The Addison-Wesley Signature Series

ВИЛЬЯМС

A Martin Fowler Signature Book

ШАБЛОНЫ КОРПОРАТИВНЫХ ПРИЛОЖЕНИЙ

ИСПРАВЛЕННОЕ ИЗДАНИЕ

Мартин Фаулер
при участии
Дейвида Райса,
Мэттью Фоммела,
Эдварда Хайета,
Роберта Ми
и Рэнди Стаффорда



AVB

Шаблоны корпоративных приложений

Patterns of Enterprise Application Architecture

Martin Fowler

With contributions from David Rice,
Matthew Foemmel, Edward Hieatt,
Robert Me, and Randy Stafford



ADDISON-WESLEY

Boston · San Francisco · New York · Toronto · Montreal
London · Munich · Paris · Madrid · Capetown · Sydney
Tokyo · Singapore · Mexico City

Шаблоны корпоративных приложений

ИСПРАВЛЕННОЕ ИЗДАНИЕ

Мартин Фаулер

при участии
Дейвида Райса, Мэттью Фоммела,
Эдварда Хайета, Роберта Ми
и Рэнди Стаффорда



Москва ♦ Санкт-Петербург ♦ Киев
2016

ББК 32.973.26-018.2.75
Ф28
УДК 681.3.07

Издательский дом "Вильямс"
Зав. редакцией *С.Н. Тригуб*
По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Фаулер, Мартин.
Ф28 Шаблоны корпоративных приложений. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2016. — 544 с. : ил. — Парал. тит. англ.
ISBN 978-5-8459-1611-2 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, если на это нет письменного разрешения издательства Addison-Wesley Publishing Company, Inc.

Authorized translation from the English language edition published by Pearson Education, Inc., Copyright © 2003

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2016

Книга отпечатана согласно договору с ООО "Дальрегионсервис".

Научно-популярное издание
Мартин Фаулер
Шаблоны корпоративных приложений

Литературный редактор *Т.П. Кайгородова*
Верстка *О.В. Мишутина*
Художественный редактор *В.Г. Павлютин*
Корректоры *Л.А. Гордиенко, Л.В. Чернокозинская*

Подписано в печать 13.05.2016. Формат 70x100/16
Гарнитура Times
Усл. печ. л. 34,0. Уч.-изд. л. 30,9
Тираж 200 экз. Заказ № 2798.

Отпечатано в АО «Первая Образцовая типография»
Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, д.1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, т/ф. 8(499)270-73-59

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1611-2 (рус.)
ISBN 0-321-12742-0 (англ.)

© Издательский дом "Вильямс", 2016
© Pearson Education, Inc., 2003

Оглавление

Предисловие 17
Введение 27

Часть I. Обзор 41
Глава 1. "Расслоение" системы 43
Глава 2. Организация бизнес-логики 51
Глава 3. Объектные модели и реляционные базы данных 59
Глава 4. Представление данных в Web 81
Глава 5. Управление параллельными заданиями 87
Глава 6. Сеансы и состояния 105
Глава 7. Стратегии распределенных вычислений 111
Глава 8. Общая картина 119

Часть II. Типовые решения 131
Глава 9. Представление бизнес-логики 133
Глава 10. Архитектурные типовые решения источников данных 167
Глава 11. Объектно-реляционные типовые решения, предназначенные для моделирования поведения 205
Глава 12. Объектно-реляционные типовые решения, предназначенные для моделирования структуры 237
Глава 13. Типовые решения объектно-реляционного отображения с использованием метаданных 325
Глава 14. Типовые решения, предназначенные для представления данных в Web 347
Глава 15. Типовые решения распределенной обработки данных 405
Глава 16. Типовые решения для обработки задач автономного параллелизма 433
Глава 17. Типовые решения для хранения состояния сеанса 473
Глава 18. Базовые типовые решения 483

Список основных источников информации 527
Предметный указатель 532

Содержание

Предисловие 17
Введение 27

Часть I. Обзор 41

Глава 1. "Расслоение" системы 43
Развитие модели слоев в корпоративных программных приложениях 44
Три основных слоя 46
Где должны функционировать слои 48

Глава 2. Организация бизнес-логики 51
Выбор типового решения 55
Уровень служб 56

Глава 3. Объектные модели и реляционные базы данных 59
Архитектурные решения 59
Функциональные проблемы 64
Считывание данных 66
Взаимное отображение объектов и реляционных структур 67
Отображение связей 67
Наследование 71
Реализация отображения 73
Двойное отображение 74
Использование метаданных 75
Соединение с базой данных 76
Другие проблемы 78
Дополнительные источники информации 79

Глава 4. Представление данных в Web 81
Типовые решения представлений 84
Типовые решения входных контроллеров 86
Дополнительные источники информации 86

Глава 5. Управление параллельными заданиями 87
Проблемы параллелизма 88
Контексты выполнения 89
Изолированность и устойчивость данных 91
Стратегии блокирования 91
Предотвращение возможности несогласованного чтения данных 93
Разрешение взаимоблокировок 94
Транзакции 95
ACID: свойства транзакций 96
Ресурсы транзакций 96
Уровни изоляции 97
Системные транзакции и бизнес-транзакции 99
Типовые решения задачи обеспечения автономного параллелизма 101
Параллельные операции и серверы приложений 102
Дополнительные источники информации 104

Глава 6. Сеансы и состояния 105
В чем преимущество отсутствия "состояния" 105
Состояние сеанса 107
Способы сохранения состояния сеанса 108

Глава 7. Стратегии распределенных вычислений 111
Соблазны модели распределенных объектов 111
Интерфейсы локального и удаленного вызова 112
Когда без распределения не обойтись 114
Сужение границ распределения 115
Интерфейсы распределения 116

Глава 8. Общая картина 119
Предметная область 120
Источник данных 121
Источник данных для сценария транзакции 121
Источник данных для модуля таблицы 122
Источник данных для модели предметной области 122
Слой представления 123
Платформы и инструменты 124
Java и J2EE 124
.NET 125
Хранимые процедуры 126
Web-службы 126
Другие модели слоев 127

Часть II. Типовые решения 131

Глава 9. Представление бизнес-логики 133

Сценарий транзакции (Transaction Script) 133
Принцип действия 133
Назначение 135
Задача определения зачтенного дохода 135
Пример: определение зачтенного дохода (Java) 136
Модель предметной области (Domain Model) 140
Принцип действия 140
Назначение 143
Дополнительные источники информации 143
Пример: определение зачтенного дохода (Java) 144
Модуль таблицы (Table Module) 148
Принцип действия 149
Назначение 151
Пример: определение зачтенного дохода (C#) 152
Слой служб (Service Layer) 156
Принцип действия 157
Разновидности "бизнес-логики" 157
Варианты реализации 157
Быть или не быть удаленному доступу 158
Определение необходимых служб и операций 158
Назначение 160
Дополнительные источники информации 160
Пример: определение зачтенного дохода (Java) 161

Глава 10. Архитектурные типовые решения источников данных 167

Шлюз таблицы данных (Table Data Gateway) 167
Принцип действия 167
Назначение 168
Дополнительные источники информации 169
Пример: класс PersonGateway (C#) 170
Пример: использование объектов ADO.NET DataSet (C#) 172
Шлюз записи данных (Row Data Gateway) 175
Принцип действия 175
Назначение 176
Пример: запись о сотруднике (Java) 178
Пример: использование диспетчера данных для объекта домена (Java) 181
Активная запись (Active Record) 182
Принцип действия 182
Назначение 184
Пример: простой класс Person (Java) 184
Преобразователь данных (Data Mapper) 187
Принцип действия 187
Обращение к методам поиска 190

Отображение данных на поля объектов домена 191
Отображения на основе метаданных 192
Назначение 192
Пример: простой преобразователь данных (Java) 193
Пример: отделение методов поиска (Java) 198
Пример: создание пустого объекта (Java) 201

Глава 11. Объектно-реляционные типовые решения, предназначенные для моделирования поведения 205
Единица работы (Unit of Work) 205
Принцип действия 206
Назначение 211
Пример: регистрация посредством изменяемого объекта (Java) 212
Коллекция объектов (Identity Map) 216
Принцип действия 216
Выбор ключей 217
Явная или универсальная? 217
Сколько нужно коллекций? 217
Куда их поместить? 218
Назначение 219
Пример: методы для работы с коллекцией объектов (Java) 219
Загрузка по требованию (Lazy Load) 220
Принцип действия 221
Назначение 223
Пример: инициализация по требованию (Java) 224
Пример: виртуальный прокси-объект (Java) 224
Пример: использование диспетчера значения (Java) 226
Пример: использование фиктивных объектов (C#) 227

Глава 12. Объектно-реляционные типовые решения, предназначенные для моделирования структуры 237
Поле идентификации (Identity Field) 237
Принцип действия 237
Выбор ключа 238
Представление поля идентификации в объекте 239
Вычисление нового значения ключа 240
Назначение 242
Дополнительные источники информации 243
Пример: числовой ключ (C#) 243
Пример: использование таблицы ключей (Java) 244
Пример: использование составного ключа (Java) 246
Класс ключа 246
Чтение 249
Вставка 252
Обновление и удаление 256
Отображение внешних ключей (Foreign Key Mapping) 258
Принцип действия 258
Назначение 261

Пример: однозначная ссылка (Java) 262
Пример: многотабличный поиск (Java) 265
Пример: коллекция ссылок (C#) 266
Отображение с помощью таблицы ассоциаций (Association Table Mapping) 269
Принцип действия 270
Назначение 270
Пример: служащие и профессиональные качества (C#) 271
Пример: использование SQL для непосредственного обращения к базе данных (Java) 274
Пример: загрузка сведений о нескольких служащих посредством одного запроса (Java) 278
Отображение зависимых объектов (Dependent Mapping) 283
Принцип действия 283
Назначение 285
Пример: альбомы и композиции (Java) 285
Внедренное значение (Embedded Value) 288
Принцип действия 289
Назначение 289
Дополнительные источники информации 290
Пример: простой объект-значение (Java) 290
Сериализованный крупный объект (Serialized LOB) 292
Принцип действия 292
Назначение 294
Пример: сериализация иерархии отделов в формат XML (Java) 294
Наследование с одной таблицей (Single Table Inheritance) 297
Принцип действия 298
Назначение 298
Пример: общая таблица игроков (C#) 299
Загрузка объекта из базы данных 301
Обновление объекта 303
Вставка объекта 303
Удаление объекта 304
Наследование с таблицами для каждого класса (Class Table Inheritance) 305
Принцип действия 305
Назначение 306
Дополнительные источники информации 307
Пример: семейство игроков (C#) 307
Загрузка объекта 307
Обновление объекта 310
Вставка объекта 311
Удаление объекта 312
Наследование с таблицами для каждого конкретного класса (Concrete Table Inheritance) 313
Принцип действия 314
Назначение 315

Пример: конкретные классы игроков (C#) 316
Загрузка объекта из базы данных 318
Обновление объекта 320
Вставка объекта 320
Удаление объекта 321
Преобразователи наследования (Inheritance Mappers) 322
Принцип действия 323
Назначение 324

Глава 13. Типовые решения объектно-реляционного отображения с использованием метаданных 325
Отображение метаданных (Metadata Mapping) 325
Принцип действия 326
Назначение 327
Пример: использование метаданных и метода отражения (Java) 328
Хранение метаданных 328
Поиск по идентификатору 330
Запись в базу данных 332
Извлечение множества объектов 334
Объект запроса (Query Object) 335
Принцип действия 336
Назначение 337
Дополнительные источники информации 337
Пример: простой объект запроса (Java) 337
Хранилище (Repository) 341
Принцип действия 342
Назначение 343
Дополнительные источники информации 344
Пример: поиск подчиненных заданного сотрудника (Java) 344
Пример: выбор стратегий хранилища (Java) 345

Глава 14. Типовые решения, предназначенные для представления данных в Web 347
Модель–представление–контроллер (Model View Controller) 347
Принцип действия 348
Назначение 350
Контроллер страниц (Page Controller) 350
Принцип действия 351
Назначение 352
Пример: простое отображение с помощью контроллера-сервлета и представления JSP (Java) 352
Пример: использование страницы JSP в качестве обработчика запросов (Java) 355
Пример: обработка запросов страницей сервера с применением механизма разделения кода и представления (C#) 358
Контроллер запросов (Front Controller) 362
Принцип действия 362
Назначение 364

Дополнительные источники информации 364
Пример: простое отображение (Java) 365
Представление по шаблону (Template View) 368
Принцип действия 369
Вставка маркеров 369
Вспомогательный объект 370
Условное отображение 370
Итерация 371
Обработка страницы 372
Использование сценариев 372
Назначение 372
Пример: использование страницы JSP в качестве представления с вынесением контроллера в отдельный объект (Java) 373
Пример: страница сервера ASP.NET (C#) 375
Представление с преобразованием (Transform View) 379
Принцип действия 379
Назначение 380
Пример: простое преобразование (Java) 381
Двухэтапное представление (Two Step View) 383
Принцип действия 383
Назначение 385
Пример: двухэтапное применение XSLT (XSLT) 390
Пример: страницы JSP и пользовательские дескрипторы (Java) 393
Контроллер приложения (Application Controller) 397
Принцип действия 398
Назначение 400
Дополнительные источники информации 400
Пример: модель состояний контроллера приложения (Java) 400

Глава 15. Типовые решения распределенной обработки данных 405
Интерфейс удаленного доступа (Remote Facade) 405
Принцип действия 406
Интерфейс удаленного доступа и типовое решение интерфейс сеанса (Session Facade) 409
Слой служб 409
Назначение 410
Пример: использование компонента сеанса Java в качестве интерфейса удаленного доступа (Java) 410
Пример: Web-служба (C#) 414
Объект переноса данных (Data Transfer Object) 419
Принцип действия 419
Сериализация объекта переноса данных 421
Сборка объекта переноса данных из объектов домена 423
Назначение 424
Дополнительные источники информации 424
Пример: передача информации об альбомах (Java) 425
Пример: сериализация с использованием XML (Java) 429

Глава 16. Типовые решения для обработки задач автономного параллелизма 433
Оптимистическая автономная блокировка (Optimistic Offline Lock) 434
Принцип действия 435
Назначение 439
Пример: слой домена с преобразователями данных (Java) 439
Пессимистическая автономная блокировка (Pessimistic Offline Lock) 445
Принцип действия 446
Назначение 450
Пример: простой диспетчер блокировки (Java) 450
Блокировка с низкой степенью детализации (Coarse-Grained Lock) 457
Принцип действия 457
Назначение 460
Пример: общая оптимистическая автономная блокировка (Java) 460
Пример: общая пессимистическая автономная блокировка (Java) 466
Пример: оптимистическая автономная блокировка корневого элемента (Java) 467
Неявная блокировка (Implicit Lock) 468
Принцип действия 469
Назначение 470
Пример: неявная пессимистическая автономная блокировка (Java) 470

Глава 17. Типовые решения для хранения состояния сеанса 473
Сохранение состояния сеанса на стороне клиента (Client Session State) 473
Принцип действия 473
Назначение 474
Сохранение состояния сеанса на стороне сервера (Server Session State) 475
Принцип действия 475
Назначение 478
Сохранение состояния сеанса в базе данных (Database Session State) 479
Принцип действия 479
Назначение 481

Глава 18. Базовые типовые решения 483
Шлюз (Gateway) 483
Принцип действия 484
Назначение 484
Пример: создание шлюза к службе отправки сообщений (Java) 485
Преобразователь (Mapper) 489
Принцип действия 490
Назначение 490
Супертип слоя (Layer Supertype) 491
Принцип действия 491
Назначение 491
Пример: объект домена (Java) 491
Отделенный интерфейс (Separated Interface) 492
Принцип действия 493
Назначение 494

Реестр (Registry) 495
Принцип действия 495
Назначение 497
Пример: реестр с единственным экземпляром (Java) 498
Пример: реестр, уникальный в пределах потока (Java) 499
Объект-значение (Value Object) 500
Принцип действия 501
Назначение 502
Совпадение названий 502
Деньги (Money) 502
Принцип действия 503
Назначение 506
Пример: класс Money (Java) 506
Частный случай (Special Case) 511
Принцип действия 512
Назначение 512
Дополнительные источники информации 512
Пример: объект NullEmployee (C#) 513
Дополнительный модуль (Plugin) 514
Принцип действия 514
Назначение 515
Пример: генератор идентификаторов (Java) 516
Фиктивная служба (Service Stub) 519
Принцип действия 519
Назначение 520
Пример: служба определения величины налога (Java) 521
Множество записей (Record Set) 523
Принцип действия 524
Явный интерфейс 524
Назначение 526

Список основных источников информации 527

Предметный указатель 532

Памяти Дениса Уильяма Фаулера (1922–2000)

посвящается

— Мартин

Предисловие

Весной 1999 года меня пригласили в Чикаго для консультаций по одному из проектов, осуществляемых силами ThoughtWorks — небольшой, но быстро развивавшейся компании, которая занималась разработкой программного обеспечения. Проект был достаточно амбициозен: речь шла о создании корпоративного лизингового приложения уровня сервера, которое должно было охватывать все аспекты проблем имущественного найма, возникающих после заключения договора: рассылку счетов, изменение условий аренды, предъявление санкций нанимателю, не внесшему плату в установленный срок, досрочное возвращение имущества и т.п. Все это могло бы звучать не так уж плохо, если не задумываться над тем, сколь разнообразны и сложны формы соглашений аренды. "Логика" бизнеса редко бывает последовательна и стройна, так как создается деловыми людьми для ситуаций, в которых какой-нибудь мелкий случайный фактор способен обусловить огромные различия в качестве сделки — от полного краха до неоспоримой победы.

Это как раз те вещи, которые интересовали меня прежде и не перестают волновать поныне: как прийти к системе объектов, способной упростить восприятие конкретной сложной проблемы. Я действительно убежден, что основное преимущество объектной парадигмы как раз и состоит в облегчении понимания запутанной логики. Разработка хорошей **модели предметной области (Domain Model, 140)**[1] для изощренной проблемы реального бизнеса весьма трудна, но ее решение приносит громадное удовлетворение.

Модель предметной области — это, однако, еще не все. Нашу модель следовало отобразить в базе данных. Как и во многих других случаях, мы использовали реляционную СУБД. Необходимо было снабдить решение пользовательским интерфейсом, обеспечить поддержку удаленных приложений и интеграцию со сторонними пакетами — и все это с привлечением новой технологии под названием J2EE, с которой никто не умел обращаться.

Несмотря на все трудности, мы обладали большим преимуществом — обширным опытом. Я длительное время проделывал аналогичные вещи с помощью C++, Smalltalk и CORBA. Многие члены команды ThoughtWorks в свое время серьезно поднаторели в Forte. В наших головах уже вертелись основные архитектурные идеи, оставалось только выплеснуть их на холст J2EE. (Теперь, по прошествии трех лет, я могу констатировать, что проект не блистал совершенством, но проверку временем выдержал очень хорошо.)

Для разрешения именно таких ситуаций и была задумана эта книга. На протяжении долгого времени мне доводилось иметь дело с массой корпоративных программных приложений. Эти проекты часто основывались на сходных идеях, эффективность которых

---

[1] Таким образом обозначается ссылка на страницу книги, где описано указанное типовое решение.

была доказана при решении сложных проблем, связанных с управлением на уровне предприятия. В книге делается попытка представить подобные проектные подходы в виде *типовых решений (patterns)*.

Книга состоит из двух частей. Первая содержит несколько ознакомительных глав с описанием ряда важных тем, имеющих отношение к сфере проектирования корпоративных приложений. Здесь бегло формулируются различные проблемы и варианты их решения. Все подробности и нюансы вынесены в главы второй части. Эту информацию уместно трактовать как справочную, и я не настаиваю на том, чтобы вы штудировали ее от начала до конца. На вашем месте я поступил бы так: детально ознакомился с материалом первой части для максимально полного понимания предмета и обратился к тем главам или типовым решениям из второй части, близкое знакомство с которыми действительно необходимо. Поэтому книгу можно воспринимать как краткий учебник (часть I), дополненный более увесистым руководством (часть II).

Итак, наше издание посвящено проектированию корпоративных программных приложений. Подобные приложения предполагают необходимость отображения, обработки и сохранения больших массивов (сложных) данных, а также реализации моделей бизнес-процессов, манипулирующих этими данными. Примерами могут служить системы бронирования билетов, финансовые приложения, пакеты программ торгового учета и т.п. Корпоративные приложения имеют ряд особенностей, связанных с подходами к решению возникающих проблем: они существенным образом отличаются от встроенных систем, систем управления, телекоммуникационных приложений, программных продуктов для персональных компьютеров и т.д. Поэтому, если вы специализируетесь в каких-либо "иных" направлениях, не связанных с корпоративными системами, эта книга, вероятно, не для вас (хотя, может быть, вам просто хочется "вкусить" нового?). За общей информацией об архитектуре программного обеспечения рекомендую обратиться к работе [33].

Проектирование корпоративных приложений сопряжено со слишком большим числом проблем архитектурного толка, и книга, боюсь, не сможет дать исчерпывающих ответов на все. Я поклонник итеративного подхода к созданию программного обеспечения. А сердцем концепции итеративной разработки является положение о том, что пользователю следует показывать первые, пусть не полные, результаты, если в них есть хоть толика здравого смысла. Хотя между написанием программ и книг существуют, мягко говоря, заметные различия, мне хотелось бы думать, что эта — далеко не всеобъемлющая — книга все-таки окажется своего рода конспектом полезных и поучительных советов. В ней освещаются следующие темы:

- "расслоение" приложения по уровням;
- структурирование логики предметной области;
- разработка пользовательского Web-интерфейса;
- связывание модулей, размещаемых в памяти (в частности, объектов), с реляционной базой данных;
- принципы распределения программных компонентов и данных.

Список тем, которых мы *не* будем касаться, разумеется, гораздо обширнее. Помимо всего остального, я предполагал обсудить вопросы проверки структуры, обмена сообщениями, асинхронных коммуникаций, безопасности, обработки ошибок, кластеризации, интеграции приложений, структурирования интерфейсов "толстых" клиентов

и прочее. Но ввиду ограничений на объем, отсутствия времени и нехватки оформившихся идей мне это не удалось. Остается надеяться, что в недалеком будущем какие-либо типовые решения в этих областях все-таки появятся. Возможно, когда-нибудь выйдет второй том книги, который вберет в себя все новое, или кто-то другой возьмет на себя труд заполнить эти и другие пробелы.

Среди всего перечисленного наиболее важной и сложной является проблема поддержки системы асинхронных коммуникаций, основанной на сообщениях. Особо острую форму она принимает при необходимости интеграции многих приложений (да и вариант системы коммуникаций для отдельно взятого приложения также "подарком" не назовешь).

В намерения автора *не* входила ориентация на какую бы то ни было конкретную программную платформу. Рассматриваемые типовые решения прошли первое испытание в конце 1980-х и начале 1990-х годов, когда я работал с C++, Smalltalk и CORBA. В конце 1990-х я начал интенсивно использовать Java и обнаружил, что те же подходы оказались приемлемыми при реализации и ранних гибридных систем Java/CORBA, и более поздних проектов на основе стандарта J2EE. Недавно я стал присматриваться к платформе Microsoft .NET и пришел к заключению, что решения вновь вполне применимы. Мои коллеги по ThoughtWorks подтвердили аналогичные выводы в отношении Forte. Я не собираюсь утверждать, что то же справедливо для *всех* платформ, современных и будущих, используемых для развертывания корпоративных приложений, но до сих пор дело обстояло именно так.

Описание большинства типовых решений сопровождается примерами кода. Выбор языка программирования обусловлен только вероятными предпочтениями читателей. Java в этом смысле выглядит наиболее привлекательно. Всякий, кто знаком с C или C++, разберется и в Java; кроме того, Java намного проще, нежели C++. Практически любой программист, использующий C++, способен воспринимать Java-код, но не наоборот. Я стойкий приверженец объектной парадигмы, поэтому речь могла идти только об одном из объектно-ориентированных языков. Итак, примеры написаны преимущественно на языке Java. Период работы над книгой совпал с этапом становления среды .NET и системы программирования C#, которая, по моему мнению, обладает многими свойствами, присущими Java. Поэтому я реализовал некоторые примеры и на C#, впрочем, с определенным риском, поскольку у разработчиков еще нет достаточного опыта взаимодействия с платформой .NET, так что идиомы ее использования, как говорится, не созрели. Оба выбранных мною языка наследуют черты C, поэтому если вы владеете одним, то сможете воспринимать — хотя бы поверхностно — и код, написанный на другом. Моей задачей было найти такой язык, который удобен для большинства разработчиков, даже если он не является их основным инструментом. (Приношу свои извинения всем, кому нравится Smalltalk, Delphi, Visual Basic, Perl, Python, Ruby, COBOL и т.д. Я знаю: вы хотите сказать, что есть языки получше, чем Java или C#. Полностью с вами согласен!)

Примеры, приведенные в книге, преследуют цель объяснить и проиллюстрировать основные идеи, лежащие в основе типовых решений. Их не нужно трактовать как окончательные результаты; если вы намерены воспользоваться ими в реальных ситуациях, вам придется проделать определенную работу. Типовые решения — удачная отправная точка, а не пункт назначения.

Для кого предназначена книга

Я писал эту книгу в расчете на программистов, проектировщиков и архитекторов, которые занимаются созданием корпоративных приложений и стремятся улучшить качество принимаемых стратегических решений.

Я подразумеваю, что большинство читателей относятся к одной из двух групп: первые, со скромными потребностями, озабочены созданием собственных программных продуктов, а вторые, более требовательные, намерены пользоваться соответствующими инструментальными средствами. Если говорить о первых, предлагаемые типовые решения помогут им сдвинуться с места, хотя затем потребуются, разумеется, и дополнительные усилия. Вторым, как я надеюсь, книга поможет понять, *что* происходит в "черном ящике" инструментальной системы, и сделать осознанный выбор в пользу того или иного поддерживаемого системой типового решения. Например, наличие средства отображения объектных структур в реляционные отнюдь не означает, что вам не придется делать самостоятельный осознанный выбор в конкретных ситуациях. В этом поможет знакомство с типовыми решениями.

Существует и третья, промежуточная, категория читателей. Им я порекомендовал бы осторожно подходить к выбору инструментальных средств. Я не раз наблюдал, как некоторые буквально погрязают в длительных упражнениях по созданию рабочей среды программирования, не имеющих ничего общего с истинными целями проекта. Если вы убеждены в правильности того, что делаете, дерзайте. Не забывайте, что многие примеры кода, приведенные в книге, намеренно упрощены для облегчения их восприятия, и вам, возможно, придется немало потрудиться, чтобы применить их в особо сложных случаях.

Поскольку типовые решения — это общеупотребительные результаты анализа повторяющихся проблем, вполне вероятно, что с *некоторыми* из них вы уже когда-либо сталкивались. Если программированием корпоративных приложений вы занимаетесь долгое время, не исключено, что вам знакомо *многое*. Я не утверждаю, что книга представляет собой коллекцию свежих знаний. Напротив, я стараюсь убедить вас в обратном: книга трактует (пусть зачастую по-новому) *старые* идеи, проверенные временем. Если вы новичок, книга, я надеюсь, поможет вам изучить предмет. Если вы уже с ним знакомы, книга поспособствует формализации и структурированию ваших знаний. Важная функция типовых решений связана с выработкой общего терминологического словаря: если вы скажете, что разрабатываемый вами класс является, например, разновидностью **интерфейса удаленного доступа (Remote Facade, 405)**, собеседникам должно быть понятно, о чем идет речь.

Благодарности

Книга многим обязана людям, с которыми мне пришлось сотрудничать в течение долгих лет. Проявления помощи были крайне разнообразны, поэтому каюсь, но порой я даже не могу вспомнить, *что* именно — значимое, принципиальное и заслуживающее упоминания на страницах книги — сообщил мне тот или иной человек, и тем не менее я признателен всем.

Начну с непосредственных участников проекта. Дейвид Райс (David Rice), мой коллега по ThoughtWorks, внес громадный вклад: добрый десяток процентов всего содержимого — это его прямая заслуга. На завершающей фазе проекта, стремясь поспеть к сроку (а Дейвид к тому же обеспечивал взаимодействие с заказчиком), мы провели вместе немало вечеров, и в одной из бесед он признался, что наконец-то понял, почему работа над книгой требует полной самоотдачи.

Еще один "мыслитель"[2], Мэттью Фоммел (Matthew Foemmel), оказался непревзойденным поставщиком примеров кода и немногословным, но острым критиком, хотя легче было бы растопить льды Арктики, нежели заставить его написать забавы ради пару абзацев текста. Я рад поблагодарить Рэнди Стаффорда (Randy Stafford) за его лепту в виде **слоя служб (Service Layer, 156)**, а также вспомнить Эдварда Хайета (Edward Hieatt) и Роберта Ми (Robert Mee). Сотрудничество с Робертом началось с того, что, просматривая материал, он обнаружил какое-то логическое несоответствие. Со временем он стал самым пристрастным и полезным критиком: его заслуги состоят не только в обнаружении каких-то пробелов, но и в их заполнении!

Я в большом долгу перед всеми, кто вошел в число официальных рецензентов книги, — Джоном Бруэром (John Brewer), Кайлом Брауном (Kyle Brown), Дженс Колдуэй (Jens Coldewey), Джоном Крапи (John Crupi), Леонардом Фенстером (Leonard Fenster), Аланом Найтом (Alan Knight), Робертом Ми, Жераром Месарашем (Gerard Meszaros), Дерк Райел (Dirk Riehle), Рэнди Стаффордом, Дейвидом Сигелом (David Siegel) и Каем Ю (Kai Yu). Я мог бы привести здесь полный список служащих ThoughtWorks — так много коллег помогали мне, сообщая о своих проектах и результатах. Многие типовые решения оформились благодаря счастливой возможности общения с талантливыми сотрудниками компании, поэтому у меня нет другого выбора, как поблагодарить коллектив в целом.

