленной порчи данных. К счастью, подобные случаи очень редки.

Случайные ошибки при вводе данных

Средства для исправления случайных ошибок, допущенных при вводе данных, должны быть простыми и удобными, поскольку подобные ситуации весьма не редки. Разумеется, нужно объяснить пользователю, как исправить ошибки. Альтернативные варианты следует предусмотреть везде, где только возможно. Рассмотрим диалоговое окно, отображаемое при неверном вводе даты: когда цифры, означающие месяц и число, переставлены местами (рис. 16-4). Система предлагает пользователю наиболее вероятный вариант даты в правильном формате, и если он подходит, то для исправления данных достаточно всего лишь щелкнуть мышью.

Когда пользователи случайно вводят неверные данные, это, как правило, либо результат оплошности, либо непонимания, какие именно данные и в каком формате требуется ввести. Диалоговое окно на рис. 16-4 объясняет, что представляет собой поле *DeliveryDate* (Дата доставки). Кроме того, пользователь может получить справочную информацию, щелкнув кнопку Help. О том, какие механизмы помо-

щи пользователю можно реализовать в системе, мы поговорим подробнее в главе 18.

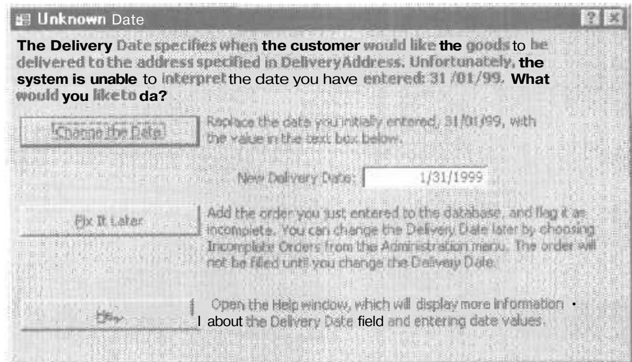


Рис. 16-4. Диалоговое окно сообщает пользователю о неверном вводе даты и предлагает возможные варианты действий

Модель системы и реальность

Чаще всего к вводу некорректных данных приводит несоответствие модели реальным условиям. Например, пользователь пытается ввести данные о заказе, который уже выполнен и доставлен клиенту, но по каким-то причинам еше не зарегистрирован в системе. Тогда дата доставки будет предшествовать дате оформления заказа, что, по всей вероятности, нарушит бизнес-правила. Если не предусмотреть стандартный способ разрешения такой ситуации и не дать четких указаний, что сделать в этом случае, пользователи могут ввести в поле, где должна быть указана дата доставки, произвольную дату, которая не противоречит бизнес-правилам и, следовательно, будет принята системой. В результате в базу данных попадет информация, не соответствующая действительности.

Итак, при проектировании системы и особенно при определении бизнес-правил следует проявлять особую осмотрительность, стараясь предугадать как можно больше нестандартных ситуаций и по возможности предоставить пользователям способы выйти из них без нарушения целостности данных. На рис. 16-5 показано диалоговое окно, которое появляется на экране пользовательского компьютера, если дата, введенная в поле *DeliveryDate*, предшествует дате, введенной в поле *OrderDate* (Дата оформления заказа). В этом примере для ввода данных в поле *OrderDate* используется элемент управления, не позво-

ляющий вводить или изменять дату вручную; по умолчанию подставляется текущая дата.

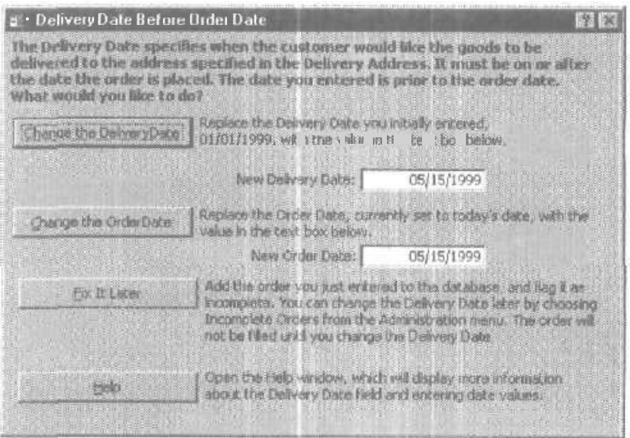


Рис. 16-5. Это диалоговое окно позволяет оформить заказ «задним числом»

Однако не все бизнес-правила можно (и нужно) обходить, если введенные данные им не соответствуют. Некоторые бизнес-правила невозможно отменить или игнорировать, и надо обязательно знать и учитывать, в каком случае и кем правило может быть нарушено. Рассмотрим пример: пользователь попытался ввести в базу данных информацию о шестом сотруднике, подчиняющемся менеджеру, который уже руководит группой из пяти человек. Однако бизнес-правило, определенное для этой системы, жестко ограничивает число сотрудников, подчиняющихся одному менеджеру, пятью человеками (рис. 16-6).

В данном случае пользователь, вводивший информацию в базу данных, не имел прав отменить действующее бизнес-правило. Однако в системе есть другое диалоговое окно, в котором сотрудник, обладающий соответствующими правами (например, управляющий или менеджер) может ввести пароль и код, авторизовав таким образом ввод данных, нарушающих бизнес-правило. На тот случай, когда управляющий или менеджер отсутствует и пользователь, вводящий информацию, не может сразу к нему обратиться, система предусматривает возможность сохранить только что сделанную запись, отложив ее авторизацию на некоторое время. Даны инструкции, как выполнитьавторизацию.

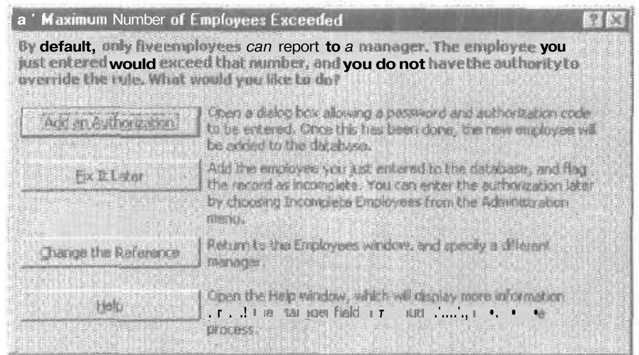


Рис. *16-6, Диалоговое* ***окно разъясняет, что только авторизованное лицо может отменить действующее бизнес-правило***

Возможность обойти бизнес-правила сделает работу с системой намного удобней, но приведет к усложнению модели данных. В приведенном примере в таблицу *Employees* (Сотрудники) придется добавить дополнительное поле с информацией об авторизации записи и включить в ограничение целостности данных ссылку на код авторизации записи.

Вы также можете реализовать механизм, позволяющий временно отменить неавторизованные записи. Один из вариантов — добавить в таблицу *Employees* поле, содержащее булево значение, которое указывает, является ли данная запись правильной. Этот флаг может использоваться для поиска записей, в которых при вводе пропущены некоторые данные (поиск проводится, когда пользователь закончил ввод данных и просматривает записи, исправляя ошибки и добавляя недостающую информацию), или для того чтобы отфильтровать неавторизованные записи при составлении отчетов.

Часто нужно объяснить, почему данная запись некорректна. Для этого следует добавить еще одно поле с соответствующим кодом: например, код MA будет означать, что запись должна быть авторизована менеджером, а код CC — что для данного заказа еще не проверен кредит. Второй подход обеспечивает большую гибкость при обработке записей. Но использование полей, содержащих код, усложнит систему, если для каждой записи использовать разные значения кода. В этом случае рационально использовать другую таблицу, содержащую все возможные значения кодов и находящуюся с первой в связи «один ко многим».

Большинство изменений, которые нужно внести в систему для поддержки авторизации записей, несложны и вполне очевидны. Од-

нако влияние, которое они могут оказать на систему в целом, зачастую трудно просчитать заранее. Например, следует ли включать в отчеты неавторизованные записи? Если нужно выборочно исключать эти записи из отчетов, коды, идентифицирующие причину отсутствия авторизации, несомненно, нужны, поскольку именно они позволяют составлять простые запросы, указывая, какие записи следует исключить. Если же неавторизованные или неполные записи нужно включать в отчеты, их при этом следует как-то особо выделять: например, сгруппировав и определив отдельный заголовок.

Если неавторизованные записи (или некоторые из них, отобранные по специальному признаку) должны быть исключены из отчетов, *нужно*, очевидно, исключить и все записи, связанные с ними. Возьмем в качестве примера случай, когда у менеджера не пять, а шесть подчиненных, причем запись о шестом еще не авторизована. Нужно ли исключить из отчета только запись о шестом подчиненном или все записи о подчиненных этого менеджера? Все эти особенности следует учесть, прежде чем возможность обойти бизнес-правило будет реализована. Конечно, такая возможность может показаться пользователям весьма удобной и полезной, но она, как правило, ведет к усложнению системы и потому не всегда оправдана, В простых системах целесообразно просто запретить ввод записи, которая не удовлетворяет бизнес-правилам, при этом пользователи должны разрешать ситуацию вне компьютерной системы. Но выбрав этот вариант, предупредите пользователя: предусмотрите выдачу соответствующего сообщения в диалоговом окне, где содержатся результаты проверки данных.

Итоги

В этой главе мы обозначили ту роль, которую играют в компьютерной системе средства поддержки целостности базы данных. Существуют два класса системных ограничений: внутренние ограничения, регулирующие физическую структуру данных и вытекающие непосредственно из реляционной модели, и бизнес-правила, определяемые предметной областью. Внутренние ограничения представляют собой жесткие правила, нарушение которых может привести к неработоспособности системы. Нарушение же бизнес-правил зачастую не приводит к подобным последствиям, и механизмы, предусматривающие возможность их обходить, могут оказаться весьма полезными. Мы также рассмотрели средства, позволяющие разрешить ситуации, когда нарушаются бизнес-правила,

В следующей главе мы подробно поговорим о процессе создания отчетов на основе имеющихся в системе данных.

Создание отчетов

Отчеты и средства, позволяющие их создавать — очень важная, едва ли не важнейшая, часть системы. Ведь разнородные сведения, хранимые в базе данных, еще не представляют ценности для пользователей, им нужны систематизированные данные. Эта глава посвящена средствам, позволяющим получать из довольно хаотичной массы данных, хранимых в памяти компьютерной системы, связную информацию.

ПРИМЕЧАНИЕ Под отчетом я отнюдь не подразумеваю всего лишь вывод на печать некоей информации, представляющей интерес для пользователя. Я вкладываю в это понятие гораздо более широкий смысл, а именно: предоставление пользователю информации на основе данных, хранимых в компьютерной системе, в удобном для восприятия виде. То есть не только как распечатанный отчет, но и как набор результатов, отображаемый в виде таблицы или списка на экране компьютера.

На заре эпохи компьютеризации, когда стоимость компьютера во много раз превышала годовую зарплату среднего клерка и время, затраченное на выполнение вычислений, ценилось поистине на вес золота, был весьма распространен простой подход к созданию отчетов: компьютерные системы выдавали несколько стандартных отчетов с заданной периодичностью. Если же требовалось изменить стандартную форму отчета или создать новую, приходилось долго осаждать сотрудников отдела информационных систем, поскольку очередь жаждущих получить от них такую программу, выстраивалась на несколько лет вперед. Задание создать «что-то вроде вот этого стандартного отчета, только отсортированного по полю *Customer*», или «отчет, где отражались бы суммарные значения по регионам», проще было дать секретарю, чем программисту.

Сегодня ситуация изменилась с точностью до наоборот. Компьютеры день ото дня все дешевеют, а рабочее время квалифицированного секретаря ценится весьма дорого. Теперь, если пользователю нужно получить «что-то вроде вот этого стандартного отчета, только с некоторыми изменениями...», он, несомненно, обратится не к секретарю, а к компьютерной системе. Соответственно, и у разработчиков баз данных прибавилось забот. Им теперь приходится удовлетворять требования пользователей в минимальный срок. Но ведь это гораздо лучше, чем корпеть над пишущей машинкой, не правда ли?

В компьютерных системах, основанных на базах данных, как правило, сочетают два подхода к созданию отчетов. Первый — разработчики заранее определяют, какие отчеты понадобятся пользователям, и предоставляют им возможность генерировать по готовым шаблонам так называемые *стандартные отчеты* (standard reports). Второй — пользователям предоставляются средства, при помощи которых они могут составлять и генерировать самостоятельно *пользовательские*, или *произвольные отчеты* (ad hoc reports).

В этой главе мы рассмотрим и стандартные, и произвольные отчеты. Но сначала я хочу более подробно остановиться на механизмах сортировки, поиска и фильтрации данных, используемых при создании отчетов.

Сортировка, поиск и использование фильтров

Средствами операторов SQL выполнить поиск, сортировку и фильтрацию данных достаточно легко, соответствующие критерии задают с помощью ключевых слов WHERE или ORDER BY оператора SELECT. Но чтобы пользователи могли задавать критерии самостоятельно, необходимо предусмотреть промежуточный слой программных компонентов. В самом деле, не вынуждать же их составлять запросы к данным при помощи языка SQL, логика которого сильно отличается от конструкций языка, на котором мыслят и общаются обычные люди. Поэтому начинающие пользователи так часто ошибаются при составлении SQL-запросов.

Приведу только один пример. Допустим, для составления отчета менеджеру по продажам нужен список всех дистрибьюторов компании в штатах Вайоминг и Флорида. Очевидно, что продумывая этот отчет, менеджер пожелает включить в список всех дистрибьюторов в штате Флорида и всех дистрибьюторов в штате Вайоминг. Но в операторе SELECT языка SQL условие WHERE, формирующее список, будет выглядеть примерно так: WHERE State = «Wyoming» OR State = «Florida». Как видите, здесь используется логический оператор OR

(или), а не логический оператор AND (и). Подобные лингвистические тонкости могут сбить с толку любого новичка!

Конечно, обучить пользователей основам языка SQL не так уж сложно. Я знаю нескольких разработчиков баз данных, которые действуют именно так, и весьма успешно. Но все же гораздо лучше предоставить пользователям интуитивный интерфейс, позволяющий составлять запросы, — такое решение не только значительно облегчит им работу с системой, но и позволит вам сэкономить силы, затрачиваемые на составление документации.

К счастью, чтобы разработать подобный интерфейс, не нужно начинать «с нуля». В Microsoft Access уже имеются соответствующие интерактивные средства, позволяющие пользователям создавать запросы к данным с использование критериев SQL. Конечно, прямой перенос механизмов пользовательского интерфейса из одного приложения в другое не всегда возможен, однако само наличие подобных средств оказывает большую помощь разработчику, позволяя ему использовать уже готовый образец. В Microsoft Visual Basic аналогичных средств не существует, однако их реализация займет не так уж много времени.

Сортировка данных

В Access реализован простой и удобный интерфейс, позволяющий быстро отсортировать данные в нужном порядке. Для указания способа сортировки имеются две команды — Sort Ascending (Сортировка в порядке возрастания) и Sort Descending (Сортировка в порядке убывания). Щелкнув мышью элемент интерфейса, по которому нужно выполнить сортировку, пользователь затем выбирает соответствующую команду в меню Records (Записи) Можно также щелкнуть соответствующую кнопку на панели управления.

Фильтр по выделенному фрагменту в одном поле

Для фильтрации данных в одном поле в Access имеются две команды: фильтр по выделенному фрагменту Filter By Selection (Фильтр по выделенному) и фильтр исключения выделенного фрагмента Filter Excluding Selection (Исключить выделенное). Эти команды похожи на те, что используются для сортировки данных — Sort Ascending и Sort Descending. Когда пользователь выделяет все содержимое одного из полей формы или задает фокус ввода на это поле и затем выбирает команду Filter By Selection, Access применяет фильтр по этому полю, отбирая только те записи, которые в точности соответствуют выделенному значению. На SQL условие WHERE, эквивалентное этому

фильтру, можно записать таким образом: **WHERE** <имя_поля> = <выделенное_значение>.

Если пользователь выделяет только несколько первых символов в поле и применяет фильтр по выделенному фрагменту, Access выбирает только те записи, которые начинаются с выделенных символов в этом поле. Рассмотрим пример — базу данных *Northwind*, поставляемую вместе с Access. Если пользователь выберет первые три символа *в* названии продукта Chartreuse verte и применит фильтр по полю *Product Name,* то на SQL условие WHERE, эквивалентное этому фильтру, можно сформулировать так: WHERE [Product Name] **LIKE** «Cha*». Этот фильтр ограничит результирующий набор тремя записями с соответствующими названиями продуктов: Chartreuse verte, Chai и Chang.

Если же пользователь выделяет несколько символов не в начале поля, а где-нибудь в середине или в конце и применяет фильтр, то Access выбирает все записи, которые содержат выделенные символы в этом поле. Например, если выбрать символы «ar» в названии продукта Chartreuse verte, то эквивалентное выражение **WHERE** на SQL запишется так: WHERE fProduct Name] LIKE «*ar*». Результат применения этого фильтра в базе данных *Northwind* содержит десять записей (рис. 17-1).

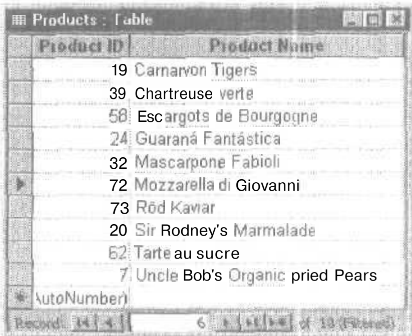


Рис. 17-1. Результат применения фильтра по выделенному фрагменту в поле Product Name

---

ПРИМЕЧАНИЕ Говоря об эквивалентах языка SQL, я имею в виду, лишь что соответствующий SQL-запрос вернет те же результаты, что и применение фильтра. Трудно с уверенностью сказать, что происходит на самом деле, когда пользователь выбирает команду Filter By Selection — составляет ли Access SQL-запрос с использованием оператора SELECT

или использует какие-то другие методы обработки данных. К сожалению, мне неизвестны описания внутренних механизмов выполнения команд, позволяющих фильтровать данные в Access.

Фильтр по заданным значениям в нескольких полях

Команда Filter By Selection предоставляет пользователю простой и удобный интерфейс, но у нес есть и недостаток — данные с ее помощью можно фильтровать только по одному полю. Чтобы еще более ограничить результирующий набор записей, пользователю придется выполнить команду Filter By Selection еще раз, задав новый критерий и используя набор результатов, полученный в результате применения первого фильтра. Это неудобно, особенно если команду приходится выполнять несколько раз, каждый раз используя новый критерий. Для пользователей, которые нуждаются в более мощных средствах фильтрации данных, в Access имеется команда Filter By Form (Изменить фильтр). На рис. 17-2 показан вид формы *Customers* из базы данных *Northwind* после того как в меню Records была выбрана эта команда.