Кайл Браун, Рейчел Рейниц (Rachel Reinitz) и Бобби Вулф (Bobby Woolf) отложили все свои дела, чтобы встретиться со мной в Северной Каролине и подробнейшим образом обсудить материалы книги. Я с удовольствием вспоминаю и наши с Кайлом телефонные беседы.

В начале 2000 года я вместе с Аланом Найтом и Каем Ю подготовил доклад для конференции Java One, который послужил первым прототипом книги. Я признателен моим соавторам, а также Джошуа Маккензи (Josh Mackenzie), Ребекке Парсонз (Rebecca Parsons) и Дейвиду Райсу за помощь, оказанную ими и тогда и позже. Джим Ньюкирк (Jim Newkirk) сделал все необходимое, чтобы познакомить меня с новым миром Microsoft .NET.

Я многому научился у специалистов, с которыми мне приходилось общаться и сотрудничать. В частности, хотелось бы поблагодарить Коллин Роу (Colleen Roe), Дейвида Мьюрэхеда (David Muirhead) и Рэнди Стаффорда за то, что они поделились со мной результатами работы над проектом системы Foodsmart в Джемстоуне. Не могу не упомянуть добрым словом всех, кто участвовал в плодотворных дискуссиях на конференциях Crested Butte, проводимых Брюсом Эклем (Bruce Eckel) в течение нескольких последних лет. Джошуа Керивски (Joshua Kerievsky) не смог выкроить время для полномасштабного рецензирования, но он прекрасно справился с функциями технического консультанта.

---

[2] Название компании ThoughtWorks можно перевести, скажем, как "работа мысли". — *Прим. пер.*

Я получил заметную помощь со стороны участников одного из читательских кружков, организованных при университете штата Иллинойс. Вот эти люди: Ариель Герценстайн (Ariel Gertzenstein), Боско Живальевич (Bosko Zivaljevic), Брэд Джонс (Brad Jones), Брайан Футей (Brian Foote), Брайан Марик (Brian Marick), Федерико Бэлэгур (Federico Balaguer), Джозеф Йодер (Joseph Yoder), Джон Брант (John Brant), Майк Хьюнер (Mike Hewner), Ральф Джонсон (Ralph Johnson) и Вирасак Витхаваскул (Weerasak Witthawaskul). Всем им большое спасибо.

Драгос Манолеску (Dragos Manolescu), экс-глава кружка, собрал собственную группу рецензентов, в которую вошли Мухаммад Анан (Muhammad Anan), Брайан Дойл (Brian Doyle), Эмад Гоше (Emad Ghosheh), Гленн Грэйссл (Glenn Graessle), Дэниел Хайн (Daniel Hein), Прабхахаран Кумаракуласингам (Prabhaharan Kumarakulasingam), Джо Куинт (Joe Quint), Джон Рейнк (John Reinke), Кевин Рейнолдз (Kevin Reynolds), Шриприя Шринивасан (Sripriya Srinivasan) и Тирумала Ваддираджу (Tirumala Vaddiraju).

Кент Бек (Kent Beck) внес столько предложений, что я затрудняюсь вспомнить их все. Он, например, придумал название для типового решения **частный случай (Special Case, 511)**. Джим Оделл (Jim Odell) был первым, кто познакомил меня с миром консалтинга, преподавания и писательства; мне не хватает слов, чтобы выразить всю глубину моей благодарности.

По мере работы над книгой я размещал ее черновики в Web. Среди тех, кто откликнулся и прислал свои вопросы, варианты возможных проблем и их решения, были Майкл Бэнкс (Michael Banks), Марк Бернстайн (Mark Bernstein), Грэхем Беррисфорд (Graham Berrisford), Бьерн Бесков (Bjorn Beskow), Брайан Борхэм (Brian Boreham), Шен Бродли (Sean Broadley), Перис Бродски (Peris Brodsky), Пол Кемпбелл (Paul Campbell), Честер Чен (Chester Chen), Джон Коукли (John Coakley), Боб Коррик (Bob Corrick), Паскаль Костэнза (Pascal Costanza), Энди Червонка (Andy Czerwonka), Мартин Дайл (Martin Diehl), Дэниел Дрейзин (Daniel Drasin), Хуан Гомес Дуазо (Juan Gomez Duaso), Дон Дуиггинз (Don Dwiggins), Питер Форман (Peter Foreman), Рассел Фриман (Russell Freeman), Питер Гэзмен (Peter Gassmann), Джейсон Горман (Jason Gorman), Дэн Грин (Dan Green), Ларс Грегори (Lars Gregori), Рик Хансен (Rick Hansen), Тобин Харрис (Tobin Harris), Рассел Хили (Russel Healey), Кристиан Геллер (Christian Heller), Ричард Хендерсон (Richard Henderson), Кайл Херменин (Kyle Hermenean), Карстен Хейл (Carsten Heyl), Акира Хирасава (Akira Hirasawa), Эрик Кон (Eric Kaun), Кирк Кнорнсчайлд (Kirk Knoernschild), Джеспер Лейдегаард (Jesper Ladegaard), Крис Лопес (Chris Lopez), Паоло Марино (Paolo Marino), Джереми Миллер (Jeremy Miller), Иван Митрович (Ivan Mitrovic), Томас Нейманн (Thomas Neumann), Джуди Оби (Judy Obee), Паоло Паровел (Paolo Parovel), Тревор Пинкни (Trevor Pinkney), Томас Рестрепо (Tomas Restrepo), Джоуэл Ридер (Joel Rieder), Мэттью Робертс (Matthew Roberts), Стефан Рук (Stefan Roock), Кен Роша (Ken Rosha), Энди Шнайдер (Andy Schneider), Александр Семенов, Стен Силверт (Stan Silvert), Джефф Суттер (Jeoff Soutter), Уолкер Термат (Volker Termath), Кристофер Тейм (Christopher Thames), Уолкер Тюро (Volker Turau), Кнут Ванхеден (Knut Wannheden), Марк Уоллес (Marc Wallace), Стефан Уэниг (Stefan Wenig), Брэд Уаймерслэдж (Brad Wiemerslage), Марк Уиндхолц (Mark Windholtz) и Майкл Юн (Michael Yoon).

Тех, кто здесь не упомянут, я тоже благодарю с не меньшей сердечностью.

Как всегда, моя самая горячая признательность любимой жене Синди (Cindy), чье общество я ценю намного больше, чем кто-либо сумеет оценить эту книгу.

И еще несколько слов

Это моя первая книга, оформленная с помощью языка XML и связанных с ним технологий и инструментов. Текст был набран в виде серии XML-документов с помощью старого доброго TextPad. Я пользовался и доморощенными шаблонами определения типа документа (Document Type Definition — DTD). Для генерации HTML-страниц Web-сайта применялся язык XSLT, а для построения диаграмм — надежный Visio, пополненный замечательными UML-шаблонами от Павла Храби (Pavel Hruby) (намного превосходящими по качеству аналоги из комплекта поставки редактора; на моем сайте приведен адрес, где их можно получить). Я написал небольшую программу, которая позволила автоматически включать примеры кода в выходной файл, избавив меня от кошмара многократного повторения операций копирования и вставки. При подготовке первого варианта рукописи я использовал XSL-FO в сочетании с Apache FOP, а позже для импорта текста в среду FrameMaker создал ряд сценариев XSLT и Ruby.

В процессе работы над книгой мне приходилось применять несколько инструментов из категории программ с открытым исходным кодом — JUnit, NUnit, ant, Xerces, Xalan, Tomcat, Jboss, Ruby и Hsql. Я искренне признателен всем авторам. Не обошлось, разумеется, и без коммерческих продуктов. В частности, я активно пользовался пакетом Visual Studio .NET и прекрасной интегрированной Java-средой разработки Idea от IntelliJ — первой, которая меня по-настоящему вдохновила со времен работы с языком Smalltalk.

Книга была приобретена для издательства Addison Wesley Майком Хендриксоном (Mike Hendrickson) и Россом Венаблзом (Ross Venables), которые руководили проектом. Я начал трудиться над рукописью в ноябре 2000 и закончил работу в июне 2002 года.

Сара Уивер (Sarah Weaver) выполняла обязанности главного редактора и координировала все стороны проекта, связанные с редактированием, верстkой, корректурой, составлением предметного указателя и получением окончательной электронной версии книги. Диана Вуд (Dianne Wood), литературный редактор, предприняла все возможное, чтобы "подчистить" словесную форму, не испортив идейного содержания. Ким Арни Мулки (Kim Arney Mulcahy) окончательно скомпоновал материал, "отшлифовал" рисунки и подготовил для передачи в типографию итоговые файлы FrameMaker. При верстке мы придерживались того же формата, который был выбран ранее для книги [18]. Шерил Фергюсон (Cheryl Ferguson), корректор, позаботилась, чтобы в тексте осталось как можно меньше ошибок, а Ирв Хершман (Irv Hershman) составил предметный указатель.

Поддержка

Эта книга отнюдь не совершенна. Несомненно, в ней есть какие-то неточности; может быть, я упустил что-то важное. Если вы найдете то, что считаете ошибочным, или сочтете, что в книгу следует включить дополнительный материал, пожалуйста, пошлите свое сообщение по адресу: `fowler@acm.org`. Обновления и исправления будут представлены на странице `http://www.martinfowler.com/eaaErrata.html`[3].

О фотографии, размещенной на обложке книги

Когда писалась эта книга, в Бостоне происходило нечто более знаменательное. Я имею в виду строительство моста Bunker Hill Bridge (попробуйте-ка втиснуть это название в небольшой дорожный указатель) через Чарлз-ривер по проекту Леонарда П. Закима (Leonard P. Zakim). Мост Закима, изображенный на обложке книги, относится к разряду подвесных, которые до сих пор не приобрели в США такого распространения, как, скажем, в Европе. Мост не особенно длинен, но зато самый широкий в мире (в своей категории) и первый в Штатах, имеющий асимметричный дизайн. Кроме того (может быть, это самое главное), он очень красив.

Мартин Фаулер,
Мелроуз, Массачусетс,
август 2002 года
`http://martinfowler.com`

[3] В переводном издании этой книги учтены исправления, опубликованные на этой Web-странице, по состоянию на 21 декабря 2003 года. — *Прим. ред.*

Ждем ваших отзывов!

Именно вас, уважаемые читатели, мы считаем главными критиками и комментаторами этой книги. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш Web-сервер и оставить там свои замечания. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши координаты:

E-mail:	info@williamspublishing.com
WWW:	http://www.williamspublishing.com

Информация для писем:

из России:	127055, Москва, ул. Лесная, д. 43, стр. 1
из Украины:	03150, Киев, а/я 152

Введение

Создание компьютерных систем (может быть, вы этого еще не знаете) — весьма непростое дело. По мере увеличения их сложности трудоемкость процессов конструирования соответствующего программного обеспечения возрастает согласно экспоненциальному закону. Как и в любой профессии, прогресс в программировании достигается только путем обучения, причем как на ошибках, так и на удачах — своих и чужих. Это издание представляет собой учебное пособие, которое поможет вам усвоить информацию и передать полученные знания другим значительно быстрее и эффективнее, чем это удавалось мне самому.

Ниже обозначен контекст книги и освещаются базовые понятия и идеи, на которых зиждется дальнейшее изложение материала.

Архитектура

Одно из странных свойств, присущих индустрии программного обеспечения, связано с тем, что какой-либо термин может иметь множество противоречивых толкований. Вероятно, наиболее многострадальным оказалось понятие *архитектура (architecture)*. Мне кажется, оно принадлежит к числу тех нарочито эффектных слов, которые употребляются преимущественно для обозначения чего-то, считающегося значительным и серьезным. (Нет, я слишком прагматичен, чтобы цинично пользоваться этим фактом, стараясь привлечь внимание читающей публики. :–))

Термин "архитектура" пытаются трактовать все, кому не лень, и всяк на свой лад. Впрочем, можно назвать два общих варианта. Первый связан с разделением системы на наиболее крупные составные части; во втором случае имеются в виду некие конструктивные решения, которые после их принятия с трудом поддаются изменению. Также растет понимание того, что существует более одного способа описания архитектуры и степень важности каждого из них меняется в продолжение жизненного цикла системы.

Время от времени Ралф Джонсон (Ralph Johnson), участвуя в списке рассылки, отправлял туда замечательные сообщения, одно из которых, касающееся проблем архитектуры, совпало с периодом завершения мною чернового варианта книги. В этом сообщении он подтвердил мое мнение о том, что архитектура — весьма субъективное понятие. В лучшем случае оно отображает общую точку зрения команды разработчиков на результаты проектирования системы. Обычно это согласие в вопросе идентификации главных компонентов системы и способов их взаимодействия, а также выбор таких решений,

которые интерпретируются как основополагающие и не подлежащие изменению в будущем. Если позже оказывается, что нечто изменить легче, чем казалось вначале, это "нечто" легко исключается из "архитектурной" категории.

В этой книге представлено мое видение основных компонентов корпоративного приложения и решений, которые должны приниматься на ранних фазах его жизни. Мне по душе трактовка системы в виде набора архитектурных *слоев (layers)* (подробнее о слоях — в главе 1, "«Расслоение» системы"). Поэтому в книге предлагаются ответы на вопросы, как осуществить декомпозицию системы по слоям и как обеспечить надлежащее взаимодействие слоев между собой. В большинстве корпоративных приложений прослеживается та или иная форма архитектурного "расслоения", но в некоторых ситуациях большее значение могут приобретать другие подходы, связанные, например, с организацией *каналов (pipes)* или *фильтров (filters)*. Однако мы сконцетрируем внимание на архитектуре слоев как на наиболее плодотворной структурной модели.

Одни типовые решения, рассмотренные ниже, можно определенно считать "архитектурными" в том смысле, что они представляют "значимые" составные части приложения и/или "основополагающие" аспекты функционирования этих частей. Другие относятся к вопросам реализации. Но я не собираюсь классифицировать, что в большей, а что в меньшей степени "архитектурно", чтобы не навязывать вам свое мнение.

Корпоративные приложения

Созданием программного обеспечения занимается множество людей. Но говорить о программном обеспечении в целом — значит, не сказать ничего, поскольку разновидностей программных приложений чрезвычайно много и каждая ситуация выдвигает особые проблемы, отличающиеся собственным уровнем сложности. Наглядным примером могут служить дискуссии с коллегами, работающими в сфере телекоммуникаций. С определенной точки зрения "мои" корпоративные приложения намного проще, нежели телекоммуникационное программное обеспечение, — мне не приходится решать чрезвычайно сложные проблемы обеспечения многопоточного функционирования и тесной интеграции аппаратных и программных компонентов. Но во всем остальном мои задачи существенно сложнее. Корпоративные приложения чаще всего имеют дело с изощренными данными большого объема и бизнес-правилами, логика которых иногда просто противоречит здравому смыслу. Хотя некоторые приемы и решения более-менее универсальны, большинство из них адекватны только в контексте определенных ситуаций.

На протяжении всей профессиональной карьеры я занимался преимущественно корпоративными приложениями, и это, разумеется, обусловило выбор типовых решений, представленных на страницах книги. (В числе близких аналогов понятия "приложение" можно назвать термин "информационная система"; читатели постарше, вероятно, вспомнят выражение "обработка данных".) Но что именно подразумевает понятие *корпоративное приложение (enterprise application)*? Точное определение сформулировать трудно, но дать смысловое толкование вполне возможно.

Начнем с примеров. К числу корпоративных приложений относятся, скажем, бухгалтерский учет, ведение медицинских карт пациентов, экономическое прогнозирование, анализ кредитной истории клиентов банка, страхование, внешнеэкономические торговые

операции и т.п. Корпоративными приложениями *не* являются средства обработки текста, регулирования расхода топлива в автомобильном двигателе, управления лифтами и оборудованием телефонных станций, автоматического контроля химических процессов, а также операционные системы, компиляторы, игры и т.д.

Корпоративные приложения обычно подразумевают необходимость долговременного (иногда в течение десятилетий) *хранения данных*. Данные зачастую способны "пережить" несколько поколений прикладных программ, предназначенных для их обработки, аппаратных средств, операционных систем и компиляторов. В продолжение этого срока структура данных может подвергаться многочисленным изменениям в целях сохранения новых порций информации без какого-либо воздействия на старые. Даже в тех случаях, когда компания осуществляет революционные изменения в парке оборудования и номенклатуре программных приложений, данные не уничтожаются, а переносятся в новую среду.

Данных, с которыми имеет дело корпоративное приложение, как правило, бывает *много*: даже скромная система способна манипулировать несколькими гигабайтами информации, организованной в виде десятков миллионов записей; и задача манипуляции этими данными вырастает в одну из основных функций приложения. В старых системах информация хранилась в виде индексированных файловых структур, подобных разработанным компанией IBM VSAM и ISAM. Сейчас для этого применяются системы управления базами данных (СУБД), большей частью реляционные. Проектирование таких систем и их сопровождение превратились в отдельные специализированные дисциплины.

Множество пользователей обращаются к данным *параллельно*. Как правило, их количество не превышает сотни, но для систем, размещенных в среде Web, этот показатель возрастает на несколько порядков. Как гарантировать возможность одновременного доступа к базе данных для всех, кто имеет на это право? Проблема остается даже в том случае, если речь идет всего о двух пользователях: они должны манипулировать одним элементом данных только такими способами, которые исключают вероятность возникновения ошибок. Большинство обязанностей по управлению параллельными заданиями принимает на себя диспетчер транзакций из состава СУБД, но полностью избавить прикладные системы от подобных забот программистам чаще всего не удается.

Если объемы данных столь велики, в приложении должно быть предусмотрено и множество различных вариантов *окон экранного интерфейса*. Вполне обычна ситуация, когда программа содержит несколько сотен окон. Одни пользователи работают с корпоративным приложением регулярно, другие обращаются к нему эпизодически, но полагаться на их техническую осведомленность в любом случае нельзя. Поэтому данные должны допускать возможность представления в самых разных формах, удобных для пользователей всех категорий. Фокусируя внимание на вопросах взаимодействия пользователей с приложением, нельзя упускать из виду и тот факт, что многие системы характеризуются высокой степенью пакетной обработки данных.

Корпоративные приложения редко существуют в изоляции. Обычно они требуют *интеграции* с другими системами, построенными в разное время и с применением различных технологий (файлы данных COBOL, CORBA, системы обмена сообщениями). Время от времени корпорации стараются провести интеграцию собственных подсистем с применением некоторой универсальной технологии. Поскольку обычно такой работе конца-краю не видно, все сводится к применению нескольких различных методов интеграции.

Ситуация усугубляется, если интеграции подлежит информация из совершенно разнородных источников.

Даже в том случае, если компания унифицирует способ интеграции своих подсистем, проблемы на этом не заканчиваются: новым камнем преткновения выступает различие в ведении бизнес-процессов и *концептуальный диссонанс* в представлении данных. В одном подразделении компании под клиентом подразумевают лицо, с которым заключено действующее соглашение; в другом к клиентам относят и тех, с кем приходилось работать прежде; в третьем учитывают только товарные сделки, но не оказанные услуги и т.п. На первый взгляд все это не кажется слишком серьезным, но если каждое из полей в сотнях типов записей обладает какой-то особой семантикой, легкие облака над вашей головой в один момент могут превратиться в грозовые тучи (достаточно представить, что тот единственный на всю компанию человек, который знал все нюансы, вдруг уволился). (Для того чтобы увидеть полную картину происходящего, следует учесть также тот факт, что данные подвержены постоянным изменениям, вносимым, как правило, без какого-либо предупреждения.)

Не стоит забывать и о том, *что* принято обозначать расплывчатым термином *бизнес-логика*. Я нахожу его забавным, поскольку могу припомнить только несколько вещей, менее логичных, нежели так называемая бизнес-логика. При создании операционной системы, например, без строгой логики не обойтись. Но бизнес-правила, которые нам сообщают, следует принимать такими, какие они есть, поскольку без серьезного политического вмешательства их не преодолеть. Мы вынуждены иметь дело со случайными наборами странных условий, которые сочетаются между собой самым непредсказуемым образом. Определенно известно только одно: вся эта мешанина еще и изменяется во времени. Разумеется, тому есть какие-то причины. Так, например, выгодный клиент может выговорить для себя особые условия оплаты кредита, отвечающие срокам поступления средств на его расчетный счет. И пару тысяч таких вот частных случаев способны сделать бизнес-логику совершенно *нелогичной*, а соответствующее программное приложение — запутанным и не поддающимся восприятию с позиций здравого смысла.

Некоторые интерпретируют понятие "корпоративное приложение" как синоним термина "большая система". Однако важно понимать, что не все подобные приложения на самом деле "велики", хотя их вклад в деятельность "корпорации" может быть довольно весомым. Многие считают, что "малые" системы причиняют меньше беспокойства, и до определенной степени это справедливо. Крах небольшого приложения обычно менее ощутим, нежели сбой крупной системы. Впрочем, я думаю, что пренебрегать совокупным влиянием многих мелких систем непозволительно. Если вы в состоянии предпринять какие-либо действия, которые помогут улучшить функционирование небольших приложений, общий результат может оказаться весьма существенным, так как нередко значимость системы далеко превосходит ее "размеры". И еще. При любой возможности старайтесь превратить "большой" проект в "малый" за счет упрощения принимаемой архитектуры и ограничения множества реализуемых функций.

Типы корпоративных приложений

При обсуждении способов проектирования корпоративных приложений и вариантов выбора типовых решений важно понимать, что двух одинаковых приложений просто не существует и различия в проблемах обусловливают необходимость применения различных средств достижения поставленных целей. Меня захлестывает волна протеста, когда я слышу советы делать что-либо "только так и не иначе". По моему убеждению, наиболее любопытная и дерзкая часть проекта, как правило, связана с анализом пространства альтернатив и взвешенным выбором в пользу одной из них. Рассмотрим для примера три довольно типичных варианта корпоративных приложений.

Первый — электронная коммерческая система типа "поставщик–потребитель" ("business to customer" — B2C); к ней обращаются с помощью Web-обозревателей, находят нужные товары, вводят необходимую информацию и осуществляют покупки. Подобная система предназначена для обслуживания большого количества пользователей одновременно, поэтому проектное решение должно быть не только эффективным по критерию использования ресурсов, но и масштабируемым: все, что требуется для повышения пропускной способности такой системы, — это приобретение дополнительного аппаратного обеспечения. Бизнес-логика подобного приложения довольно прямолинейна: прием заказа, незамысловатые финансовые операции и отправка уведомления о доставке. Необходимо, чтобы каждый мог быстро и легко обратиться к системе, поэтому Web-интерфейс должен быть предельно простым и доступным для воспроизведения с помощью максимально широкого диапазона обозревателей. Информационный источник включает базу данных для хранения заказов и, возможно, некий механизм обмена данными с системой складского учета для получения информации о наличии товаров и их отгрузке.

Сопоставьте рассмотренную систему с программой, автоматизирующей учет соглашений имущественного найма. В некоторых аспектах последняя намного проще, нежели приложение электронной коммерции B2C, поскольку круг ее пользователей, работающих одновременно, существенно *уже* — скажем, не более сотни. В чем она сложнее, так это в бизнес-логике. Составление ежемесячных счетов за аренду, обработка различных событий (таких, как преждевременный возврат имущества и просроченный платеж), проверка вводимой информации при заключении договора — все это достаточно трудоемкие функции. В индустрии лизинга успех во многом определяется выбором одного из множества вариантов, незначительно отличающихся от классических примеров сделок, которые заключались в прошлом. И бизнес-логика этой предметной области весьма сложна, поскольку правила игры слишком свободны.

Подобная система отличается сложностью и в отношении пользовательского интерфейса. Зачастую требования к интерфейсу таковы, что их нельзя удовлетворить только средствами HTML; приходится пользоваться интерфейсными средствами, предоставляемыми более традиционной моделью "толстого" клиента. Усложнение процедур взаимодействия пользователя с программой вынуждает применять и более изощренные варианты транзакций: например, оформление договора аренды может продолжаться несколько часов, и все это время пользователь выполняет одну логическую транзакцию. Схема базы данных также заметно расширяется и может включать несколько сотен таблиц и соединений с внешними пакетами, предназначенными для оценки стоимости активов и цены аренды.

В качестве третьего примера рассмотрим простейшую систему учета расходов для небольшой компании, обладающую самой незатейливой логикой. Пользователи системы (их всего несколько) могут обращаться к ней с помощью стандартизованного Web-интерфейса. Источником информации является база данных с двумя-тремя таблицами. Однако даже к такой тривиальной задаче нельзя относиться легкомысленно. Вероятно, систему потребуется сконструировать очень быстро, в то же время принимая во внимание возможности ее роста за счет будущего включения модулей финансового анализа и налогообложения, генерации отчетной документации, взаимодействия с другими приложениями и т.п. Попытки применения тех же архитектур, которые приемлемы в двух предыдущих случаях, в данной ситуации чреваты существенным замедлением темпов работ. Если программа обеспечивает получение тех или иных преимуществ (а такими должны быть все корпоративные приложения), задержка с ее внедрением в эксплуатацию означает прямые финансовые потери. Поэтому отнюдь не хотелось бы принимать решения, которые воспрепятствуют развитию системы в дальнейшем. Однако, если оснастить систему дополнительными службами с прицелом на будущее, но сделать это неправильно, новый уровень сложности как раз и может затруднить ее эволюцию, приостановить процесс внедрения и отсрочить получение преимуществ. Несмотря на "незначительность" каждого отдельного приложения такого класса, все они, рассматриваемые в совокупности, способны оказать серьезное положительное (или отрицательное) влияние на показатели деятельности "корпорации".

Реализация каждой из трех упомянутых систем сопряжена с трудностями, но трудности эти в каждом случае специфичны. Как следствие, нам не удастся предложить единую архитектуру, приемлемую во всех ситуациях. Выбирая архитектуру и варианты проектирования, следует принимать во внимание особенности конкретной системы. Вот почему на страницах книги вы не найдете универсальных рецептов, пригодных на все случаи жизни. Напротив, наличие многих типовых решений не избавляет вас от необходимости выбора из нескольких альтернатив. Даже если вы отдадите предпочтение какому-либо решению, вам, возможно, придется модифицировать его, чтобы приспособить к реальным условиям. Нельзя работать над проектом корпоративного приложения бездумно. Все, на что способна какая бы то ни было книга, — это дать некую исходную информацию, оттолкнувшись от которой вы можете начать мыслить самостоятельно.

Все сказанное применимо и к выбору инструментальных средств программирования. Приступая к работе, следует — думаю, это очевидно — максимально сузить круг инструментов. Однако не стоит забывать, что разные средства проявляют свои лучшие качества в разных ситуациях. Избегайте использования инструментов, предназначенных для других целей, иначе они не только не принесут пользы, но и причинят вред.

Производительность

Многие архитектурные решения напрямую связаны с аспектами *производительности (performance)* системы. Чтобы говорить о производительности, я предпочитаю увидеть работающую систему, измерить ее характеристики и, учитывая полученные результаты, применить строго определенные процедуры оптимизации. Однако некоторые архитектурные решения определяют параметры производительности таким образом, что

устранение возможных проблем средствами оптимизации затрудняется. Именно поэтому к принятию таких решений всегда стоит подходить с большой ответственностью.

В книгах, подобных этой, трудно толковать вопросы производительности, поскольку любой совет не может приниматься как незыблемый факт до тех пор, пока не будет испытан в конкретных условиях. Я слишком часто сталкивался с ситуациями, когда вариант проектирования принимался или отвергался по причинам, связанным с производительностью, а затем замеры параметров осуществлялись в реальной среде и полностью опровергали прежние выводы.

На страницах книги я приведу всего несколько рекомендаций по повышению производительности приложений, включая и такую, которая касается минимизации количества удаленных вызовов. Однако какими бы неоспоримыми все они ни казались, вы обязаны всегда проверять их на практике. Помимо того, в некоторых примерах кода я пожертвовал соображениями эффективности выполнения в угоду удобочитаемости. Выполняя оптимизацию, не забывайте проверять параметры производительности до и после, иначе вы добьетесь только того, что код станет более трудным для восприятия.

Существует одно важное обстоятельство: значительное изменение конфигурации приложения способно перечеркнуть все ранее установленные факты, касающиеся вопросов производительности. Поэтому, обновив версию виртуальной машины, СУБД, аппаратного обеспечения или любого другого компонента системы, вы обязаны заново провести оптимизацию и убедиться, что принятые меры возымели действие. Нередко бывает и так, что приемы оптимизации, с успехом применявшиеся в прошлом, в новых условиях вызывают обратный эффект.

Еще одна проблема любой дискуссии по вопросам производительности состоит в том, что многие термины употребляются и интерпретируются непоследовательно и неверно. Наиболее характерный пример — понятие, описывающее *способность приложения к масштабированию (scalability)*: тут можно назвать не менее пяти-шести различных толкований. Ниже рассмотрены термины, применяемые в этой книге.

Время отклика (response time) — промежуток времени, который требуется системе, чтобы обработать запрос извне, подобный щелчку на кнопке графического интерфейса или вызову функции API сервера.

Быстрота реагирования (responsiveness) — скорость подтверждения запроса (не путать с временем отклика — скоростью обработки). Эта характеристика во многих случаях весьма важна, поскольку интерактивная система, пусть даже обладающая нормальным временем отклика, но не отличающаяся высокой быстротой реагирования, всегда вызывает справедливые нарекания пользователей. Если, прежде чем принять очередной запрос, система должна полностью завершить обработку текущего, параметры времени отклика и быстроты реагирования, по сути, совпадают. Если же система способна подтвердить получение запроса раньше, ее быстрота реагирования выше. Например, применение динамического индикатора состояния процесса копирования повышает быстроту реагирования экранного интерфейса, хотя никак не сказывается на значении времени отклика.

Время задержки (latency) — минимальный интервал времени до получения какого-либо отклика (даже если от системы более ничего не требуется). Параметр приобретает особую важность в распределенных системах. Если я "прикажу" программе ничего не делать и сообщить о том, когда именно она закончит это "ничегонеделание", на персональном компьютере ответ будет получен практически мгновенно. Если же программа

выполняется на удаленной машине, придется подождать, вероятно, не менее нескольких секунд, пока запрос и ответ проследуют по цепочке сетевых соединений. Снизить время задержки разработчику прикладной программы не под силу. Фактор задержки — главная причина, побуждающая минимизировать количество удаленных вызовов.

Пропускная способность (throughput) — количество данных (операций), передаваемых (выполняемых) в единицу времени. Если, например, тестируется процедура копирования файла, пропускная способность может измеряться числом байтов в секунду. В корпоративных приложениях обычной мерой производительности служит число транзакций в секунду (transactions per second — tps), но есть одна проблема — транзакции различаются по степени сложности. Для конкретной системы необходимо рассматривать смесь "типовых" транзакций.

В контексте рассмотренных терминов под *производительностью* можно понимать один из двух параметров — *время отклика* или *пропускную способность*, в частности тот, который в большей степени отвечает природе ситуации. Иногда бывает трудно судить о производительности, если, например, использование некоторого решения повышает пропускную способность, одновременно увеличивая время отклика. С точки зрения пользователя, значение быстроты реагирования может оказаться более важным, нежели время отклика, так что улучшение быстроты реагирования ценой потери пропускной способности или возрастания времени отклика вполне способно повысить производительность.

Загрузка (load) — значение, определяющее степень "давления" на систему и измеряемое, скажем, количеством одновременно подключенных пользователей. Параметр загрузки обычно служит контекстом для представления других функциональных характеристик, подобных времени отклика. Так, нередко можно слышать выражения наподобие следующего: "время отклика на запрос составляет 0,5 секунды для 10 пользователей и 2 секунды для 20 пользователей".

Чувствительность к загрузке (load sensitivity) — выражение, задающее зависимость времени отклика от загрузки. Предположим, что система А обладает временем отклика, равным 0,5 секунды для 10–20 пользователей, а система В — временем отклика в 0,2 секунды для 10 пользователей и 2 секунды для 20 пользователей. Это дает основание утверждать, что система А обладает меньшей чувствительностью к загрузке, нежели система В. Можно воспользоваться и термином *ухудшение (degradation)*, чтобы подчеркнуть факт меньшей устойчивости параметров системы В.

Эффективность (efficiency) — удельная производительность в пересчете на одну единицу ресурса. Например, система с двумя процессорами, способная выполнить 30 tps, более эффективна по сравнению с системой, оснащенной четырьмя аналогичными процессорами и обладающей продуктивностью в 40 tps.

Мощность (capacity) — наибольшее значение пропускной способности или загрузки. Это может быть как абсолютный максимум, так и некоторое число, при котором величина производительности все еще превосходит заданный приемлемый порог.

Способность к масштабированию (scalability) — свойство, характеризующее поведение системы при добавлении ресурсов (обычно аппаратных). Масштабируемой принято считать систему, производительность которой возрастает пропорционально объему приобщенных ресурсов (скажем, вдвое при удвоении количества серверов). *Вертикальное масштабирование (vertical scalability, scaling up)* — это увеличение мощности отдельного сервера (например, за счет увеличения объема оперативной памяти). *Горизонтальное*

масштабирование (horizontal scalability, scaling out) — это наращивание потенциала системы путем добавления новых серверов.

Следует отметить, что существует проблема: выбор проектного решения оказывает не одинаковое влияние на факторы производительности. Рассмотрим для примера две программные системы, работающие на однотипных серверах: первая (назовем ее Swordfish) обладает мощностью в 20 tps, а вторая, Camel, — в 40 tps. Какая система более производительна? Какая из них в большей мере способна к масштабированию? На основании имеющейся информации мы не можем ответить на вопрос, касающийся возможностей масштабирования. Единственное, что понятно, — Camel более эффективна в конфигурации с одним сервером. Добавим еще один сервер и допустим, что производительность Swordfish теперь исчисляется величиной 35 tps, а продуктивность Camel возрастает до 50 tps. Мощность Camel все еще выше, но параметры масштабирования Swordfish выглядят предпочтительнее. Продолжив пополнение систем серверами, мы обнаружим, что производительность Swordfish всякий раз возрастает на 15 tps, в то время как аналогичный показатель Camel — только на 10 tps. Полученные данные позволяют прийти к заключению, что Swordfish лучше поддается горизонтальному масштабированию, хотя Camel демонстрирует большую эффективность при количестве серверов, меньшем пяти.

Проектируя корпоративную систему, часто следует уделять больше внимания обеспечению средств масштабирования, а не наращиванию мощности или повышению эффективности. Производительность масштабируемой системы всегда удастся увеличить, если такая потребность действительно возникнет. Помимо того, добиться возможности масштабирования проще. Нередко проектировщикам и программистам приходится изворачиваться, чтобы увеличить мощность системы для конкретной аппаратной конфигурации, хотя, вероятно, было бы дешевле приобрести дополнительное оборудование. Если, скажем, система Camel стоит дороже, чем Swordfish, и разница в стоимости покрывает цену нескольких серверов, то Swordfish обойдется дешевле даже в том случае, когда необходимый уровень производительности составляет всего 40 tps. Сейчас модно сетовать на возрастающие требования к аппаратному обеспечению, непременно сопровождающие выпуск очередных версий популярных программ, и я присоединяю свой голос к хору возмущенных потребителей, когда при установке последней версии Word мне велят обновить конфигурацию компьютера. Но использовать более новое оборудование зачастую просто выгоднее, нежели заставлять программу "крутиться" на устаревшей технике. Если корпоративное приложение поддается масштабированию, добавить несколько серверов дешевле, чем приобрести услуги нескольких программистов.

Типовые решения

Концепция *типовых решений (patterns)* известна уже давно, и мне не хотелось бы выступать здесь с очередным очерком истории ее развития. Однако я не собираюсь упускать удачную возможность рассказать о собственной точке зрения на предмет.

Какого бы то ни было общеупотребительного определения типового решения не существует. Вероятно, лучше всего начать с цитаты из книги Кристофера Александера (Christopher Alexander), вдохновившей не одно поколение энтузиастов: "Каждое типовое решение описывает некую повторяющуюся проблему и ключ к ее разгадке, причем таким

образом, что вы можете пользоваться этим ключом многократно, ни разу не придя к одному и тому же результату" [1]. К. Александер — архитектор, и говорит он о сооружениях, но данное им определение, по моему мнению, прекрасно подходит и для типовых решений в программировании. Основа типового решения — подход, достаточно общий и эффективный для того, чтобы обеспечить преодоление периодически возникающих проблем определенной природы. Типовое решение можно воспринимать и в виде некоторой рекомендации: искусство создания типовых решений состоит в вычленении и кристаллизации таких относительно независимых рекомендаций, которые допускают возможность более или менее обособленного применения.

Типовые решения уходят своими корнями в практику. Чтобы распознать их, необходимо присмотреться к тому, что и как делают другие, изучить полученные результаты и выявить их главную составляющую. Это нелегко, но коль скоро хорошие типовые решения найдены, они обретают большую ценность. Для меня эта ценность в данный момент проявляется в том, что дает основание написать книгу, которую будут использовать как руководство к действию. Вам не обязательно читать ее (или любое другое аналогичное издание) от корки до корки, чтобы убедиться в полезности предлагаемых рекомендаций. Достаточно краткого знакомства, которое позволит почувствовать смысл проблем и соответствующих решений. От вас не требуется досконально знать все детали, кроме тех, которые помогут подыскать в книге подходящее типовое решение. И лишь потом вам, возможно, понадобится изучить его более подробно.

Как только вы ощутили потребность в типовом решении, следует подумать, как применить его в конкретных обстоятельствах. Основная особенность типовых решений состоит в том, что ими нельзя пользоваться слепо; вот почему многие инструментальные оболочки, основанные на типовых решениях, и получаемые с их помощью результаты так убоги. Мне кажется, что типовые решения можно сравнить с полуфабрикатом: чтобы получить удовольствие от еды, вам придется довершить приготовление блюда по собственному рецепту. Используя типовое решение, я "поджариваю" его всякий раз по-другому и наслаждаюсь отменным и своеобразным вкусом.

Каждое типовое решение относительно независимо от других, но говорить об их полной взаимной изолированности нельзя. Зачастую одно решение непосредственно приводит к другому либо может применяться только в контексте некоего базового решения. Так, например, решение **наследование с таблицами для каждого класса (Class Table Inheritance, 305)** обычно используется совместно с решением **модель предметной области (Domain Model, 140)**. Границы между решениями не отличаются четкостью, но я предпринял все возможное для того, чтобы представить каждое решение в самодостаточной форме. Поэтому, если кто-то предлагает "применить решение **единица работы (Unit of Work, 205)**", будет довольно проследовать к соответствующему разделу книги, не читая всего остального.

Если у вас есть опыт проектирования корпоративных приложений, многие типовые решения, о которых речь пойдет ниже, вам, вероятно, уже знакомы. Надеюсь, вы не будете разочарованы (ваше возможное недоумение я уже пытался предупредить в предисловии). Типовые решения — не научные открытия; они в большой мере представляют собой обобщения результатов, накопленных в соответствующей области. Поэтому более уместно говорить о "поиске" типового решения, а не о его "изобретении". Моя роль состоит в обнаружении общих подходов, выделении главного и оформлении полученных выводов. Ценность типового решения для проектировщика заключается не в новизне

идеи; главное — это помощь в описании и сообщении этой идеи. Если собеседники осведомлены о том, *что* именно представляет собой **интерфейс удаленного доступа (Remote Facade, 405)**, одному из них достаточно произнести фразу "этот класс является **интерфейсом удаленного доступа**", чтобы расставить все точки над *i*. А новичку вы сможете подсказать, что, дескать, "для этой задачи больше подойдет **объект переноса данных (Data Transfer Object, 419)**", и он, обратившись к странице 646 настоящей книги, найдет все необходимое. Классификация типовых решений способствует созданию словаря проектировщика, а потому выбору их названий следует уделять должное внимание.

В главе 18, "Базовые типовые решения", упомянут ряд базовых типовых решений универсального характера; они используются при описании всех остальных решений, ориентированных на корпоративные приложения.

Структура типовых решений

Каждому автору приходится выбирать форму представления типовых решений. Некоторые принимают за основу подходы, которые изложены в классических книгах, посвященных типовым решениям (например, [1, 20, 34]). Другие изобретают что-то свое. Я долго не мог составить твердое мнение о том, в чем же состоит превосходство того или иного варианта над остальными. С одной стороны, меня не могло удовлетворить нечто столь же краткое, как форма из [20]; с другой — описание каждого решения должно было бы уместиться в пределах небольшого раздела книги. Ниже описан компромиссный результат, к которому я пришел после длительных размышлений.

Первое — название решения. Выбор подходящего имени довольно важен, поскольку одна из целей, преследуемых при использовании типовых решений, состоит в создании словаря, который облегчает взаимодействие проектировщиков в ходе выполнения проекта. Например, если я сообщу, что мой Web-сервер реализован на основе **контроллера запросов (Front Controller, 362)** и **представления с преобразованием (Transform View, 379)**, и эти типовые решения будут знакомы и вам, у вас наверняка сложится однозначное представление об архитектуре приложения.

За именем следуют аннотация и эскиз. Аннотация состоит из одного-двух предложений и содержит краткое описание типового решения; эскиз служит визуальной иллюстрацией решения и часто (хотя не всегда) представляет собой UML-диаграмму. Аннотация и эскиз предназначены для создания наглядного "образа" решения и облегчают его запоминание. Если у вас уже есть решение (в том смысле, что оно вам известно, хотя, возможно, вы не знаете его названия), аннотация и эскиз — это, пожалуй, все, с помощью чего можно его быстро отыскать.

Далее вкратце обозначается проблема, которая стимулировала появление типового решения. Если таких проблем несколько, выбирается более характерная. Затем приводятся два обязательных раздела.

В разделе *Принцип действия* освещаются особенности реализации решения. Изложение ведется в стиле, в наименьшей мере зависящем от специфики какой бы то ни было конкретной платформы. Там, где подобная информация все-таки целесообразна, соответствующий текст выделен с помощью отступов, сразу привлекающих внимание читателя. Чтобы упростить восприятие материала, в некоторых случаях здесь приводятся дополнительные UML-диаграммы.

Раздел *Назначение* содержит сведения о том, при каких обстоятельствах следует применять типовое решение, а также о его сравнительных преимуществах и недостатках.

Многие из решений, упомянутых в этой книге, взаимозаменяемы: таковыми являются, скажем, **контроллер страниц (Page Controller, 350)** и **контроллер запросов (Front Controller, 362)**. Часто одинаково пригодными оказываются несколько решений, поэтому, обнаружив одно из них, я спрашиваю себя, в каких случаях им не стоило бы пользоваться. Ответ на этот вопрос нередко подводит меня к выбору альтернативного решения.

В разделе *Дополнительные источники информации* приводятся ссылки на отдельные библиографические источники, имеющие отношение к обсуждаемой теме. Внимание ограничивается тем материалом, который, как мне кажется, способен оказать наибольшую помощь в освоении особенностей типового решения. Из рассмотрения исключены источники, не содержащие ничего нового, работы, с которыми я не знаком лично, публикации в редких или экзотических изданиях, а также нестабильные и сомнительные гиперссылки.

В конце раздела, посвященного тому или иному решению, по мере необходимости могут быть приведены *простые примеры* использования, проиллюстрированные фрагментами кода на языках Java и/или C#. Выбор языков программирования (это уже отмечалось в предисловии) обусловлен вероятными предпочтениями большинства читателей. Очень важно понимать, что любой пример *не есть* типовое решение. При реализации решения оно *не* будет выглядеть точно так же, как пример, поэтому примеры не следует трактовать как некую разновидность макросов. Я намеренно пытался упрощать их настолько, насколько это было возможно, так что соответствующие типовые решения предстанут перед вами в такой прозрачной форме, какую я только мог вообразить. Стремясь к простоте, я силился продемонстрировать в примерах сущность типовых решений и старался избегать деталей, которые могли бы отвлечь внимание от главного. Все частные проблемы опущены, но имейте в виду, что они могут быстро всплыть в условиях конкретной конфигурации системы. Вот почему типовые решения всегда следует испытывать на практике.

Вместо рассмотрения одного или нескольких сквозных примеров я решил прибегнуть к простым независимым примерам. Последние легче воспринимать, хотя не всегда понятно, как использовать совместно. Сквозной пример демонстрирует проблемы в их связи, но не удобен для быстрого знакомства с природой конкретного решения. Я не исключаю возможности компромисса, но мне найти его не удалось.

Код примеров написан так, чтобы обеспечить простоту восприятия. В результате некоторые вещи остались, что называется, "за кадром" (в частности, так произошло с обработкой ошибок — вопросом, недостаточно хорошо проработанным мною для того, чтобы предложить здесь какое-либо типовое решение). Каждый пример обставлен слишком большим количеством упрощающих условий, и поэтому я посчитал нецелесообразным обеспечить возможность загрузки образцов кода с моего Web-сайта.

Ограничения предлагаемых в книге типовых решений

Как упоминалось в предисловии, предлагаемую коллекцию типовых решений *нельзя* трактовать как исчерпывающее руководство по разработке корпоративных программных приложений. Я рассматривал бы эту книгу не с позиций полноты, а с точки зрения полезности. Предмет слишком серьезен для одной головы, тем более для одной публикации.

Представленные здесь сведения касаются всех типовых решений в области корпоративных приложений, известных мне на данный момент. Я не собираюсь утверждать, что полностью осведомлен обо всех их разновидностях и взаимозависимостях. Книга

отражает уровень моего сегодняшнего опыта с учетом тех знаний, которые были приобретены в процессе ее написания. Но мысль не стоит на месте, и опыт, разумеется, будет пополняться и после выхода книги в свет. Программирование — бесконечный процесс, и в нем нет предела совершенству.

Собираясь воспользоваться типовыми решениями, не забывайте, что они только отправная точка, а не пункт назначения. Ни один автор не в состоянии предвидеть всего многообразия форм и вариантов программных проектов. Я написал эту книгу, чтобы помочь вам сдвинуться с места, так что извлекайте уроки. Помните, что любое типовое решение — это начало трудного пути, и только вам решать, стоит ли брать на себя ответственность (и не лишать себя удовольствия) пройти его до конца.

ЧАСТЬ I

Обзор

Глава 1

"Расслоение" системы

Концепция *слоев (layers)* — одна из общеупотребительных моделей, используемых разработчиками программного обеспечения для разделения сложных систем на более простые части. В архитектурах компьютерных систем, например, различают слои кода на языке программирования, функций операционной системы, драйверов устройств, наборов инструкций центрального процессора и внутренней логики чипов. В среде сетевого взаимодействия протокол FTP работает на основе протокола TCP, который, в свою очередь, функционирует "поверх" протокола IP, расположенного "над" протоколом Ethernet.

Описывая систему в терминах архитектурных слоев, удобно воспринимать составляющие ее подсистемы в виде "слоеного пирога". Слой более высокого уровня пользуется службами, предоставляемыми нижележащим слоем, но тот не "осведомлен" о наличии соседнего верхнего слоя. Более того, обычно каждый промежуточный слой "скрывает" нижний слой от верхнего: например, слой 4 пользуется услугами слоя 3, который обращается к слою 2, но слой 4 не знает о существовании слоя 2. (Не в каждой архитектуре слои настолько "непроницаемы", но в большинстве случаев дело обстоит именно так.)

Расчленение системы на слои предоставляет целый ряд преимуществ.

- Отдельный слой можно воспринимать как единое самодостаточное целое, не особенно заботясь о наличии других слоев (скажем, для создание службы FTP необходимо знать протокол TCP, но не тонкости Ethernet).
- Можно выбирать альтернативную реализацию базовых слоев (приложения FTP способны работать без каких-либо изменений в среде Ethernet, по соединению PPP или в любой другой среде передачи информации).
- Зависимость между слоями можно свести к минимуму. Так, при смене среды передачи информации (при условии сохранения функциональности слоя IP) служба FTP будет продолжать работать как ни в чем не бывало.
- Каждый слой является удачным кандидатом на стандартизацию (например, TCP и IP — стандарты, определяющие особенности функционирования соответствующих слоев системы сетевых коммуникаций).
- Созданный слой может служить основой для нескольких различных слоев более высокого уровня (протоколы TCP/IP используются приложениями FTP, telnet, SSH и HTTP). В противном случае для каждого протокола высокого уровня пришлось бы изобретать собственный протокол низкого уровня.

Схема расслоения обладает и определенными недостатками.

- Слои способны удачно инкапсулировать многое, но не все: модификация одного слоя подчас связана с необходимостью внесения каскадных изменений в остальные слои. Классический пример из области корпоративных программных приложений: поле, добавленное в таблицу базы данных, подлежит воспроизведению в графическом интерфейсе и должно найти соответствующее отображение в каждом промежуточном слое.
- Наличие избыточных слоев нередко снижает производительность системы. При переходе от слоя к слою моделируемые сущности обычно подвергаются преобразованиям из одного представления в другое. Несмотря на это, инкапсуляция нижележащих функций зачастую позволяет достичь весьма существенного преимущества. Например, оптимизация слоя транзакций обычно приводит к повышению производительности всех вышележащих слоев.

Однако самое трудное при использовании архитектурных слоев — это определение содержимого и границ ответственности каждого слоя.

Развитие модели слоев в корпоративных программных приложениях

По молодости лет мне не довелось пообщаться с ранними версиями систем пакетной обработки данных, но, как мне кажется, тогда люди не слишком задумывались о каких-то там "слоях". Писалась программа, которая манипулировала некими файлами (ISAM, VSAM и т.п.), и этим, собственно говоря, функции приложения исчерпывались.

Понятие слоя приобрело очевидную значимость в середине 1990-х годов с появлением систем *клиент/сервер (client/server)*. Это были системы с двумя слоями: клиент нес ответственность за отображение пользовательского интерфейса и выполнение кода приложения, а роль сервера обычно поручалась СУБД. Клиентские приложения создавались с помощью таких инструментальных средств, как Visual Basic, PowerBuilder и Delphi, предоставлявших в распоряжение разработчика все необходимое, включая экранные компоненты, обслуживающие интерфейс SQL: для конструирования окна было достаточно перетащить на рабочую область необходимые управляющие элементы, настроить параметры доступа к базе данных и подключиться к ней, используя таблицы свойств.

Если задачи сводились к простым операциям по отображению информации из базы данных и ее незначительному обновлению, системы клиент/сервер действовали безотказно. Проблемы возникли с усложнением логики предметной области — бизнес-правил, алгоритмов вычислений, условий проверок и т.д. Прежде все эти обязанности возлагались на код клиента и находили отражение в содержимом интерфейсных экранов. Чем сложнее становилась логика, тем более неуклюжим и трудным для восприятия делался код. Воспроизведение элементов логики на экранах приводило к дублированию кода, и тогда при необходимости внести простейшее изменение приходилось "прочесывать" всю программу в поисках одинаковых фрагментов.

Одной из альтернатив было описание логики в тексте хранимых процедур, размещаемых в базе данных. Языки хранимых процедур, однако, отличались ограниченными возможностями структуризации, что вновь негативно сказывалось на качестве кода. Помимо

того, многие отдали предпочтение реляционным системам баз данных, поскольку используемый в них стандартизованный язык SQL открывал возможности безболезненного перехода от одной СУБД к другой. Хотя воспользовались ими на практике только единицы, мысль о возможной смене поставщика СУБД, не связанной со сколько-нибудь ощутимыми затратами, согревала всех. А наличие жесткой зависимости языков хранимых процедур от конкретных версий систем фактически разрушало эти надежды.

По мере роста популярности систем клиент/сервер набирала силу и парадигма объектно-ориентированного программирования, давшая сообществу ответ на сакраментальный вопрос о том, куда "девать" бизнес-логику: перейти к системной архитектуре с *тремя* слоями, в которой слой *представления* отводится пользовательскому интерфейсу, слой *предметной области* предназначен для описания бизнес-логики, а третий слой представляет *источник данных*. В этом случае удалось бы разнести интерфейс и логику, поместив последнюю на отдельный уровень, где она может быть структурирована с помощью соответствующих объектов.

Несмотря на предпринятые усилия, движение под знаменем объектной ориентации в направлении трехуровневой архитектуры было еще слишком робким и неуверенным. Многие проекты оказывались чрезмерно простыми, что отнюдь не вызывало у программистов желания покинуть наезженную колею систем клиент/сервер и связать себя новыми обязательствами. Помимо того, средства разработки приложений клиент/сервер с трудом поддерживали трехуровневую модель вычислений либо не предоставляли подобных инструментов вовсе.

Радикальный сдвиг произошел с появлением Web. Всем внезапно захотелось иметь системы клиент/сервер, где в роли клиента выступал бы Web-обозреватель. Если, однако, вся бизнес-логика приложения сосредоточивалась в коде *толстого* клиента, при переходе к Web-интерфейсу приходилось пересматривать ее полностью. А в удачно спроектированной трехуровневой системе достаточно было просто заменить уровень представления, не затрагивая слой предметной области. Позже, с появлением Java, все увидели объектно-ориентированный язык, претендующий на всеобщее признание. Появившиеся инструментальные средства конструирования Web-страниц были в меньшей степени связаны с SQL и потому более подходили для реализации третьего уровня.

При обсуждении вопросов расслоения программных систем нередко путают понятия *слоя (layer)* и *уровня*, или *яруса (tier)*. Часто их употребляют как синонимы, но в большинстве случаев термин *уровень* трактуют, подразумевая *физическое* разделение. Поэтому системы клиент/сервер обычно описывают как двухуровневые (в общем случае "клиент" действительно отделен от сервера физически): клиент — это приложение для настольной машины, а сервер — процесс, выполняемый сетевым компьютером-сервером. Я применяю термин *слой*, чтобы подчеркнуть, что слои вовсе *не* обязательно должны располагаться на разных машинах. Отдельный слой бизнес-логики может функционировать как на персональном компьютере "рядом" с клиентским слоем интерфейса, так и на сервере базы данных. В подобных ситуациях речь идет о двух узлах сети, но о трех слоях или уровнях. Если база данных локальна, все три слоя могут соседствовать и на одном компьютере, но даже в этом случае они должны сохранять свой суверенитет.

Три основных слоя

В этой книге внимание акцентируется на архитектуре с тремя основными слоями: *представление (presentation)*, *домен (предметная область, бизнес-логика) (domain)* и *источник данных (data source)*. В табл. 1.1 приведено их краткое описание (названия заимствованы из [9]).

Таблица 1.1. *Основные слои*

Слой	Функции
Представление	Предоставление услуг, отображение данных, обработка событий пользовательского интерфейса (щелчков кнопками мыши и нажатий клавиш), обслуживание запросов HTTP, поддержка функций командной строки и API пакетного выполнения
Домен	Бизнес-логика приложения
Источник данных	Обращение к базе данных, обмен сообщениями, управление транзакциями и т.д.

Слой *представления* охватывает все, что имеет отношение к общению пользователя с системой. Он может быть настолько простым, как командная строка или текстовое меню, но сегодня пользователю, вероятнее всего, придется иметь дело с графическим интерфейсом, оформленным в стиле толстого клиента (Windows, Swing и т.п.) или основанным на HTML. К главным функциям слоя представления относятся отображение информации и интерпретация вводимых пользователем команд с преобразованием их в соответствующие операции в контексте домена (бизнес-логики) и источника данных.

Источник данных — это подмножество функций, обеспечивающих взаимодействие со сторонними системами, которые выполняют задания в интересах приложения. Код этой категории несет ответственность за мониторинг транзакций, управление другими приложениями, обмен сообщениями и т.д. Для большинства корпоративных приложений основная часть логики источника данных сосредоточена в коде СУБД.

Логика *домена* (*бизнес-логика* или логика *предметной области*) описывает основные функции приложения, предназначенные для достижения поставленной перед ним цели. К таким функциям относятся вычисления на основе вводимых и хранимых данных, проверка всех элементов данных и обработка команд, поступающих от слоя представления, а также передача информации слою источника данных.

Иногда слои организуют таким образом, чтобы бизнес-логика полностью скрывала источник данных от представления. Чаще, однако, код представления может обращаться к источнику данных непосредственно. Хотя такой вариант менее безупречен с теоретической точки зрения, в практическом отношении он нередко более удобен и целесообразен: код представления может интерпретировать команду пользователя, активизировать функции источника данных для извлечения подходящих порций информации из базы данных, обратиться к средствам бизнес-логики для анализа этой информации и осуществления необходимых расчетов и только затем отобразить соответствующую картинку на экране.

Часто в рамках приложения предусматривают несколько вариантов реализации каждой из трех категорий логики. Например, приложение, ориентированное на использование как интерфейсных средств толстого клиента, так и командной строки, может (и, вероятно, должно) быть оснащено двумя соответствующими версиями логики представления. С другой стороны, различным базам данных могут отвечать многочисленные слои источников данных. На отдельные "пакеты" может быть поделен даже слой бизнес-логики (скажем, в ситуации, когда алгоритмы расчетов зависят от типа источника данных).

До сих пор предполагалось обязательное наличие пользователя. Возникает закономерный вопрос: что произойдет, если в управлении программным приложением человек участия не принимает (примерами могут служить и новейшие Web-службы, и традиционный процесс пакетной обработки)? В этом случае в роли пользователя приложения выступает сторонняя клиентская программа и становится очевидным сходство между слоями представления и источника данных: оба они задают связь приложения с внешним миром. Именно этот вариант логики лежит в основе типового решения **шестигранная архитектура (Hexagonal Architecture)** Алистера Коуберна (Alistair Cockburn) [39], которое трактует любую систему как ядро, окруженное интерфейсами к внешним системам. В **шестигранной архитектуре**, где все, что находится снаружи, — это не что иное, как интерфейсы к внешним субъектам, исповедуется симметричный подход к проблеме, в отличие от асимметричной схемы расслоения, которой придерживаюсь я.

Эта асимметрия, однако, кажется мне полезной, поскольку, как я полагаю, следует различать интерфейс, предлагаемый в виде службы другим, и сторонние службы, которыми пользуетесь вы сами. Собственно говоря, в этом и состоит реальное различие между слоями представления и источника данных. Представление — это внешний интерфейс к службе, который приложение открывает стороннему потребителю: либо человеку-оператору, либо программе. Источник данных — это интерфейс к функциям, которые предлагаются приложению внешней системой. Я нахожу очевидные выгоды в том, что интерфейсы трактуются как неодинаковые, поскольку это различие заставляет по-особому воспринимать каждую из служб.

Хотя три основных слоя — представление, бизнес-логика и источник данных — можно обнаружить в любом корпоративном приложении, способ их разделения зависит от степени сложности этого приложения. Простой сценарий извлечения порции информации из базы данных и отображения ее в контексте Web-страницы можно описать одной процедурой. Но я все равно попытался бы выделить в нем три слоя — пусть даже, как в этом случае, распределив функции каждого слоя по разным подпрограммам. При усложнении приложения это дало бы возможность разнести код слоев по отдельным классам, а позже разбить множество классов на пакеты. Форма расслоения может быть произвольной, но в любом корпоративном приложении слои *должны* быть идентифицированы.