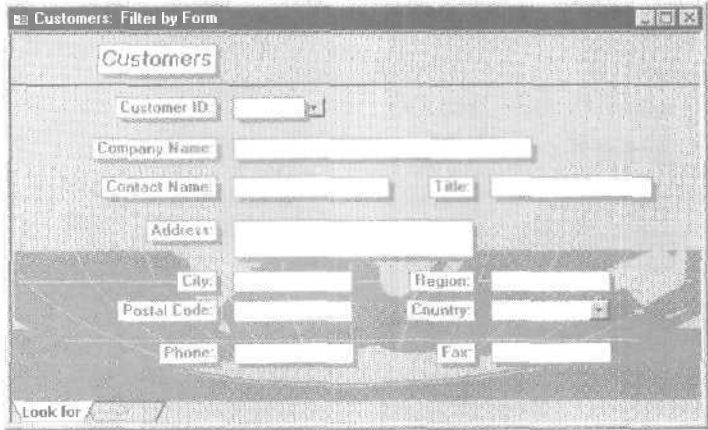


Рис. 17-2. Команда Filter By Form позволяет фильтровать записи по нескольким полям

Окно, открываемое по команде Filter By Form, позволяет задать значения полей, используемых для фильтров. В нем имеется несколько вкладок, на каждой из которых можно указывать значения полей. Все значения, заданные на вкладке Look For (Найти), будут объединены логической операцией И (AND), а ко всем параметрам фильтра, указанным на отдельных вкладках Or (Или), будет применена ло-

гическая операция ИЛИ (OR). Данная концепция построения интерфейса, хотя и вполне продуманная, порой все же трудна для восприятия пользователями. Дело в том, что использование логических операций И и ИЛИ для фильтрации наборов данных и значение, вкладываемое в союзы «и» и «или» в большинстве европейских языков, различаются.

По умолчанию в диалоговом окне, открываемом по команде Filter By Form, каждое текстовое поле исходной формы представлено как комбинированное окно, содержашее все текущие значения, хранимые в этом поле. Вывод списка, содержащего все значения, можно отключить. Для этого следует на вкладке, позволяющей задать свойства текстового поля, для параметра Filter Lookup (Применение автофильтра) выбрать значение Never (Никогда). Когда при задании критериев фильтра используется вывод полей исходной формы в виде комбинированного окна, при фильтрации записей в результирующий набор включаются записи, в которых значения соответствующих полей точно совпадают со значением, выбранным пользователем в комбинированном окне. Если же автофильтр не используется (то есть для параметра Filter Lookup выбрано значение Never), пользователь может ввести в это поле значение, используемое для поиска записей, точно совпадающих с ним, или выражения для поиска: например LIKE «CHA*» или IS NOT NULL. Решение, использовать ли автофильтр или предоставить пользователям право самостоятельно задавать критерии поиска, зависит от того, сколько записей будет содержать комбинированное окно автофильтра (очевидно, заставлять пользователя просматривать 100 тыс. записей не имеет смысла), а также от необходимой гибкости интерфейса.

Расширенный фильтр

Команда Filter By Form и интерфейс, связанный с ней, в большинстве случаев удовлетворяют потребности и пользователей, и разработчиков. Но если пользователи знакомы со средствами создания запросов в Access, или если вы решили предоставить пользователям возможность выбирать значения для фильтра из комбинированного списка и вводить выражения, используемые в качестве критериев фильтра, то больше подойдут средства расширенного фильтра. Для этого в меню Records выберите команду Advanced Filter/Sort (Расширенный фильтр) и задайте параметры фильтра в соответствующих полях открывшегося диалогового окна (рис. 17-3).

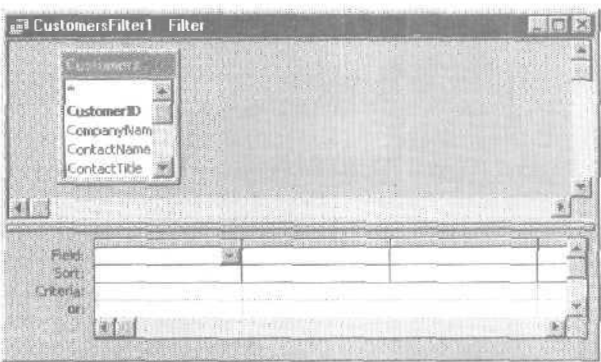


Рис. 17-3. Диалоговое окно, открывающееся при выборе команды Advanced Filter/Sort при работе с формой Customers базы данных Northwind

Некоторые элементы окна расширенного фильтра совпадают с элементами окна запроса, открываемого в режиме конструктора. Однако в отличие от интерфейса построения запроса, интерфейс расширенного фильтра позволяет задавать лишь параметры, используемые в выражениях WHERE и ORDER BY оператора SELECT. Он не разрешает модифицироавать структуру возвращаемого набора записей: например, изменить список полей или выполнить соединение таблицы с другими наборами записей.

Интерфейс средств расширенной сортировки Access достаточно сложен, и создать интерфейс с аналогичными возможностями в Visual Basic нелегко. При необходимости предоставить пользователю такой интерфейс лучше выберите в качестве средства разработки Access или готовый продукт третьей фирмы, который можно встроить в систему.

Средство построения запросов Microsoft English Query

Если при проектировании базы данных используется не механизм СУБД Access, a SQL Server, в качестве средства построения запросов. очевидно, будет применен Microsoft English Query. Кроме пользовательского интерфейса, позволяющего выполнять сортировку и фильтрацию данных, средство English Query предоставляет командный интерфейс, дающий возможность составлять запросы к данным на естественном языке.

Для реализации возможностей English Query в своем приложении свяжите терминологию предметной области, моделируемой в системе, со схемой базы данных. Для этого создайте объект, который в English Query называется *файлом приложения* (application file). Это не

трудно, хотя задача не столь тривиальна, как может показаться на первый взгляд. Создание этого файла похоже на создание предметного указателя справочной системы: нужно составить полный список слов, которые будут использоваться для называния сущностей схемы базы данных и их атрибутов.

Дальнейшая интеграция English Query в приложение базы данных несложна. Приложение просто передает вопрос пользователя, сформулированный на английском языке, механизму обработки запросов English Query, и в результате обработки получает готовый оператор SQL.

Но лишь в теории все выглядит так просто, на практике далеко не всегда удается избежать ошибок и затруднений. Если пользователь, формулируя вопрос, использует термины, не включенные в файл приложения, то в приложение, работающее с базами данных, будет возвращено сообщение об ошибке.

Microsoft English Query — надежное средство, подчас незаменимое для систем баз данных. Его интерфейс, позволяющий формулировать вопросы на английском языке, особенно удобен, если схема базы данных достаточно сложна, а большую часть ее пользователей составят новички.

Стандартные отчеты

Практически в каждой компьютерной системе, работающей с базами данных, есть определенный набор отчетов, которые можно продумать и спланировать заранее. Как правило, большинство таких стандартных отчетов определяется на этапе анализа рабочих процессов в системе. Однако не следует ограничиваться только результатами системного анализа — нужно спросить у пользователей, какие еще отчеты они считают необходимым реализовать.

Отчеты в виде списков и подробные отчеты

Используйте отчеты в виде списков и подробные отчеты. *Отчет ввиде списка* представляет собой списки всех экземпляров сущностей (проще говоря, полный набор всех записей), имеющихся в базе данных. Иногда достаточно составить эти списки в алфавитном порядке и вывести на экраны пользовательских компьютеров. Гораздо чаще, однако, такие списки приходится определенным образом группировать. Например, менеджеру по продажам нужен список покупателей, сгруппированный по штатам, где проживают эти покупатели; или по регионам; или по фамилиям торговых агентов, обслуживавших этих покупателей. Если исходная таблица, на основе которой составляет-

ся список, содержит много записей (например, несколько тысяч), предоставьте пользователю возможность просмотреть и распечатать лишь определенную их часть, которую он выберет самостоятельно. Так, торговому агенту, скорее всего, не понадобится полный список всех покупателей, зарегистрированных в системе — ему нужны будут лишь те, кого обслуживает он сам.

Отчет в виде списка содержит лишь некоторые сведения о каждом экземпляре сущности. *Подробный отчет* содержит всю (или почти всю) информацию, относящуюся к данной сущности. При этом также часто полезно предоставить пользователю возможность выбрать записи, которые он хочет распечатать, с помощью поля со списком, позволяющим выбрать несколько значений одновременно. Этот элемент управления удобен — ведь выбранные значения не обязательно следуют одно за другим, а могут находиться в разных местах списка.

Поле со списком применимо не всегда. Если таблица содержит несколько сотен или тысяч записей, используйте несколько полей со списками, позволяющих пользователю ограничить набор отображаемых записей. Можно использовать и другие элементы управления. Например, спроектируйте пользовательский интерфейс, подобно окну управления печатью Microsoft Word, где имеется возможность задать диапазон выводимых на печать страниц. Тогда пользователи получат право выбирать диапазон выводимых в окне отчета или распечатываемых записей: например, задав диапазон выводимых на печать записей при помощи специального символа в текстовом поле. Выделить же диапазон записей, отмеченный символами тире, или отдельные записи, разделенные запятыми, совсем не сложно.

Отчеты, использующие агрегированные данные

В этих отчетах агрегированные данные различным образом комбинируются и сравниваются между собой. Типичные примеры таких отчетов — распределение объемов продаж по регионам с указанием доли от объема продаж, приходящейся на каждого из торговых агентов; или число покупателей, которые приобрели продукты, относящиеся к указанной категории. Эти отчеты гораздо труднее реализовать, чем отчеты в виде списков и подробные отчеты, однако они намного информативней и поэтому используются чаще.

Отчеты, использующие агрегированные данные, удобно выдавать пользователю в графическом представлении. Для графической интерпретации данных в отчетах можно применить Microsoft Graph и другие графические средства. Но я рекомендую использовать графическое представление данных лишь в дополнение к обычному их пред-

ставлению в виде таблиц или текста и ни в коем случае не заменять графикой данные в текстовом формате. Данные из отчета о числе продаж, представленные в текстовом виде, можно импортировать в другое приложение (например, Microsoft Excel) и использовать для статистического анализа и прочих манипуляций.

Отчеты на основе форм пользовательского интерфейса

Кроме схемы базы данных, при создании отчетов также используют формы пользовательского интерфейса. Некоторые отчеты в системе могут создаваться только на основе форм. Кроме того, на мой взгляд, возможность распечатать не только отчеты, но и большинство форм, имеющихся в системе, весьма полезна, а порой и просто незаменима. Реализовать ее очень легко, а пользователям она наверняка пригодится.

Нередко подробный отчет, в который включена вся информация, относящаяся к отдельной сущности, содержит те же сведения, что и одна из форм системы. В этом случае, очевидно, можно не создавать отдельного отчета на основе этой формы, а использовать подробный отчет. Однако часто подробные отчеты или иначе отформатированы, или содержат гораздо больший объем информации, чем отчеты, полученные на основе отдельных форм пользовательского интерфейса. В таких случаях лучше реализовать оба вида отчетов — и подробные, и полученные из форм. Для печати разных видов отчетов можно предусмотреть разные возможности: например, по умолчанию выводить на печать отчет, полученный на основе данной формы (для его печати пользователю нужно щелкнуть соответствующую кнопку на панели инструментов), а для печати подробного отчета — использовать специальную команду меню.

Интерфейс для создания отчетов

Разработка средств для создания отчетов в пользовательском интерфейсе не доставляет значительных трудностей. Если для создания отчета используется команда Print Report (Печать отчета), то существует несколько способов включить ее в пользовательский интерфейс: как команду меню, как кнопку на панели управления или как кнопку в пользовательской форме. Как правило, в системе предусмотрено несколько видов отчетов, и некоторые из них выводятся на печать из одной формы. Поэтому команда, при помощи которой отчет выводится на печать, должна ясно объяснять пользователю, какой именно отчет будет выведен при ее использовании.

Удобней всего включить команду в состав меню. В этом случае вполне достаточно указать названия отчетов в меню Reports (Отчеты). Если для печати отчетов используется кнопка на панели управ-

ления, название отчета можно вывести на всплывающей подсказке этой кнопки. Не следует указывать во всплывающей подсказке или в названии кнопки только название отчета, который будет распечатан — обязательно укажите и команду, выполняемую при щелчке кнопки. Так, всплывающая подсказка или надпись на кнопке Print Customer Listing (Печать списка покупателей) гораздо информативнее, чем просто Customer Listing (Список пользователей). Пункты меню, напротив, должны быть как можно более краткими. Так, пункт Customer Listing в меню Reports вполне уместен и достаточно информативен. Данные рекомендации относительно названий пунктов меню и других элементов интерфейса соответствуют общепринятым стандартам, используемым при разработке интерфейса для Windows.

Теперь остается ответить еще на один, не менее важный вопрос; в каком виде отчет будет представлен пользователю. Обычно СУБД содержит не менее нескольких десятков, а то и сотен, разнообразных отчетов. Включать их все как отдельные пункты в одно меню, очевидно, бессмысленно. Поэтому чаще всего проектировщики ограничивают список отчетов теми, которые имеют непосредственное отношение к текущему пользовательскому контексту. В самом деле, пользователю вряд ли понадобится распечатать список телефонных номеров всех сотрудников компании в то время, когда он будет занят вводом информации о сделанных клиентами заказах. Поэтому отчет, содержащий различные справочные данные о сотрудниках компании, не следует включать в форму ввода данных о заказах.

Если же вы все-таки хотите предоставить пользователям одновременный доступ ко всем отчетам, существующим в системе, сделайте это при помощи отдельного диалогового окна, где будет размещен список всех отчетов, сгруппированных по различным категориям. Из этого списка пользователь выберет отчет, который захочет распечатать. Данная реализация удобна, и когда нужно распечатать сразу несколько отчетов. В этом случае предпочтительнее использовать поле со списком, позволяющим выбрать несколько значений одновременно. Пользователь выберет из списка отчеты, которые хочет распечатать, и щелкнув кнопку на панели инструментов, перейдет к выполнению другой работы.

Для отчетов, которые, как правило, распечатываются группами (например, для ежемесячных) я применяю схожий метод — включаю в интерфейс диалоговое окно, где указаны все виды отчетов, выбранные по умолчанию. Пользователи могут добавить к этому списку отчеты, генерируемые периодически или только в определенных случа-

ях или отменить выбор одного из стандартных отчетов, включенных в список,

Конечно, работать со списком, который содержит несколько групп отчетов, немного труднее и дольше, чем с меню, из которого выбирается нужный отчет. Но это компенсируется существенным преимуществом. Например, часто возникает необходимость повторно распечатать один из отчетов. И тогда средство, позволяющее выбирать как целый ряд отчетов, так и отдельный отчет покажется пользователю очень удобным.

Обработка ошибок принтера

Система должна предоставлять возможность исправлять ошибки и разрешать различные ситуации, связанные с принтерами. Иногда реализовать эту возможность непросто, например, если пользователь захочет распечатать все отчеты, которые еше не были распечатаны. Проблема в том, что средства разрабатываемой системы не позволяют точно определить, какие отчеты уже были распечатаны, а какие — нет, Достоверно известно лишь, какие из них были отправлены на печать и помещены в буферное устройство принтера. Но если данные были помешены в буфер, это отнюдь не означает, что отчет распечатан (например, в лотке принтера закончилась бумага),

Некоторые разработчики подходят к проблеме ошибок, возможных в процессе печати, так: после отправки отчета или группы отчетов на печать, система просит у пользователя подтвердить, правильно ли был напечатан отчет. Данный подход нельзя считать оптимальным, уже потому, что пользователь будет вынужден ждать завершения печати отчетов, прежде чем сможет продолжать работу с системой. Это не доставит значительных неудобств, если используется локальный принтер. Но если отчеты отправлены на сетевой принтер, где существует длинная очередь заданий на печать, то пользователь может провести не один час в бездействии, прежде чем продолжит работу. А то, что большинство отчетов все же удается распечатать с первого раза, делает такую задержку и вовсе неоправданной.

Я рекомендую обрабатывать ошибки, связанные с принтером, как исключения — таковыми по сути, они и являются. В ситуации, когда требуется распечатать только те отчеты, которые еще не были напечатаны, придется добавлять одно поле в соответствующую таблицу базы данных. Если от пользователя требуется подтверждение, что отчеты распечатаны правильно, то очевидно, нужно добавить поле Yes/No или булеву константу. При другом способе подтверждения печати отчета можно хранить в соответствующем поле таблицы дату распе-

чатки или номер задания печати, присвоенный принтером. А чтобы регистрировать проблемы, возникающие при печати, добавьте дополнительную команду, позволяющую пользователю ввести комментарий, если при печати отчета возникали какие-либо проблемы. Этот комментарий можно связать с распечатываемым отчетом или с соответствующим заданием печати.

Для отчетов, печатаемых не чаще одного раза в день, допустимо использовать текущую дату в качестве флага. Но поскольку всегда может понадобиться распечатать один и тот же отчет два и более раз в день, гораздо лучше генерировать уникальный номер для каждого задания печати. Если при печати отчета возникли какие-либо неполадки или ошибки, пользователь просто выберет соответствующее задание из списка выполняемых, и система снова задаст в поле, обозначающем номер задания печати, значение *Null* для этого задания. После этого все те задания, номера которых содержат *Null*, автоматически будут включены в следующее задание печати. Или же система составит список всех заданий, при выполнении которых произошла ошибка, и предоставит пользователю возможность выбрать те из них, которые нужно попытаться распечатать повторно.

Каким образом пользователь отличит одно задание печати от другого? Вы можете вывести номер задания печати в нижнем колонтитуле отчета. Я в подобных ситуациях предпочитаю создать системную таблицу, где хранится название отчета, номер задания печати, дата распечатки, а также имя и фамилия пользователя, который выдал это задание печати (если они известны). Гораздо удобнее вывести пользователю список имеющихся заданий печати с подробными комментариями, вместо того чтобы заставлять его запоминать номера заданий.

Иногда возможность повторно распечатать какую-либо запись отсутствует или ограничена внешними условиями. Например, во многих системах автоматизации бухгалтерского учета предусмотрена возможность или даже обязательная процедура создания месячного отчета. Но с началом нового календарного месяца некоторые текущие значения, хранимые в вычисляемых полях, могут обнуляться с последующим обновлением в конце месяца, Пользователи такой системы уже не смогут заново составить отчет за предыдущий месяц, это будет сопряжено со значительными трудностями. Подобную концепцию разработки следует признать неудачной.

Итак, процесс печати достаточно сложен и может сопровождаться различными ошибками и затруднениями. Поэтому я по возможности стараюсь разделить процессы создания отчетов и обновления записей в таблицах, сделав их абсолютно независимыми друг от друга (ра-

зумеется, за исключением обновления тех полей таблиц, где хранится признак успешной печати отчета). Если же сам рабочий процесс определен таким образом, что обновление записи жестко связано с печатью отчета, то наиболее разумно приостановить дальнейшее выполнение всех задач в рамках этого рабочего процесса, пока пользователь, сгенерировавший отчет и отправивший его на печать, не подтвердит, что отчет успешно распечатан.

Подтверждение успешного завершения печати может ожидаться в фоновом режиме, что позволит не прерывать работу с системой. Например, поместите на панель задач специальный значок, щелкнув который, пользователь откроет диалоговое окно подтверждения окончания печати. После этого завершится процесс обновления таблиц в системе. В любом случае, запрашивая у пользователя подтверждение успешного завершения печати, не забудьте дать ему подробные инструкции и пояснения.

Печать автоматическая и по команде пользователя

При создании стандартных отчетов очень важен способ вывода их на печать: будут ли они распечатываться автоматически или по команде пользователя, или же следует реализовать обе эти возможности. В большинстве случаев я предпочитаю печать по требованию пользователя. Единственное исключение из этого правила — случай, когда отчет является частью рабочего процесса, например, когда счет-фактура автоматически составляется и высылается покупателю после ввода всей информации о заказе.

Отчеты, являющиеся частью рабочих процессов (например, счета-фактуры) можно выводить на печать и по отдельности, и группами. Например, распечатать каждый счет сразу же после того, как пользователь введет всю информацию о заказе. Или же объединить в одну группу все счета, которые еще не были распечатаны, и отправить их на печать, когда пользователь закончит ввод данных о принятых заказах. Первый вариант уместен, если используется локальный принтер, а второй — если принтер сетевой. Кроме того, полезно предоставить пользователю возможность выбрать один из двух способов (и принтеров) — ведь конфигурация сети может изменяться, а локальные и сетевые принтеры — добавляться и удаляться.

Некоторые разработчики предпочитают автоматически создавать отчеты, которые должны выводиться на печать через определенные промежутки времени — например, каждую неделю или в конце каждого месяца. Я не возражаю против такого способа, однако предпочитаю в дополнение к автоматической печати предоставить пользо-

вателю возможность распечатывать эти отчеты самостоятельно, по отдельной команде.

Автоматическое создание и печать отчетов достаточно сложны. В системе должны быть реализованы механизмы расчета времени, чтобы печать отчета не пришлась на нерабочий день. Очевидно, нужно предоставить пользователям право вносить изменения в расписание, определяющее время печати отчета; а также отслеживать, был ли отчет распечатан в указанное время, или произошли какие-либо ошибки. Для многопользовательских систем следует определить пользователей (или специальные группы пользователей), которые имеют право выдавать команду, подтверждающую вывод автоматически сгенерированных отчетов на печать. Нужна также корректная обработка следующей ситуации: ни один из обладающих соответствующими правами пользователей не зарегистрировался в системе в тот день, когда должны быть распечатаны автоматически генерируемые отчеты.

Итак, если принять во внимание все возможные трудности, связанные с автоматической генерацией и печатью отчетов, вариант, когда пользователь просто выбирает в меню пункт Print Weekly Reports (Печать еженедельных отчетов), покажется тривиальным. В любом случае, предусмотрите возможность повторно генерировать отчеты по команде пользователя в случае ошибки принтера. Включите эту команду в меню или реализуйте ее как элемент управления в диалоговом окне, сообщающем пользователю о проблемах, возникших при печати.

Вряд ли, однако, имеет смысл реализовать автоматическую генерацию отчетов только потому, что пользователи могут забыть распечатать еженедельный, ежемесячный или квартальный отчет — это почти исключено. Но полезно предусмотреть возможность создать этот отчет после указанной даты, если пользователь все-таки о нем забудет. Например, вы можете задать определенный период для отдельного вида отчетов и использовать его по умолчанию, разрешив пользователям изменять это значение. В этом случае разрешите пользователям переносить дату создания отчета не только назад, но и вперед.

Пользовательские отчеты

Иногда удается настолько четко определить и разграничить рабочие процессы, поддерживаемые СУБД, что можно заранее спланировать все отчеты, которые будут создаваться в системе. В этом случае, очевидно, можно ограничиться стандартными отчетами. Однако чаще приходится предоставлять пользователям большую свободу — само-

стоятельно задать определенные параметры отчетов или даже создавать новые отчеты.

Насколько гибкими будут средства создания отчетов в вашей системе, зависит от того, какие требования к отчетам предъявляют пользователи. Процесс и концепции создания пользовательских отчетов могут существенно различаться. Вы можете предоставить пользователям широкий набор средств для создания отчетов, а можете ограничить их возможности всего лишь заданием нескольких дополнительных критериев для фильтров, используемых в стандартных отчетах.

Средства создания отчетов

Если для создания отчетов вы используете Access или средство, разработанное сторонними фирмами-производителями, реализовать в системе пользовательские средства для создания отчетов будет несложно. Однако новое средство для создания отчетов Microsoft Data Reports, поставляемое вместе е Visual Basic версии 6, не годится для создания пользовательских отчетов, поскольку для работы с ним требуется установить средства разработки на каждый из клиентских компьютерОЕ:.

Средства для создания отчетов предоставляют пользователям широчайшие возможности создавать произвольные формы и виды отчетов по своему усмотрению. К сожалению, подобная гибкость не дается даром. Во-первых, может довольно сильно увеличиться стоимость системы. Например, для многопользовательской системы, где полная версия Access должна быть установлена на десятки, а то и сотни компьютеров, стоимость лицензий Access никак нельзя считать пренебрежимо малой.

Высокая стоимость средств для создания пользовательских отчетов не единственный фактор, ограничивающий их применение. Отнюдь не достаточно просто предоставить средства для создания произвольных отчетов — нужно еще научить пользователей с ними работать. Чтобы составить произвольный отчет, недостаточно просто знать, как работать ео средствами для создания отчетов, нужно разобраться в структуре базы данных, чтобы включить в отчет именно те данные, которые нужны. Такое обучение требует времени, а пользователи — занятые люди. У них есть работа, которую они должны выполнять, и создание произвольных отчетов отнюдь не входит в список их непосредственных служебных обязанностей.

Настраиваемые пользовательские отчеты

Чтобы максимально упростить процесс создания отчетов и сделать его удобным для пользователей, часто пишут специальные утилиты. Та-

кое решение порой рациональней, чем включение в систему коммерческого программного обеспечения, предназначенного для создания отчетов. Теоретически возможно создать утилиту, обладающую столь же широкими возможностями, как и средства Access, и поддерживающую схему базы данных разрабатываемой системы. Однако это требует много сил и времени. Поэтому разработчики обычно предпочитают более простые решения и включают в систему готовые шаблоны отчетов. А пользователь может выбрать, какие именно данные включить в отчет.

Мастер отчетов в Access близок к модели настраиваемых пользовательских отчетов, однако все-таки представляет собой неплохой пример использования шаблонов для этой цели. Он включает разнообразные возможности форматирования, и при этом пользователи избавлены от трудностей, связанных с самостоятельным созданием отчетов — они могут выбирать из готовых шаблонов и стилей. После того как отчет с помощью мастера создан, пользователи могут продолжать работу с ним посредством стандартного интерфейса Access.

Возможность отдельно задать шаблон и стиль отчета, как правило, удовлетворяет большинство пользователей. Чтобы еще более упростить создание настраиваемых отчетов, комбинируйте различные стили и шаблоны — пользователю останется только выбрать подходящий вариант. Большинство пользователей хотят получить отчеты, лишь немного отличающиеся от стандартных, чаще всего порядком сортировки или другим критерием фильтрации данных.

Мой собственный подход к созданию отчетов отличается от того, который применяется в мастере Access, Я выделяю два компонента в пользовательских отчетах и условно называю их «формат» и «критерии». К формату я отношу элементы отчета, определяющие макет и стиль, а также поля таблиц, которые должны быть включены в отчет (и возможно, таблицы и запросы, используемые для создания отчета). Реализация компонентов формата довольно проста; как правило, это отчет, созданный в Access, или объект Visual Basic Data Report, модифицируемый в зависимости от заданных пользователем критериев (как правило, они определяют порядок сортировки и параметры фильтра). Кроме того, я обычно добавляю к отчетам средства, позволяющие сохранить заданные критерии, чтобы использовать их впоследствии (подробнее об этом я еще расскажу). Иногда полезно предоставить пользователям возможность различным образом группировать данные при составлении отчетов, однако в большинстве случаев это не нужно.

Поскольку при таком подходе пользователи имеют дело только с двумя компонентами, требующих задания определенных параметров, для генерации пользовательских отчетов я использую диалоговое окно (рис. 17-4). Если вы решите последовать моему примеру, то полагаю, вам придется лишь немного изменить вид этого окна, чтобы адаптировать его к пользовательскому интерфейсу своей системы.

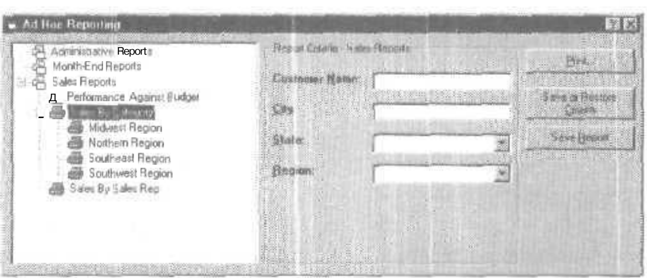


Рис. 17-4. Диалоговое окно для создания пользовательских отчетов

Диалоговое окно разделено на три области. В левой находится окно просмотра с древовидной структурой, где перечислены все имеющиеся виды форматов. Пользователю предлагается выбрать подходящий формат для отчета. Если список имеющихся форматов достаточно компактен и прост, можно не использовать древовидную структуру, а дать возможность выбрать элементы форматирования из списка или с помощью переключателей. В рассматриваемом примере форматы представлены в виде дерева и сгруппированы по категориям. Это удобно, если список форматов, определенных в системе, достаточно велик. Сгруппируйте форматы по категориям, например «Административные отчеты», «Ежемесячные отчеты», «Статистика продаж» и т. п.

Еще один уровень иерархической структуры в левой области окна — отчеты. В данном контексте отчет есть не что иное, как набор сохраненных критериев и формат. Например, указав для параметра *Region* (Регион) в отчете по статистике продаж значение *Southwest* (Юго-Западный), пользователь может сохранить этот отчет под названием *Southwest Region Sales* (Продажи в Юго-Западном регионе). Порой, если приходится задавать достаточно много параметров, это существенно экономит время.

В центральной части диалогового окна на рис. 17-4 пользователь может задать критерии для сортировки и фильтрации данных. Иногда удается спроектировать пользовательский интерфейс так, чтобы с помощью одного набора элементов управления задавать параметры всех

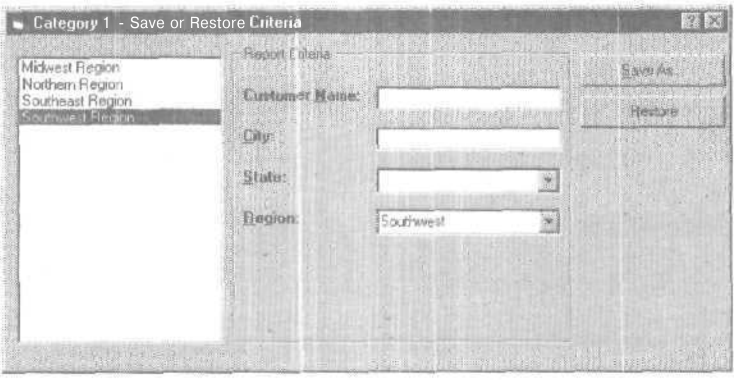
форматов, а также делать недоступными для пользователя те элементы, которые не применимы к выбранному типу формата. Порой для каждого формата приходится создавать уникальный набор критериев — и соответственно, набор элементов управления, позволяющий задать численные значения для этих критериев. Я предпочитаю по возможности придерживаться варианта, где тип формата определяет набор доступных пользователю элементов интерфейса. При этом все форматы подразделяются на отдельные категории, определяющие, какие элементы управления доступны пользователю, а какие — нет.

В правой части диалогового окна на рис. 17-4 размещено несколько кнопок. Щелкнув кнопку Print (Печать), пользователь открывает вспомогательное окно, позволяющее задать параметры печати — число копий отчета, принтер, на котором он будет распечатан и т. д. Если в вашей системе не предусмотрены такие возможности управления печатью, разместите в диалоговом окне создания пользовательского отчета две кнопки — Print и Print Preview (Предварительный просмотр), позволяющие просмотреть созданный отчет на экране и отправить его на печать. При этом пользователю не нужно переходить к вспомогательному окну параметров печати.

Щелкнув кнопку Save Or Restore Criteria (Сохранить или восстановить критерии), пользователь откроет одноименное диалоговое окно (рис. 17-5). В его центральной части находится группа элементов управления, позволяющих выбирать критерии для отчета. Она полностью идентична группе элементов в центральной части основной формы составления пользовательских отчетов — диалогового окна на рис. 17-4. В списке, размещенном в левой части (рис. 17-5), представлены сохраненные пользователями критерии отчетов. Когда пользователь выбирает одно из названий в этом списке, в средней части окна отображаются соответствующие параметры. Щелкнув кнопку Save As (Сохранить как), пользователь может ввести новое имя и сохранить критерий, задав для него соответствующие параметры в центральной части диалогового окна. Кнопка Restore позволяет восстановить прежние параметры выбранного критерия и загрузить его в основную форму создания пользовательских отчетов.

Реализовать возможность сохранять пользовательские критерии достаточно просто. Нужно создать для каждой категории по одной таблице с числом полей, соответствующим числу элементов управления диалогового окна. В многопользовательской системе следует определиться, будут ли использоваться одни и те же критерии для всех пользователей, или каждый получит возможность создать свой собственный набор критериев, Если создаваемые критерии будут общи-

ми для всех пользователей, то соответствующие таблицы следует хранить в основной базе данных, то есть там же, где и остальные совместно используемые данные. Если же каждый пользователь создает свой собственный набор критериев, таблицы можно хранить локально, в базе данных клиентского приложения (или локальной базе данных, если для создания клиентского приложения использовался не Microsoft Access, а другое средство разработки).



Рас,** 17-5. Диалоговое окно, позволяющее **сохранять новые и** восстанавливать **прежние** критерии **пользовательских отчетов

Оба этих подхода отнюдь не являются взаимоисключающими. Вы можете реализовать систему так, чтобы каждый пользователь выбирал, как из общих, так и из своего собственного набора критериев. Чтобы реализовать такую возможность, либо добавьте поле, содержащее фамилию и имя пользователя, в таблицу, где хранятся критерии, либо отобразите пользователю результат выполнения операции UNION над общей и локальной таблицами. Я предпочитаю, чтобы фамилии и имена пользователей хранились в таблице критериев — это облегчает поддержку системы, когда один и тот же пользователь регистрируется и работает за разными компьютерами, а не на одном рабочем месте.

Пользователи могут хранить критерии, распределив их по категориям отчетов. При этом сохраненные критерии применимы к любому отчету, относящемуся к данной категории (разумеется, если спецификации критериев общие для всех отчетов).

Но иногда целесообразно связать критерии с определенным форматом отчета (рис. 17-4). Реализация такого подхода также не представляет сложностей. Потребуется создать таблицы для каждого типа критерия (можно использовать таблицы, созданные для сохранения

критериев) и промежуточную таблицу, которая связывает форматы отчетов и определенные критерии (рис. 17-6).

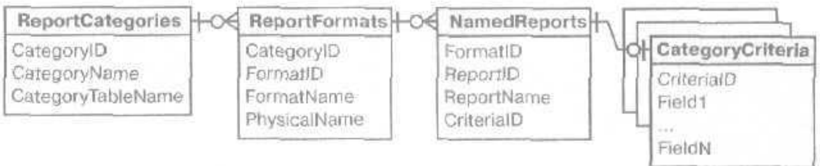


Рис. 17-6. Структура таблиц, используемых для сохранения и связывания критериев и отчетов

В таблице *ReportCategories* (Категории отчета) есть поле *CategoryTableName* (Имя таблицы категории). Это позволяет определить в системе ту таблицу, которая содержит критерии, относящиеся к указанной категории формата отчета. Если структура критериев одинакова для всех отчетов, это поле в таблице не нужно. Если же для разных форматов используются разные наборы критериев, оно обязательно должно быть включено в таблицу *ReportFormats.*

Важно, что данная структура таблиц (рис. 17-6), позволяет конфигурировать отчеты на этапе исполнения программы. Поля *FormatName* (Название формата) и *PhysicalName* (Имя соответствующего объекта) включены в таблицу *Report Formats* для поддержки этой возможности. Если форматы отчетов определяются как объекты базы данных (например, отчеты Access) или как отдельные файлы (другие средства создания отчетов), вы можете использовать «обходной путь», чтобы реализовать возможность добавить отчет в систему на этапе исполнения клиентского приложения. Для этого нужно разместить список форматов, отображаемый в форме пользовательского приложения, в таблице *ReportFormats*, а не использовать жестко заданный список.

Чтобы добавить в систему новый отчет, создайте новый объект, определяющий формат (в Access это отчет, в других средствах создания отчетов — отдельный файл) и затем добавьте запись в системную таблицу *ReportFormats.* Новый формат будет автоматически включен в систему и доступен всем пользователям, причем это никак не повлияет на функциональность ядра системы. Поле *FormatName* содержит название формата, отображаемое пользователю в списке форматов, а поле *PhysicalName* — реальное имя объекта и путь к нему (если объекты хранятся не в самой базе данных).

Описанный подход позволяет существенно упростить пользовательский интерфейс по сравнению с вариантом, когда для создания

нестандартных отчетов пользователям приходится прибегать к помощи специальных программных средств. Но хотя данный подход более прост и удобен для пользователя, он подходит не для всех систем. Например, пользователям может понадобиться создавать новые отчеты, добавляя дополнительные критерии фильтров к стандартным отчетам, имеющимся в системе. Эти критерии можно добавить в систему, просто задав дополнительные элементы управления в диалоговом окне параметров печати нестандартного отчета, вместо того чтобы разрабатывать специальные утилиты для создания настраиваемых пользовательских отчетов.