Помимо необходимости разделения на слои, существует правило, касающееся взаимоотношения слоев: зависимость бизнес-логики и источника данных от уровня представления не допускается. Другими словами, в тексте приложения не должно быть вызовов функций представления *из* кода бизнес-логики или источника данных. Правило позволяет упростить возможность адаптации слоя представления или замены его альтернативным вариантом с сохранением основы приложения. Связь между бизнес-логикой и источником данных, однако, не столь однозначна и во многом определяется выбором типовых решений для архитектуры источника данных.

Самым сложным в работе над бизнес-логикой является, вероятно, выбор того, *что* именно и *как* следует относить к тому или иному слою. Мне нравится один неформальный тест. Вообразите, что в программу добавляется принципиально отличный слой, например интерфейс командной строки для Web-приложения. Если существует некий набор функций, которые придется продублировать для осуществления задуманного, значит, здесь логика домена "перетекает" в слой представления. Можно сформулировать тест иначе: нужно ли повторять логику при необходимости замены реляционной базы данных XML-файлом?

Хорошим примером ситуации является система, о которой мне когда-то рассказали. Представьте, что приложение отображает список выделенных красным цветом названий товаров, объемы продаж которых возросли более чем на 10% в сравнении с уровнем прошлого месяца. Допустим, программист разместил соответствующую логику непосредственно в слое представления, решив здесь же сопоставлять уровни продаж текущего и прошлого месяца и изменять цвет, если разность превышает заданный порог.

Проблема заключается в том, что в слой преставления вводится несвойственная ему логика предметной области. Чтобы надлежащим образом разделить слои, нужен метод бизнес-логики, отображающий факт превышения уровня продаж определенного продукта на заданную величину. Метод должен осуществить сравнение уровней продаж по двум месяцам и возвратить значение булева типа. В коде слоя представления достаточно вызвать этот метод и, руководствуясь полученным результатом, принять решение об изменении цвета отображения. В этом случае процесс разбивается на две части: выявление факта, который может служить основанием для изменения цвета, и собственно изменение.

Впрочем, мне не хотелось бы выглядеть сухим доктринером. Рецензируя эту книгу, Алан Найт (Alan Knight) как-то признался, что он "разрывался между тем, считать ли передачу подобной функции в ведение пользовательского интерфейса первым шагом на скользкой дорожке, ведущей прямиком в преисподнюю, либо вполне разумным компромиссным решением, с которым не согласился бы только отъявленный буквоед". Причина, которая беспокоит нас обоих, состоит в том, что на самом деле верны *оба* вывода!

Где должны функционировать слои

На протяжении всей книги речь идет о логических слоях, т.е. о расчленении системы на отдельные части. Подобное разделение полезно даже тогда, когда все слои функционируют на одной машине. Впрочем, существуют ситуации, где различия в поведении системы могут быть обусловлены принципами ее физической организации.

В большинстве случаев существует только два варианта размещения и выполнения компонентов корпоративных приложений — на персональном компьютере и на сервере.

Зачастую самым простым является функционирование кода всех слоев системы на сервере. Это становится возможным, например, при использовании HTML-интерфейса, воспроизводимого Web-обозревателем. Основным преимуществом сосредоточения всех частей приложения в одном месте является то, что при этом максимально упрощаются процедуры исправления ошибок и обновления версий. В этом случае не приходится беспокоиться о внесении соответствующих изменений на всех компьютерах, об их совместимости с другими приложениями и синхронизации с серверными компонентами.

Общие аргументы в пользу размещения каких-либо слоев на компьютере клиента состоят в повышении *быстроты реагирования (responsiveness)* приложения и в обеспечении возможности локальной работы. Чтобы код сервера смог отреагировать на действия, предпринимаемые пользователем на клиентской машине, требуется определенное время. А если пользователю необходимо быстро опробовать несколько вариантов и немедленно увидеть результат, продолжительность сетевого обмена становится серьезным препятствием. Помимо того, приложению требуется сетевое соединение как таковое. В частности, размышляя над этими строками, я находился в десяти километрах от ближайшего пункта, где можно было бы подключиться к сети, но хотелось бы, чтобы сетевой доступ обеспечивался везде. Может быть, в обозримом будущем так и случится, но что делать жителям какой-нибудь тьмутаракани, которые не желают ждать, пока кто-то из операторов беспроводной связи удосужится обеспечить "покрытие" их Богом забытого селения? А поддержка возможностей локального функционирования выдвигает особые требования, но боюсь, что они выбиваются из контекста этой книги.

Приняв к сведению все приведенные соображения, можно исследовать альтернативы, рассматривая слой за слоем. Слой источника данных лучше всегда располагать на сервере. Исключение составляет случай, когда функции сервера дублируются в коде "очень толстого" клиента для обеспечения средств локального функционирования системы. При этом предполагается, что изменения, вносимые в раздельные источники данных на клиентской машине и на сервере, подлежат синхронизации посредством механизма репликации. Однако, как уже упоминалось выше, обсуждение подобных вопросов придется отложить до лучших времен или передать инициативу другому автору.

Решение о том, где должен функционировать слой представления, большей частью зависит от предпочтений в выборе типа пользовательского интерфейса. Применение интерфейса толстого клиента автоматически влечет за собой необходимость размещения слоя представления на клиентской машине. Использование Web-интерфейса означает, что логика представления сосредоточена на сервере. Существуют и исключения, например удаленное управление клиентским программным обеспечением (таким, как X-сервер в UNIX) с запуском Web-сервера на настольном компьютере, но они редки.

Если речь идет о создании системы типа "поставщик–потребитель" ("business to customer" — B2C), у вас просто нет выбора. К серверу может подключиться любой, и вы вряд ли будете мириться с потерей посетителя только из-за того, что он использует какое-то экзотическое программное или аппаратное обеспечение. Поэтому целесообразно все функции сконцентрировать на сервере, а клиенту передавать материал в формате HTML, полностью готовый для воспроизведения с помощью Web-обозревателя. Подобное архитектурное решение ограничено в том, что реализация самой незначительной логики пользовательского интерфейса требует обращения к серверу, а это не может не сказаться на быстроте реагирования приложения. Уменьшить зависимость от сервера можно за счет применения фрагментов кода на языках сценариев Web-обозревателя (подобных JavaScript) и загружаемых аплетов, но подобные меры снижают уровень совместимости обозревателей и вызывают другие проблемы. Чем более "чист" код HTML, тем проще жизнь.

Вряд ли ваша жизнь будет простой даже в том случае, если каждый из настольных компьютеров вашей компании настроен, как утверждает начальник отдела информационных технологий, "максимально тщательно". Необходимость поддержки клиентского программного обеспечения в актуальном состоянии и требование исключить даже малую

вероятность его несовместимости с другими программами — это серьезные проблемы, которые проявляются и в тривиальных ситуациях.

Основной повод для применения интерфейсов толстого клиента — сложность задач и невозможность создания полноценных полезных приложений иной архитектуры. Однако популярность Web-интерфейсов неуклонно растет, а потребность в использовании толстых клиентов, напротив, снижается. Могу сказать одно: пользуйтесь Web-интерфейсами, если можете, и обращайтесь к средствам толстого клиента, если без них никак не обойтись.

А как быть с кодом бизнес-логики? Его можно активизировать или целиком на сервере, или полностью в контексте клиентской части, или используя смешанный стиль. И вновь вариант "все на сервере" наиболее привлекателен с точки зрения удобства сопровождения системы. Передача каких-либо бизнес-функций клиенту может быть обусловлена только, скажем, необходимостью повышения быстроты реагирования интерфейса системы или потребностью в средствах поддержки локального функционирования.

Если в рамках клиента необходимо выполнять какие-либо функции логики предметной области, прежде всего уместно рассмотреть возможность поручения клиенту *всех* таких функций. Подобный вариант очень похож на выбор интерфейса толстого клиента. Запуск Web-сервера на клиентской машине ненамного повысит быстроту реагирования приложения, хотя даст возможность использовать его в локальном режиме. Где бы ни находился код бизнес-логики, его следует сохранять в отдельных модулях, не связанных со слоем представления, используя одно из типовых решений — **сценарий транзакции (Transaction Script, 133)** или **модель предметной области (Domain Model, 140)**. Передача клиенту всего кода бизнес-логики сопровождается — и это уже отмечалось — усложнением процедур обновления системы.

Расщепление множества бизнес-функций между сервером и клиентом выглядит как наихудшее решение, поскольку в общем случае затрудняет идентификацию того или иного фрагмента логики. Основная причина, побуждающая применять подобную архитектуру, может состоять в том, что клиенту необходимо владеть только какой-то частью бизнес-логики. Главное — изолировать эту порцию кода в отдельном модуле, не зависящем от других частей системы. Это даст возможность активизировать код и на компьютере клиента, и на сервере, если такая потребность возникнет позже. Такой подход, разумеется, требует дополнительных усилий, но они оправданны.

После выбора узлов обработки необходимо попытаться обеспечить выполнение всего кода, относящегося к каждому отдельному узлу, в рамках единого процесса, функционирующего либо на одном узле, либо в пределах кластера из нескольких узлов. Не стоит делить слои по разрозненным процессам, если в этом нет насущной необходимости. В противном случае вам придется иметь дело с решениями типа **интерфейса удаленного доступа (Remote Facade, 405)** и **объекта переноса данных (Data Transfer Object, 419)**, а это чревато потерей производительности и повышением сложности.

Важно помнить, что подобные вещи относятся к числу тех, которые Дженс Коулдвей (Jens Coldewey) метко окрестила *катализаторами сложности (complexity boosters)*: это распределенная обработка, многопоточные вычисления, сочетание радикально различных концепций (например, "объектной ориентации" и "реляционной модели"), межплатформенное взаимодействие и обеспечение предельно высокого уровня быстродействия. Решение любой из названных задач сопряжено с большими затратами. Конечно, иногда приходится их нести, но это должно рассматриваться как исключение, а не правило.

Глава 2

Организация бизнес-логики

Рассматривая структуру логики *предметной области* (или *бизнес-логики*) приложения, я изучаю варианты распределения множества предусматриваемых ею функций по трем типовым решениям: **сценарий транзакции (Transaction Script, 133)**, **модель предметной области (Domain Model, 140)** и **модуль таблицы (Table Module, 148)**.

Простейший подход к описанию бизнес-логики связан с использованием **сценария транзакции** — процедуры, которая получает на вход информацию от слоя представления, обрабатывает ее, проводя необходимые проверки и вычисления, сохраняет в базе данных и активизирует операции других систем. Затем процедура возвращает слою представления определенные данные, возможно, осуществляя вспомогательные операции для форматирования содержимого результата. Бизнес-логика в этом случае описывается *набором* процедур, по одной на каждую (составную) операцию, которую способно выполнять приложение. Типовое решение **сценарий транзакции**, таким образом, можно трактовать как сценарий действия, или бизнес-транзакцию. Оно не обязательно должно представлять собой единый фрагмент кода. Код делится на подпрограммы, которые распределяются между различными **сценариями транзакции**.

Типовое решение **сценарий транзакции** отличается следующими преимуществами:

- представляет собой удобную процедурную модель, легко воспринимаемую всеми разработчиками;
- удачно сочетается с простыми схемами организации слоя источника данных на основе типовых решений **шлюз записи данных (Row Data Gateway, 175)** и **шлюз таблицы данных (Table Data Gateway, 167)**;
- определяет четкие границы транзакции.

С возрастанием уровня сложности бизнес-логики типовое решение **сценарий транзакции** демонстрирует и ряд недостатков. Если нескольким транзакциям необходимо осуществлять схожие функции, возникает опасность дублирования фрагментов кода. С этим явлением удается частично справиться за счет вынесения общих подпрограмм "за скобки", но даже в этом случае большая часть дубликатов остается на месте. В итоге приложение может выглядеть как беспорядочная мешанина без отчетливой структуры.

Конечно, сложная логика — это удачный повод вспомнить об объектах, и объектно-ориентированный вариант решения проблемы связан с использованием **модели предметной области**, которая, по меньшей мере в первом приближении, структурируется преимущественно вокруг основных сущностей рассматриваемого домена. Так, например, в лизинговой системе следовало бы создать классы, представляющие сущности "аренда", "имущество", "договор" и т.д., и предусмотреть логику проверок и вычислений: так, например, объект, представляющий сущность "имущество", вероятно, уместно снабдить логикой вычисления стоимости.

Выбор **модели предметной области** в противовес **сценарию транзакции** — это как раз та смена парадигмы программирования, о которой так любят говорить апологеты объектного подхода. Вместо использования одной подпрограммы, несущей в себе всю логику, которая соответствует некоторому действию пользователя, каждый объект наделяется только функциями, отвечающими его природе. Если прежде вы не пользовались **моделью предметной области**, процесс обучения может принести немало огорчений, когда в поисках нужных функций вам придется метаться от одного класса к другому.

Различие между двумя рассматриваемыми подходами на простом примере описать довольно сложно, но я все-таки попытаюсь это сделать. Простейший способ состоит в сопоставлении *диаграмм последовательностей (sequence diagrams)*, соответствующих обоим вариантам решения одной проблемы (рис. 2.1 и 2.2).

В задаче определения зачтенного дохода от продажи каждого продукта в рамках заданного контракта (подробнее это рассматривается в главе 9, "Представление бизнес-логики") алгоритм вычислений зависит от типа продукта. Вычислительный метод должен распознать тип продукта, применить подходящий алгоритм, а затем создать соответствующий объект, представляющий значение дохода. (Аспекты взаимодействия с базой данных для простоты здесь не рассматриваются.)

Из рис. 2.1 видно, что все обязанности возлагаются на метод **сценария транзакции**, получающий данные от объектов типа **шлюз таблицы данных**. На рис. 2.2 показано, напротив, несколько объектов, каждый из которых несет ответственность за выполнение своей доли функций, а результат генерируется объектом "Стратегия определения зачтенного дохода".

Ценность **модели предметной области** состоит в том, что, освоившись с подходом, вы получаете в свое распоряжение множество приемов, позволяющих поладить с возрастающей сложностью бизнес-логики "цивилизованным" путем. Например, с увеличением количества алгоритмов расчета дохода достаточно создавать новые объекты "Стратегия определения зачтенного дохода". При использовании **сценария транзакции** в аналогичной ситуации пришлось бы добавлять в логику метода дополнительные условия. Если вы настолько же неравнодушны к объектам, как и я, то наверняка предпочтете **модель предметной области** даже в самых простых случаях.

Стоимость практической реализации **модели предметной области** обусловлена степенью сложности как самой модели, так и конкретного варианта слоя источника данных. Чтобы добиться успехов в применении модели, новичкам придется затратить немало времени: некоторым требуется несколько месяцев работы над соответствующим проектом, прежде чем их стиль мышления перестроится в нужном направлении. После приобретения опыта работать становится намного проще — в вас даже просыпается энтузиазм. Через все это прошел и я, и все другие, кто так же одержим объектной парадигмой. Впрочем, сбросить груз привычек и сделать шаг вперед многим, к сожалению, так и не удается.

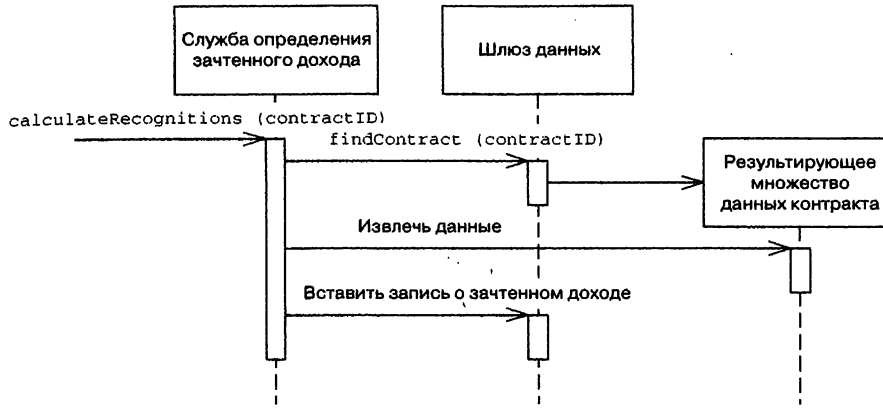


Рис. 2.1. Вычисление зачтенного дохода с помощью **сценария транзакции**

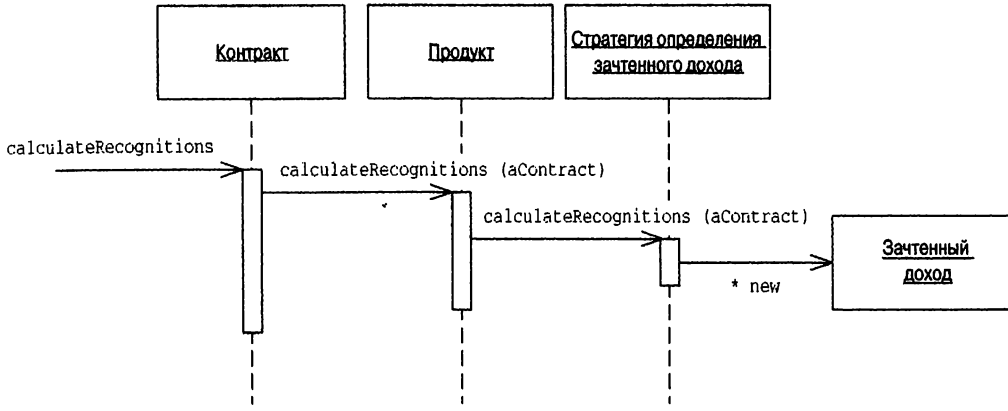


Рис. 2.2. Вычисление зачтенного дохода с помощью **модели предметной области**

Разумеется, каким бы ни был подход, необходимость отображать содержимое базы данных в структуры памяти и наоборот все еще остается. Чем более "богата" **модель предметной области**, тем сложнее становится аппарат взаимного отображения объектных структур в реляционные (обычно реализуемый на основе типового решения **преобразователь данных (Data Mapper, 187)**). Сложный слой источника данных стоит дорого — в финансовом смысле (если вы приобретаете услуги сторонних разработчиков) или в отношении затрат времен (если беретесь за дело самостоятельно), но если он у вас есть, считайте, что добрая половина проблемы уже решена.

Существует и третий вариант структуризации бизнес-логики, предусматривающий применение типового решения **модуль таблицы**; он показан на рис. 2.3.

На первый взгляд это типовое решение очень похоже на **модель предметной области**: в обоих случаях создаются отдельные классы, представляющие контракт, продукт и зачтенный доход. Принципиальное различие заключается в том, что **модель предметной области** содержит по одному объекту контракта для каждого контракта, зафиксированного

в базе данных, а **модуль таблицы** является единственным объектом. **Модуль таблицы** применяется совместно с типовым решением **множество записей (Record Set, 523)**. Посылая запросы к базе данных, пользователь прежде всего формирует объект **множество записей**, а затем создает объект контракта, передавая ему **множество записей** в качестве аргумента (см. рис. 2.3). Если потребуется выполнять операции над отдельным контрактом, следует сообщить объекту соответствующий идентификатор (ID).

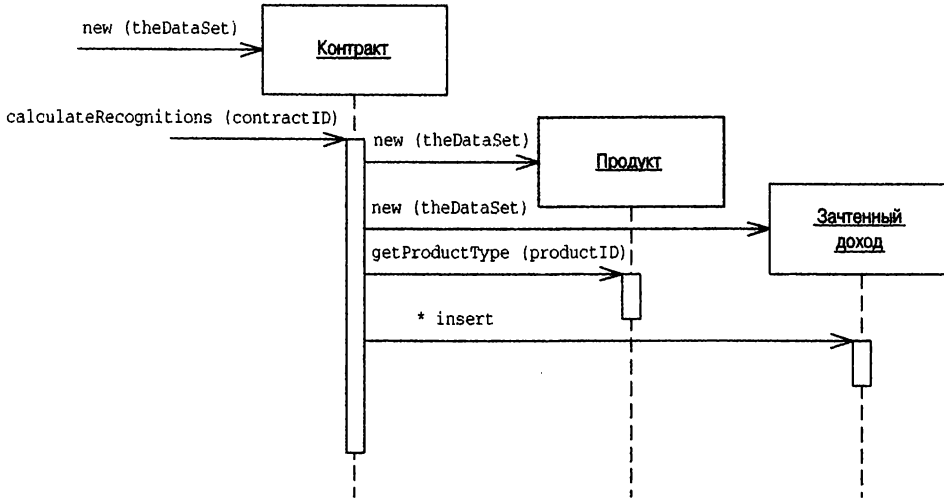


Рис. 2.3. Вычисление зачтенного дохода с помощью **модуля таблицы**

Модуль таблицы во многих смыслах представляет собой промежуточный вариант, компромиссный по отношению к **сценарию транзакции** и **модели предметной области**. Организация бизнес-логики вокруг таблиц, а не в виде прямолинейных процедур облегчает структурирование и возможность поиска и удаления повторяющихся фрагментов кода. Однако решение **модуль таблицы** не позволяет использовать многие технологии (скажем, *наследование (inheritance)*, *стратегии (strategies)* и другие объектно-ориентированные типовые решения), которые применяются в **модели предметной области** для уточнения структуры логики.

Наибольшее преимущество **модуля таблицы** состоит в том, *как* это решение сочетается с остальными аспектами архитектуры. Многие графические интерфейсные среды позволяют работать с результатами обработки SQL-запроса, организованными в виде **множества записей**. Поскольку решение **модуль таблицы** также основано на использовании **множества записей**, открывается возможность выполнения запроса, манипулирования его результатом в контексте **модуля таблицы** и передачи данных графическому интерфейсу для отображения. Некоторые платформы, в частности Microsoft COM и .NET, поддерживают именно такой стиль разработки.

Выбор типового решения

Итак, какому из трех типовых решений отдать предпочтение? Ответ неочевиден и во многом зависит от степени сложности бизнес-логики. На рис. 2.4 показан один из тех неформальных графиков, которые всегда действуют мне на нервы, когда приходится наблюдать презентации, созданные средствами PowerPoint. Причиной раздражения служит отсутствие единиц измерения величин, представляемых координатными осями. Впрочем, в данном случае подобный график кажется уместным, так как помогает визуализировать критерии сопоставления решений.

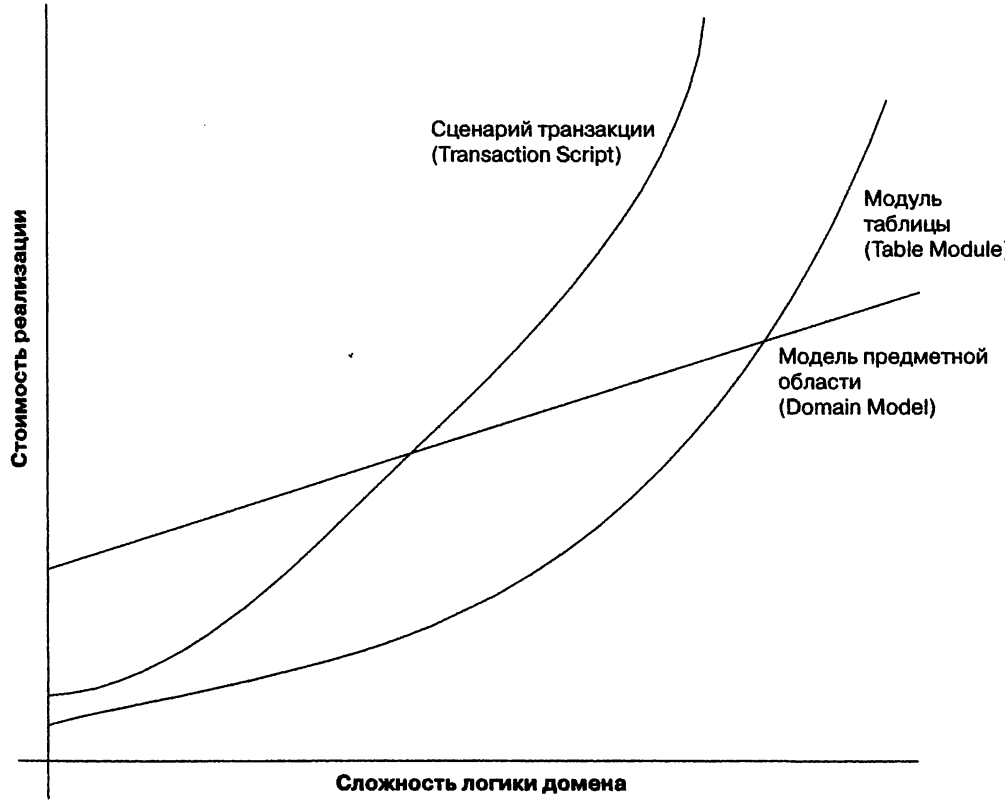


Рис. 2.4. Зависимость стоимости реализации различных схем организации бизнес-логики от ее сложности

Если логика приложения проста, **модель предметной области** менее соблазнительна, поскольку затраты на ее реализацию не окупаются. Но с возрастанием сложности альтернативные подходы становятся все менее приемлемыми: трудоемкость пополнения приложения новыми функциями увеличивается по экспоненциальному закону.

Конечно, необходимо разобраться, к какому именно сегменту оси *x* относится конкретное приложение. Было бы совсем неплохо, если бы я имел право сказать, что **модель предметной области** следует применять в тех случаях, когда сложность бизнес-логики

составляет, например, 7,42 или больше. Однако *никто* не знает, как измерять сложность бизнес-логики. Поэтому на практике проблема обычно сводится к выслушиванию мнений сведущих людей, которые способны хоть как-то проанализировать ситуацию и прийти к осмысленным выводам.

Существует ряд факторов, влияющих на кривизну линий графика. Наличие команды разработчиков, уже знакомых с **моделью предметной области**, позволяет снизить величину начальных затрат — хотя и не до уровней, характерных для двух других зависимостей (последнее обусловлено сложностью слоя источника данных). Таким образом, чем опытнее вы и ваши коллеги, тем больше у вас оснований для применения **модели предметной области**.

Эффективность **модуля таблицы** серьезно зависит от уровня поддержки структуры **множества записей** в конкретной инструментальной среде. Если вы работаете с чем-то наподобие Visual Studio .NET, где многие средства построены именно на основе модели **множества записей**, это обстоятельство наверняка придаст **модулю таблицы** дополнительную привлекательность. Что касается **сценария транзакции**, то я вообще не вижу доводов в пользу его применения в контексте .NET. Более того, если бы не специальная поддержка схемы **множества записей**, в этой ситуации я не стал бы возиться и с **модулем таблицы**.

Как только архитектура выбрана (пусть и не окончательно), изменить ее со временем становится все труднее. Поэтому целесообразно приложить определенные усилия, чтобы *заранее* решить, каким путем двигаться. Если вы начали со **сценария транзакции**, но затем поняли свою ошибку, не мешкая обратитесь к **модели предметной области**. Если вы с нее и начинали, переход к **сценарию транзакции** вряд ли окажется удачным, если только вы не сможете серьезно упростить слой источника данных.

Три типовых решения, которые мы бегло рассмотрели, *не* являются взаимоисключающими альтернативами. На самом деле **сценарий транзакции** нередко используется для некоторого фрагмента бизнес-логики, а **модель предметной области** или **модуль таблицы** — для оставшейся части.

Уровень служб

Один из общих подходов к реализации бизнес-логики состоит в расщеплении слоя предметной области на два самостоятельных слоя: "поверх" **модели предметной области** или **модуля таблицы** располагается **слой служб (Service Layer, 156)**. Обычно это целесообразно только при использовании **модели предметной области** или **модуля таблицы**, поскольку слой домена, включающий лишь **сценарий транзакции**, не настолько сложен, чтобы заслужить право на создание дополнительного слоя. Логика слоя представления взаимодействует с бизнес-логикой исключительно при посредничестве **слоя служб**, который действует как API приложения.

Поддерживая внятный интерфейс приложения (API), **слой служб** подходит также для размещения логики управления транзакциями и обеспечения безопасности. Это дает возможность снабдить подобными характеристиками каждый метод **слоя служб**. Для таких целей обычно применяются файлы свойств, но атрибуты .NET предоставляют удобный способ описания параметров непосредственно в коде.

Основное решение, принимаемое при проектировании **слоя служб**, состоит в том, какую часть функций уместно передать в его ведение. Самый "скромный" вариант —

представить **слой служб** в виде промежуточного интерфейса, который только и делает, что направляет адресуемые ему вызовы к нижележащим объектам. В такой ситуации **слой служб** обеспечивает API, ориентированный на определенные *варианты использования (use cases)* приложения, и предоставляет удачную возможность включить в код функции-оболочки, ответственные за управление транзакциями и проверку безопасности.

Другая крайность — в рамках **слоя служб** представить большую часть логики в виде **сценариев транзакции**. Нижележащие объекты домена в этом случае могут быть тривиальными; если они сосредоточены в **модели предметной области**, удастся обеспечить их однозначное отображение на элементы базы данных и воспользоваться более простым вариантом слоя источника данных (скажем, **активной записью (Active Record, 182)**).

Между двумя указанными полюсами существует вариант, представляющий собой *больше*, нежели "смесь" двух подходов: речь идет о модели *"контроллер–сущность" ("controller–entity")* (ее название обязано своим происхождением одному общеупотребительному приему, основанному на результатах из [22]). Главная особенность модели заключается в том, что логика, относящаяся к отдельным транзакциям или вариантам использования, располагается в соответствующих **сценариях транзакции**, которые в данном случае называют контроллерами (или службами). Они выполняют роль входных контроллеров в типовых решениях **модель–представление–контроллер (Model View Controller, 347)** и **контроллер приложения (Application Controller, 397)** (вы познакомитесь с ними позже) и поэтому называются также *контроллерами вариантов использования (use case controller)*. Функции, характерные одновременно для нескольких вариантов использования, передаются *объектам-сущностям (entities)* домена.