Стандартные письма

Стандартные письма — это особый вид пользовательских отчетов. При их рассылке изменяются только фамилия и имя адресата, но не текст. Пользователь должен составить текст письма, используя готовые шаблоны, уже хранящиеся в системе, и выбрать из списка адресов те, по которым должны рассылаться письма.

Я не считаю пользовательские отчеты СУБД лучшим средством для составления писем: и тех, которые содержат один и тот же текст, и тех, содержимое которых меняется в зависимости от ситуации. Хотя возможности форматирования отчетов в СУБД весьма широки и их список продолжает пополняться, все же они не сравнимы с возможностями, предоставляемыми текстовыми процессорами. Кроме того, большинству пользователей необходимо редактировать текст письма, однако предоставить им возможность изменять текст, хранимый в базе данных, нецелесообразно, поскольку все изменения, внесенные в стандартный текст, невозможно отменить впоследствии. Вот почему в большинстве случаев нерационально использовать отчеты для составления стандартных писем.

И все же это совсем не означает, что базы данных совсем бесполезны в данном случае. Как раз наоборот, в них весьма удобно хранить адреса и фамилии, а также стандартные фрагменты текста. Но при составлении письма лучше пользоваться не формой отчета, а выгружать всю необходимую информацию в текстовый процессор, например в Microsoft Word, позволяющий легко форматировать и редактировать текст письма перед печатью и отправкой.

Организовать обмен данными между текстовым процессором и базой данных очень легко. Вы можете использовать возможность слияния данных из различных документов Microsoft Office или команду MailMerge, указав таблицу или запрос Access в качестве источника данных. Затем при помощи Visual Basic for Applications (VBA) вставьте в документ данные из базы и выведите готовый документ на экран

пользовательского компьютера для дальнейшего редактирования или отправьте его на печать.

В некоторых случаях понадобится возможность регистрации и учета всех стандартных писем, имеющихся в системе. Если пользователи не имеют права вносить исправления в стандартные письма перед отправкой на печать или рассылкой, то незачем хранить сами эти письма в базе данных. В этом случае в базе данных должен быть зарегистрирован только факт отправки письма, а также дата отправки и фамилия отправителя. Но если пользователи составляют текст писем из набора стандартных фрагментов, вы можете смоделировать этот процесс при помощи сложной сущности (рис. 17-7).

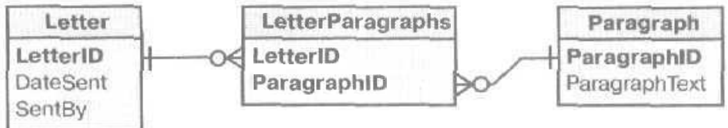


Рис.** 17-7. **Структура, позволяющая хранить абзацы стандартного текста, которые впоследствии могут быть включены в письма

Если же нужна возможность редактировать письма перед отправкой их на печать, то хранить текст писем в базе данных нецелесообразно, поскольку ни Access, ни Microsoft SQL Server не предоставляют удобных и быстрых средств работы с большими фрагментами текста. В этом случае в базе данных следует хранить только имена физических файлов писем и путь к этим файлам. Разумеется, в системе должны быть предусмотрены механизмы отслеживания и обработки ситуаций, когда зарегистрированный файл был перемешен, переименован или удален. Как правило, в этом случае от пользователя требуется выполнить какие-либо дополнительные действия или ответить на вопросы системы.

Итоги

Мы рассмотрели различные аспекты предоставления пользователю информации на основе данных, хранимых в системе. Как правило, под этим подразумевается создание отчетов, которые впоследствии могут быть распечатаны. Другие виды предоставления информации пользователю — в виде формы или набора данных, помещенных в таблицу.

В первой части главы обсуждались различные возможности сортировки и фильтрации данных, предоставляемые Microsoft Access. Даже если вы не используете Access в качестве инструмента разработки, приведенные примеры окажутся весьма полезны — ведь подоб-

ные возможности очень часто необходимы пользователям, и их реализуют во многих компьютерных системах.

Мы рассмотрели также различные виды стандартных отчетов, в том числе отчеты в виде списков и подробные отчеты, а также отчеты, использующие агрегированные данные, и отчеты на основе форм пользовательского интерфейса. Я рассказала о различных способах интеграции средств создания отчетов в интерфейс пользователя, а также об устранении ошибок, возникших при создании отчетов. Мы подробно остановились на некоторых аспектах создания пользовательских отчетов, и обсудили конкретные способы реализации этих отчетов. И в заключение, были рассмотрены широкие возможности поиска и обработки данных, предоставляемые СУБД, для создания стандартных писем, Как вы могли убедиться, возможности современных компьютерных систем позволяют сочетать мощные средства работы с данными с возможностями текстовых процессоров.

В следующей главе будут обсуждаться вопросы, связанные с поддержкой пользователя. Особое внимание мы уделим средствам поддержки пользователей, реализуемым в пользовательском интерфейсе клиентской части приложений.

Поддержка пользователя

Поддержка пользователя — достаточно широкое понятие. Все, о чем мы говорили в третьей части книги, подпадает под это определение. Ваша цель при проектировании интерфейса — создать для пользователя удобное рабочее место. Например, целостность данных, которую мы обсуждали в главе 16, позволяет облегчить пользователю безошибочный ввод данных. В этой главе мы рассмотрим более обшие формы поддержки пользователя, применяемые при построении интерфейса системы.

В главе 12 мы говорили, что людей, которые будут работать с вашей системой, можно разделить на три группы. Неопытным пользователям нужна информация о том, что делает система. Опытные пользователи хотят знать, как выполняются конкретные задачи, а пользователи-эксперты — как ускорить работу. Для каждой группы пользователей требования к поддержке различны. Вводный курс обучения, столь полезный для начинающих, вызовет зевоту у опытных пользователей и раздражение — у экспертов.

Поэтому следует не просто рассматривать технические аспекты поддержки, но и учитывать нюансы требований различных групп пользователей. Помните также, что уровень подготовки пользователей меняется по мере их работы с системой. Так что пусть диалоговое окно, выводимое на экран при загрузке программы и приглашающее пользователей-новичков ознакомиться с основами системы, содержит флажок Don't show me again (Не показывать в следующий раз). Включив этот флажок, опытный пользователь отменит вывод диалогового окна на экран при последующих загрузках программы. Сообщения интерактивных подсказок обычно никого не раздражают, но все же следует предусмотреть механизм их отключения.

Старайтесь не отсылать пользователя к документации. Интерфейс должен предоставлять полную информацию о реализованных в нем функциях,

Большинство систем позволяют выполнить одно и то же действие разными способами. Например, сохранить изменения в записи можно, выбрав пункт Save (Сохранить) в меню File (Файл), нажав соответствующую кнопку на панели управления или при помощи сочетания клавиш Ctrl-S. Каждый из этих способов называется *вектором команды,* а каждый вектор команды чаще использует определенная группа пользователей. Новички, еще не знакомые со всеми особенностями системы, охотней задействуют меню, более опытные — графические элементы управления, а эксперты — сочетания клавиш. Чтобы пользователь смог подобрать для себя наиболее подходящий способ, интерфейс должен предоставлять полную информацию о возможных векторах команд.

На рис. 18-1 показан пункт меню File системы Microsoft Access 2000, выводимый по умолчанию, Заметьте, что пункт Save показывает также и все вектора команд. Легко запоминающаяся комбинация Alt-F-S выделена с помощью подчеркивания, буква S — первая в слове Save <буква F в пункте File также подчеркнута). Показана также соответствующая кнопка меню и комбинация клавиш Ctrl-S. Предоставляя столь подробную информацию, интерфейс способствует обучению пользователя в процессе работы.

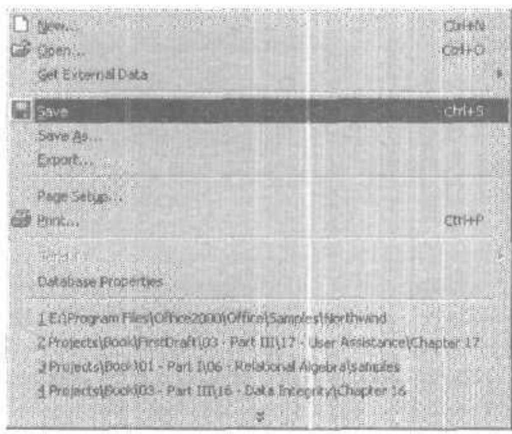


Рис. 18-1. Пункт меню File приложения Access 2000 предоставляет информацию о всех векторах команды Save

ПРИМЕЧАНИЕ Microsoft Visual Basic не позволяет показывать изображения командных кнопок в соответствующих пунктах меню, хотя существуют компоненты ActiveX сторонних производителей, которые поддерживают такую возможность. Парадигма создания меню в Access 2000 позволяет вставлять в пункты меню изображения, однако реализована эта возможность столь неуклюже, что вам почти наверняка придется использовать Button Editor для создания собственных графических элементов. И Visual Basic, и Access позволяют легко привязывать к командам определенные комбинации клавиш.

Демонстрация пользователю возможных векторов команд не решает всех проблем, так как это пассивный механизм. Все механизмы поддержки пользователей можно разделить на *пассивные,* которые являются частью пользовательского механизма, *механизмы реакции на действия пользователя* (или *реактивные*), и *активные,* которые пытаются «угадать» текущие потребности пользователя. Мы рассмотрим в этой главе все три вида и закончим ее кратким обзором учебных материалов.

Пассивные механизмы поддержки

Пассивные механизмы — это ярлыки, указатели, пояснения, которые включены в интерфейс. В отличие от реактивных, пассивные механизмы не требуют от пользователя каких-либо действий.

К пассивным механизмам поддержки относятся ярлыки элементов управления, названия пунктов меню и даже названия форм. Поэтому очень важно хорошо продумать названия элементов интерфейса — они должны как можно более ясно отражать их назначение,

Именами ряд пассивных механизмов не исчерпывается. В него входят легко запоминающиеся сочетания клавиш, всплывающие подсказки, строки состояний и другие элементы.

Запоминающиеся сочетания клавиш

Сочетания клавиш позволяют выполнять операции наиболее быстро. Комбинации клавиш выполняют роль пассивного механизма поддержки и всегда выделяются подчеркиванием в названиях соответствующих элементов управления. Любой более-менее опытный пользователь Microsoft Windows знаком с парадигмой «сочетание клавиши Alt и подчеркнутого символа».

При разработке системы назначьте определенное сочетание клавиш каждому элементу меню каждого элемента управления. Определите, какой символ вы будете использовать в этом сочетании. Суще-

ствуют определенные правила, определяющие выбор символа. Таки-ми символами могут быть:

- первая буква названия элемента управления или пункта меню;
- вторая согласная в названии элемента управления;
- звонкий гласный в названии элемента управления.

И в Access, и в Visual Basic определить сочетание клавиш нетрудно: просто вставьте специальный символ (&) в название команды или элемента управления перед тем символом, который будет использоваться в сочетании клавиш. Обе системы выделяют этот символ подчеркиванием. (Чтобы использовать символ «&» непосредственно в сочетании клавиш, вам придется вставить перед ним еще один такой символ: команда меню Nuts & Bolts будет отображаться в интерфейсе пользователя как Nuts _Bolts, а вот комбинация Nuts && Bolts — как Nuts & Bolts).

Всплывающие подсказки

Назначение всплывающей подсказки очевидно — проинформировать пользователя о назначении кнопок на панели и тех элементов, которые не сопровождаются надписями (рис. 18-2).

Рис. 18-2. Всплывающая подсказка для кнопки Save (Сохранить) панели инструментов Access Form View

---

ПРИМЕЧАНИЕ Когда всплывающая подсказка используется для элементов управления, не являющихся кнопками панели инструментов, ее называют всплывающим ярлыком. И всплывающие ярлыки, и всплывающие подсказки работают совершенно одинаково, и я буду говорить о них как о всплывающих подсказках.

Всплывающие подсказки делают работу с системой значительно удобней. Если вам когда-нибудь приходилось выбирать картинку, адекватно объясняющую назначение элемента управления, вы знаете как это трудно. Одно дело — выбрать графический элемент для кнопки Save (Сохранить) — практически каждому пользователю знакомо изображение дискеты в пакете Microsoft Office. А вот какое изображение подойдет для кнопки Open Customer Form, которая открывает

форму ввода информации о заказчике? Вы можете поместить на эту кнопку маленькую человеческую фигуру, но что тогда использовать для форм Employees (Сотрудники) и Vendors (Поставщики)? Изображения на кнопках могут ввести пользователя в заблуждение.

К счастью, всплывающие подсказки решают эту проблему. Они дают возможность пользователю легко запомнить назначение кнопок. Например в Microsoft Access кнопка с изображением рыбы открывает форму Customers (Покупатели). (Интересно, что проектировщики хотели этим сказать?) Разумеется, пользователь, впервые увидевший эту панель инструментов, не сможет без помощи всплывающей подсказки определить назначение кнопки с изображением рыбы. Но получив подсказку, он в дальнейшем будет ориентироваться без труда.

Чтобы вставить всплывающую подсказку в Access или Visual Basic, достаточно присвоить соответствующее значение одному из свойств элементов управления. Текст всплывающей подсказки должен быть коротким — не больше двух-трех слов. Помните, что основное назначение подсказки — напомнить пользователю о назначении элемента, а не обучить его работе с системой.

Все подсказки уникальны. Если элемент управления дублирует пункт меню, текст их подсказок должен быть одинаковым. Если элемент управления не дублирует никакого пункта меню, используйте в качестве подсказки наиболее подходящий по смыслу глагол. Если панель управления содержит три кнопки, которые выводят на экран формы ввода данных о заказчиках, поставщиках и сотрудниках, примените в качестве подсказок соответственно Customers, Suppliers и Employees. Если же одна из кнопок отвечает за вывод информации о заказчике на экран, а другая — за печать этой информации, вам придется использовать фразы типа Open Customer Form (Открыть форму Заказчик) или Maintain Customers (Поддержка заказчиков) — для первой, и Print Customers (Распечатать информацию о заказчиках) — для второй кнопки.

При применении изображений в пользовательском интерфейсе очень важно соблюдать ряд правил и разграничивать изображения, представляющие данные и действия. Я обычно провожу такое разграничение, а затем использую комбинацию изображений из этих двух групп там, где это нужно. Если изображение перечеркивающего объект крестика связано с операцией удаления, а изображение человека — с таблицей заказчиков, разумно использовать комбинацию этих изображений, чтобы обозначить действие «удалить запись о заказчике» (рис. 18-3).

Рис. 18-3. Такая комбинация изображений интуитивно понятна большинству пользователей

Строка состояния

Строка состояния — стандартное средство пассивной помощи пользователю. Располагающиеся в нижней части окон, строки состояний отображают различную важную информацию, например, нажата ли клавиша Num Lock или Caps Lock. Строки состояний отображают также пользовательские сообщения.

Заметьте, что форма *Categories* развернута на весь экран, так что строку состояния в нижней части экрана можно принять за часть эгой формы. На самом деле строка состояния принадлежит главному окну Access (рис. 18-4).

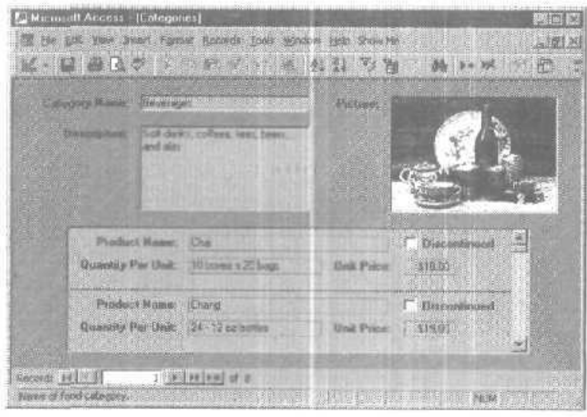


Рис. 18-4, Строка состояния отображается в нижней части главного окна

Access позволяет отображать информацию об элементах управления. Для этого поместите соответствующий текст в свойстве StatusBarText этих элементов. Если вы не сделали этого для элемента управления, связанного с полем какой-либо таблицы, в строке состояния будет отображаться значение свойства поля Description (которое вы можете задать, редактируя свойства таблицы в режиме Design view). Насколько мне известно, Access не позволяет задать свойство StatusBarText во время исполнения программы на уровне кода, по-

этому его нельзя использовать для отображения на экране пользовательских сообщений.

В Visual Basic можно управлять отображаемым в строке состояния текстом, явно задав свойство Text элемента управления StatusBar. Поэтому, в отличие от Access. вы можете использовать строку состояния для отображения пользовательских сообщений. Жаль только, что текст, отображаемый пользователю, необходимо включить непосредственно в код программы, а способа явно связать строку состояния и элемент управления, чтобы отображать нужную информацию автоматически, в Visual Basic не существует. Тем не менее, это не слишком большая плата за ту гибкость, которую предоставляет рабочая среда.

Строка состояния никак не влияет на работу пользователя, в отличие, например, от диалогового окна. Она не перехватывает управление клавиатурой или мышью, выводя на экран сообщения. Фактически, в Access (а также и в Visual Basic, если вы позаботились о соответствующей функциональности) пользователь может запретить вывод на экран сообщений с помощью строки состояния.

В Visual Basic строка состояний сообщит пользователю о происходящих фоновых процессах, а также об ошибках времени выполнения. Пользователь может сделать строку состояния невидимой, так что не используйте ее для вывода информации, на которую пользователь обязан отреагировать, — он может не заметить такого сообщения.

Например, чтобы сообщить пользователю, что он ввел неправильные данные, можно использовать строку состояния, выделив ее красным цветом. Но так как место в строке состояния ограничено, для вывода информации большого объема обычно используют окно системного сообщения,

Реактивные механизмы поддержки пользователя

Реактивные механизмы помощи, в отличие от пассивных, выдают информацию только в ответ на определенные действия пользователя. Опытные пользователи применяют их чаще, чем новички, которые либо не подозревают об их существовании, либо не знают, как вызвать подсказку. По этой причине реактивные механизмы не слишком подходят для оказания помоши типа «что это такое и как этим пользоваться», которая нужна большинству начинающих.

Большинство реактивных механизмов представляет собой разновидность контекстных подсказок. Существует несколько парадигм их реализации, из которых мы рассмотрим всего две: традиционная контекстная подсказка и всплывающие ярлыки. В некотором смысле,

сообщения об ошибках также можно отнести к реактивным механизмам, и мы обсудим их в конце раздела.

Контекстная справка, вызываемая пользователем

Традиционная контекстная справка — это, как правило, пользовательская документация, доступная в электронном виде (она удобней, чем печатный документ). Пользователи могут перейти от одного раздела справки к другому, просто щелкнув мышью нужный элемент в окне содержания справки. С другой стороны, такую документацию нельзя взять с собой и читать как книгу.

Тщательно спроектированная контекстная помощь позволяет воспользоваться всеми ее преимуществами. Создание справочной системы — это отдельная сложная задача, и мы не рассматриваем ее здесь. Я лишь дам несколько рекомендаций, касающихся справочных систем для баз данных, и порекомендую соответствующую литературу.

Итак, во-первых, никогда не следует рассматривать справочную систему как элемент, встроенный в пользовательский интерфейс. То есть ваша программа должна быть независимой от справочной системы.

Помните, что новички могут не подозревать о существовании справочных систем, и им никогда не придет в голову нажать клавишу F1, если они попали в затруднительную ситуацию. Вы должны рассматривать справочную систему как дополнение к встроенной системе помощи, а не как ее замену. Многие проектировщики думают, что создав справочную систему, они избавляют себя от необходимости проектировать встроенную систему подсказок. Это ошибка — программа без системы подсказок крайне неудобна в использовании,

Следующий вопрос — выбор способа поддержки пользователя. Грубо контекстные справочные системы можно разделить на две категории: ориентированные на выполняемые задачи и ориентированные на выполняемые функции. Системы, ориентированные на задачи, подсказывают пользователю, как достичь конкретной цели, например, распечатать счет-фактуру или отчет о встрече, Функционально-ориентированные системы поясняют детали выполнения конкретной операции (например, команды Print) или функции элемента управления (например, поля *CustomerID* в форме пользовательского интерфейса). Обычно эти два вида справочных систем в пользовательской документации соответствуют разделам «Руководство для пользователя» и «Справочный раздел»,

Если ваша система поддерживает очень сложные рабочие процессы, лучше реализовать *справочную систему, ориентированную на выполнение задач.* Тем не менее, помните о необходимости встроенной справочной системы. Не следует выводить на экран алфавитный спи-

сок доступных форм, как это сделано в окне базы данных Access, и рассчитывать, что пользователь непременно прочтет контекстную справку, объясняющую, в какой последовательности нужно загружать формы.

Помните, что контекстные справки не заменяют обучающих материалов. Единственная цель контекстной справки — подсказать пользователю, как действовать, а не когда или почему.

Если операция выполняется в несколько приемов, лучше не пытаться дать объяснение на одной странице, а разбить справку на несколько разделов. Вынесите в заголовок наиболее существенные моменты, а желающим ознакомиться с подробностями предоставьте ссылки на соответствующие разделы.

Функционально ориентированные справочные системы поясняют пользователю, что и почему он должен делать. Большинство таких справочных систем скорее объясняют назначение элементов данных и элементов управления, нежели содержат информацию о функциональности. Некоторые СУБД, например Access, позволяют получить справку о встроенных функциях (таких, как Mid$).

Для работы с СУБД действительно нужны справочные разделы с информацией о сущностях, их атрибутах и ограничениях. Могут понадобиться и пояснения, касающиеся функциональности элементов управления: например, как осуществлять навигацию по дереву или как получить значение даты с помощью встроенного календаря.

При создании справочных разделов, объясняющих структуру данных в системе, важно представлять, в какой ситуации она может понадобиться пользователю. Если пользователь просматривает форму ввода заказов и видит элемент, называющийся Desired Delivery Date (Желательная дата доставки), вряд ли он станет нажимать клавишу F1, чтобы узнать о назначении элемента — оно и так ясно. Специальная справка в данном случае — напрасная потеря времени.

Почему пользователю может потребоваться помощь? Может быть, он не понимает, почему это поле уже заполнено — объясните ему, что такое значение по умолчанию, и как его изменить. Или покупатель попросил доставить ему товар «в любое время после первого числа текущего месяца» — объясните пользователю, что он должен ввести самую раннюю дату поставки. Попытайтесь взглянуть на систему глазами самого пользователя.

Подсказки типа «Что это такое»

Эти подсказки очень напоминают контекстные справки, за исключением способа вызова. Подсказку вызывают, щелкнув кнопку с символом вопроса на панели инструментов и выбрав затем элемент управления, справку о котором пользователь хочет получить (рис. 18-5).

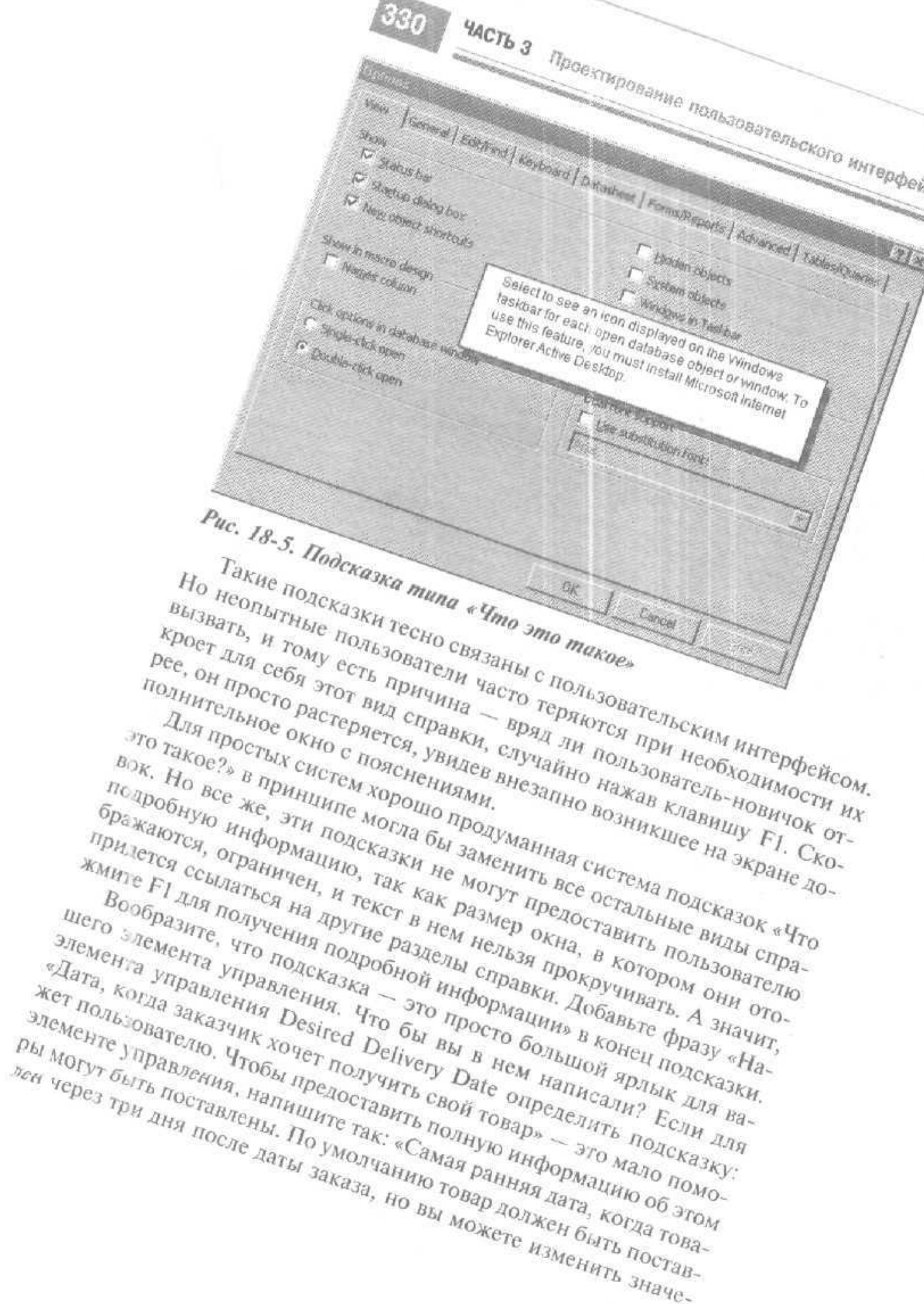


Рис. 18-5. Подсказка типа «Что это такое»

Такие подсказки тесно связаны с пользовательским интерфейсом. Но неопытные пользователи часто теряются при необходимости их вызвать, и тому есть причина — вряд ли пользователь-новичок откроет для себя этот вид справки, случайно нажав клавишу F1. Скорее, он просто растеряется, увидев внезапно возникшее на экране дополнительное окно с пояснениями.

Для простых систем хорошо продуманная система подсказок «Что это такое?» в принципе могла бы заменить все остальные виды справок. Но все же, эти подсказки не могут предоставить пользователю подробную информацию, так как размер окна, в котором они отображаются, ограничен, и текст в нем нельзя прокручивать. А значит, придется ссылаться на другие разделы справки. Добавьте фразу «Нажмите F1 для получения подробной информации» в конец подсказки.

Вообразите, что подсказка — это просто большой ярлык для вашего элемента управления. Что бы вы в нем написали? Если для элемента управления Desired Delivery Date определить подсказку: «Дата, когда заказчик хочет получить свой товар» — это мало поможет пользователю. Чтобы предоставить полную информацию об этом элементе управления, напишите так: «Самая ранняя дата, когда товары могут быть поставлены. По умолчанию товар должен быть поставлен через три дня после даты заказа, но вы можете изменить значе-

ние даты, щелкнув мышью поле и введя другую дату. Нажмите F1 для получения подробной информации».

Звуковые сигналы

Звуковые сигналы, выдаваемые компьютером — мощное средство оповещения, но проектировщик может использовать их не только во благо, но и во зло. Простейший пример неадекватного использования звуковых сигналов — гудок, оповещающий пользователя, что он совершил недопустимые действия.

Вместо того чтобы запугивать пользователя резкими звуками, лучше сообщите ему что-нибудь приятное. Например, если введенные данные соответствуют всем необходимым требованиям, пусть компьютер негромко произнесет что-нибудь вроде «OK, все верно». Но если возникла какая-то проблема, лучше всего вывести сообщение об ошибке в строке состояния, а не «звонить во все колокола», раздражая пользователя и окружающих громкими звуками.

На мой взгляд, лучше всего иллюстрирует эти слова пример, авторство которого принадлежит Алану Куперу. Когда вы нажимаете клавишу, клавиатура издает тихий щелчок. Вы, скорее всего, не обращаете внимания на эти звуки, но если клавиатура вдруг откажется «щелкать», обязательно поймете, что что-то не так.

Не опасайтесь, что звуковые подтверждения, выдаваемые при правильном вводе данных, создадут излишний шум в офисе. Да, фраза «OK, все верно» должна выделяться на фоне офисного шума. Но мне приходилось использовать звуковые сообщения при проектировании систем телефонной связи, когда в одной комнате находилось более 100 одновременно работающих операторов, и никаких проблем не возникало.

Сообщения об ошибках

Очень жаль, но большинство пользователей не считает сообщения об ошибках одним из видов помощи, воспринимая их, скорее, как заслуженное наказание. Конечно, это не так, ведь каждая ошибка — это очередной повод помочь пользователю. Хорошо воспитанный человек, прося о помощи, не пытается сразу же обругать помощника, не доказывает, что в его проблемах виноват кто-то еще. Он ясно, доходчиво и вежливо объясняет возникшую проблему, и то, почему он обратился за помощью.

«Хорошо воспитанная» компьютерная система ведет себя так же. Она даже должна быть более вежливой, чем человек. Когда возникает ошибка времени исполнения, система обязана:

- объяснить пользователю на понятном ему языке, что произошло;
- вежливо попросить о помощи;
- не просить пользователя сделать то, что система должна сделать самостоятельно;
- описать все действия, которые мог бы предпринять пользователь.

Иногда системы безнадежно «зависают», или недостаток оперативной памяти или дискового пространства заставляет пользователя вмешиваться в работу программы. На этот случай вы практически ничего не можете предпринять, кроме как предусмотреть вывод на экран соответствующего сообщения, чтобы проинформировать пользователя о том, что же он должен сделать.

Ясное и написанное понятным языком сообщение поможет выйти из ситуации с наименьшими потерями. Если пользователь осознал суть проблемы, он скорее всего избежит ее в дальнейшем — если, конечно, подобной ситуации можно избежать вообще.

Объяснив пользователю дальнейшие действия, вы поможете ему справиться с проблемой самостоятельно. Не предоставляйте ему возможности выбора, в которых он скорее всего запутается. Помните то, что очевидно для вас как для специалиста, вовсе не очевидно для пользователя. Не бойтесь объяснений, но и не увлекайтесь деталями.

Сообщение должно быть написано вежливо. Пользователи оценят дружелюбие системы и, может быть, простят вам ваши ошибки. Помните, никто не совершенен.

Активная помощь

Пассивные и реактивные механизмы все более широко применяются в компьютерной индустрии, по мере того как растет наш опыт взаимодействия пользователей и компьютеров. Последняя категория механизмов поддержки — *активные механизмы*, но число систем, которые их используют, пока невелико.

Принцип активной помощи прост: система отслеживает действия пользователя и в случае необходимости сама предлагает ему помощь или подсказку, Например, Office Assistant (Помощник из приложений Microsoft Office) предлагает пользователям контекстные справки в зависимости от их действий.

Очень интересен такой механизм активной помоши, как интеллектуальный агент. Это программа, которой пользователь делегирует выполнение определенных задач. Ее обычно используют в Web-приложениях для ответа на просьбы типа: «Найди мне этот товар за наименьшую цену» или «Предложи книгу, которая мне понравится». Но ничто не ограничивает область применения интеллектуальных аген-

сок доступных форм, как это сделано в окне базы данных Access, и рассчитывать, что пользователь непременно прочтет контекстную справку. объясняющую, в какой последовательности нужно загружать формы.

Помните, что контекстные справки не заменяют обучающих материалов. Единственная цель контекстной справки — подсказать пользователю, как действовать, а не когда или почему.

Если операция выполняется в несколько приемов, лучше не пытаться дать объяснение на одной странице, а разбить справку на несколько разделов. Вынесите в заголовок наиболее существенные моменты, а желающим ознакомиться с подробностями предоставьте ссылки на соответствующие разделы.

Функционально ориентированные справочные системы поясняют пользователю, что и почему он должен делать. Большинство таких справочных систем скорее объясняют назначение элементов данных и элементов управления, нежели содержат информацию о функциональности. Некоторые **СУБД,** например Access, позволяют получить справку о встроенных функциях (таких, как Mid$).

Для работы с СУБД действительно нужны справочные разделы с информацией о сущностях, их атрибутах и ограничениях. Могут понадобиться и пояснения, касающиеся функциональности элементов управления: например, как осуществлять навигацию по дереву или как получить значение даты с помощью встроенного календаря.

При создании справочных разделов, объясняющих структуру данных в системе, важно представлять, в какой ситуации она может понадобиться пользователю. Если пользователь просматривает форму ввода заказов и видит элемент, называющийся Desired Delivery Date (Желательная дата доставки), вряд ли он станет нажимать клавишу F1, чтобы узнать о назначении элемента — оно и так ясно. Специальная справка в данном случае — напрасная потеря времени.

Почему пользователю может потребоваться помощь? Может быть, он не понимает, почему это поле уже заполнено — объясните ему, что такое значение по умолчанию, и как его изменить. Или покупатель попросил доставить ему товар «в любое время после первого числа текущего месяца» — объясните пользователю, что он должен ввести самую раннюю дату поставки. Попытайтесь взглянуть на систему глазами самого пользователя.

Подсказки типа «Что это такое»

Эти подсказки очень напоминают контекстные справки, за исключением способа вызова. Подсказку вызывают, щелкнув кнопку с символом вопроса на панели инструментов и выбрав затем элемент управления, справку о котором пользователь хочет получить (рис. 18-5).

Рис. 18-5. Подсказка типа « Что это такое»

Такие подсказки тесно связаны с пользовательским интерфейсом. Но неопытные пользователи часто теряются при необходимости их вызвать, и тому есть причина — вряд ли пользователь-новичок откроет для себя этот вид справки, случайно нажав клавишу F1. Скорее, он просто растеряется, увидев внезапно возникшее на экране дополнительное окно с пояснениями.

Для простых систем хорошо продуманная система подсказок «Что это такое?» в принципе могла бы заменить все остальные виды справок. Но все же, эти подсказки не могут предоставить пользователю подробную информацию, так как размер окна, в котором они отображаются, ограничен, и текст в нем нельзя прокручивать. А значит, придется ссылаться на другие разделы справки. Добавьте фразу «Нажмите F1 для получения подробной информации» в конец подсказки.

Вообразите, что подсказка — это просто большой ярлык для вашего элемента управления. Что бы вы в нем написали? Если для элемента управления Desired Delivery Date определить подсказку: «Дата, когда заказчик хочет получить свой товар» — это мало поможет пользователю. Чтобы предоставить полную информацию об этом элементе управления, напишите так: «Самая ранняя дата, когда товары могут быть поставлены. По умолчанию товар должен быть поставлен через три дня после даты заказа, но вы можете изменить значе-

ние даты, щелкнув мышью поле и введя другую дату. Нажмите F1 для получения подробной информации».

Звуковые сигналы

Звуковые сигналы, выдаваемые компьютером — мощное средство оповещения, но проектировщик может использовать их не только во благо, но и во зло. Простейший пример неадекватного использования звуковых сигналов — гудок, оповещающий пользователя, что он совершил недопустимые действия.

Вместо того чтобы запугивать пользователя резкими звуками, лучше сообщите ему что-нибудь приятное. Например, если введенные данные соответствуют всем необходимым требованиям, пусть компьютер негромко произнесет что-нибудь вроде «OK, все верно». Но если возникла какая-то проблема, лучше всего вывести сообщение об ошибке в строке состояния, а не «звонить во все колокола», раздражая пользователя и окружающих громкими звуками.

На мой взгляд, лучше всего иллюстрирует эти слова пример, авторство которого принадлежит Алану Куперу. Когда вы нажимаете клавишу, клавиатура издает тихий щелчок. Вы, скорее всего, не обращаете внимания на эти звуки, но если клавиатура вдруг откажется «щелкать», обязательно поймете, что что-то не так.