Хотя модель "контроллер–сущность" распространена довольно широко, я отношусь к ней с прохладцей. Контроллеры вариантов использования, подобные любому **сценарию транзакции**, негласно поощряют дублирование фрагментов кода. Моя точка зрения такова: если вы сказали "а", решив прибегнуть к **модели предметной области**, то будьте любезны произнести и "б", отведя этому решению доминирующую роль. Единственное достойное исключение, вероятно, связано с изначальным использованием **сценария транзакции** совместно со **шлюзом записи данных**. В такой ситуации имеет смысл передать повторяющиеся функции **шлюзам записи данных**, преобразовав их в простую **модель предметной области** с помощью решения **активная запись**. Впрочем, я бы так не делал (может быть, к этому меня вынудила бы только необходимость модернизации существующего приложения).

Здесь речь не идет о том, что вы никогда не должны использовать объекты служб, содержащие бизнес-логику; просто я хочу подчеркнуть, что создавать на их основе фиксированный слой кода необязательно. Процедурные служебные объекты подчас весьма удобны для представления логики, но я склоняюсь к тому, чтобы применять их только по мере необходимости, а не в виде архитектурного слоя.

Таким образом, на вашем месте я предпочел бы самый тонкий **слой служб**, какой только возможен (если он вообще нужен). Обычно же я добавляю его только тогда, когда он действительно необходим. Впрочем, мне знакомы хорошие специалисты, которые *всегда* применяют **слой служб**, содержащий взвешенную долю бизнес-логики, так что этим моим советом вы можете благополучно пренебречь. Рэнди Стаффорд (Randy Stafford) добился впечатляющих успехов на ниве проектирования программного обеспечения, используя модели с "богатым" **слоем служб**, поэтому я и попросил его написать одноименный раздел этой книги.

Глава 3

Объектные модели и реляционные базы данных

Роль слоя источника данных состоит в том, чтобы обеспечить возможность взаимодействия приложения с различными компонентами инфраструктуры для выполнения необходимых функций. Главная составляющая подобной проблемы связана с поддержкой диалога с базой данных — в большинстве случаев *реляционной*. До сих пор изрядные массивы данных все еще хранятся в устаревших форматах (скажем, в файлах ISAM и VSAM), но практически все разработчики современных приложений, предусматривающих связь с системами баз данных, ориентируются на реляционные СУБД.

Одной из самых серьезных причин успеха реляционных систем является поддержка ими SQL — наиболее стандартизованного языка коммуникаций с базой данных. Хотя сегодня SQL все более обрастает раздражающе несовместимыми и сложными "улучшениями", поддерживаемыми различными поставщиками СУБД, синтаксис ядра языка, к счастью, остается неизменным и доступным для всех.

Архитектурные решения

В первую обойму архитектурных решений входят, в частности, и такие, которые оговаривают способы взаимодействия бизнес-логики с базой данных. Выбор, который делается на этом этапе, имеет далеко идущие последствия и отменить его бывает трудно или даже невозможно. Поэтому он заслуживает наиболее тщательного осмысления. Нередко подобными решениями как раз и обусловливаются варианты компоновки бизнес-логики.

Несмотря на широкую поддержку корпоративными приложениями, использование SQL сопряжено с определенными трудностями. Многие разработчики просто не владеют SQL и потому, пытаясь сформулировать эффективные запросы и команды, сталкиваются с проблемами. Помимо того, все без исключения технологии внедрения предложений SQL в код на языке программирования общего назначения страдают теми или иными

изъянами. (Безусловно, было бы лучше осуществлять доступ к содержимому базы данных с помощью неких механизмов уровня языка разработки приложения.) А администраторы баз данных хотели бы уяснить нюансы обработки SQL-выражений, чтобы иметь возможность их оптимизировать.

По этим причинам разумнее обособить код SQL от бизнес-логики, разместив его в специальных классах. Удачный способ организации подобных классов состоит в "копировании" структуры каждой таблицы базы данных в отдельном классе, который формирует **шлюз (Gateway, 483)**, поддерживающий возможности обращения к таблице. Теперь основному коду приложения нет необходимости что-либо "знать" о SQL, а все SQL-операции сосредоточиваются в компактной группе классов.

Существует два основных варианта практического использования типового решения **шлюз**. Наиболее очевидный — создание экземпляра **шлюза** для каждой записи, возвращаемой в результате обработки запроса к базе данных (рис. 3.1). Подобный **шлюз записи данных (Row Data Gateway, 175)** представляет собой модель, естественным образом отображающую объектно-ориентированный стиль восприятия реляционных данных.

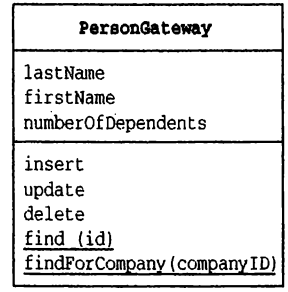


Рис. 3.1. Для каждой записи, возвращаемой запросом, создается экземпляр **шлюза записи данных**

Во многих средах поддерживается модель **множества записей (Record Set, 523)** — одна из основополагающих структур данных, имитирующая табличную форму представления содержимого базы данных. Инструментальными системами предлагаются даже графические интерфейсные элементы, реализующие схему **множества записей**. С каждой таблицей базы данных следует сопоставить соответствующий класс типа **шлюз таблицы данных (Table Data Gateway, 167)** (рис. 3.2), который содержит методы активизации запросов, возвращающих **множество записей**.

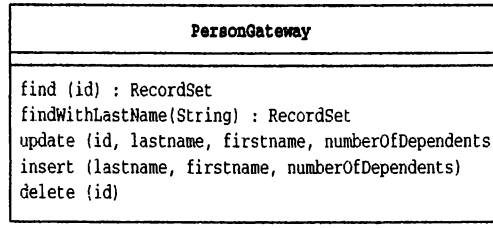


Рис. 3.2. Для каждой таблицы в базе данных создается экземпляр **шлюза таблицы данных**

Я склоняюсь к необходимости применения какого-либо из "шлюзовых" решений даже в простых приложениях (чтобы убедиться в этом, достаточно одного взгляда на мои сценарии на языках Ruby и Python), поскольку убежден в несомненной целесообразности четкого разделения кода SQL и бизнес-логики.

Тот факт, что **шлюз таблицы данных** удачно сочетается с **множеством записей**, обеспечивает такому варианту шлюза очевидные преимущества при использовании **модуля таблицы (Table Module, 148)**. Это типовое решение следует иметь в виду и при работе с хранимыми процедурами. Нередко предпочтительно осуществлять доступ к базе данных только при посредничестве хранимых процедур, а не с помощью прямых обращений. В подобной ситуации, определяя **шлюз таблицы данных** для каждой таблицы, следует предусмотреть и коллекцию соответствующих хранимых процедур. А я создал бы в памяти еще и дополнительный **шлюз таблицы данных**, выполняющий функцию оболочки, которая могла бы скрыть механизм вызовов хранимых процедур.

При использовании **модели предметной области (Domain Model, 140)** возникают другие альтернативы. В этом контексте уместно применять и **шлюз записи данных**, и **шлюз таблицы данных**. Впрочем, по моему мнению, в данной ситуации эти решения могут оказаться не вполне целенаправленными или просто несостоятельными.

В простых приложениях **модель предметной области** представляет отнюдь не сложную структуру, которая может содержать по одному классу домена в расчете на каждую таблицу базы данных. Объекты таких классов часто снабжены умеренно сложной бизнес-логикой. В этом случае имеет смысл возложить на каждый из них и ответственность за ввод-вывод данных, что, по существу, равносильно применению решения **активная запись (Active Record, 182)** (рис. 3.3). Это решение можно воспринимать и так, будто, начав со **шлюза записи данных**, мы добавили в класс порцию бизнес-логики (может быть, когда обнаружили в нескольких **сценариях транзакции (Transaction Script, 133)** повторяющиеся фрагменты кода).

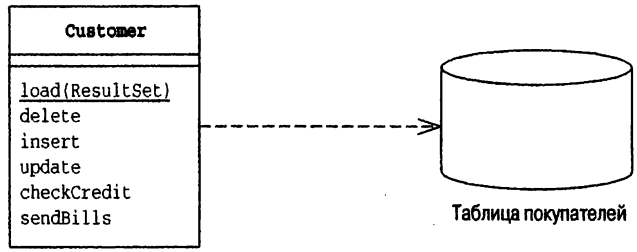


Рис. 3.3. При использовании решения **активная запись** *объект класса домена "осведомлен" о том, как взаимодействовать с таблицами базы данных*

В подобного рода ситуациях дополнительный уровень опосредования, формируемый за счет применения **шлюза**, не представляет большой ценности. По мере усложнения бизнес-логики и возрастания значимости богатой **модели предметной области** простые решения типа **активная запись** начинают сдавать свои позиции. При разнесении бизнес-логики по все более мелким классам взаимно однозначное соответствие между классами домена и таблицами базы данных постепенно теряется. Реляционные базы данных не поддерживают механизм наследования, что затрудняет применение *стратегий (strategies)* [20] и других развитых объектно-ориентированных типовых решений. А когда бизнес-

логика становится все более беспорядочной, желательно иметь возможность ее тестирования без необходимости постоянно обращаться к базе данных.

С усложнением **модели предметной области** все эти обстоятельства вынуждают создавать промежуточные функциональные уровни. Так, решение типа **шлюза** способно устранить некоторые проблемы, но оно все еще оставляет нас с **моделью предметной области**, тесно привязанной к схеме базы данных. В результате при переходе от полей **шлюза** к полям объектов домена приходится выполнять определенные преобразования, которые приводят к усложнению объектов домена.

Более удачный вариант состоит в том, чтобы изолировать **модель предметной области** от базы данных, возложив на промежуточный слой всю полноту ответственности за отображение объектов домена в таблицы базы данных. Подобный **преобразователь данных (Data Mapper, 187)** (рис. 3.4) обслуживает все операции загрузки и сохранения информации, инициируемые бизнес-логикой, и позволяет независимо варьировать как **модель предметной области**, так и схему базы данных. Это наиболее сложное из архитектурных решений, обеспечивающих соответствие между объектами приложения и реляционными структурами, но его неоспоримое преимущество заключается в полном обособлении двух слоев.

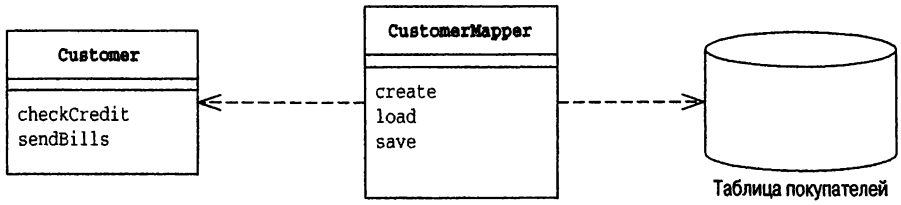


Рис. 3.4. **Преобразователь данных** *изолирует объекты домена от базы данных*

Я не рекомендовал бы рассматривать **шлюз** в качестве основного механизма сохранения данных в контексте **модели предметной области**. Если бизнес-логика действительно проста и степень соответствия между классами и таблицами базы данных высока, удачной альтернативой окажется типовое решение **активная запись**. Если же вам приходится иметь дело с чем-то более сложным, самым лучшим выбором, вероятнее всего, будет **преобразователь данных**.

Рассмотренные типовые решения нельзя считать взаимоисключающими. В большинстве случаев речь будет идти о механизме сохранения информации из неких структур памяти в базе данных. Для этого придется выбрать *одно* из перечисленных решений — смешение подходов чревато неразберихой. Но даже если в качестве инструмента доступа к базе данных применяется, скажем, **преобразователь данных**, для создания оболочек таблиц или служб, трактуемых как внешние интерфейсы, вы вправе использовать, например, **шлюз**.

Здесь и ниже, употребляя слово *таблица (table)*, я имею в виду, что обсуждаемые приемы и решения применимы в равной мере ко всем данным, отличающимся табличным характером: к *хранимым процедурам (stored procedures)*, *представлениям (views)*, а также к (промежуточным) результатам выполнения "традиционных" запросов и хранимых процедур. К сожалению, какой-либо общий термин, который охватывал бы все эти понятия, отсутствует. (Под представлением я подразумеваю *виртуальную таблицу (virtual table)* — то же толкование принято и в SQL; запросы к обычным и виртуальным таблицам обладают одним и тем же синтаксисом.)

Операции *обновления (update)* содержимого источников, не являющихся хранимыми таблицами, очевидно, более сложны, поскольку представление далеко не всегда можно модифицировать напрямую — вместо этого приходится манипулировать таблицами, на основе которых оно создано. В этом случае инкапсуляция представления/запроса с помощью подходящего типового решения — хороший способ сосредоточить логику операций обновления в одном месте, что делает использование виртуальных структур более простым и безопасным.

Впрочем, проблемы все еще остаются. Непонимание природы формирования виртуальных таблиц приводит к несогласованности операций: если последовательно выполнить операции обновления двух различных структур, построенных на основе одних и тех же хранимых таблиц, вторая операция способна повлиять на результаты первой. Конечно, в логике обновления можно предусмотреть соответствующие проверки, направленные на то, чтобы избежать возникновения подобных эффектов, но какую цену за все это придется заплатить?

Я обязан также упомянуть о самом простом способе сохранения данных, который можно использовать даже в наиболее сложных **моделях предметной области**. Еще на заре эпохи объектов многие осознали, что существует фундаментальное расхождение между реляционной и объектной моделями; это послужило стимулом создания *объектно-ориентированных* СУБД, расширяющих парадигму до аспектов сохранения информации об объектах на дисках. При работе с объектно-ориентированной базой данных вам не нужно заботиться об отображении объектов в реляционные структуры. В вашем распоряжении набор взаимосвязанных объектов, а о том, как и когда их считывать или сохранять, "беспокоится" СУБД. Помимо того, с помощью механизма транзакций вы вправе группировать операции и управлять совместным доступом к объектам. Для программиста все это выглядит так, словно он взаимодействует с неограниченным пространством объектов, размещенных в оперативной памяти.

Главное преимущество объектно-ориентированных баз данных — повышение производительности разработки приложений: хотя какие-либо воспроизводимые тестовые показатели мне не известны, в неформальных беседах постоянно высказываются суждения о том, что затраты на обеспечение соответствия между объектами приложения и реляционными структурами традиционной базы данных составляют приблизительно треть от общей стоимости проекта и не снижаются в течение всего цикла сопровождения системы.

В большинстве случаев, однако, объектно-ориентированные базы данных применения не находят, и основная причина такого положения вещей — риск. За реляционными СУБД стоят тщательно разработанные, хорошо знакомые и проверенные жизнью технологии, поддерживаемые всеми крупными поставщиками систем баз данных на протяжении длительного времени. SQL представляет собой в достаточной мере унифицированный интерфейс для разнообразных инструментальных средств. (Если вас заботят проблемы продуктивности разработки и функционирования прикладных программ, предусматривающих необходимость взаимодействия с СУБД, я могу сказать только одно: с какими бы то ни было убедительными сравнительными характеристиками производительности объектно-ориентированных и реляционных систем я не знаком.)

Если у вас нет возможности или желания воспользоваться объектно-ориентированной базой данных, то, делая ставку на **модель предметной области**, вы должны серьезно изучить варианты приобретения инструментов отображения объектов в реляционные структуры. Хотя рассмотренные в этой книге типовые решения сообщат вам многое из

того, что необходимо знать для конструирования эффективного **преобразователя данных**, на этом поприще от вас все еще потребуются значительные усилия. Поставщики коммерческих инструментов, предназначенных для "связывания" объектно-ориентированных приложений с реляционными базами данных, затратили многие годы на разрешение подобных проблем, и их программные продукты во многом более изобретательны, гибки и мощны, нежели те, которые можно "состряпать" кустарным образом. Сразу оговорюсь, что они недешевы, но затраты на их возможную покупку следует сопоставить со значительными расходами, которые вам придется понести, избрав путь самостоятельного создания и сопровождения этого слоя программного кода.

Нельзя не вспомнить о попытках разработки образцов слоя кода в стиле объектно-ориентированных СУБД, способного взаимодействовать с реляционными системами. В мире Java таким "зверем", например, является JDO, но о достоинствах или недостатках этой технологии пока нельзя сказать ничего определенного. У меня слишком малый опыт ее применения, чтобы приводить на страницах этой книги какие-либо категорические заключения.

Даже если вы решились на приобретение готовых инструментов, знакомство с представленными здесь типовыми решениями вовсе не помешает. Хорошие инструментальные системы предлагают обширный арсенал альтернатив "подключения" объектов к реляционным базам данных, и компетентность в сфере типовых решений поможет вам прийти к осмысленному выбору. Не думайте, что система в одночасье избавит вас от всех проблем; чтобы заставить ее работать эффективно, вам придется приложить изрядные усилия.

Функциональные проблемы

Когда речь заходит о взаимном отображении объектов и таблиц реляционной базы данных, внимание обычно сосредоточивается на структурных аспектах, т.е. на том, как следует соотносить таблицы и объекты. Однако я совершенно уверен, что самая трудная часть проблемы — это выбор и обоснование архитектурной и функциональной составляющих.

Обсудив основные варианты архитектурных решений, рассмотрим функциональную (поведенческую) сторону, в частности вопрос о том, как обеспечить загрузку различных объектов и сохранение их в базе данных. На первый взгляд это не кажется слишком сложной задачей: объект можно снабдить соответствующими методами загрузки ("load") и сохранения ("save"). Именно такой путь целесообразно избрать, например, при использовании решения **активная запись (Active Record, 182)**.

Загружая в память большое количество объектов и модифицируя их, система должна следить за тем, какие объекты подверглись изменению, и гарантировать сохранение их содержимого в базе данных. Если речь идет всего о нескольких записях, это просто. Но по мере увеличения числа объектов растут и проблемы: как быть, скажем, в такой далеко не самой сложной ситуации, когда необходимо создать записи, которые должны ссылаться на ключевые значения друг друга?

При выполнении операций считывания или изменения объектов система должна гарантировать, что состояние базы данных, с которым вы имеете дело, остается согласованным. Так, например, на результат загрузки данных не должны влиять изменения,

вносимые другими процессами. В противном случае итог операции окажется непредсказуемым. Подобные вопросы, относящиеся к дисциплине управления параллельными заданиями, весьма серьезны. Они обсуждаются в главе 5, "Управление параллельными заданиями".

Типовым решением, имеющим существенное значение для преодоления такого рода проблем, является **единица работы (Unit of Work, 205)**, использование которой позволяет отследить, какие объекты считываются и какие модифицируются, и обслужить операции обновления содержимого базы данных. Автору прикладной программы нет нужды явно вызывать методы сохранения — достаточно сообщить объекту **единица работы** о необходимости *фиксации (commit)* результатов в базе данных. Типовое решение **единица работы** упорядочивает все функции по взаимодействию с базой данных и сосредоточивает в одном месте сложную логику фиксации. Его лучшие качества проявляются именно тогда, когда интерфейс между приложением и базой данных становится особенно запутанным.

Решение **единица работы** удобно воспринимать в виде объекта, действующего как контроллер процессов отображения объектов в реляционные структуры. В отсутствие такового роль контроллера, принимающего решения о том, когда и как загружать и сохранять объекты приложения, обычно выполняет слой бизнес-логики.

В процессе загрузки данных необходимо тщательно следить за тем, чтобы ни один из объектов не был считан дважды, иначе в памяти будут созданы два объекта, соответствующих одной и той же записи таблицы базы данных. Попробуйте обновить каждую из них, и неприятности не заставят себя ждать. Чтобы уйти от проблем, необходимо вести учет каждой считанной записи, а поможет в этом типовое решение **коллекция объектов (Identity Map, 216)**. Каждый раз при необходимости считывания порции данных вначале следует проверить, не содержится ли она в **коллекции объектов**. Если информация уже загружалась, можно предусмотреть возврат ссылки на нее. В этом случае любые попытки изменения данных будут скоординированы. Еще одно преимущество — возможность избежать дополнительного обращения к базе данных, поскольку **коллекция объектов** действует как кэш-память. Не забывайте, однако, что главное назначение **коллекции объектов** — учет идентификационных номеров объектов, а не повышение производительности приложения.

При использовании **модели предметной области (Domain Model, 140)** связанные объекты загружаются совместно таким образом, что операция считывания одного объекта инициирует загрузку другого. Если связями охвачено много объектов, считывание любого из них приводит к необходимости загружать из базы данных целый граф объектов. Чтобы исключить подобное неэффективное поведение системы, необходимо умерить аппетит, сократив количество загружаемых объектов, но оставить за собой право завершения операции, если потребность в дополнительной информации действительно возникнет. Типовое решение **загрузка по требованию (Lazy Load, 220)** предполагает использование специальных меток вместо ссылок на реальные объекты. Существует несколько вариаций схемы, но во всех случаях реальный объект загружается только тогда, когда предпринимается попытка проследовать по ссылке, которая его адресует. Решение **загрузка по требованию** позволяет оптимизировать число обращений к базе данных.

Считывание данных

Рассматривая проблему считывания информации из базы данных, я предпочитаю трактовать предназначенные для этого методы в виде *функций поиска (finders)*, скрывающих посредством соответствующих входных интерфейсов SQL-выражения формата "select". Примерами подобных методов могут служить `find(id)` или `findForCustomer(customer)`. Разумеется, если ваше приложение оперирует тремя десятками выражений "select" с различными критериями выбора, указанная схема становится чересчур громоздкой, но такие ситуации, к счастью, редки.

Принадлежность методов зависит от вида используемого интерфейсного типового решения. Если каждый класс, обеспечивающий взаимодействие с базой данных, привязан к определенной *таблице*, в его состав наряду с методами вставки и замены уместно включить и методы поиска. Если же объект класса соответствует отдельной *записи* данных, требуется иной подход.

В этом случае можно попробовать сделать методы поиска статическими, но за это придется заплатить некоторой долей гибкости, в частности вам более не удастся в целях тестирования заменить базу данных **фиктивной службой (Service Stub, 519)**. Чтобы избежать подобных проблем, лучше предусмотреть специальные классы поиска, включив в состав каждого из них методы, обеспечивающие инкапсуляцию тех или иных SQL-запросов. В результате выполнения запроса метод возвращает коллекцию объектов, соответствующих определенным записям данных.

Применяя методы поиска, следует помнить, что они выполняются в контексте состояния базы данных, а не состояния объекта. Если после создания в памяти объектов-записей данных, отвечающих некоторому критерию, вы активизируете запрос, который предполагает поиск записей, удовлетворяющих тому же критерию, то очевидно, что объекты, созданные, но не зафиксированные в базе данных, в результат обработки запроса не попадут.

При считывании данных проблемы производительности могут приобретать первостепенное значение. Необходимо помнить несколько эмпирических правил.

Старайтесь при каждом обращении к базе данных извлекать немного больше записей, чем нужно. В частности, не выполняйте один и тот же запрос повторно для приобретения дополнительной информации. Почти всегда предпочтительнее получить как можно больше данных, но нужно отдавать себе отчет в том, что при использовании пессимистической стратегии управления параллельными заданиями это может привести к ненужному блокированию большого количества записей. Например, если необходимо найти 50 записей, удовлетворяющих определенному условию, лучше выполнить запрос, возвращающий 200 записей, и применить для отбора искомых дополнительную логику, чем инициировать 50 отдельных запросов.

Другой способ исключить необходимость неоднократного обращения к базе данных связан с применением операторов *соединения (join)*, позволяющих с помощью одного запроса извлечь информацию из нескольких таблиц. Итоговый набор записей может содержать больше информации, чем требуется, но скорость его получения, вероятно, выше, чем в случае выполнения нескольких запросов, возвращающих в результате те же данные. Для этого следует воспользоваться **шлюзом (Gateway, 483)**, охватывающим информацию из нескольких таблиц, которые подлежат соединению, или

преобразователем данных (Data Mapper, 187), позволяющим загрузить несколько объектов домена с помощью единственного вызова.

Применяя механизм соединения таблиц, имейте в виду, что СУБД способны эффективно обслуживать запросы, в которых соединению подвергнуто не более трех-четырех таблиц. В противном случае производительность системы существенно падает, хотя воспрепятствовать этому, по меньшей мере частично, удается, например, с помощью разумной стратегии кэширования представлений.

Работу СУБД можно оптимизировать самыми разными способами, включая компактное группирование взаимосвязанных данных, тщательное проектирование индексов и кэширование порций информации в оперативной памяти. Все эти технологии, однако, выбиваются из контекста книги (хотя должны быть присущи сфере профессиональных интересов хорошего администратора баз данных).

Во всех случаях, тем не менее, следует учитывать особенности конкретных приложений и базы данных. Наставления общего характера хороши до определенного момента, когда необходимо как-то организовать способ мышления, но реальные обстоятельства всегда вносят свои коррективы. Достаточно часто СУБД и серверы приложений обладают сложными механизмами кэширования, затрудняющими дать хоть сколько-нибудь точную оценку производительности приложения. Никакое правило не обходится без исключений, так что тонкой настройки и доводки системы избежать не удастся.

Взаимное отображение объектов и реляционных структур

Во всех разговорах об объектно-реляционном отображении обычно и прежде всего имеется в виду обеспечение взаимно однозначного соответствия между объектами в памяти и табличными структурами базы данных на диске. Подобные решения, как правило, не имеют ничего общего с вариантами **шлюза таблицы данных (Table Data Gateway, 167)** и находят ограниченное применение совместно с решениями типа **шлюза записи данных (Row Data Gateway, 175)** и **активной записи (Active Record, 182)**, хотя все они, вероятно, окажутся востребованными в контексте **преобразователя данных (Data Mapper, 187)**.

Отображение связей

Главная проблема, которая обсуждается в этом разделе, обусловлена тем, что связи объектов и связи таблиц реализуются по-разному. Проблема имеет две стороны. Во-первых, существуют различия в способах представления связей. Объекты манипулируют связями, сохраняя ссылки в виде адресов памяти. В реляционных базах данных связь одной таблицы с другой задается путем формирования соответствующего *внешнего ключа (foreign key)*. Во-вторых, с помощью структуры коллекции объект способен сохранить *множество* ссылок из одного поля на другие, тогда как правила нормализации таблиц базы данных допускают применение только *однозначных* ссылок. Все это приводит к расхождениям в структурах объектов и таблиц. Так, например, в объекте, представляющем сущность "заказ", совершенно естественно предусмотреть коллекцию ссылок на объекты, описывающие заказываемые товары, причем последним нет необходимости ссылаться

на "родительский" объект заказа. Но в схеме базы данных все обстоит иначе: запись в таблице товаров должна содержать внешний ключ, указывающий на запись в таблице заказов, поскольку заказ не может иметь многозначного поля.

Чтобы решить проблему различий в способах представления связей, достаточно сохранять в составе объекта идентификаторы связанных с ним объектов-записей, используя типовое решение **поле идентификации (Identity Field, 237)**, а также обращаться к этим значениям, если требуется прямое и обратное отображение объектов и ключей таблиц базы данных. Это довольно скучно, но вовсе не так трудно, если усвоить основные приемы. При считывании информации из базы данных для перехода от идентификаторов записей к объектам используется **коллекция объектов (Identity Map, 216)**. Связи, задаваемой внешним ключом, отвечает типовое решение **отображение внешних ключей (Foreign Key Mapping, 258)**, устанавливающее подходящую связь одного объекта с другим (рис. 3.5). Если в **коллекции объектов** ключа нет, необходимо либо считать его из базы данных, либо применить вариант **загрузки по требованию (Lazy Load, 220)**. При сохранении информации объект фиксируется в таблице базы данных в виде записи с соответствующим ключом, а все ссылки на другие объекты, если таковые существуют, заменяются значениями **полей идентификации** этих объектов.

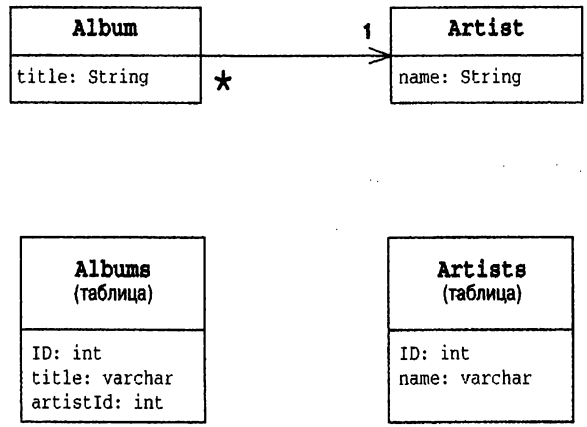


Рис. 3.5. Пример использования типового решения **отображение внешних ключей** *для реализации однозначной ссылки*

Если объект способен содержать коллекцию ссылок, требуется более сложная версия **отображения внешних ключей**, пример которой представлен на рис. 3.6. В этом случае, чтобы отыскать все записи, содержащие идентификатор объекта-источника, необходимо выполнить дополнительный запрос (или воспользоваться решением **загрузка по требованию**). Для каждой считанной записи, удовлетворяющей критерию поиска, создается соответствующий объект, и ссылка на него добавляется в коллекцию. Для сохранения коллекции следует обеспечить сохранение каждого объекта в ней, гарантируя корректность значений внешних ключей. Операция может быть довольно сложной, особенно в том случае, когда приходится отслеживать динамику добавления объектов в коллекцию и их удаления. В крупных системах с подобной целью применяются подходы, основанные на метаданных (об этом речь идет несколько позже). Если объекты

коллекции не используются вне контекста ее владельца, для упрощения задачи можно обратиться к типовому решению **отображение зависимых объектов (Dependent Mapping, 283)**.