Не опасайтесь, что звуковые подтверждения, выдаваемые при правильном вводе данных, создадут излишний шум в офисе. Да, фраза «OK, все верно» должна выделяться на фоне офисного шума. Но мне приходилось использовать звуковые сообщения при проектировании систем телефонной связи, когда в одной комнате находилось более 100 одновременно работающих операторов, и никаких проблем не возникало.

Сообщения об ошибках

Очень жаль, но большинство пользователей не считает сообщения об ошибках одним из видов помощи, воспринимая их, скорее, как заслуженное наказание. Конечно, это не так, ведь каждая ошибка — это очередной повод помочь пользователю. Хорошо воспитанный человек, прося о помощи, не пытается сразу же обругать помощника, не доказывает, что в его проблемах виноват кто-то еще. Он ясно, доходчиво и вежливо объясняет возникшую проблему, и то, почему он обратился за помощью.

«Хорошо воспитанная» компьютерная система ведет себя так же. Она даже должна быть более вежливой, чем человек. Когда возникает ошибка времени исполнения, система обязана;

- объяснить пользователю на понятном ему языке, что произошло;
- вежливо попросить о помощи;
- не просить пользователя сделать то, что система должна сделать самостоятельно;
- описать все действия, которые мог бы предпринять пользователь.

Иногда системы безнадежно «зависают», или недостаток оперативной памяти или дискового пространства заставляет пользователя вмешиваться в работу программы. На этот случай вы практически ничего не можете предпринять, кроме как предусмотреть вывод на экран соответствующего сообщения, чтобы проинформировать пользователя о том, что же он должен сделать.

Ясное и написанное понятным языком сообщение поможет выйти из ситуации с наименьшими потерями. Если пользователь осознал суть проблемы, он скорее всего избежит ее в дальнейшем — если, конечно, подобной ситуации можно избежать вообще.

Объяснив пользователю дальнейшие действия, вы поможете ему справиться с проблемой самостоятельно. Не предоставляйте ему возможности выбора, в которых он скорее всего запутается. Помните: то, что очевидно для вас как для специалиста, вовсе не очевидно для пользователя. Не бойтесь объяснений, но и не увлекайтесь деталями.

Сообщение должно быть написано вежливо. Пользователи оценят дружелюбие системы и, может быть, простят вам ваши ошибки. Помните, никто не совершенен.

Активная помощь

Пассивные и реактивные механизмы все более широко применяются в компьютерной индустрии, по мере того как растет наш опыт взаимодействия пользователей и компьютеров. Последняя категория механизмов поддержки — *активные механизмы,* но число систем, которые их используют, пока невелико.

Принцип активной помощи прост: система отслеживает действия пользователя и в случае необходимости сама предлагает ему помощь или подсказку. Например, Office Assistant (Помощник из приложений Microsoft Office) предлагает пользователям контекстные справки в зависимости от их действий,

Очень интересен такой механизм активной помощи, как интеллектуальный агент. Это программа, которой пользователь делегирует выполнение определенных задач. Ее обычно используют в Web-приложениях для ответа на просьбы типа: «Найди мне этот товар за наименьшую цену» или «Предложи книгу, которая мне понравится». Но ничто не ограничивает область применения интеллектуальных аген-

тов только Web-приложениями. Можно спроектировать программу, помогающую студентам планировать свое расписание, например: «Занятия по математике — до обеда, а языковые курсы — вечером».

Microsoft предлагает два способа организации интеллектуального агента — с помощью Microsoft Office Assistant и Microsoft Agent. Большинство пользователей знакомы с «говорящей скрепкой для бумаг» — Microsoft Office Assistant, но не каждый знает, что Office Assistant позволяет обращаться к его интерфейсу из других программ. Office Assistant доступен только из приложений Microsoft Office и не распространяется вместе с механизмом баз данных Microsoft Access.

Если вы работаете с Access (или любым другим средством разработки, поддерживающим элементы Microsoft ActiveX, включая Visual Basic), можете скачать Microsoft Agent SDK с Web-узла Microsoft. Этот пакет обладает расширенными возможностями, по сравнению с Office Assistant.

Microsoft Agent — очень занятная игрушка. Вы можете создавать свои заставки-агенты с помощью SDK. Microsoft Agent также поддерживает голосовые сообщения. Мне очень нравится включенный в поставку Microsoft SQL Server 7.0 интерфейс Microsoft Agent к приложению Microsoft English Query — он обеспечивает обработку SQL-запросов. Вообразите, что база данных представлена в виде заставки Microsoft Agent — каково?

Однако ни Office Assistant, ни Microsoft Agent не обеспечивают поддержку реакции на действия пользователя. Они предоставляют интерфейс взаимодействия с пользователем, но обязанность связать действия пользователя и реакцию на них — целиком возлагается на разработчика.

Обучение пользователя

Обучение пользователя состоит в предоставлении ему необходимых материалов, класса и квалифицированного преподавателя. Но список этим не исчерпывается.

Если вы решили организовать обучение в форме компьютерных курсов, помните, для какой аудитории они предназначены. Начинающих пользователей прежде всего интересует, что может сделать система, а не то, как она работает.

Объем курсов зависит от сложности системы и от размеров бюджета, выделенного на обучение. Многие системы требуют лишь простых и кратких вводных курсов. Более сложные программы — создания больших структурированных обучающих систем, с разбиением на главы и контрольными вопросами.

Опытные пользователи хотят знать, как выполнять те или иные операции, и обычно им хватает встроенной контекстной помощи. Фактически, разница между помощью и обучением достаточно расплывчата. Тем не менее, сложные системы могут потребовать создания специального курса и для опытных пользователей.

Вам следует отделять понятие обучающего курса от собственно системы. Обучение должно происходить в режиме реального времени, но для этого никогда не следует использовать реально действующее приложение.

Итоги

В этой главе мы изучили три вида механизмов поддержки пользователя: встроенный в пользовательский интерфейс пассивный механизм, реактивный механизм, откликающийся на действия пользователя, и активный механизм, действия которого определяются системой.

Мы рассмотрели три типа пассивных механизмов: запоминающиеся сочетания клавиш, ярлыки и строки состояний. Реактивные механизмы реализуют в форме подсказок или разъяснений в режиме реального времени, они могут включать звуковые эффекты или сообщения об ошибках. Наконец, мы обсудили сравнительно новый механизм поддержки пользователей — активный, и вкратце — проблемы обучения пользователей.

Словарь терминов

абстрактная сущность (abstract entity) — сущность, моделирующая связи между другими сущностями.

агрегатная функция (aggregate function) — функция SQL, возвращающая суммарные значения.

активная помощь пользователю (proactive user assistance) — механизм помощи пользователю, при создании которого стараются предугадать, где и какая именно помощь может потребоваться.

альтернативный ключ (alternate key) — ключ-кандидат в отношении, который не используется в качестве внешнего ключа таблицы.

аномалия обновления (update anomaly) — ошибка операции манипулирования данными, возникающая в результате недочетов при создании модели данных.

атрибут (attribute) — столбец отношения.

база данных (database) — схема базы данных и хранимые данные.

базовое отношение (base relation) — отношение, которое на физическом уровне представлено таблицей в базе данных.

бизнес-ограничение (business constraint) — ограничение, основанное на предметной области.

бизнес-правило (business rule) — ограничение целостности, основанное на предметной области, а не вытекающее из реляционной теории,

булево выражение (Boolean expression) — выражение, результатом которого может быть либо значение *True,* либо *False.*

вектор команды (command vector) — один из способов выполнения команды в пользовательском интерфейсе: например, при помощи элемента меню или кнопки на панели управления.

внешнее отношение (foreign relation) — отношение, получающее внешний ключ от другого участника связи.

внешнее соединение (outer join) — соединение, возвращающее все записи, присутствующие во внутреннем соединении, а также все записи, присутствующие в одном или в обоих **других** участниках соединения.

внутреннее ограничение (intrinsic constraint) — ограничение, определяющее физическую структуру базы данных.

внутреннее соединение (inner join) — соединение, возвращающее записи, только если результатом

выполнения операции является значение *True*.

главное отношение (primary relation) — отношение, первичный ключ которого хранится в другом участвующем в связи отношении.

двойная связь (binary relationship) — связь, в которой два участника.

декартово произведение (Cartesian product) — реляционная операция, выполняющая соединение каждой записи одного набора записей с каждой записью другого набора.

декларативная целостность (declarative integrity) — метод определения ограничений целостности, при котором ограничения явно объявляются при определении таблиц.

декомпозиция без потерь (lossless decomposition) возможность разделить отношения так, чтобы при объединении получившихся в результате этого отношений информация не была утеряна.

домен (domain) — диапазон значений, которые может принимать атрибут.

домены совместимых типов (type-compatible domains) — домены, для которых допускается логическое сравнение.

естественное соединение (natural join) — особый случай эквисоединения; при этом соединение основано на операции равенства, в нем участвуют все общие и только один набор общих полей включается в набор результатов.

заголовок отношения (relation heading) — определение атрибута и домена, размещаемое в верхней части отношения.

задача (task) — отдельный шаг в рабочем процессе.

замыкание (closure) — принцип, декларирующий, что результатом всех операций над отношением является отношение, над которым можно выполнять другие операции.

запись (record) - физическое представление кортежа.

запрос (query) — производное отношение в Microsoft Access.

каскадное обновление (cascading update) — автоматическое обновление сущностей во внешнем отношении при изменении соответствующей сущности главного отношения.

ключ-кандидат (candidate key) - один или несколько атрибутов, уникально идентифицирующих отношение.

кортеж (tuple) — строка в отношении.

левое внешнее соединение (left outerjoin) — внешнее соединение, возвращающее все поля первого набора записей в операторе SELECT.

модель данных (data model) — концептуальное описание предметной области в терминах реляционной модели.

мощность отношения (cardinality of a relation) — число строк в отношении.

мощность связи (cardinality of a relationship) — максимальное число экземпляров сущности, которые могут участвовать в отношении.

набор записей (recordset) — в Microsoft Access — физическое представление отношений.

обычная сущность (regular entity) — сущность, которая может существовать независимо от ее участия в связи.

ограничение на уровне домена (domain constraint) — ограничение целостности, определяющее диапазон допустимых для домена значений.

ограничение на уровне сущности (entity constraint) — ограничение целостности, гарантирующее действительность сущностей, моделируемых в системе.

ограничение целостности (integrity constraint) — правило, поддерживающее целостность данных.

ограничение, определенное для базы данных (database constraint) — ограничение целостности, ссылающееся на множество отношений.

осиротевшая сущность (orphan entity) — слабая сущность, не связанная ни с одной сущностью в главном отношении.

отношение (relation) — логическая конструкция, в которой данные представлены в виде строк и столбцов.

пассивная помощь пользователю (passive user assistance) — механизм помощи пользователю, встроенный в интерфейс.

первичный ключ (primary key) — ключ-кандидат отношения, позволяющий уникально идентифицировать записи в таблице.

поле (field) — представление атрибута в базе данных на физическом уровне.

полное внешнее соединение (full outer join) — внешнее соединение, возвращающее все поля обоих участников соединения.

пользовательский отчет (ad hoc report) — отчет, форму которого может задать пользователь.

правое внешнее соединение (right outer join) — внешнее соединение, возвращающее все поля второго набора записей, перечисленного в операторе SELECT.

предметная область (problem space) — часть реального мира, моделируемая приложением базы данных.

представление (view) — производное отношение в Microsoft SQL Server.

приложение базы данных (database application) - набор форм и отчетов, с которыми работает пользователь.

производное отношение (derived relation) — виртуальное отноше-

ние, определяемое в терминах других отношений.

промежуточная таблица (junction table) — таблица, представляющая отношение в базе данных.

простой ключ (simple key) -- ключ-кандидат, состоящий из родного атрибута.

рабочий процесс (work process) - некая деятельность, выполняемая *с* использованием приложения базы данных.

размерность отношения (degree of a relation) — число столбцов в отношении.

размерность связи (degree of a relationship) — число участников связи.

реактивные механизмы помощи пользователю (reactive user assistance) — помощь, вызываемая каким-либо действием пользователя: например, неправильным вводом запроса для контекстной справки.

реальная сущность (concrete entity) — сущность, моделирующая объект или событие реального мира.

реляционная разность (relational difference) — реляционная операция, возвращающая только те записи одного набора, которые совпадают с записями другого набора.

реляционное деление (relational divide) — соединение, возвращающее все записи первого набора, значения которых совпадают со всеми соответствующими значениями второго набора.

реляционное объединение (relational union) — конкатенация двух наборов записей.

реляционное пересечение (relational intersection) — реляционная операция, возвращающая записи, общие для двух наборов.

связь (relationship) — связь, существующая между двумя или более сущностями.

система баз данных (database system) — приложение базы данных, механизм СУБД и база данных.

скалярное значение (scalar value) — одиночное, неповторяющееся значение.

слабая сущность (weak entity) -- сущность, которая может существовать, только если участвует в связи.

составная сущность (composite entity) — сущность предметной области, моделируемая одним или несколькими отношениями.

составной ключ (composite key) — ключ-кандидат, состоящий из двух или более атрибутов.

ссылочная целостность (referential integrity) — ограничения целостности, гарантирующие целостность связей между сущностями.

стандартный отчет (standard report) — отчет, определяемый и реализуемый как часть приложения базы данных.

сущность (entity) — любой объект или понятие, информация о котором должна храниться в системе.

схема (schema) — физическая схема таблиц в системе баз данных.

схема базы данных (database schema) — физическая схема таблиц в базе данных.

таблица (table) - физическое представление отношения в схеме базы данных.

тело отношения (relation body) - кортежи, составляющие отношение.

тета-соединение (theta-join) — теоретически: любое соединение, основанное на операторе сравнения; как правило, термин используется для обозначения соединений, основанных на операторах, отличных от оператора равенства.

трехзначная логика (three-valued logic) — логическая модель, в которой любое выражение может иметь одно из трех значений: *True*, *False* или *Null*.

тройная связь (ternary relationship) — связь, число участников которой равно трем.

унарная связь (unary relationship) — связь отношения с самим собой.

участник связи (participant) — сущность, для которой существует связь с другой сущностью,

целостность данных (data integrity) - правила, используемые в базе данных, и гарантирующие, что хранимые данные, даже если не являются точными, по крайней мере правдоподобны.

целостность транзакций (transaction integrity) — ограничение целостности, реализующее проверку правильности множественных операций с базой данных.

целостность, обеспечиваемая процедурами, или процедурная целостность (procedural integrity) — метод усиления целостности данных, в основе которого лежит создание процедур, автоматически выполняемых при обновлении, добавлении или удалении данных.

эквисоединение (equi-join) — соединение двух таблиц, в основе которого реляционный оператор равенства.

Рекомендуемая литература

Часть I. Теория реляционных баз данных

Date C. J. An Introduction to Database Systems. 7th ed. Reading. Mass.: Addison-Wesley Publishing Company. 1999.

Date C. J, and Darwen Huge. Foundation for Object/Relational Databases: The Third Manifesto. Reading. Mass.: Addison-Wesley Publishing Company. 1998.

Fleming Candace C. and von Halle Barbara. Handbook of Relational Database Design. Reading. Mass.: Addison-Wesley Publishing Company. 1989.

Teorey Toby J. Database Modeling & Design. 3rd ed. San Francisco: Morgan Kaufmann Publishers, 1999.

Часть II. Проектирование реляционных систем баз данных

Gilb Tom and Susanna Finzi. Principles of Software Engineering Management. Reading. Mass.: Addison-Wesley Publishing Company. 1988.

Haught Dan and Ferguson Jim. Microsoft Jet Database Engine Programmer's Guide. 2nd ed. Redmond-Wash.: Microsoft Press. 1997.

McConnell Steve. Rapid Development. Redmond-Wash.: Microsoft Press. 1996.

Pressman Roger S. Software Engineering: A Practitioner's Approach. 3rd ed. New York: McGraw-Hill. 1992.

Sommerville Ian. Software Engineering. 6th ed. Reading, Mass.: Addison-Wesley Publishing Company. 1996.

Soukup Ron. Inside Microsoft SQL Server 6.5. Redmond-Wash.: Microsoft Press. 1997[1].

[1] Эта книга издана и на русском языке: *Саукап Рон.* Основы Microsoft **SQL** Server 6.5. М.; «Русская редакция». 1998. — *Прим, переводчика.*

Часть III. Проектирование пользовательского интерфейса

Cooper Alan. About Face: The Essentials of User Interface Design. Foster City, Cal.: IDG Books Worldwide. 1995.

Heckel Paul. The Elements of Friendly Software Design. New York: Warner Books. 1991.

Mandel Paul. The Elements of User Interface Design. New York: John Wiley & Sons. 1997.

Microsoft Corporation. The Windows Interface Guidelines for Software Design. Redmond-Wash.: Microsoft Press. 1998.

Shneiderman Ben. Designing the User Interface: Strategies for Effective Human-Computer Interaction. Reading. Mass.: Addison-Wesley Publishing Company. 1980.

Предметный указатель

D

Data Access Objects **(DAO)**
см. модель, доступа к данным объектная, DAO

M

Microsoft Access 2–5, 194
Microsoft ActiveX 121, 261
Microsoft ActiveX Data Objects (ADO) *см.* модель, доступа к данным объектная, ADO
Microsoft Jet 7–10, 55, 81-89, 91–93, 178; *см. также* механизм СУБД
Microsoft SQL Server 2-3, 8-10, 55, 81–97, 113-114, 178, 194; *см, также* механизм СУБД

R

Remote Data Objects (RDO)
см, модель, доступа к данным объектная, RDO

S

Structured Query Language (SQL) 33, 94-99

А

агрегирование 109–110, 257, 300-301
алгебра реляционная 94
— оператор
— — CUBE 109, 113–114
— — ROLLUP 109–113
— — TRANSFORM 109-112
— — ограничения 97–98
— операция
— — агрегирование 109–110
— — декартово произведение 108-109
— — деление 104
— — над множествами 104
— — объединение 105
— — переименование 109–111
— — пересечение 106–108
— — проекция 98
— — разность 107-108
— — расширение 109–110
— — соединение 98–104
альтернативный ключ
см. ключ, альтернативный
анализ стоимостный 137–141
архитектура
- данных 171, 181–188
— — двухуровневая 186–188
— — многоуровневая 171, 188–189
— — одноуровневая 182–185
— программная 171–172
— — многоуровневая 172
— — трехуровневая 172-174
— — четырехуровневая 174–178
- СУБД 236
атрибут 12-25, 34, 165-168, 245-255
— моделирование 156–159
— связываемый 76, 160
— тип данных 21–22

Б

база данных 2–5
— анализ 117
— внутренние ограничения 275–276
— диаграмма 55–56
— инструменты разработки 9
— манипулирование данными 7
— модель 28-33
— нормальная форма 34
— объект 154–161
— представление данных 10
— проектирование 122–124, 142
— реляционная 2–3; *см. также* механизм СУБД
— средства доступа к данным 7-8
— **структура 34**
-схема 3–5, 81, 94, 122, 171, 178–192, 207–208
— физическая модель 3, 5
— хранение данных 7
бизнес-правило 72-73, 80, 86, 145–148, 163-165, 172–180, 261, 276–277, 286-291
— нарушение ограничений 287-291
бизнес-процесс 129, 143
бизнес-требование 118, 130-132

Д

данные 154
— агрегированные 257
— архитектура 181–182
— декларативная целостность 85. *см, также* целостность, данных
— денормализация 190
— «договорное значение» 82
— доступ 7–8
— — ADO 8
— — DAO 8
— **-ODBC 8**
— — OLE DB 8
— — RDO 8
-зашита 196–200
— зависимость
— — многозначная 48
— — соединения 49–50
— значение *Null* 82-84, 95, 109-110, 280–281
— — в ключевых и индексных полях 91
— логические операции 84
— модель 2-3, 13-14, 23-24, 31–34, 52-57, 93, 154, 242
— — звездообразная 9
— — иерархическая 9
— — концептуальная 171, 191–192, 207
— — реляционная 2–3, 9-10. 57
— — сетевая 9
— — обработка 84
-защита 2, 196-200
— — на уровне пользователя 197-198
— — от случайных ошибок 287-288
— — регистрация системных событий 199–200
— — требования системной безопасности 196
— — уровень общего доступа 197
— значение по умолчанию 89
— иерархическая **структура** 252-254
— избыточность 28-30, 34, 42, 48, 78
— логическая структура 171
— манипулирование 7-9, 80

— маска ввода 272–273
— моделирование 40-41, 74–75, 129–130, 155
— модель 31, 122, 242
— неопределенные значения 74-78, 81-83
— нормализация 170
— обработка 210
— ограничение
— — диапазона значений 168, 261
— — формата 272–273
— отображение пользователю 247, 251, 256–258, 267
— набор значений 263–264
— отсутствующие значения 74-75, 81–83
— переменной длины 279
— проверка правильности 178-180, 192-193
— — при вводе 261
— произвольные значения 262
— процедурная целостность 85
— сортировка 193, 294
— структура 238
— тип 21-23, 39–41, 75, 259-261
— — дата 269-270
— — совместимые 22–23, 39
— — текстовый 269–272
— — числовой 269
— физическая структура 122, 171, 275
— фильтрация 293–298
— формат 170, 260-261
— функциональная зависимость 37-38, 46-47
— хранение 7–9
— целостность 48, 72–73, 81, 178–181, 275
— — базы 79–80, 93
— — домена 85–86
— — нарушение 85
— — переходов 75–76
— — сущности 76–78, 86-90
— — транзакций 80
— — ограничения 275
— — преобразования 93
— — транзакции 93
декартово произведение 108-109
декомпозиция 34, 42, 50, 78
диаграмма
— рабочих процессов 206, *см. также* рабочий процесс
— «сущности–связи» 25–27, 53-56, 165, 207, 228, 243
— — средства построения 55-56
диалоговая панель управления см. интерфейс пользователя, диалоговая панель управления
документация проектная 203-204
— контроль за изменениями 210-211
— средства создания 210
домен 13, 21-23, 39-41, 73-75, 165-168
— анализ 167–170

Е

естественное соединение *см.* соединение, естественное

З

зависимость *см.* данные, зависимость
заголовок отношения *см.* отношение
задача 142–143, 240, *см. также* рабочий процесс

запись
— блокирование 227
— каскадное обновление 79
запрос 94, 194–196, 298
— параметрический 195
— связанный [04
— создание с помощью Microsoft English Query 298–299
защита данных *см.* данные, защита

И

избыточностьданных *см.* данные, избыточность
индекс 184–187, 193-196
— влияние на производительность 184-186, 194-196
— кластерный 185
— уникальный 91
Интернет-архитектура 189–191
— тонкий клиент 189
— толстый клиент 191
интерфейс пользователя 208, 213-215
— архитектура 228–241
— — многодокументная 234-239
— — однодокументная 231—232, 239
— — проект 238–239
— — рабочая книга 232–233
— — стиль Microsoft Outlook 233-234
— вектор команды 317
— диалоговая панель управления 237-238
— «дружественный» 217, 229-231
— мастер 240–241
— — поддержка рабочих процессов 240-241
— — сохранение данных при вводе 241
— модель 208
— — декларируемая 215–216
— — ментальная пользователя 215–216, 237, 259
— — реализации 215–216
— навигация 250–251
— отображение
— — данных 244–245
— — модели «сущности-связи» 242-243
-отчет 308–310
— поддержка
— — контекста 245–246
— — пользователя 316–328
— — рабочих процессов 228
— последовательный 224–226
— представление
— — данных 259
— — связей 246-248
— проектирование 214–215, 219–224
— — форм 226-227
— прототип 208–209
— спецификация 209–210
— тестирование 216
— форма
— — вспомогательная 245–246, 249
— — модальная 246
— — основная 237–238
— — подчиненная 248–250, 267
— элементуправления
— — выбор и размещение 244-247, 259, 262
— — выключатель 262
— — дерево 267-268
— — иерархическая таблица 267
— — кнопка–выключатель 266

— — кнопка с зависимой фиксацией 266
— — комбинированное окно 263-265
— — комбинированный список 264
— — настраиваемый табличный элемент 267
— — окно списка 263, 265–266
— — переключатель 263
— — полоса прокрутки 271
— — раскрывающийся список 264-265
— — связанная пара списков 268-269
— — текстовое поле с настраиваемыми свойствами 273
— — флажок 263–64
интранет–архитектура 189–191

К

клиент
— толстый *см.* Интернет–архитектура, толстый клиент
— тонкий *см.* Интернет-архитектура, тонкий клиент
клиент-серверная система 5, 186–188; *см. также* приложение, клиент–серверная архитектура
клиентское приложение *см.* приложение, клиентское
ключ 35-38, 193
— альтернативный 36–37
— внешний 78–79, 193
— искусственный 37
— кандидат 35–46, 63, 66, 78. 244
— ограничение уникальности 91
— первичный 36–37, 43, 91. 178, 185–186, 193, 283-284
— полный 42
— простой 36
— составной 36, 42, 46
конкатенация 224
контекст пользователя 257
кортеж 12

Л

логика, трехзначная 95–96
— операторы 96–97

М

механизм СУБД 5-7, 55, 179-180
- Microsoft Jet 7, 182-185
— Microsoft SQL Server 7
многодокументная архитектура *см.* интерфейс пользователя, архитектура, многодокументная
многозначная зависимость *см.* данные, зависимость, многозначная
модель
— данных *см.* данные, модель
— доступа к данным объектная 7-8, 190
— — ADO 8, 190
— — DAO 8
— — RDO 8
— жизненного цикла 117
— — водопада 118
— — спиральная 119
— — эволюционная разработка 120–121
— реляционная 2–3, 9–10, 34, 82, 94, 275, *см. также* данные, модель

— служб *см.* архитектуру, программная
мощность
— отношения 12
— связи 54, 69–70

Н

набор результатов 10–12
нормализация 34–39, 45–52, 170; *см. также* нормальная форма
нормальная форма *см.* форма, нормальная

О

объектная модель доступа к данным *см.* модель, доступа к данным, объектная
ограничение 98, 163–165, 192, 277–286
-CHECK 192
— внешнего ключа 91–92
— внутреннее 275-279
— диапазона значений данных 168, 281-282
— длины данных 278–279
— на уровне атрибута 192
— на уровне домена 73–75, 192, *см. также* целостность, данных, домена
— на уровне сущности 192, 282–286
— переходов 75–76
— процедурное 80
— ссылочной целостности 78–79, *см. также* целостность, ссылочная
— типа данных 278
— формата данных 278
— целостности 72–74, 275, *см. также* целостность, данных
— — бизнес-правила 275–276
— — внутреннее 275–276
— — на уровне базы данных 79-80
— — на уровне домена 74–75
однодокументная архитектура *см.* интерфейс пользователя, архитектура, однодокументная
окно приложения 231–238, 240-241
— активное 236-237
— дочернее 235-236
— основное 234–239
оператор
— CUBE 109, 113–114
— ROLLUP 109, 112–113
— TRANSFORM 109–112
операция
— над данными 40
— над отношением 10
отношение 9–13
— абстрактное 57–58
— атрибут 12
— базовое 94
— заголовок 12
— ключ 35–36
— «многие ко многим» 67
— моделирование 29–33, 52, 65-68
— мощность 12
— «один ко многим» 68, 252
— производное 94
— размерность 12
— ссылающееся 55–57, 63, 66, 192-193
— ссылочное 56–57, 63, 66, 192-193
— тело 12
— участие в связи 54–55
отчет
— на основе форм пользовательского интерфейса 301

— обработка ошибок принтера 303-306
— подробный 299-301
— пользовательский 293, 306-307
— — критерии построения 31j
— — настраиваемый 307–313
— представление пользователю 302-303
— с использованием агрегированных данных 300–301
— средства создания 292–293, 301-302, 307
— — сортировка данных 294
— — фильтры 293-298
— стандартный 293, 299–301
— форматирование 300

П

пейджинг 190
первичный ключ *см.* ключ, первичный
пересечение *см.* алгебра реляционная
поддержка пользователя 316-317
— активный механизм 327–328
— — интеллектуальный агент 327-328
— пассивный механизм 318–322
— — всплывающие подсказки 319-320
— — сочетания клавиш 318–319
— — строка состояния 321–322
— реактивный механизм 322-327
— — звуковой сигнал 326
— — контекстная справка 323-324
— сообщения об ошибках 326-327
пользователь
— интервьюирование 143–144, 150–152
— начинающий 217
— обучение 328–329
— опытный 218
— эксперт 218
пользовательский интерфейс *см,* интерфейс пользователя
предметная область 3, 13–14, 28, 163–164
представление 194–195
приложение
— баз данных 5, 8–9
— — физическая архитектура 171
— клиент–серверная архитектура 5, 186–188; *см. также* архитектура, данных, двухуровневая
— клиентское 5, 236
— логическая структура 171
— пользовательское 5
— — реализация ограничения целостности 85
— серверное 5
— средства разработки 8–9
принцип замыкания 10–11, *см. также* операция, над отношением
программная архитектура *см.* архитектура, программная
проектная документация *см.* документация, проектная
проекция 98
промежуточная таблица *см.* таблица, **промежуточная**
процедура 85

Р

рабочий процесс 79–80, 122, 143-152, 154-155, 163-165, 195, 206, 228, 232, 238, 242, 254
- анализ 143-144, 148–152
— документирование 150–152
— задача 142–152
размерность
— отношения 12
— связи 24, 52
разность *см.* алгебра, реляционная
реляционная алгебра *см.* алгебра, реляционная
реляционная модель *см.* модель, реляционная
риски проекта 204

С

связанный запрос *см.* запрос, связанный
связь 23-25, 50-52, 72, 92, 191-193, 243, 247, 259
— атрибуты 163
— двойная 24, 52
— «многие ко многим» 24–27, 64–65, 242, 255-257
— моделирование 55–70, 156-163
— мощность 54, 69–70, 160-162
— направление 68–69
— необязательная 53–54
— неявная 24
— обязательная 53–54, 160–162
— «один к одному» 24–27, 58–63, 92, 195, 242, 247
— «один ко многим» 24–27, 248-250
— ограничения 163
— реализация в пользовательском интерфейсе 246-253
— полная 53
— размерность 24, 52
- тип 53-55, 58
- тройная 24, 52, 67-69
— унарная 52, 65–66
— участник 24, 52, 160
— частичная 53
— явная 24
система баз данных 2-5
— архитектура 171–172
— границы применения 124–130, 136–137, 204
— документирование 202–205
— критерии 124, 131-135
— масштаб 136–137
— направления разработки 135-136
— пользовательский интерфейс 136
— реляционная 2, *см. также* база данных
- цель 124-130, 135, 204
система управления базами данных (СУБД) *см.* система баз данных
системная архитектура 171–172, 192; *см. также* архитектура
скаляр 9–11, 39
служба 172
совместимые типы данных *см.* тип данных, совместимые; данные, тип, совместимые
соединение 98-99, 100-109
— внешнее 103
— — левое 103
— — полное 103-104
— — правое 103
— естественное 100
— таблиц 51; *см. также* таблица, соединение
— эквисоединение 99

— тета-соединение 101–104
сортировка данных
 см. данные, сортировка
справочная система *см.* интерфейс пользователя; поддержка пользователя
средства доступа к данным
 см. данные, доступ
ссылающееся отношение
 см, отношение, ссылающееся
ссылочная таблица
 см. таблица, ссылочная
ссылочная целостность
 см. целостность, ссылочная
ссылочное отношение
 см. отношение, ссылочное
стоимостный анализ
 см. анализ стоимостный
структура данных *см.* данные, структура
сущность 13-20, 72, 245-247
— абстрактная 15
— атрибут 14–19
— взаимодействие с другими 163–165
— диаграмма состояний перехода 57-58
— зависимая 163
— конкретная 15
— моделирование 13-23, 59–64, 156–159, 243
— надтип 24
— обычная 53
— повторный анализ 163
— подтип 14–15, 24
— подчиненная 247
— простая 243, 247
-связь 23, 52–54, 72
— — с предметной областью 163–164
— — техника представления 54
— — тип 24
— слабая 53
— участие
— — в рабочих процессах 164-165
— — в связи 24, 52-53
— целостность 90
схема базы данных *см.* база данных, схема
сценарий пользовательский 152-153

Т

таблица 10, 190–191, 243
— временная 190
— подчиненная 253–254, 267
— промежуточная 65, 255-257, 311
— связанная 92
— соединение 51, 193
— сортировка данных 193
— ссылающаяся 78
— ссылочная 78
тело отношения *см.* отношение, тело
тета–соединение *см.* соединение, тета–соединение
тип данных 21–23, 39–41, 87-89, 168-369, 279;
 см. также данные, тип
— логический 73, 261–262
— ограничения 262
— определяемый пользователем 85-86, 89
— совместимые 22–23, 39
— текстовый 260–261
— физический 74
— числовой 261
толстый клиент *см.* Интернет-архитектура, толстый клиент

тонкий клиент *см.* Интернет-архитектура, тонкий клиент
транзакция 80-81
— зависимость 81
— откат 81
трехзначная логика *см.* логика, трехзначная
трехуровневая архитектура *см.* архитектура, программная, трехуровневая
триггер 85, 91, 178, 191

У

участник связи *см.* связь, участник

Ф

фильтр 293-298
— по выделенному фрагменту 294-295
— по заданным значениям в полях 296-297
— расширенный 297-298
форма
— нормальная 34
— — Бойса–Кодда 45–49
— — вторая 42–43, 49
— — первая 39-40
— — пятая 45, 49–51
— — третья 43–45, 170
— — четвертая 45, 48–49
формат данных *см.* данные, формат
функциональная зависимость *см.* данные, функциональная зависимость

Х

хранимая процедура 191–196

Ц

целостность; *см. также* данные, целостность
— базы данных 79
— данных 72-86, 85, 93
— — декларативная 85
— — на уровне атрибута 76
— — на уровне базы данных 79-80 ,
— — на уровне домена 75–77, 85-86
— — на уровне переходов 75-76
— — на уровне сущности 76–78, 86-90
— — нарушение 84–85
— — ограничение 72-75, 86-90, 178, 277-286
— — процедурная 85
— — реализация 81
— -ссылочная 78-79, 82-83, 86, 91–92, 193, 282–285
— домена 73-75
цель проекта 122, 124, 204

Ч

четырехуровневая архитектура *см.* архитектура, программная, четырехуровневая

Э

эквисоединение *см.* соединение, эквисоединение

Об авторе

Ребекка Райордан - признанный эксперт в области разработки и внедрения систем баз данных. Опыт ее работы в данной области составляет почти два десятка лет. Ребекка работает независимым консультантом по разработке и поддержке систем баз данных и систем, поддерживающих рабочие процессы. В 1998 г. корпорация Microsoft присвоила ей статус MVP за активную работу в группах Интернет-новостей.

В настоящее время Ребекка Райордан проживает в Амстердаме (Нидерланды). Связаться с ней можно по адресу: rebeccar@ibm.net

ЛИЦЕНЗИОННОЕ СОГЛАШЕНИЕ MICROSOFT

(прилагаемый к книге компакт-диск)

ЭТО ВАЖНО - ПРОЧИТАЙТЕ ВНИМАТЕЛЬНО. Настоящее лицензионное соглашение (далее «Соглашение") является юридическим документом, оно заключается между Вами (физическим или юридическим лицом) и Microsoft Corporation (далее «корпорация Microsoft») на указанный выше продукт Microsoft, который включает программное обеспечение и может включать сопутствующие мультимедийные и печатные материалы, а также электронную документацию (далее «Программный Продукт»). Любой компонент, входящий в Программный Продукт, который сопровождается отдельным Соглашением, подпадает под действие именно того Соглашения, а не условий, изложенных ниже. Установка, копирование или иное использование данного Программного Продукта означает принятие Вами данного Соглашения. Если Вы не принимаете его условия, го не имеете права устанавливать, копировать или както иначе использовать этот Программный Продукт.

ЛИЦЕНЗИЯ НА ПРОГРАММНЫЙ ПРОДУКТ

Программный Продукт защищен законами Соединенных Штатов по авторскому праву и международными договорами по авторскому праву, а также другими законами и договорами по правам на интеллектуальную собственность.