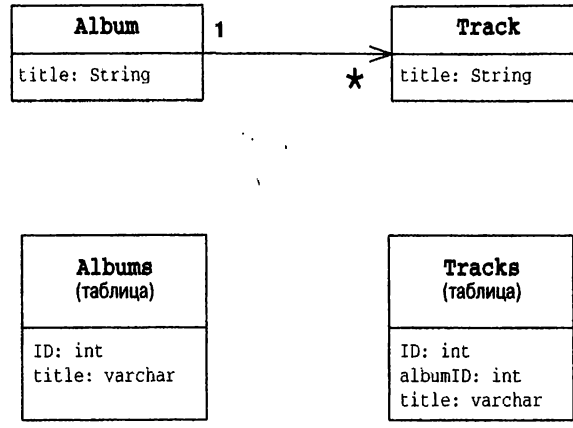


Рис. 3.6. Пример использования типового решения **отображение внешних ключей** *для реализации коллекции ссылок*

Совсем иная ситуация возникает, когда связь относится к категории "многие ко многим", т.е. коллекции ссылок присутствуют "с обеих сторон" связи. Примером может служить множество служащих и общий набор профессиональных качеств, отдельными из которых обладает каждый служащий. В реляционных базах данных проблема описания такой структуры преодолевается за счет создания дополнительной таблицы, содержащей пары ключей связанных сущностей (именно это предусмотрено в типовом решении **отображение с помощью таблицы ассоциаций (Association Table Mapping, 269)**), пример которого показан на рис. 3.7.

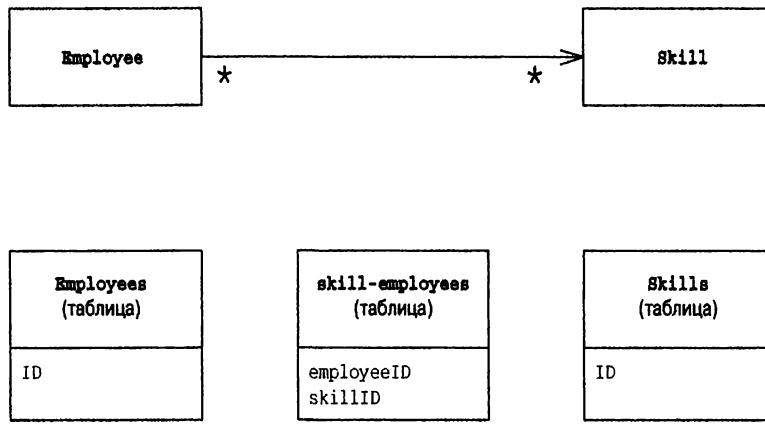


Рис. 3.7. Пример использования типового решения **отображение с помощью таблицы ассоциаций** *для реализации связи "многие ко многим"*

При работе с коллекциями принято полагаться на критерий упорядочения, с учетом которого она создана. В объектно-ориентированных языках программирования общей практикой является использование типов упорядоченных коллекций, подобных спискам и массивам. Однако задача сохранения в реляционной базе данных содержимого коллекции с произвольным критерием упорядочения остается сложной. В качестве способов ее решения можно порекомендовать использование неупорядоченных множеств или определение критерия упорядочения на этапе выполнения запроса к коллекции (последний подход связан с большими затратами).

В некоторых случаях проблема обновления данных усугубляется из-за необходимости выполнять условия *целостности на уровне ссылок (referential integrity)*. Современные СУБД позволяют *откладывать (defer)* проверку таких условий до завершения транзакции. Если "ваша" система такую возможность предоставляет, грех ею не воспользоваться. Если нет, система инициирует проверку после каждой операции записи. В такой ситуации вы обязаны соблюдать верный порядок прохождения операций. Не вдаваясь в детали, напомню, что один из подходов связан с построением и анализом графа, описывающего такой порядок, а другой состоит в задании жесткой схемы выполнения операций непосредственно в коде приложения. Иногда это позволяет снизить вероятность возникновения ситуаций *взаимоблокировки (deadlock)*, для разрешения которых приходится осуществлять *откат (rollback)* тех или иных транзакций.

Для описания связей между объектами, преобразуемых во внешние ключи, используется типовое решение **поле идентификации**, но далеко не все связи объектов следует фиксировать в базе данных именно таким образом. Разумеется, небольшие **объекты-значения (Value Object, 500)**, описывающие, скажем, диапазоны дат или денежные величины, нецелесообразно представлять в отдельных таблицах базы данных. **Объект-значение** уместнее отображать в виде **внедренного значения (Embedded Value, 288)**, т.е. набора полей объекта, "владеющего" **объектом-значением**. При необходимости загрузки данных в память **объекты-значения** можно легко создавать заново, не утруждая себя использованием **коллекции объектов**. Сохранить **объект-значение** также несложно — достаточно зафиксировать значения его полей в таблице объекта-владельца.

Можно пойти дальше и предложить модель группирования **объектов-значений** с сохранением их в таблице базы данных в виде единого поля типа **крупный сериализованный объект (Serialized LOB, 292)**. Аббревиатура LOB происходит от словосочетания Large OBject и означает "крупный объект"; различают *крупные двоичные объекты (Binary LOBs — BLOBs)* и *крупные символьные объекты (Character LOBs — CLOBs)*. Сериализация множества объектов в виде XML-документа — очевидный способ сохранения иерархических объектных структур. В этом случае для считывания исходных объектов будет достаточно одной операции. При выполнении большого количества запросов, предполагающих поиск мелких взаимосвязанных объектов (например, для построения диаграмм или обработки счетов), производительность СУБД часто резко падает, и **крупные сериализованные объекты** позволяют существенно снизить загрузку системы.

Недостаток такого подхода состоит в том, что в рамках "чистого" SQL сконструировать запросы к отдельным элементам сохраненной структуры не удастся. На помощь может прийти XML, позволяющий внедрить выражения формата XPath в вызовы SQL, хотя такой подход на данный момент пока не стандартизован. **Крупные сериализованные объекты** лучше всего применять для хранения относительно небольших групп объектов. Злоупотребление этим подходом приведет к тому, что база данных со временем будет напоминать файловую систему.

Наследование

Выше уже упоминались составные иерархические структуры, которые традиционно плохо отображаются средствами реляционных систем баз данных. Существует и другая разновидность иерархий, еще более усугубляющих страдания приверженцев реляционной модели: речь идет об иерархиях классов, создаваемых на основе механизма *наследования (inheritance)*. Поскольку SQL не предоставляет стандартизованных инструментов поддержки наследования, придется вновь прибегнуть к аппарату отображения. Существует три основных варианта представления структуры наследования: "одна таблица для всех классов иерархии" (**наследование с одной таблицей (Single Table Inheritance, 297)**) — рис. 3.8; "таблица для каждого конкретного класса" (**наследование с таблицами для каждого конкретного класса (Concrete Table Inheritance, 313)**) — рис. 3.9; "таблица для каждого класса" (**наследование с таблицами для каждого класса (Class Table Inheritance, 305)**) — рис. 3.10.

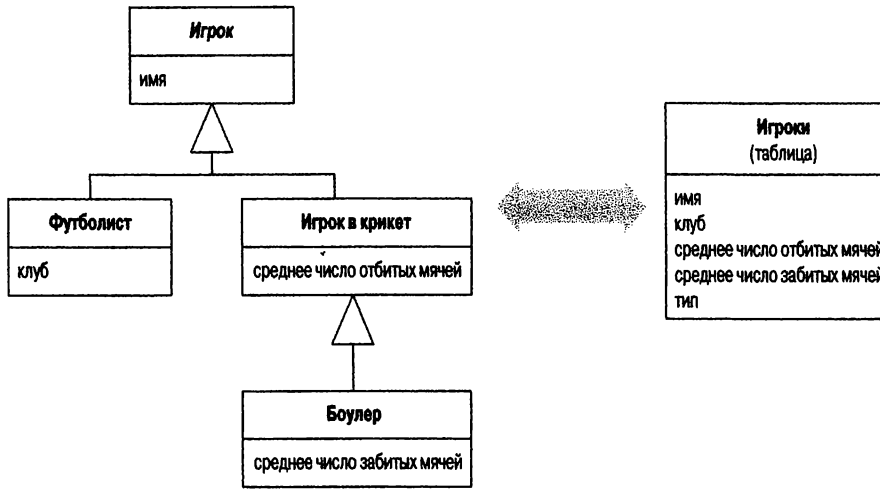


Рис. 3.8. Типовое решение **наследование с одной таблицей** *предусматривает сохранение значений атрибутов всех классов иерархии в одной таблице*

Возможен компромисс между необходимостью дублирования элементов данных и потребностью в ускорении доступа к ним. Решение **наследование с таблицами для каждого класса** — самый простой и прямолинейный вариант соответствия между классами и таблицами базы данных, но для загрузки информации об отдельном объекте в этом случае приходится осуществлять несколько операций соединения (join), что обычно сопряжено со снижением производительности системы. Решение **наследование с таблицами для каждого конкретного класса** позволяет обойти соединения, предоставляя возможность считывания всех данных об объекте из единственной таблицы, но существенно препятствует внесению изменений. При любой модификации базового класса нельзя забывать о необходимости соответствующего преобразования всех таблиц дочерних классов (и кода, обеспечивающего корректное отображение). Изменение самой иерархической структуры способно вызвать еще более серьезные проблемы. Помимо того, отсутствие таблицы для базового класса может усложнить управление ключами. Что касается **наследования с одной**

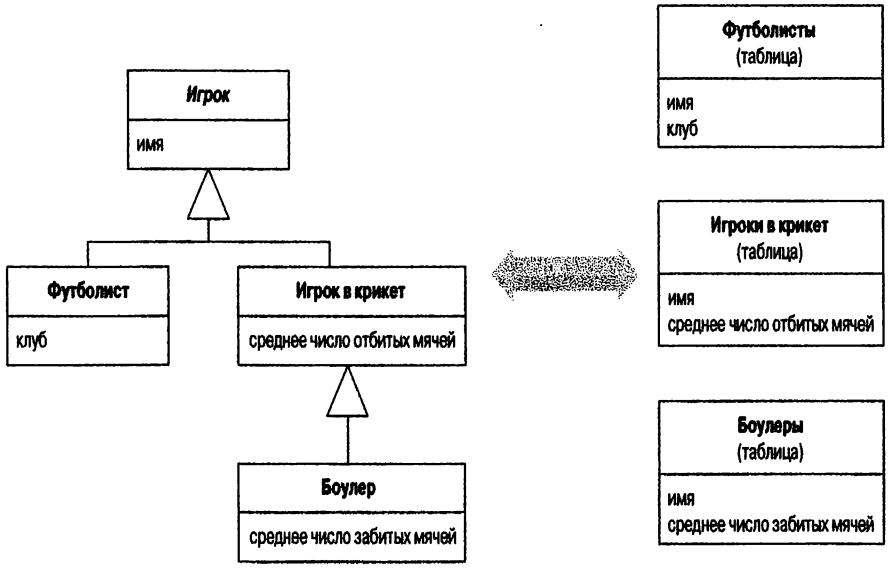


Рис. 3.9. Типовое решение **наследование с таблицами для каждого конкретного класса** *предусматривает использование отдельных таблиц для каждого конкретного класса иерархии*

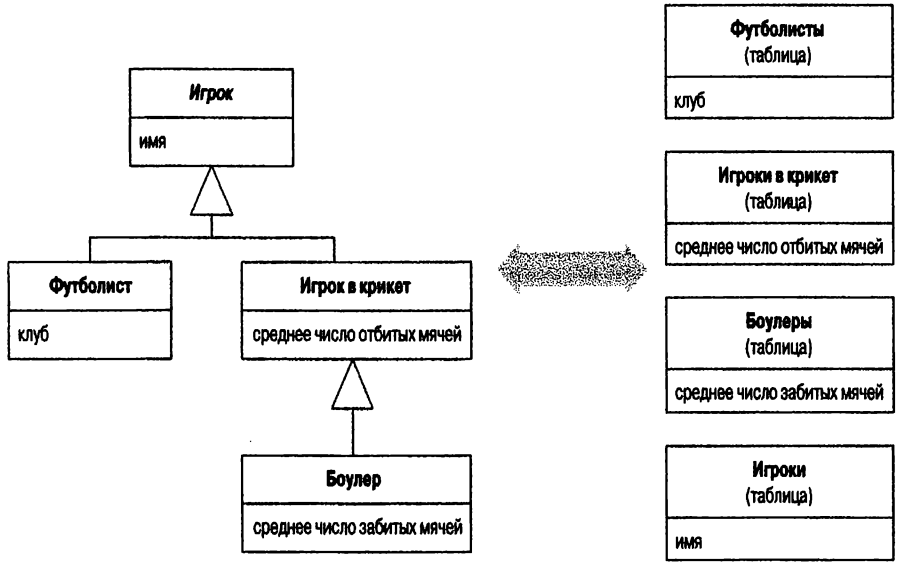


Рис. 3.10. Типовое решение **наследование с таблицами для каждого класса** *предусматривает использование отдельных таблиц для каждого класса иерархии*

таблицей, то самым большим недостатком этого решения является нерациональное расходование дискового пространства, поскольку каждая запись таблицы содержит поля для атрибутов всех созданных дочерних классов и многие из этих полей остаются пустыми.

(Впрочем, некоторые СУБД "умеют" осуществлять сжатие неиспользуемых областей.) Большой размер записи приводит и к замедлению ее загрузки. Преимущество же **наследования с одной таблицей** заключается в том, что все данные, относящиеся к любому классу, сосредоточены в одном месте, а это значительно упрощает возможность внесения изменений и позволяет избежать операций соединения.

Три упомянутых решения не являются взаимоисключающими — их вполне можно сочетать в рамках одной и той же иерархии классов: например, информацию о наиболее важных классах уместно объединить посредством **наследования с одной таблицей**, а для других классов воспользоваться решением **наследование с таблицами для каждого класса**. Разумеется, совмещение решений повышает сложность их применения.

Среди названных способов отображения иерархии наследования трудно выделить какой-либо один. Как и при использовании всех других типовых решений, необходимо принять во внимание конкретные обстоятельства и требования. Моим первым выбором был бы вариант **наследования с одной таблицей** как наиболее простой в реализации и устойчивый к многочисленным модификациям, а двумя другими я пользовался бы по мере необходимости, чтобы избавиться от неподходящих или заведомо лишних полей. Лучше всего, однако, побеседовать с администратором базы данных, знакомым со всеми ее нюансами и тонкостями, и прислушаться к советам, которые он вам даст.

Здесь и далее в примерах и типовых решениях подразумевается модель *единичного наследования (single inheritance)*. Сегодня парадигма *множественного наследования (multiple inheritance)* теряет популярность и во многих языках все чаще изымается из обихода. При использовании интерфейсов Java и .NET проблемы, сопутствующие применению инструментов множественного наследования, все еще о себе напоминают. В обсуждаемых здесь типовых решениях подобные аспекты специально не оговариваются. Впрочем, достаточно сказать, что поладить с отображением иерархий множественного наследования вам поможет "трио" решений, рассмотренных выше. **Наследование с одной таблицей** предполагает размещение атрибутов всех базовых классов и интерфейсов в одной большой таблице, при использовании **наследования с таблицами для каждого класса** создаются таблицы для каждого интерфейса и суперкласса, а реализация **наследования с таблицами для каждого конкретного класса** связана с включением данных обо всех базовых классах и интерфейсах в каждую таблицу конкретного класса.

Реализация отображения

Отображение объектов в реляционные структуры, по существу, сводится к одной из трех общих ситуаций:

- готовой схемы базы данных нет, и ее можно выбирать;
- приходится работать с существующей схемой, которую не разрешается изменять;
- схема задана, но возможность ее модификации оговаривается дополнительно.

В простейшем случае, когда схема создается самостоятельно, а бизнес-логика отличается умеренной сложностью, оправдан подход, основанный на **сценарии транзакции (Transaction Script, 133)** или **модуле таблицы (Table Module, 148)**; таблицы могут создаваться

с помощью традиционных инструментов проектирования баз данных. Для исключения кода SQL из бизнес-логики применяется **шлюз записи данных (Row Data Gateway, 175)** или **шлюз таблицы данных (Table Data Gateway, 167)**.

Используя **модель предметной области (Domain Model, 140)**, остерегайтесь структурировать приложение с оглядкой на схему базы данных и больше заботьтесь об упрощении бизнес-логики. Воспринимайте базу данных только как инструмент сохранения содержимого объектов. Наибольший уровень гибкости взаимного отображения объектов и реляционных структур обеспечивает **преобразователь данных (Data Mapper, 187)**, но это типовое решение отличается повышенной сложностью. Если структура базы данных изоморфна **модели предметной области**, можно воспользоваться **активной записью (Active Record, 182)**.

Хотя первоочередное построение модели представляет собой удобный способ восприятия предметной области, этот вариант действий правомерен только тогда, когда он реализуется в течение короткого итеративного цикла. Провести шесть месяцев за построением **модели предметной области**, не предусматривающей применения базы данных, а затем в один момент прийти к заключению о том, что сохранять кое-какую информацию все-таки было бы неплохо, не только несерьезно, но и весьма рискованно. Опасность заключается в том, что, если проект продемонстрирует изъяны на уровне производительности, исправить положение будет непросто. Каждую полуторамесячную или даже более краткую итерацию разработки модели и приложения следует сопровождать принятием решений, касающихся уточнения и расширения схемы базы данных. В этом случае у вас всегда будет самая свежая информация о том, как интерфейс взаимодействия приложения с базой данных ведет себя на практике. Итак, решая любую частную задачу, в первую очередь необходимо думать о **модели предметной области**, не забывая немедленно интегрировать каждую ее часть в базу данных.

Если схема базы данных создана загодя, в вашем распоряжении примерно те же альтернативы, но процесс несколько отличается. Если бизнес-логика проста, ее слой располагается "поверх" создаваемых классов **шлюза записи данных** или **шлюза таблицы данных**, имитирующих функции базы данных. При повышении уровня сложности бизнес-логики приходится прибегать к помощи **модели предметной области**, дополненной **преобразователем данных**, который должен обеспечить сохранение содержимого объектов приложения в существующей базе данных.

Двойное отображение

Время от времени встречаются ситуации, когда для получения схожих данных приходится взаимодействовать с несколькими источниками. Это могут быть базы данных с приблизительно одинаковым содержимым, но незначительно различающимися схемами (следует заметить: чем меньше и тоньше различия, тем сильнее головная боль), а также совершенно иные формы существования информации (например, комбинация XML-документов, CICS-транзакций и реляционных таблиц).

Наиболее простой вариант действий — создать по одному слою отображения для каждого источника данных. Однако, если информация схожа, она будет во многом дублироваться. В таком случае уместно рассмотреть схему двухэтапного отображения. На первом этапе структуры данных в памяти преобразуются в "усредненную" логическую схему хранения, нивелирующую различия в форматах представления данных во всех источниках. На втором этапе логическая схема отображается в физические с учетом их частных особенностей.

Дополнительный этап оправдывает себя только в том случае, когда источники данных отличаются заметной общностью. Отображение логической схемы хранения данных в физическую можно воспринимать в виде **шлюза (Gateway, 483)**, а для перехода от формы представления данных в памяти к логической схеме использовать любые приемы отображения.

Использование метаданных

Наличие в тексте приложения повторяющихся фрагментов кода — признак его плохо продуманной структуры. Общие функции можно вынести "за скобки", например средствами наследования и делегирования — одними из главных механизмов объектно-ориентированного программирования, но существует и более изощренный подход, связанный с **отображением метаданных (Metadata Mapping, 325)**.

Типовое решение **отображение метаданных** основано на вынесении функций отображения в файл метаданных, задающий соответствие между столбцами таблиц базы данных и полями объектов. Наличие метаданных позволяет не повторять код отображения и заново генерировать его по мере необходимости. Даже пустяковый фрагмент метаданных, подобный приведенному ниже, способен весьма выразительно сообщить большой объем информации.

```
<fieldName = "customer", targetClass = "Customer",
↳dbColumn = "custID", targetTable = "customers",
↳lowerBound = "1", upperBound = "1", setter = "loadCustomer"/>
```

С помощью этой информации вам удастся определить операции считывания и записи данных, автоматического формирования любых SQL-выражений, поддержки множественных связей и т.п. Вот почему модель метаданных находит широкое применение в коммерческих инструментальных средствах отображения объектов и реляционных структур.

Отображение метаданных предлагает все необходимое для конструирования запросов в терминах прикладных объектов. Типовое решение **объект запроса (Query Object, 335)** позволяет создавать запросы на основе объектов и данных в памяти, не требуя знаний SQL и деталей реляционной схемы, и применять инструменты **отображения метаданных** для трансляции таких выражений в соответствующие конструкции SQL.

Воспользуйтесь этими решениями на наиболее ранних стадиях проекта, и вам удастся применить одну из форм **хранилища (Repository, 341)**, полностью скрывающего базу данных от взгляда извне. Любые запросы к базе данных могут трактоваться как **объекты запроса** в контексте **хранилища**: в этой ситуации разработчику приложения неведомо, откуда извлекаются объекты — из памяти или из базы данных. Решение **хранилище** весьма удачно сочетается с богатыми **моделями предметной области (Domain Model, 140)**.

Соединение с базой данных

В большинстве случаев приложение взаимодействует с базой данных при посредничестве некоторого объекта *соединения (connection)*. Прежде чем выполнять команды и запросы к базе данных, соединение обычно необходимо *открыть*. Объект соединения должен пребывать в открытом состоянии на протяжении всего сеанса работы с базой данных. По завершении обслуживания запроса возвращается объект типа **множество записей (Record Set, 523)**. Некоторые интерфейсы позволяют манипулировать полученным **множеством записей** даже после *закрытия* соединения, а некоторые требуют наличия активного соединения. Если действия выполняются в рамках транзакции, последняя, как правило, ограничена контекстом определенного соединения, которое должно оставаться открытым, как минимум, до завершения транзакции.

Во многих средах открытие выделенного соединения сопряжено с расходованием строго ограниченных ресурсов, поэтому применение находят так называемые *пулы соединений (connection pools)*. В этом случае приложение не открывает и закрывает соединение, а запрашивает его из пула и освобождает, когда оно больше не требуется. Сегодня поддержка пула соединений обеспечивается большинством вычислительных платформ, поэтому потребность в самостоятельной реализации подобной структуры возникает редко. Если все-таки вам приходится этим заниматься, прежде всего выясните, действительно ли применение пула повышает производительность системы. Нередко среда способна обеспечить более быстрое создание нового соединения, нежели повторное использование соединения из пула.

Платформы, поддерживающие механизм пула соединений, часто скрывают последний посредством дополнительных интерфейсов, так что операции создания нового объекта соединения и выделения объекта из пула для разработчика прикладной системы выглядят совершенно идентично. Аналогично, закрытие соединения в реальности может означать возврат соединения в общий пул, доступный для других процессов. Подобная инкапсуляция, безусловно, является положительным решением. Ниже будут употребляться термины "открытие" и "закрытие", но имейте в виду, что в конкретной ситуации они могут означать соответственно "захват" соединения из пула и его "освобождение" с возвращением в пул.

Неважно, как создаются соединения, но ими необходимо надлежащим образом управлять. Поскольку, как уже отмечалось, речь в данном случае идет о дорогостоящих вычислительных ресурсах, соединения следует закрывать сразу по завершении их использования. Помимо того, если применяется аппарат транзакций, необходимо гарантировать, что каждая команда внутри определенной транзакции активизируется при посредничестве одного и того же объекта соединения.

Наиболее общая рекомендация такова: следует запросить объект соединения явно, используя вызов, обращенный к пулу или диспетчеру соединений, а затем ссылаться на этот объект в каждой команде, адресуемой к базе данных; по окончании использования объект необходимо закрыть. Но как поручиться, что ссылки на объект соединения присутствуют везде, где необходимо, и не забыть закрыть соединение по окончании работы?

Решение первой части проблемы состоит в передаче объекта соединения в любую команду в виде явно задаваемого параметра. К сожалению, при этом может оказаться, что объект соединения не будет передан единственному методу, расположенному в стеке

вызовов на несколько уровней ниже, а присутствует во всевозможных вызовах методов. В этой ситуации целесообразно вспомнить о **реестре (Registry, 495)**. Поскольку вам наверняка не хочется, чтобы соединением пользовались несколько вычислительных потоков, необходимо применять вариант **реестра** уровня одного потока.

Даже если вы не так забывчивы, как я, то и в этом случае согласитесь, что требование явного закрытия соединений довольно обременительно, поскольку его слишком легко оставить без внимания, что, разумеется, недопустимо. Соединение нельзя закрывать и после выполнения *каждой* команды, так как первая же попытка сделать подобное внутри транзакции приведет к ее откату.

Оперативная память, как и соединения, относится к числу ресурсов, которые надлежит возвращать системе сразу по завершении их использования. В современных средах разработки управление памятью и функции сборки мусора осуществляются автоматически. Поэтому один из способов закрытия соединения состоит в том, чтобы довериться сборщику мусора. При таком подходе соединение закрывается, если в ходе сборки мусора утилизируются объект соединения как таковой и все объекты, которые на него ссылаются. Удобно то, что здесь применяется та же хорошо знакомая схема управления памятью. Однако есть и проблема: соединение закрывается только тогда, когда сборщик мусора действительно утилизирует память, а это может произойти гораздо позже, чем исчезнет последняя ссылка, адресующая объект соединения, т.е. в ожидании закрытия он проведет немало времени. Возникнет подобная проблема или нет, во многом определяется особенностями среды разработки, которой вы пользуетесь.

Если рассуждать в более широком смысле, я не стал бы слишком полагаться на механизм сборки мусора. Другие схемы — даже с принудительным закрытием соединений — кажутся более надежными. Впрочем, сборку мусора можно считать своего рода страховочным вариантом: как говорится, лучше позже, чем никогда.

Поскольку соединения логически тяготеют к транзакциям, удобная стратегия управления ими состоит в трактовке соединения как неотъемлемого "атрибута" транзакции: оно открывается в начале транзакции и закрывается по завершении операций фиксации или отката. Транзакции известно, с каким соединением она взаимодействует, и потому вы можете сосредоточиться на транзакции, более не заботясь о соединении как таковом. Поскольку завершение транзакции обычно имеет более "видимый" эффект, чем завершение соединения, вы вряд ли забудете ее зафиксировать (а если и забудете, то, поверьте мне, быстро об этом вспомните). Один из вариантов совместного управления транзакциями и соединениями демонстрируется в типовом решении **единица работы (Unit of Work, 205)**.

Если речь идет о выполнении операций вне транзакций (например, о чтении заведомо неизменяемых данных), для каждой операции можно использовать "свежий" объект соединения. Совладать с любыми проблемами, касающимися использования объектов соединений с короткими периодами существования, поможет схема организации пула соединений.

При необходимости обработки данных в автономном режиме в контексте **множества записей** можно открыть соединение для загрузки информации в структуру данных, закрыть его и приступить к обработке. По завершении следует открыть новое соединение, активизировать транзакцию и сохранить результаты в базе данных. Если действовать по такой схеме, придется позаботиться о синхронизации данных с той информацией, которая изменялась другими транзакциями в период обработки содержимого

множества записей. Обсуждению подобных вопросов, имеющих отношение к проблематике управления параллельными заданиями, посвящена глава 5, "Управление параллельными заданиями".

Особенности процедур управления соединениями во многом обусловлены параметрами конкретных среды разработки и приложения.

Другие проблемы

В некоторых примерах кода, приведенных ниже, используются конструкции SQL-запросов вида `select * from`, а в отдельных случаях предложения `select` содержат списки выражений. Некоторые драйверы баз данных неспособны справиться с обработкой выражений `select *`, особенно при добавлении новых столбцов или перестановке существующих. Хотя для многих современных систем эти проблемы не характерны, было бы неразумно применять конструкции `select *`, если для доступа к столбцам используются их порядковые номера, так как любое переупорядочение столбцов приведет к разрушению кода. Если же столбцы адресуются по наименованиям, все нормально. Собственно говоря, такие запросы и легче воспринимать. Единственный недостаток подобного подхода заключается в некотором снижении производительности, что, впрочем, еще нужно установить опытным путем.

Если вы все-таки ссылаетесь на столбцы по номерам, удостоверьтесь, что эти обращения соответствуют SQL-определениям соответствующих таблиц. При работе со **шлюзом таблицы данных (Table Data Gateway, 167)** следует использовать наименования столбцов. Помимо того, удобно иметь простые процедуры для тестирования операций создания, чтения, обновления и удаления данных в контексте каждой из применяемых вами структур отображения. Это поможет выявлять случаи расхождения между кодом и реляционными схемами.

Всегда полезнее приложить усилия для создания статического предварительно откомпилированного SQL-кода, нежели пользоваться динамическими конструкциями SQL, которые всякий раз приходится компилировать заново. В большинстве СУБД те или иные средства предварительной компиляции обязательно предусмотрены. Еще одно правило: не пользуйтесь механизмом сцепления строк для объединения текста нескольких SQL-запросов.

Многие СУБД поддерживают пакетное выполнение команд SQL. В примерах эти средства не отображены, но в реальных проектах к ним непременно следует обращаться. Как именно — зависит от характеристик конкретной платформы.

В примерах объекты соединений заменены объектом "DB" типа **реестр (Registry, 495)**. Технология создания соединения во многом определяется спецификой СУБД и среды разработки, поэтому вносите в код те изменения, которые отвечают вашей ситуации. И еще. Ни в одном из типовых решений, кроме тех, которые относятся к сфере управления параллельными заданиями, не упоминалось о транзакциях. Поэтому приспосабливайте подходы и образцы кода к собственным условиям.

Дополнительные источники информации

С необходимостью отображения объектов в реляционные структуры сталкиваются все, кто так или иначе связан с программированием и базами данных, и потому неудивительно, что этому предмету посвящено море литературы. Удивительно как раз другое: ни одна из публикаций не истолковывает эту тему с современных позиций, цельно и полно, поэтому, собственно, я и уделил ей здесь так много внимания.

Хорошо то, что источник идей, откуда можно черпать вдохновение для собственного творчества, весьма широк и глубок. Среди тех, кого благодарные последователи обобрали до нитки, присутствуют [4, 10, 23, 41]. Настоятельно рекомендую проштудировать названные работы и вам — они станут хорошим дополнением к материалу этой книги.

Глава 4

Представление данных в Web

Одно из самых впечатляющих изменений, происшедших с корпоративными приложениями за последние несколько лет, связано с появлением пользовательских интерфейсов в стиле Web-обозревателей. Они принесли с собой ряд преимуществ: исключили потребность в применении специального клиентского программного слоя, обобщили и унифицировали процедуры доступа и упростили проблему конструирования Web-приложений.

Разработка Web-приложения начинается с настройки программного обеспечения Web-сервера, что обычно сводится к созданию некоторого файла настроек, определяющего, какие *адреса URL (Uniform Resource Locators — URLs)* обслуживаются теми или иными программами. Нередко сервер способен управлять множеством программ, которое можно динамически пополнять, размещая программы в подходящих каталогах. Функции Web-сервера состоят в интерпретации адреса URL запроса и передаче управления соответствующей программе. Существует две основные формы представления программы Web-сервера — *сценарий (script)* и *страница сервера (server page)*.

Сценарий состоит из функций или методов, предназначенных для обработки запросов HTTP. Типичными примерами могут служить сценарии CGI и сервлеты Java. Подобная программа способна выполнять практически все то же, что и традиционное приложение. Сценарий часто разбивается на подпрограммы и пользуется сторонними службами. Он получает данные с Web-страницы, проверяя строковый объект HTTP-запроса и вычленяя из него регулярные выражения; простота реализации подобных функций с помощью языка Perl снискали последнему славу одного из наиболее адекватных средств разработки сценариев CGI. В иных случаях, например при использовании сервлетов Java, программист получает доступ к информации запроса через интерфейс ключевых слов, что нередко значительно удобнее. Результатом работы Web-сервера служит другая — ответная — строка, образуемая сценарием с привлечением обычных функций поточного вывода.

Задача формирования кода HTML посредством команд поточного вывода не очень привлекательна для программистов, а непрограммистам она вообще не по силам, хотя они с удовольствием взялись бы за Web-дизайн с помощью других инструментов. Это естественным образом подводит к модели страниц сервера, где функции программы сводятся к возврату порции текстовых данных. Страница содержит текст HTML с "вкраплениями" исполняемого кода. Подобный подход, реализуемый, например, в PHP, ASP и JSP, особенно удобен, если требуется незначительная дополнительная обработка текста с учетом реакции пользователя.

Поскольку модель сценариев лучше подходит для интерпретации запросов, а схема страниц сервера — для форматирования ответов, вполне разумно применять их совместно. На самом деле это довольно старая идея, впервые реализованная в пользовательских интерфейсах на основе типового решения **модель–представление–контроллер (Model View Controller, 347)**, пример которого представлен на рис. 4.1.

Решение находит широкое применение, но зачастую трактуется неверно (это особенно характерно для приложений, написанных до появления Web). Основная причина состоит в неоднозначном толковании термина "контроллер". Он употребляется во многих контекстах, и ему придается самый разный смысл, иногда совершенно противоречащий тому, который заключен в решении **модель–представление–контроллер**. Вот почему, говоря об этом решении, я предпочитаю использовать словосочетание *входной контроллер (input controller)*.

Входной контроллер принимает запрос и извлекает из него информацию. Затем он передает бизнес-логику надлежащему объекту модели, который обращается к источнику данных и выполняет действия, предусмотренные в запросе, включая сбор информации, необходимой для ответа. По завершении функций он передает управление входному контроллеру, который, анализируя полученный результат, принимает решение о выборе варианта представления ответа. Управление и соответствующие данные передаются представлению. Взаимодействие входного контроллера и представления зачастую осуществляется не в виде прямых вызовов, а при посредничестве некоторого объекта HTTP-сеанса, который служит для передачи данных в обоих направлениях.

Основной довод в пользу применения решения **модель–представление–контроллер** состоит в том, что оно предусматривает полное отмежевание модели от Web-представления. Это упрощает возможности модификации существующих и добавления новых представлений. А размещение логики в отдельных объектах **сценария транзакции (Transaction Script, 133)** и **модели предметной области (Domain Model, 140)** облегчает их тестирование. Это особенно важно, когда в качестве представления используется страница сервера.

Здесь наступает черед практического применения второго варианта толкования термина "контроллер". Во многих версиях пользовательского интерфейса объекты представления отделяются от объектов домена промежуточным слоем объектов **контроллера приложения (Application Controller, 397)**, назначением которого является управление потоком функций приложения и выбор порядка демонстрации интерфейсных экранов. **Контроллер приложения** выглядит как часть слоя представления либо как самостоятельная "прослойка" между уровнями представления и предметной области. **Контроллеры приложения** могут быть реализованы независимо от какого бы то ни было частного представления, и тогда их удается использовать повторно для различных представлений. Такая схема приемлема в случаях, когда различные представления снабжены одинаковой логикой и инструментами навигации, но особенно хороша, если представления обладают разной логикой.

Контроллеры приложения нужны далеко не всем системам. Они удобны, когда приложение отличается богатством логики, касающейся порядка воспроизведения экранов интерфейса и навигации между ними, либо отсутствием простой зависимости между страницами и сущностями предметной области. Когда же порядок следования страниц несуществен, необходимости в использовании **контроллеров приложения** практически не возникает. Вот простой тест, позволяющий определить, целесообразно ли применять **контроллеры приложения**: если потоком функций системы управляет машина, ответ положителен, а если пользователь — отрицателен.

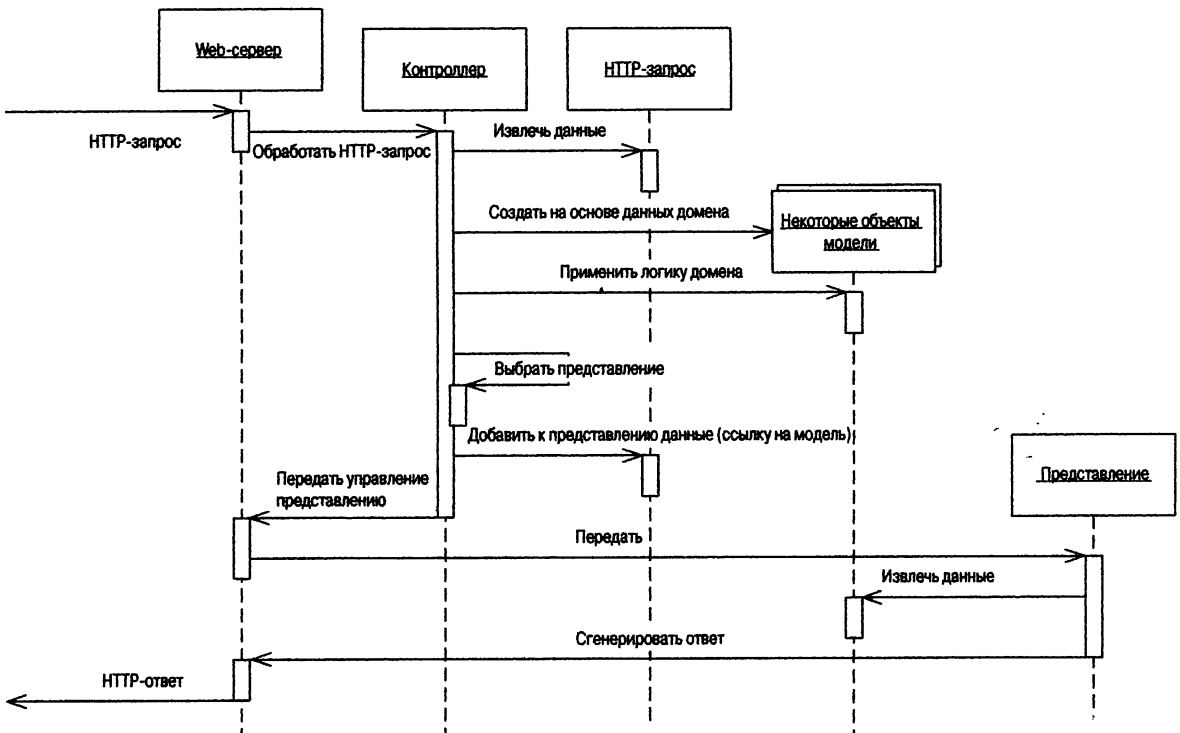


Рис. 4.1. Укрупненная диаграмма последовательностей, иллюстрирующая взаимодействие модели, представления и входного контроллера в структуре Web-сервера: контроллер обрабатывает запрос, инициирует реализацию бизнес-логики и передает управление представлению для формирования ответа, основанного на модели

Типовые решения представлений

Когда речь заходит о возможных способах реализации представлений, внимания заслуживают три основных типовых решения: **представление с преобразованием (Transform View, 379)**, **представление по шаблону (Template View, 368)** и **двухэтапное представление (Two Step View, 383)**. По существу, выбор сводится к одному из двух вариантов — воспользоваться **представлением с преобразованием** или **представлением по шаблону** в базовой одноэтапной версии либо усложнить любое из них до уровня **двухэтапного представления**.

Я обычно начинаю с выбора между **представлением по шаблону** и **представлением с преобразованием**. Первое позволяет оформлять представление в соответствии со структурой страницы и вставлять в нее специальные маркеры, отмечающие позиции фрагментов динамического содержимого. **Представление по шаблону** поддерживается целым рядом популярных платформ, многие из которых основаны на модели страниц сервера (скажем, ASP, JSP и PHP) и позволяют внедрять в текст страницы код на полнофункциональном языке программирования. Такое решение отличается мощностью и гибкостью; если код сложен и запутан, задача сопровождения системы чрезвычайно затрудняется. Поэтому использование технологий страниц сервера требует аккуратности и последовательности в обособлении логики кода от структуры страницы, которое зачастую достигается при посредничестве *вспомогательного (helper)* объекта.

Примером реализации **представления с преобразованием** может служить XSLT. Эта технология оказывается весьма эффективной, если данные домена сохраняются в формате XML или допускают быстрое преобразование в XML. Входной контроллер выбирает подходящую таблицу стилей XSLT и применяет ее к XML-коду, описывающему модель.

Если представление задается с помощью процедурных сценариев, для написания кода можно пользоваться **представлением с преобразованием**, **представлением по шаблону** или "смесью" двух подходов в любой подходящей пропорции.

Далее принимается решение о том, использовать одноэтапную версию представления (рис. 4.2) или прибегнуть к соответствующей форме **двухэтапного представления**. Одноэтапный вариант предусматривает преимущественно по одному компоненту представления для каждого интерфейсного экрана приложения. Код представления получает данные домена и преобразует их в формат HTML. Я говорю "преимущественно", поскольку схожие "логические экраны" (logical screens) могут делить представления между собой. Впрочем, в большинстве ситуаций уместно полагать, что каждое представление относится к одному экрану.

Двухэтапное представление (рис. 4.3) разделяет процесс на две стадии: на первой на основе данных домена формируется логический экран, который на второй стадии трансформируется в код HTML. Для каждого экрана существует одно представление первого этапа, но для приложения в целом — только одно представление второго этапа.

Достоинство **двухэтапного представления** заключается в том, что решение о варианте преобразования в HTML принимается в одном месте. Это существенно облегчает задачу внесения глобальных изменений, поскольку для модификации каждого экрана достаточно отредактировать данные единственного объекта. Разумеется, воспользоваться таким преимуществом удастся только в том случае, когда логическое представление остается постоянным, т.е. различные экраны компонуются по одному принципу. Сайты, спроектированные по излишне сложным схемам, единообразием логической структуры обычно не отличаются.

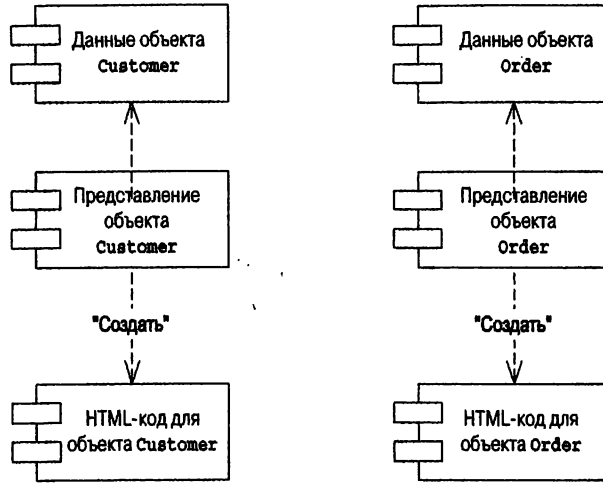


Рис. 4.2. Пример одноэтапного представления

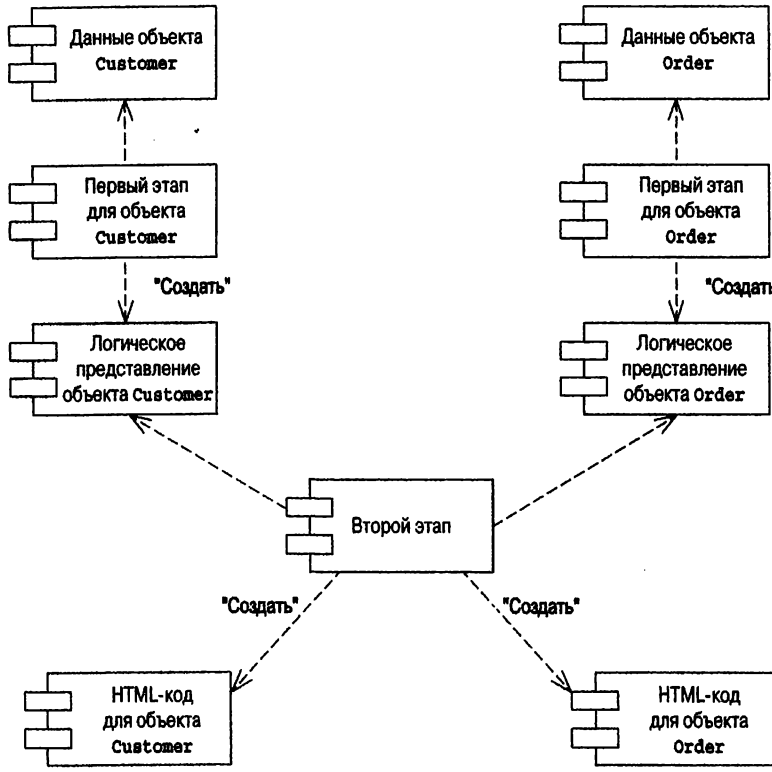


Рис. 4.3. Пример **двухэтапного представления**

Типовое решение **двухэтапное представление** хорошо проявляет себя в ситуациях, где службы Web-приложения используются многочисленными клиентами (например, посетителями сайта системы бронирования авиабилетов). Удовлетворяя требованиям компоновки одного логического экрана, каждая из версий клиентского приложения, отвечающая определенному варианту реализации второго этапа представления, может иметь другой внешний вид. Таким же образом **двухэтапное представление** может быть использовано и для обслуживания разных устройств вывода, когда необходимо предусмотреть отдельные реализации второго этапа представления, скажем, для обычного Web-обозревателя и карманного компьютера. К сожалению, наличие общего логического экрана может сыграть отрицательную роль в случае, когда речь идет о двух существенно отличающихся пользовательских интерфейсах (например, об интерфейсе Web-обозревателя и сотового телефона).

Типовые решения входных контроллеров

Существует два типовых решения проблемы организации входных контроллеров. Наиболее общий подход состоит в создании объекта входного контроллера для каждой страницы Web-сайта. В простейшем случае подобный **контроллер страниц (Page Controller, 350)** можно оформить в виде страницы сервера, сочетая в нем функции представления и входного контроллера. Во многих ситуациях, однако, легче выделить входной контроллер в самостоятельный объект. Нередко взаимно однозначного соответствия между **контроллерами страниц** и представлениями не существует. Точнее говоря, следует иметь **контроллер страниц** для каждого действия, где действием является кнопка или гиперссылка. В большинстве случаев действия соответствуют страницам, но бывает и так, что ссылка, например, указывает на разные страницы в зависимости от определенного условия.

На входной контроллер возлагаются две основные обязанности: обработка HTTP-запроса и принятие решения о том, что с ним делать дальше, которые зачастую имеет смысл разделить, поручив первую функцию странице сервера, а вторую — вспомогательному объекту. В то же время типовое решение **контроллер запросов (Front Controller, 362)** предусматривает использование единственного объекта, предназначенного для обработки *всех* запросов. Обработчик интерпретирует полученный адрес URL, определяет, с какого рода запросом он имеет дело, и создает отдельный объект для дальнейшего обслуживания запроса. Таким образом удается централизовать деятельность по обработке всех HTTP-запросов в рамках единого объекта и избежать необходимости изменения конфигурации Web-сервера в случае модификации структуры действий сайта.

Дополнительные источники информации

В большинстве книг по Web-технологиям найдется пара глав, посвященных удачным образцам организации Web-серверов (подобная информация, однако, часто изобилует ненужными деталями). Прекрасный пример Java-проекта обсуждается в главе 9 руководства [9]. Лучший источник информации о других типовых решениях — книга [3]; многие из них не привязаны к Java и носят универсальный характер. Терминология, касающаяся разделения функций входного контроллера и контроллера приложения, заимствована из [25].

Глава 5

Управление параллельными заданиями

Мартин Фаулер и Дейвид Райс

Параллельное (concurrent) выполнение операций — одна из наиболее сложных дисциплин в области разработки программного обеспечения. Проблемы параллелизма возникают всякий раз, когда, скажем, несколько процессов или потоков вычислений предпринимают попытки манипуляций одними и теми же элементами данных. Восприятие множества параллельных операций затруднено, поскольку перечислить все возможные сценарии развития событий, способные привести к тем или иным неприятностям, крайне сложно. Что бы вы ни делали, всегда кажется, что какие-то вещи упущены. Более того, параллельные операции трудно тестировать. Всем нам нравятся средства автоматического тестирования, сопровождающие процессы разработки программного обеспечения от начала и до конца, но найти тесты, которые смогли бы убедить в достаточной надежности параллельного кода, практически невозможно.

Забавно и парадоксально, что с возрастанием степени параллелизма операций в корпоративных приложениях разработчики все меньше заботятся о сопутствующих проблемах. Причиной подобного легкомысленного отношения служит наличие готовых подсистем — диспетчеров транзакций. Модель транзакций позволяет избежать массы трудностей: если вы манипулируете данными внутри транзакции, будьте уверены, что ничего плохого с ними не случится.

Впрочем, это не значит, что проблемами управления параллельными заданиями можно полностью пренебречь, так как многие аспекты взаимодействия приложения и системы нельзя свести к контексту единой транзакции, обращенной к базе данных, — во многих случаях приходится иметь дело с данными, модифицируемыми несколькими транзакциями. Последняя задача получила название *автономного параллелизма (offline concurrency)*.

Еще одна ситуация, в которой феномен параллелизма проявляет свою угрожающую сущность, связана с серверами приложений, поддерживающими множество одновременно протекающих потоков вычислений. Впрочем, автору прикладной программы беспокоиться не о чем — все обязанности принимает на себя серверная платформа.

Чтобы разобраться в проблемах, надлежит освоиться по меньшей мере с некоторыми базовыми понятиями; с этого и начнем. Однако не стоит трактовать эту главу как сколько-нибудь полное введение в технологии управления параллельными операциями, так как для достижения подобной цели потребовалась бы, как минимум, толстенная книга. Вы познакомитесь с аспектами параллелизма, имеющими отношение к корпоративным программным приложениям. Затем на ваш суд будут представлены типовые решения в области управления параллельными заданиями в автономном режиме и некоторые подходы к обеспечению многопоточного функционирования серверов приложений.

На протяжении всей главы для иллюстрации концепций параллелизма приводятся примеры из области, которая, вероятно, вам хорошо знакома: речь идет о системах контроля версий исходного кода, которые применяются командами разработчиков для координации вносимых изменений. (Между прочим, если вы не осведомлены о подобных системах, то не сможете плодотворно работать над корпоративными приложениями.)

Проблемы параллелизма

Обозначим некоторые принципиальные проблемы обеспечения параллельной работы программных приложений. На преодоление именно этих проблем направлены усилия систем управления параллельными заданиями. Впрочем, трудности связаны не только с параллелизмом — управляющие системы, разрешая одно, часто усугубляют другое!

Самый простой для восприятия пример — *утраченные изменения (lost updates)*. Предположим, что некий Мартин открывает для редактирования файл с исходным кодом программы, намереваясь за пару минут подправить текст метода `checkConcurrency`. В то же самое время ни о чем не подозревающий Дейвид обращается к тому же файлу, чтобы модифицировать метод `updateImportantParameter`. Дейвид справляется со своим заданием очень быстро — настолько быстро, что, приступив к работе позже Мартина, успевает завершить ее раньше... к несчастью. В момент открытия Мартином файл еще не содержал изменений, внесенных Дейвидом, поэтому, сохраняя результат, Мартин запишет поверх содержимого файла свою копию данных, и редакция Дейвида будет безвозвратно потеряна.

А эффект *несогласованного чтения (inconsistent read)* возникает в ситуациях, когда считываются две корректные сами по себе порции данных, которые, однако, не могут существовать в один и тот же момент времени. Допустим, тому же вездесущему Мартину захотелось узнать, сколько классов содержит пакет поддержки параллельной работы, который состоит из двух вложенных пакетов, реализующих протоколы блокирования и протоколирования действий транзакций. Мартин заглядывает в пакет блокирования и видит в нем семь классов. В этот момент звонит телефон, и приятель, которому просто нечего делать, зовет попить пивка. Пока Мартин разбирается с ним, неутомимый Дейвид, наконец, устраняет досадную ошибку в коде метода четырехфазного блокирования и вводит два новых класса в пакет блокирования и три, в добавление к имеющимся пяти, — в пакет протоколирования. Чувство долга берет верх над жаждой, и Мартин, положив трубку и возвратившись к начатому делу, обнаруживает в пакете протоколирования восемь классов и приходит к глубокомысленному выводу о том, что общее количество классов равно пятнадцати.

Конечно, 15 — это неверный ответ. Правильным был бы такой: 12 до внесения изменений Дейвидом и 17 — после. Любой из них корректен, хотя в какой-то момент времени, возможно, неактуален, но число 15, полученное на основе несогласованных выводов, не отвечает действительности ни при каких обстоятельствах.

Обе проблемы выражаются в потере *достоверности (correctness)* информации и в некорректном поведении системы, чего можно было бы избежать, если бы два человека не обращались к одним и тем же данным одновременно. Впрочем, если бы речь шла только о недостоверности, проблемы не были бы столь серьезными. Собственно говоря, можно упорядочить действия таким образом, чтобы только один субъект имел право обращаться к порции данных в определенный момент времени. Это поможет сберечь достоверность информации, но снизит возможности параллельного выполнения операций по ее обработке. Основополагающая проблема любой модели программирования параллельных операций заключается не только в сохранении корректности данных, но и в обеспечении максимальной степени параллелизма (*живучести (liveness)* системы). Зачастую приходится поступаться корректностью в угоду параллельности; величина такой жертвы определяется, с одной стороны, серьезностью нештатных ситуаций, возникающих из-за возможной недостоверности или неактуальности данных, а с другой — действительным уровнем потребностей в выполнении операций в параллельном режиме.

Это далеко не все проблемы, которые могут подстерегать разработчиков систем параллельной обработки данных, но, как можно полагать, главные. Для их разрешения используются различные управляющие механизмы. Увы, за все приходится платить. Нередко процесс преодоления одних проблем способствует появлению других — хотя и менее серьезных, но неизбежных. Отсюда следует один важный вывод: если проблемы, связанные с параллельной обработкой данных, "терпимы", постарайтесь отказаться от идеи использования управляющего механизма.

Контексты выполнения

Все операции, выполняемые системой, протекают в определенном *контексте*, причем зачастую более чем в одном. Общепринятой терминологии для обозначения контекстов выполнения не существует, и потому ниже введены определения, которые будут использоваться в дальнейшем.

В аспекте взаимодействия программной системы с внешним миром можно выделить два важных контекста — запрос и сеанс. *Запрос (request)* соответствует отдельно взятому обращению к системе со стороны внешнего субъекта-клиента. Обработка запроса является прерогативой сервера; обычно подразумевается, что клиент, инициировавший запрос, ожидает ответа на него. При использовании некоторых протоколов клиенту разрешается прерывать запрос до получения ответа, но такие ситуации встречаются сравнительно редко. Чаще клиент имеет возможность послать другой запрос, противоречащий исходному (скажем, отправить запрос с заказом, а затем отменить заказ). По мнению клиента, связь двух запросов вполне очевидна, но конкретный протокол может и не донести эту информацию до сервера.

Сеанс (session) — это долговременный процесс взаимодействия клиента и сервера. В частном случае сеанс может включать только один запрос, но более характерна ситуация, когда он охватывает серию запросов, посылаемых пользователем в логической

последовательности. Обычно сеанс начинается с подключения клиента к системе и служит средой выполнения некоторого множества действий, включая отправку запросов к базе данных и осуществление одной или нескольких бизнес-транзакций (о них речь идет ниже). В завершение сеанса пользователь отключается от системы явно либо просто закрывает приложение, полагая, что система отреагирует на это надлежащим образом.

Программное обеспечение сервера корпоративных приложений может выступать в двух ипостасях — как сервер для клиента приложения и как клиент других систем. Поэтому нужно иметь в виду и возможность одновременного протекания нескольких сеансов с участием сервера (например, HTTP-сеанса с клиентом и сеанса взаимодействия с СУБД).

Рассмотрим другую пару понятий, происходящих из предметной области операционных систем, — "процесс" и "поток вычислений". *Процесс (process)* — это всеобъемлющий контекст выполнения, обеспечивающий высокий уровень изоляции охватываемых им данных от внешнего мира. *Поток вычислений (thread)* — более "легковесный" активный агент; в контексте одного процесса может функционировать целое множество потоков. Модель многопоточного программирования находит самое широкое применение, поскольку обеспечивает высокий уровень использования вычислительных ресурсов благодаря возможности формирования многочисленных запросов в русле единого процесса. Однако потоки, как правило, имеют доступ к одним и тем же массивам оперативной памяти, что приводит к естественным проблемам. Некоторые среды позволяют управлять доступом к ресурсам и создавать *изолированные потоки (isolated threads)*, которые, в частности, способны обращаться к собственным областям памяти.

Одна из трудностей восприятия контекстов выполнения состоит в том, что они в действительности не упорядочены в такой степени, как того хотелось бы. В теории каждый сеанс должен быть связан исключительно с одним процессом на протяжении всего времени его протекания. Поскольку процессы в достаточной мере изолированы друг от друга, это могло бы снизить опасность возникновения конфликтов из-за параллелизма. На сегодня, однако, не известна ни одна серверная платформа, которая функционирует подобным образом. Ближайшая альтернатива связана с активизацией нового процесса для обработки каждого поступившего запроса; именно такая схема использовалась в ранних Web-системах, основанных на сценариях Perl. Сейчас ее стараются избегать, так как старт процесса сопряжен с расходованием избыточных ресурсов. Но обычной практикой является обработка процессом только одного запроса в каждый момент времени; это позволяет избавиться от многих проблем, обусловленных параллельным функционированием нескольких потоков.

При работе с системой баз данных следует различать еще один важный контекст выполнения — контекст *транзакции (transaction)*. Транзакции способны соединять в себе несколько запросов, которые клиенту хотелось бы трактовать как единый запрос. Команды, составляющие транзакцию, могут быть адресованы приложением к СУБД (системные транзакции) или пользователем к приложению (бизнес-транзакции). Эти термины обсуждаются ниже.

Изолированность и устойчивость данных

Проблемы параллельного выполнения программ известны уже давно, и за это время предложено немало вариантов их решения. Для корпоративных приложений особенно важны два решения: поддержка *изолированности (isolation)* и обеспечение *устойчивости (immutability)* данных.

Подобные проблемы возникают в ситуациях, когда несколько активных агентов, таких, как процессы или потоки вычислений, обращаются к одной и той же порции информации. Один из вариантов разрешения возможных конфликтов состоит в изоляции данных таким образом, чтобы к любому их элементу мог адресоваться только один агент. Процессы прикладной программы протекают так же, как их аналоги уровня операционной системы: процесс получает в свое безраздельное владение участок оперативной памяти, и только этот процесс способен считывать и сохранять ассоциированные с ним данные. Схожим образом действуют и схемы блокирования файлов, применяемые во многих популярных приложениях. Если файл открыт Мартином, в этот период никто другой открыть его уже не сможет; в крайнем случае система разрешит обратиться к файлу в режиме "только для чтения" и просмотреть ту версию данных, которая соответствовала началу сеанса работы Мартина. На протяжении всего сеанса никто не сможет изменить содержимое файла и увидеть какую бы то ни было промежуточную информацию, сохраняемую Мартином.

Поддержка *изолированности* — весьма важный технологический прием, позволяющий снизить вероятность возникновения ошибок. Слишком часто разработчики сами загоняют себя в угол, используя инструментальные средства управления, которые вынуждают неотрывно держать руку на пульсе событий. Заставляя программу функционировать в некой изолированной зоне, мы избавляемся от такой необходимости. Таким образом, добротное проектирование приложения предполагает успешный поиск таких зон и перенос возможно большей части кода в их контекст.