1, **ОБЪЕМ ЛИЦЕНЗИИ.** Настоящее Соглашение дает Вам право:

a) **Программный продукт.** Вы можете установить и использовать одну копию Программного Продукта на одном компьютере. Основной пользователь компьютера, на котором установлен данный Программный Продукт, может сделать только для себя вторую копию и использовагь ее на портативном компьютере.

b) **Хранение или использование в сети.** Вы можете также скопировать или установить экземпляр Программного Продукта на устройстве хранения, например на сетевом сервере, исключительно для установки или запуска данного Программного Продукта на других компьютерах в своей внутренней сети, но тогда Вы должны приобрести лицензии на каждый такой компьютер. Лицензию на данный Программный продукт нельзя использовать совместно или одновременно на других компьютерах.

c) **License Pak.** Если Вы купили эту лицензию в составе Microsoft License Pak, можете сделать ряд дополнительных копий программного обеспечения, входящего в данный Программный Продукт, и использовать каждую копию так, как было описано выше. Кроме того, Вы получаете право сделать соответствующее число вторичных копий для портативного компьютера в целях, также оговоренных выше.

d) **Примеры** кода. Это относится исключительно к отдельным частям Программного Продукта, заявленным как примеры кода (далее «Примеры»), если таковые входят в состав Программного Продукта.

i) **Использование и модификация.** Microsoft дает Вам право использовать и модифицировать исходный код Примеров при условии соблюдения пункта (d)(iii) ниже. Вы не имеете права распространять в виде исходного кода ни Примеры, ни их модифицированную версию.

ii) Распространяемые файлы. При соблюдении пункта (d)(iii) Microsoft дает Вам право на свободное от отчислении копирование и распространение в виде объектного кода Примеров или их модифицированной версии, кроме тех частей (или их модифицированных версий), которые оговорены в файле Readme, относящемся к данному Программному Продукту, как не подлежащие распространению.

iii) Требования к распространению файлов. Вы можете распространять файлы, разрешенные к распространению, при условии, что: а) распространяете их в виде объектного кода только в сочетании со своим приложением и как его часть; б) не используете название, эмблему или товарные знаки Microsoft для продвижения своего приложения; в) включаете имеющуюся в Программном Продукте ссылку на авторские права в состав этикетки и заставки своего приложения; г) согласны освободить от ответственности и взять на себя защиту корпорации Microsoft от любых претензий или преследований по закону, включая судебные издержки, если таковые возникнут в результате использования или распространения Вашего приложения; и д) не допускаете дальнейшего распространения конечным пользователем своего приложения. По поводу отчислений и других условий лицензии применительно к иным видам использования или распространения распространяемых файлов обращайтесь в Microsoft.

2. ПРОЧИЕ ПРАВА И ОГРАНИЧЕНИЯ

- Ограничения на **реконструкцию, декомпиляцию** и **дизассемблирование.** Вы не имеете права реконструировать, декомпилировать или дизассемблировать данный Программный Продукт, кроме того случая, когда такая деятельность (только в той мере, которая необходима) явно разрешается соответствующим законом, несмотря на это ограничение.

- Разделение компонентов. Данный Программный Продукт лицензируется как единый продукт. Его компоненты нельзя отделять друг от друга для использования более чем на одном компьютере.

- Аренда. Данный Программный Продукт нельзя сдавать в прокат, передавать во временное пользование или уступать для использования в иных целях.

- Услуги по технической поддержке. Microsoft может (но не обязана) предоставить Вам услуги по технической поддержке данного Программного Продукта (далее «Услуги»). Предоставление Услуг регулируется соответствующими правилами и программами Microsoft, описанными в руководстве пользователя, электронной документации и/или других материалах, публикуемых Microsoft. Любой дополнительный программный код, предоставленный в рамках Услуг, следует считать частью данного Программного Продукта и подпадающим под действие настоящего Соглашения. Что касается технической информации, предоставляемой Вами корпорации Microsoft при использовании ее Услуг, то Microsoft может задействовать эту информацию в деловых целях, в том числе для технической поддержки продукта и разработки. Используя такую техническую информацию, Microsoft не будет ссылаться на Вас.

- Передача прав на программное обеспечение. Вы можете безвозвратно уступить все права, регулируемые настоящим Соглашением, при условии, что не оставите себе никаких копий, передадите все составные части данного Программного Продукта (включая компоненты, мультимедийные и печатные материалы, любые об-

новления, Соглашение и сертификат подлинности, если таковой имеется) и принимающая сторона согласится с условиями настоящего Соглашения.

- **Прекращение действия** Соглашения. Без ущерба для любых других прав Microsoft может прекратить действие настоящего Соглашения, если Вы нарушите его условия. В этом случае Вы должны будете уничтожить все копии данного Программного Продукта вместе со всеми его компонентами.

3. АВТОРСКОЕ ПРАВО. Все авторские права и право собственности на Программный Продукт (в том числе любые изображения, фотографии, анимации, видео, аудио, музыку, текст, примеры кода, распространяемые файлы и апплеты, включенные в состав Программного Продукта) и любые его копии принадлежат корпорации Microsoft или ее поставщикам. Программный Продукт охраняется законодательством об авторских правах и положениями международных договоров. Таким образом, Вы должны обращаться с данным Программным Продуктом, как с любым другим материалом, охраняемым авторскими правами, с тем исключением, что Вы можете установить Программный Продукт на один компьютер при условии, что храните оригинал исключительно как резервную или архивную копию. Копирование печатных материалов, поставляемых вместе с Программным Продуктом, запрещается.

ОГРАНИЧЕНИЕ ГАРАНТИИ

ДАННЫЙ ПРОГРАММНЫЙ ПРОДУКТ (ВКЛЮЧАЯ ИНСТРУКЦИИ ПО ЕГО ИСПОЛЬЗОВАНИЮ) ПРЕДОСТАВЛЯЕТСЯ БЕЗ КАКОЙ-ЛИБО ГАРАНТИИ. КОРПОРАЦИЯ MICROSOFT СНИМАЕТС СЕБЯ ЛЮБУЮ ВОЗМОЖНУЮ ОТВЕТСТВЕННОСТЬ, ВТОМЧИСЛЕ ОТВЕТСТВЕННОСТЬ ЗА КОММЕРЧЕСКУЮ ЦЕННОСТЬ ИЛИ СООТВЕТСТВИЕ ОПРЕДЕЛЕННЫМ ЦЕЛЯМ. ВЕСЬ РИСК ПО ИСПОЛЬЗОВАНИЮ ИЛИ РАБОТЕ С ПРОГРАММНЫМ ПРОДУКТОМ ЛОЖИТСЯ НА ВАС.

НИ ПРИ КАКИХ ОБСТОЯТЕЛЬСТВАХ КОРПОРАЦИЯ MICROSOFT. ЕЕ РАЗРАБОТЧИКИ, А ТАКЖЕ ВСЕ, ЗАНЯТЫЕ В СОЗДАНИИ, ПРОИЗВОДСТВЕ И РАСПРОСТРАНЕНИИ ДАННОГО ПРОГРАММНОГО ПРОДУКТА, НЕ НЕСУТ ОТВЕТСТВЕННОСТИ ЗА КАКОЙ-ЛИБО УЩЕРБ (ВКЛЮЧАЯ ВСЕ, БЕЗ ИСКЛЮЧЕНИЯ, СЛУЧАИ УПУЩЕННОЙ ВЫГОДЫ. НАРУШЕНИЯ ХОЗЯЙСТВЕННОЙ ДЕЯТЕЛЬНОСТИ, ПОТЕРИ ИНФОРМАЦИИ ИЛИ ДРУГИХ УБЫТКОВ) ВСЛЕДСТВИЕ ИСПОЛЬЗОВАНИЯ ИЛИ НЕВОЗМОЖНОСТИ ИСПОЛЬЗОВАНИЯ ДАННОГО ПРОГРАММНОГО ПРОДУКТА ИЛИ ДОКУМЕНТАЦИИ, ДАЖЕ ЕСЛИ КОРПОРАЦИЯ MICROSOFT БЫЛА ИЗВЕЩЕНА О ВОЗМОЖНОСТИ ТАКИХ ПОТЕРЬ, ТАК КАК В НЕКОТОРЫХ СТРАНАХ НЕ РАЗРЕШЕНО ИСКЛЮЧЕНИЕ ИЛИ ОГРАНИЧЕНИЕ ОТВЕТСТВЕННОСТИ ЗА НЕПРЕДНАМЕРЕННЫЙ УЩЕРБ, УКАЗАННОЕ ОГРАНИЧЕНИЕ МОЖЕТ ВАС НЕ КОСНУТЬСЯ.

РАЗНОЕ

Настоящее Соглашение регулируется законодательством штата Вашингтон (США), кроме случаев (и лишь в той мере, насколько это необходимо) исключительной юрисдикции того государства, на территории которого используется Программный Продукт. Если у Вас возникли какие-либо вопросы, касающиеся настоящего Соглашения, или если Вы желаете связаться с Microsoft по любой другой причине, пожалуйста, обращайтесь в местное представительство Microsoft или пишите по адресу: Microsoft Sales Information Center, One Microsoft Way, Redmond, WA 98052-6399.

Ребекка Райордан

Основы реляционных баз данных

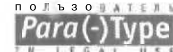
Перевод с английского под общей редакцией ***Н. Б. Желновой***

Технический редактор ***С. В, Дергачев***

Компьютерная верстка ***Д. В. Петухов***

Дизайнер обложки ***Е. В. Козлова***

Оригинал-макет выполнен с использованием издательской системы Adobe PageMaker 6.0



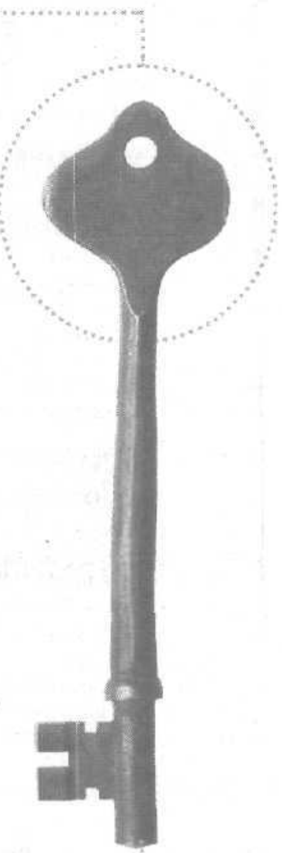
Главный редактор ***А. И. Козлов***

Подготовлено к печати издательско-торговым домом «Русская Редакция»

Лицензия ЛР № 066422 от 19.03.99 г.
Подписано в печать с готовых диапозитивов 18.09.01 г.
Формат 60 x 90 1/16. Физ. печ. л. 24.
Тираж 3000 экз. Заказ № 190.

ОАО «Санкт-Петербургская типография № 6».
193144, Санкт-Петербург, ул. Моисеенко, 10.
Телефон отдела маркетинга 271-35-42.

информация из ПЕРВЫХ РУК!



Из каждого номера Windows 2000 Magazine/RE Вы будете получать самую последнюю информацию о том, как добиться от Windows 2000 Professional и Windows 2000 Server максимальной производительности. Вы узнаете, какие существуют программные продукты для Windows 2000 и как они работают, как Windows 2000 Professional или Windows 2000 Server ведут себя в сети, куда обращаться за помощью и многое, многое другое.

Как подписчик журнала, Вы будете иметь бесплатный доступ к электронным приложениям "Профессионалам Windows NT/2000" и "SQL Server Magazine ONLINE". В них Вы найдете дополнительную информацию по операционным системам семейства Windows 2000 и продуктам, работающим на их основе.

ЗАЧЕМ ИСКАТЬ?

МОЖНО ПОДПИСАТЬСЯ!

www.win2000mag.ru letters@win2000mag.ru

Подписка во всех отделениях связи по каталогу Роспечати – 79741, АПР – 38185

117036 г. Москва,
ул. Шверника, д. 4
тел.: (095) 232-0023
e-mail: info@softline.ru

Все для разработки ПО

Почему опытные разработчики приобретают нужные для их работы программы в компании SoftLine?

- Их привлекают низкие цены, т.к. компания работает напрямую с вендорами.
- Их привлекает имеющаяся возможность получения демо-версий и обновлений.
- В выборе программ им помогают каталог Soft Line-direct и сайт www.softline.ru.
- Большая часть ассортимента SoftLine для разработчиков недоступна в других компаниях.

Какие этапы разработки охватывает программное обеспечение, поставляемое SoftLine?

- Проектирование программ (Microsoft, CA/Platinum, Rational, SilverRun, Quest).
- Совместная работа (Centura, Merant, Microsoft).
- Управление проектами (PlanisWare, PlanView, Microsoft).
- Написание кода (среды разработки Allaire, Borland, IBM, Microsoft, компоненты Allround Automation, ComponentOne, Crystal Decisions, Janus, Sitraka, Stingray).
- Оптимизация кода (Compaq, Fuji, Intel, MainSoft, Sun, Sybase, Tenberry).
- Отладка и тестирование (NuMega, Intuitive Systems, Segue).
- Упаковка приложений (InstallShield, Wise Solutions).
- Развертывание и поддержка (Remedy, RoyalBlue, CA, Network Associates).
- Обучение пользователей (Adobe, Allen Communications, click2learn.com, eHelp, Macromedia, Quest, Ulead).

SoftLine — это свобода выбора

Обратившись в SoftLine, вы в кратчайшие сроки решите проблемы с программным обеспечением. Получив консультацию менеджеров, часть из которых знакома с работой разработчиков не понаслышке (на собственном опыте), вы подберете все необходимое для работы в вашей области - от интегрированной среды RAD - до готовых компонент. При этом мы оставим выбор идеологии разработки за вами - например, для регулярного получения информации о продуктах и технологиях, вы сможете подписаться на Microsoft Developer Network, Sun Developer Essentials или на нашу собственную рассылку компакт-дисков - SoftLine Support Subscription, предоставляющую обновления и демо-версии всех ведущих производителей. Компания SoftLine также поможет вам в выборе обучающих курсов.

Microsoft
Borland

<allaire>

NuMega

InstallShield.
SOFTWARE CORPORATION

ComponentOne

ПРОФЕССИОНАЛЬНЫЙ ЖУРНАЛ

ПРОГРАММИСТ

Профессиональный журнал, посвященный исключительно вопросам разработки. Наши авторы - профессиональные программисты, которые делятся с читателями «секретами мастерства». Мы публикуем материалы о самых современных технологиях и средствах разработки, статьи о принципах и методах, теории и практике программирования. Мы предлагаем статьи на самые разные «программерские» темы, любого уровня сложности

Подписка

Индекс в каталоге агентства "Роспечать" - 80467, в каталоге "Пресса России" - 45775.

www.programme.ru '
info@programme.ru

HARD'n'SOFT

ПОПРОБУЙ ЖЕЛЕЗО НА ВКУС!

В НАШЕМ ЖУРНАЛЕ И ТОЛЬКО ДЛЯ ВАС:

- НОВОСТИ КОМПЬЮТЕРНОЙ ИНДУСТРИИ
- ПОДРОБНОСТИ О СОВРЕМЕННЫХ И ПЕРСПЕКТИВНЫХ ТЕХНОЛОГИЯХ
- ТЕСТЫ И ОБЗОРЫ АППАРАТНЫХ И ПРОГРАММНЫХ ПРОДУКТОВ
- КОНСУЛЬТАЦИИ ЭКСПЕРТОВ, ВСТРЕЧИ С ИНТЕРЕСНЫМИ ЛЮДЬМИ
- ВИКТОРИНЫ С ЦЕННЫМИ ПРИЗАМИ
- CD-ПРИЛОЖЕНИЕ С ПОЛЕЗНЫМИ УТИЛИТАМИ

НАШИ ИНДЕКСЫ: HARD'N'SOFT - 73140, HARD N SOFT * CD - 26067

Издательство «Русская Редакция» — партнер Microsoft Press в России — предлагает широкий выбор литературы по современным информационным технологиям. Мы переводим на русский язык бестселлеры ведущих издательств мира, а также сотрудничаем с компетентными российскими авторами.

Наши книги Вы можете приобрести

- **в Москве:**
  - «Библио-Глобус» ул. Мясницкая, 6, теп : (095) 926-3557
  - «Московский дом книги» ул Новый Арбат, 8 ТЕЛ (095) 290-4507
  - «Дом технической книги» Ленинский пр-т, 40, тел : (095) 137-6019
  - «Молодая гвардия» ул. большая Полянка, 28 тел.: (095) 238-5001
  - "Дом книги на Соколе» Ленинградский пр т, 78, тел • (095) 152-4511
  - «Мир печати» ул 2-я Тверская-Ямская, 54. тел.: (095) 978-5047
  - Торговый дои книги «Москва» уп Тверская, 8, теп (095] 229-6463
  - «Алекс я К°» Магазин «Книги» г. Зеленоград, Панфиловский пр-т, 1106Б, тел.: (095) 532-9669

Фирменный магазин «Компьютерная и деловая книга»
Москва, Ленинский проспект, строение 38, тел.: (095) 778-7269

- **в Санкт-Петербурге:**
  ЗАО «Диалект», тел : (312) 247-1483
- **в Новосибирске:**
  ООО «Топ-книга», тел. (3832) 36-1026
- **в Набережных Челнах:**
  000 «Аспект-С», тел.: (8552) 58-8013
- **в Алма-Ате (Казахстан):**
  ЧП Болат Амреев, тел : 8-327-290-191-25, (3272) 26-1404
- **в Киеве (Украина):**
  ООО Издательство «Ирина», тел.: (044) 269-0423
  «Техническая книга на Петровке», тел (044) 464-6895
- **в Минске (Белоруссия):**
  000 «Попурри» тел.: 8-10-375-17-2225726
  ОАО «Аргфинанс» тел 8-10-375-17-2366716

Интернет-магазин http://www.ITbook.ru

тел.: (095) 142-0571; **тел./факс:** (095) 145-4519
e-mail: info@rusedit.ru; **http://** www.rusedit.ru

ОСНОВЫ РЕЛЯЦИОННЫХ БАЗ ДАННЫХ

Эта книга поможет Вам разрабатывать коммерческие программные продукты с использованием современных технологий. В ней изложены теоретические сведения о реляционных базах данных и основные принципы проектирования приложений, работающих с базами данных. Особое внимание уделено собственно процессу разработки СУБД и используемым при этом технологиям.
Книга содержит множество примеров, полезных как начинающим, так и опытным разработчикам.

На прилагаемом компакт-диске:

- **документы и шаблоны Microsoft Word, которые помогут Вам в создании документации при разработке СУБД;**
- **тематические статьи из обзоров Microsoft и Microsoft Knowledge Base.**

ISBN 5-7502-0150-3

9 785750 201501

Базовый курс
Теория и практика

Издание содержит:

- **основные понятия, а также принципы и методы, положенные в основу реляционной модели;**
- **подробное описание возможностей, предоставляемых широко распространенными средствами разработки реляционных баз данных — Microsoft Access, Microsoft SQL Server, а также полный обзор средств Microsoft Visual Basic для создания высокоэффективных приложений, работающих с базами данных;**
- **ответы на типичные вопросы и способы устранения проблем, возникающих при разработке баз данных;**
- **советы по созданию удобного пользовательского интерфейса для приложений баз данных;**
- **сведения о разработке структуры данных на основе рабочих процессов.**

Сайт издательства: **www.rusedit.ru**
Интернет-магазин: **www.ITbook.ru**