Проблемы, связанные с параллельным доступом к данным, возникают только тогда, когда общие фрагменты таких данных подвержены изменениям. Один из естественных способов предотвращения потенциальных конфликтов заключается в обнаружении *устойчивых (immutable)* элементов информации. Совершенно очевидно, что придать статус устойчивости *всем* данным не удастся, поскольку основное назначение многих систем как раз и состоит в обеспечении средств модификации информации. Но если определить *некоторые* порции данных как устойчивые или, по меньшей мере, устойчивые в продолжение некоторых периодов, можно ослабить ограничения параллельного доступа. Еще одна альтернатива — отделить приложения, функционирующие в режиме "только для чтения", от всех остальных и заставить их работать с копиями источников данных, что значительно упростит логику управления доступом.

Стратегии блокирования

Что может произойти при наличии определенных данных, которые подвержены изменениям и не допускают возможности изоляции? Если говорить в самом широком

смысле, существует два вида стратегий управления параллельными заданиями — *оптимистические (optimistic)* и *пессимистические (pessimistic)*.

Предположим, что Мартину и Дейвиду позарез нужно редактировать файл `Customer` одновременно. Используя схему *оптимистического блокирования (optimistic locking)*, тот и другой могут получить в свое распоряжение копии файла и свободно их править. Завершая работу первым, Дейвид без проблем сохраняет изменения в основном файле. Если в то же самое время записать данные в файл пытается и Мартин, система контроля версий, обнаружив конфликт, должна запретить эту операцию и позволить Мартину принять осмысленное решение по выходу из сложившейся ситуации. При реализации стратегии *пессимистического блокирования (pessimistic locking)* первый пользователь, захвативший файл, препятствует открытию файла всеми другими пользователями. Если в таком случае более проворным окажется Мартин, Дейвид не сможет работать с файлом до тех пор, пока тот не будет освобожден.

Стратегии удобно воспринимать следующим образом: оптимистическое блокирование дает возможность *обнаруживать* конфликты, в то время как пессимистическое позволяет их *предотвращать*. В реальных системах контроля версий исходного программного кода применяются обе стратегии, хотя сегодня большинству разработчиков по нраву модель оптимистического блокирования. (Некоторые довольно резонно утверждают, что оптимистическое блокирование не является "блокированием" в привычном смысле слова, но, на наш взгляд, этот термин слишком удобен и широко распространен, чтобы его можно было легко отвергнуть.)

Обоим подходам присущи как достоинства, так и недостатки. Изъян схемы пессимистического блокирования состоит в снижении степени параллелизма операций. Когда Мартин работает с заблокированным файлом, всем остальным желающим (в частности, Дейвиду) приходится ждать своей очереди. Если вам приходилось пользоваться системами контроля версий, основанными на модели пессимистического блокирования, вы согласитесь, что неопределенно долгое ожидание способно порой довести до бешенства. В ситуации с корпоративными данными проблема еще более обостряется, поскольку редактируемые порции информации не позволяется даже считывать; что же говорить тогда о возможности их совместного параллельного изменения!

Стратегия оптимистического блокирования предоставляет гораздо больше свободы, так как блокировка удерживается только в течение периода фиксации изменений. Проблемы появляются при возникновении конфликтов версий данных. Каждый, кто обращается к файлу после внесения изменений Дейвидом, должен проверить оставленную им версию файла, определить, как осуществить слияние собственных результатов с редакцией Дейвида, и зарегистрировать новую версию. Если речь идет об исходном программном коде, все это не особенно трудно: во многих случаях системы контроля версий способны осуществить слияние автоматически, а если по каким-либо причинам это невозможно, они предлагают удобные средства поиска и воспроизведения внесенных исправлений. Но выполнить слияние нескольких версий одной и той же порции бизнес-данных намного сложнее, поэтому зачастую проще поступиться затратами времени и сил и все начать сначала.

Главное, *что* следует учитывать, осуществляя выбор между оптимистическим и пессимистическим блокированием, — это частота возникновения и степень опасности конфликтов. Если конфликты относительно редки или их последствия незначительны, обычно разумно предпочесть оптимистическую стратегию, поскольку она обеспечивает

высокий уровень параллелизма операций и более проста в реализации. Если же конфликты грозят головной болью, уместнее избрать технологию пессимистического блокирования.

Ни один из подходов не свободен от недостатков, серьезных и не очень. Более того, используя любую стратегию блокирования, вы можете легко спровоцировать новые проблемы, ничуть не проще тех, с которыми вы пытались совладать. Оставим обсуждение всех тонкостей предмета авторам соответствующих специализированных изданий, а здесь отметим лишь отдельные моменты.

Предотвращение возможности несогласованного чтения данных

Рассмотрим следующую ситуацию. Мартин редактирует текст класса `Customer`, добавляя некоторые вызовы методов класса `Order`. Тем временем Дейвид исправляет интерфейс класса `Order`. Дейвид компилирует код и фиксирует его новую версию; Мартин, как ни странно, делает то же самое. Но общая часть кода теперь неверна, поскольку Мартину неизвестно, что класс `Order` подвергся негласной модификации. Одни системы контроля версий способны распознать подобные попытки *несогласованного чтения (inconsistent read)*, в то время как другие для достижения согласованности требуют вмешательства извне.

Проблемой несогласованного чтения нередко пренебрегают просто потому, что повышенное внимание уделяют ситуациям, чреватым утратой внесенных изменений. В рамках технологий пессимистического блокирования разработаны удачные варианты решения проблемы за счет использования специфических режимов блокирования, сопровождающих операции чтения и записи. В первом случае применяют *общие (shared)* блокировки, а во втором — *монопольные (exclusive)*. Допускается одновременный захват многих общих блокировок одной и той же порции данных, причем, если хотя бы одна общая блокировка активизирована, монопольную блокировку запросить не удастся. Если же кто-то захватил монопольную блокировку, все попытки получения любых блокировок тех же данных будут отвергнуты. Подобная схема пессимистического блокирования полностью устраняет опасность несогласованного чтения.

Механизмы выявления конфликтов, используемые в стратегиях оптимистического блокирования, обычно основаны на неких схемах маркирования данных. В качестве маркеров используются те или иные хронологические признаки или счетчики. Для обнаружения утраченных изменений система сравнивает маркер версии измененных данных с маркером версии общих исходных данных. Если маркеры совпадают, изменение вступает в силу.

Выявление фактов несогласованного чтения осуществляется по тому же принципу: каждая часть считанных данных нуждается в маркере, который сравнивается с маркером оригинала. Любое несоответствие указывает на наличие конфликта.

Контроль за считыванием информации часто сопряжен с ненужными проблемами из-за разногласий на почве обращения к данным, которые в действительности того не заслуживают. Чтобы уменьшить тяжесть подобного бремени, необходимо четко размежевать данные по степени их важности. Трудность заключается в том, что не всегда ясно, каково истинное назначение тех или иных элементов информации. Почтовый код в адресе может выглядеть как излишество, но если представить, что при расчете налогов учитывается местожительство, за доступом к адресной информации придется следить так же

тщательно, как и за обращениями к более "важным" данным. Определение того, *что* достойно внимания и *что* нет, — задача весьма серьезная, не зависящая от разновидности применяемой системы управления параллельными заданиями.

Другой способ преодоления проблемы несогласованного чтения данных состоит в использовании типового решения **операции чтения с временными признаками (Temporal Reads)**, предполагающего, что при каждом чтении данные обозначаются неким хронологическим признаком или неизменяемой меткой, а СУБД возвращает данные в той редакции, которая соответствует определенному признаку или метке. Подобный механизм поддерживается только несколькими СУБД, но в системах контроля версий исходного программного кода он находит более широкое применение. Основное препятствие связано с необходимостью сохранения полной истории изменений, что требует дополнительных затрат времени и дискового пространства. Поэтому такое решение более приемлемо для систем контроля версий и менее — для систем баз данных. Впрочем, оно может оказаться весьма уместным при реализации приложений для близкой вам предметной области. За подробной информацией обращайтесь к работам [16, 37].

Разрешение взаимоблокировок

Одна из частных, но весьма важных и трудных проблем, сопутствующих применению стратегий пессимистического блокирования, связана с возникновением *взаимоблокировок (deadlocks)*. Предположим, что Мартин приступает к редактированию файла `Customer`, а Дейвид берется за правку файла `Order`. В какой-то момент Дейвид осознает, что для завершения работы ему необходимо несколько изменить и файл `Customer`, но Мартин владеет блокировкой этого файла, и Дейвиду приходится ждать ее освобождения. В то же время Мартин приходит к мысли о том, что неплохо было бы немного подкорректировать файл `Order`, но этому мешает блокировка, удерживаемая Дейвидом. Наши герои попадают в ситуацию взаимоблокировки: ни один не может продвинуться дальше до тех пор, пока его "соперник" не завершит свою часть работы. В случаях, подобных рассмотренному, проблема выглядит не так уж угрожающе и ее несложно предотвратить, но если цепочку взаимозависимостей составляет множество людей или программ, тогда, к сожалению, "вечер перестает быть томным".

Существует целый ряд методов разрешения взаимоблокировок. Методы одной группы предусматривают выявление ситуаций взаимоблокировки по мере их возникновения. В подобных случаях выбирается *процесс-жертва (victim)*, который вынужден прервать свою деятельность и освободить все занятые им блокировки, чтобы дать возможность остальным участникам конфликта продолжить работу. Задача обнаружения взаимоблокировок в самой общей постановке весьма сложна; к тому же участи "жертв" не позавидуешь. Другой подход предполагает задание для каждой блокировки определенного лимита времени. Если процесс исчерпал свой лимит, но функции до конца так и не выполнил, он принудительно прерывается с потерей блокировок и всех достигнутых результатов и, по существу, становится той же жертвой. Контроль лимитов реализовать гораздо проще, нежели механизм выявления взаимоблокировок, но, если, скажем, один процесс, обращающийся к многим данным и ресурсам, обладает существенно большим лимитом в сравнении с остальными, многие из них совершенно необоснованно будут принесены в жертву даже при отсутствии реальной опасности взаимоблокировки.

Средства контроля лимитов времени и обнаружения взаимоблокировок призваны разрешать трудные ситуации *после* их возникновения, но есть и такие инструменты,

которые не должны допускать самой возможности их появления. Риск попадания во взаимоблокировку существенно возрастает, когда процесс, уже владеющий блокировками, пытается приобрести новые или повысить уровень блокирования, скажем, с общего до монопольного. Поэтому один из способов предотвратить угрозу состоит в том, чтобы все блокировки, необходимые процессу, захватывались им в самом начале цикла работы.

Вполне возможно установить правило, регламентирующее порядок захвата блокировок всеми процессами, действующими в системе, например запрашивать блокировки файлов в соответствии с лексикографическим порядком следования их названий. В этом случае Дейвид, получивший блокировку файла `Order`, позже уже не сможет обратиться к файлу `Customer`, не нарушив регламент, и потому станет очевидной "жертвой".

Можно придерживаться и такой простой, но в некоторых случаях весьма эффективной стратегии: если Мартин пытается приобрести захваченную Дейвидом блокировку, он сразу же приносится в жертву.

Если вы чрезмерно консервативны и осмотрительны, то можете применять одновременно несколько схем: например, заставить процессы приобретать все блокировки сразу и при этом отвести каждому из них соответствующий лимит времени на тот случай, если взаимоблокировка при каких-то обстоятельствах все-таки возникнет. Подобные меры на первый взгляд кажутся избыточными, но очень часто они вполне разумны и оправданны.

Довольно легко поддаться соблазну и сконструировать замечательную во всех отношениях схему разрешения взаимоблокировок, а затем вдруг с ужасом осознать, что какая-то цепочка событий не была учтена. Поэтому при разработке корпоративных приложений следует отдавать предпочтение простым и испытанным решениям. Их использование, возможно, приведет к необоснованным жертвам среди сообщества процессов, но это лучше, чем непредсказуемые последствия неверных или вовсе отсутствующих решений.

Транзакции

Транзакции (transactions) — основной инструмент управления параллельными процессами в корпоративных приложениях. Слово "транзакция" часто приходит на ум в связи с коммерческими и банковскими операциями. Обращение к банкомату, ввод пароля и получение наличности — это транзакция. Транзакциями можно считать, скажем, внесение платы за коммунальные услуги или покупку бокала пива в ближайшем ларьке.

Думаю, эти примеры хорошо характеризуют природу типичной транзакции. Во-первых, транзакция представляет собой ограниченную последовательность действий с явно определенными начальной и завершающей операциями[1]. Так, транзакция, связанная с банкоматом, начинается с размещения магнитной карты в считывающем устройстве и завершается выдачей денег либо сообщения о несоответствии запрошенной суммы остатку на счете. Во-вторых, все ресурсы, затрагиваемые транзакцией, пребывают в согласованном состоянии в момент ее начала и остаются в таковом после ее завершения: любитель пива, например, лишается какой-то части карманных денег, но получает удовольствие от созерцания янтарного напитка и утоления жажды. Баланс сохраняется.

[1] Процесс потребления пива и все вытекающие последствия, несомненно, связаны с фактом посещения ларька, но частью рассматриваемой "транзакции", вероятно, не являются. — *Прим. пер.*

То же можно сказать и о стороне продавца: опустошая пивные бочки, он наполняет свой кошелек.

Помимо того, транзакция должна либо выполняться целиком, либо не выполняться вовсе. Банковская система не может вычесть сумму из остатка на счете до тех пор, пока в выходной лоток банкомата не будет выдана соответствующая пачка купюр.

ACID: свойства транзакций

Транзакции в программных системах часто описывают в терминах свойств, обозначаемых общей аббревиатурой *ACID*.

- *Atomicity (атомарность)*. В контексте транзакции либо выполняются все действия, либо не выполняется ни одно из них. Частичное или избирательное выполнение недопустимо. Например, если клиент банка переводит сумму с одного счета на другой и в момент между завершением расходной и началом приходной операции сервер терпит крах, система должна вести себя так, будто расходной операции не было вовсе. Система должна либо осуществить обе операции, либо не выполнить ни одной. *Фиксация (commit)* результатов служит свидетельством успешного окончания транзакции; *откат (rollback)* приводит систему в состояние, в котором она пребывала до начала транзакции.
- *Consistency (согласованность)*. Системные ресурсы должны пребывать в целостном и непротиворечивом состоянии как до начала транзакции, так и после ее окончания.
- *Isolation (изолированность)*. Промежуточные результаты транзакции должны быть закрыты для доступа со стороны любой другой действующей транзакции до момента их фиксации. Иными словами, транзакция протекает так, будто в тот же период времени других параллельных транзакций не существует.
- *Durability (устойчивость)*. Результат выполнения завершенной транзакции не должен быть утрачен ни при каких условиях.

Ресурсы транзакций

В большинстве случаев под транзакциями в корпоративных приложениях подразумеваются последовательности операций, описывающие процессы взаимодействия с базами данных. Но существует и множество других объектов, управляемых с помощью механизмов поддержки транзакций, например, очереди сообщений или заданий на печать, банкоматы и т.д. Таким образом, термин "ресурсы транзакций" служит для обозначения всего, что может быть затребовано параллельно протекающими процессами, определяемыми с помощью модели транзакций. Наиболее распространенным ресурсом транзакций являются базы данных. Поэтому для наглядности и краткости упоминаются именно они, хотя все сказанное вполне применимо и к другим видам ресурсов.

Для обеспечения высокого уровня пропускной способности современные системы управления транзакциями проектируются в расчете на максимально короткие транзакции. Обычно из практики исключаются транзакции, которые охватывают действия по обработке нескольких запросов; если же подобной ситуации избежать не удается, решения реализуются на основе схемы *длинных транзакций (long transactions)*.

Чаще, однако, границы транзакции совпадают с моментами начала и завершения обработки одного запроса. Подобная *транзакция запроса (request transaction)* — весьма удачная модель, и многие среды поддерживают простой и естественный синтаксис ее описания.

Альтернативное решение состоит в том, чтобы как можно дольше откладывать процедуру открытия транзакции. При использовании подобной *отсроченной транзакции (late transaction)* все операции чтения могут выполняться до момента ее начала, который наступает только при необходимости осуществления операций, связанных с внесением каких бы то ни было изменений. Это позволяет минимизировать длину транзакции, но лишает возможности на протяжении продолжительных периодов времени применять какие-либо средства управления параллельными операциями. Такая стратегия сопряжена с повышением вероятности возникновения эффекта несогласованного чтения и потому используется сравнительно редко (если только не существует серьезных доводов в ее пользу).

Применяя транзакции, следует понимать, *что* именно надлежит блокировать. Во многих случаях диспетчер транзакций СУБД блокирует *отдельные* записи таблицы базы данных, вовлеченные в операцию, что позволяет сохранить высокий уровень параллелизма при доступе к таблице. Но если транзакция пытается блокировать слишком большое количество записей таблицы, число запросов на блокировку превышает допустимый лимит и система распространяет действие блокировки на таблицу в целом, приостанавливая выполнение конкурирующих транзакций. Подобное *расширение блокировки (lock escalation)* способно серьезно сократить потенциал параллельного доступа к данным, и именно поэтому, например, не стоит создавать некую таблицу "объектов" для данных на уровне **супертипа слоя (Layer Supertype, 491)** предметной области. Такая таблица — самый подходящий кандидат для расширения блокировки, что практически запрещает другим процессам и потокам иметь доступ к базе данных.

Уровни изоляции

Для того чтобы повысить степень параллелизма операций, ограничения взаимной обособленности транзакций определенным образом ослабляют. Полностью изолированные друг от друга транзакции принято называть *упорядочиваемыми (serializable)*. Результат выполнения транзакций в таком случае не зависит от того, протекают ли они строго последовательно по одной в каждый момент времени либо все вместе и параллельно. Если продолжить рассмотрение примера с подсчетом классов в программном пакете, начатое выше в этой главе, нетрудно убедиться, что упорядочиваемость транзакций гарантирует Мартину получение результата, который соответствовал бы либо ситуации завершения транзакции Мартина до начала транзакции Дейвида — 12, либо случаю старта транзакции Мартина после окончания транзакции Дейвида — 17. Выбор данного уровня изоляции не способен обусловить получение конкретного результата (в этом примере — одного из двух возможных), но по крайней мере гарантирует его корректность.

Большинство систем управления транзакциями основаны на стандарте SQL, который предусматривает возможность использования четырех уровней изоляции. Упорядочиваемый уровень является самым строгим, а каждый из трех других допускает возможность возникновения тех или иных эффектов несогласованного чтения. Напомним условие примера, в котором Мартин подсчитывает классы двух пакетов, реализующих протоколы блокирования и протоколирования действий транзакций, а Дейвид их модифицирует. Предположим, что до внесения изменений Дейвидом пакет блокирования

содержит семь классов, а пакет протоколирования — пять; после завершения транзакции Дейвида эти значения возрастают до девяти и восьми соответственно. Вначале Мартин работает с пакетом блокирования, затем Дейвид модифицирует оба пакета, после чего Мартин обращается к пакету протоколирования.

Если выбран уровень изоляции с поддержкой *упорядочиваемых* транзакций, система гарантирует, что в результате Мартин получит одно из двух значений — 12 или 17, причем оба вполне корректны. Хотя нельзя поручиться, что каждое выполнение этого сценария приведет к одному и тому же исходу, один из двух предсказуем наверняка.

Следующим по степени строгости является уровень *повторяемого чтения (repeatable-read)*, разрешающий наличие *фантомных (phantom)* записей или объектов, которые могли быть добавлены в таблицу или коллекцию параллельными транзакциями. Наш доверчивый Мартин открывает пакет блокирования и видит в нем семь классов. Чуть позже коварный Дейвид завершает свою транзакцию, и Мартин обнаруживает в пакете протоколирования восемь классов. Суммарный результат явно неверен: на его качество повлияло присутствие фантомных объектов, адекватных только части, но не всей транзакции бедного Мартина.

Уровень изоляции *чтение фиксированных данных (read-committed)* разрешает операции *неповторяемого чтения (unrepeatable-read)*. Представим, что Мартин обращает внимание на некие итоговые записи, а не на классы как таковые. Используя операцию неповторяемого чтения, он находит в пакете блокирования итоговую запись, содержащую значение семь. Неустанный Дейвид осуществляет свою фиксацию, а Мартин, как и прежде, обращаясь к пакету протоколирования, считывает итог, равный восьми. (Если бы Мартин мог повторить операцию чтения записи из пакета блокирования, он, разумеется, получил бы новое значение — девять.) Системе баз данных проще обнаруживать операции неповторяемого чтения, нежели объекты-фантомы, так что уровень повторяемого чтения обеспечивает большую меру корректности данных, но меньшую степень параллелизма, чем уровень чтения фиксированных данных.

На уровне *чтения нефиксированных данных (read-uncommitted)*, гарантирующем только самую слабую степень изоляции, позволено выполнение операций *чтения мусора (dirty reads)*. Иными словами, транзакция может "видеть" промежуточные данные, сохраненные, но не зафиксированные другими транзакциями. Это чревато возникновением ошибок двух категорий. Во-первых, Мартин может заглянуть в пакет блокирования в тот момент, когда Дейвид уже добавил в него один новый класс, но еще не успел сохранить другой. В результате он придет к неверному выводу, что количество классов в пакете равно восьми. Вторая опасность связана с тем, что Дейвид имеет право внести изменения, а затем аннулировать их, осуществив откат транзакции, и Мартин, возможно, увидит то, чего позже в действительности уже не будет.

В табл. 5.1 перечислены потенциальные изъяны, присущие каждому уровню изоляции.

Если вас в первую очередь беспокоит проблема достоверности информации, следует применять уровень изоляции с поддержкой упорядочения транзакций. Однако имейте в виду, что это приведет к резкому снижению степени параллелизма операций и пропускной способности системы в целом. Поэтому лучше попытаться отыскать разумный компромисс.

Кроме того, вас никто не заставляет использовать одинаковые уровни изоляции для всех транзакций — выбирайте те, которые в наибольшей мере отвечают вашим потребностям.

Таблица 5.1. *Уровни изоляции и возможные эффекты несогласованного чтения*

Уровень изоляции	Чтение мусора	Неповторяемое чтение	Фантомные объекты
Чтение нефиксированных данных	Да	Да	Да
Чтение фиксированных данных	Нет	Да	Да
Повторяемое чтение	Нет	Нет	Да
С поддержкой упорядочения транзакций	Нет	Нет	Нет

Системные транзакции и бизнес-транзакции

Те транзакции, о которых шла речь до сих пор и которые упоминаются в большинстве случаев, называют *системными (system transactions)*. Именно такие транзакции поддерживаются СУБД и специализированными системами управления. Транзакция базы данных — это группа SQL-команд, обрамленная инструкциями начала и завершения. Если, скажем, четвертая от начала транзакции команда приводит к нарушению некоторого ограничения целостности, система должна аннулировать результаты выполнения первых трех команд и уведомить процесс-инициатор о неудачном завершении транзакции. Если все четыре команды обработаны успешно, их результаты становятся достоянием других процессов одновременно, а не каждый в отдельности. Технологии управления системными транзакциями в достаточной мере стандартизованы и доступны для разработчиков прикладных программ.

Однако смысл системных транзакций остается скрытым для пользователей бизнес-систем. Например, с точки зрения посетителя банковского Web-портала, транзакция состоит из процедуры регистрации, выбора счета, задания суммы, определения вида операции и щелчка на кнопке OK. Подобная последовательность действий называется *бизнес-транзакцией (business transaction)* и должна обладать теми же свойствами ACID, что и аналогичная системная транзакция. Если пользователь прерывает выполнение сценария до щелчка на кнопке OK, любые изменения в состоянии системы подлежат безусловной отмене; если транзакция завершается успешно, все ее промежуточные результаты фиксируются на уровне системы только после щелчка на кнопке OK.

Для поддержки свойств ACID бизнес-транзакции необходимо выполнить ее целиком в рамках одной системной транзакции. К сожалению, бизнес-транзакция зачастую предусматривает обработку многих запросов, поэтому для ее реализации потребуется *длинная (long)* системная транзакция. Но во многих случаях эффективность таких транзакций оставляет желать лучшего.

Если ваша система не предполагает функционирования многих параллельных процессов, без длинных транзакций можно обойтись. Впрочем, это не значит, что вам их непременно следует обходить стороной. При использовании длинных транзакций удается избежать многих проблем, хотя и ценой частичной утраты возможности масштабирования. А процесс преобразования длинных транзакций в короткие часто оказывается крайне сложным и неоднозначным.

Тем не менее во многих корпоративных приложениях длинные транзакции *не* применяются. В таких случаях приходится принимать на себя ответственность за поддержку свойств ACID бизнес-транзакции, т.е. решать проблему обеспечения *параллелизма в автономном режиме (offline concurrency)*. Любое взаимодействие бизнес-транзакции с таким ресурсом транзакции, как база данных, выполняется внутри системной транзакции (тем самым обеспечивается целостность этого ресурса). Как будет показано ниже, для адекватного воплощения бизнес-транзакции простого перечня системных транзакций недостаточно — программное приложение должно обеспечить надлежащие средства их "склеивания".

Свойства *атомарности* и *устойчивости* бизнес-транзакций поддерживать значительно легче, нежели другие свойства ACID: достаточно реализовать фазу фиксации бизнес-транзакции с помощью системной транзакции. Прежде чем предпринимать попытку фиксации всех внесенных изменений в наборе записей, бизнес-транзакция должна активизировать системную транзакцию. Только системная транзакция поможет гарантировать, что результаты всех модификаций будут зафиксированы цельно и надежно. Единственной потенциально сложной задачей в этом случае является поддержка точного набора измененных данных в течение всего жизненного цикла бизнес-транзакции. Если приложение основано на **модели предметной области (Domain Model, 140)**, проследить за изменениями поможет типовое решение **единица работы (Unit of Work, 205)**. Однако при размещении бизнес-логики в контексте **сценария транзакции (Transaction Script, 133)** операции по учету изменений придется выполнять вручную, но этот факт, вероятно, не создаст много проблем, поскольку выбор решения **сценарий транзакции** говорит сам за себя: бизнес-транзакция, видимо, не отличается логической сложностью.

Намного сложнее сберечь в рамках бизнес-транзакции ACID-свойство *изолированности*. Изъяны в качестве взаимной изоляции бизнес-транзакций, в свою очередь, приводят к нарушениям *согласованности*. Требование согласованности подразумевает, что бизнес-транзакция не должна оставлять набор записей данных в некорректном состоянии. Внутри одной бизнес-транзакции ответственность приложения за обеспечение согласованности сводится к выполнению всех оговоренных бизнес-правил. В пределах нескольких транзакций приложение должно гарантировать, что изменения, вносимые одной из них, не повлияют на результаты работы остальных.

Наряду с очевидными проблемами противоречивости операций изменения, существуют более тонкие, связанные с несогласованностью операций чтения. Когда информация считывается несколькими системными транзакциями, сложно гарантировать ее согласованность. Степень рассогласования считанных данных может быть настолько велика, что способна привести к сбою системы.

Бизнес-транзакции тесно связаны с сеансами взаимодействия пользователя с системой. Обычно уместно полагать, что все бизнес-транзакции протекают в рамках одного сеанса. Хотя с формальной точки зрения вполне возможно спроектировать систему, в которой для реализации одной бизнес-транзакции требуется несколько сеансов, делать это не рекомендуется, поскольку такой путь наверняка заведет вас в тупик.

Типовые решения задачи обеспечения автономного параллелизма

Возможности существующих инструментов контроля за системными транзакциями следует использовать как можно шире. Принимаясь за управление параллельными операциями, перекрывающими границы системных транзакций, вы вступаете в мутные воды, кишащие виртуальными медузами, акулами, пираньями и другими малосимпатичными тварями, встреча с которыми не сулит ничего хорошего. К сожалению, наличие принципиального несоответствия между системными и бизнес-транзакциями нередко означает, что этой неприятной участи вам не избежать. Однако рассматриваемые ниже типовые решения, вероятно, ее облегчат.

Впрочем, пользоваться ими следует только в случае крайней необходимости. Если есть возможность, скажем, отобразить все бизнес-транзакции в виде одной системной транзакции либо воспользоваться длинными транзакциями, пренебрегая гипотетической потребностью масштабирования приложения в будущем, не упускайте этот шанс. Перекладывая заботу об управлении параллельными операциями на штатную специализированную систему, вы избавитесь от массы неприятностей. Предлагаемые решения — та спасительная соломинка, за которую вы сможете ухватиться, если не сможете освободиться от подобных обязанностей. Принимая во внимание сложную природу параллелизма, еще раз напомним, что каждое типовое решение — это начальная точка, а не пункт назначения. И хотя нам оно может казаться полезным, не станем пропагандировать его как панацею от всех бед.

Первое предлагаемое типовое решение — **оптимистическая автономная блокировка (Optimistic Offline Lock, 434)** — реализует оптимистическую стратегию управления параллельными бизнес-транзакциями. Ему отводится первое место ввиду относительной простоты использования и обеспечения высшей степени параллелизма. Одно из ограничений **оптимистической автономной блокировки** состоит в том, что неудачное завершение бизнес-транзакции удается обнаружить только при попытке фиксации ее результатов, а при определенных обстоятельствах такое запаздывание обходится слишком дорого. Пользователь, которому приходится тратить целый час на ввод всей информации о договоре найма, чтобы с досадой обнаружить роковую ошибку, допущенную в самом начале сеанса, вряд ли останется в восторге от общения с системой, поощряющей подобные безобразия. Альтернативное решение — **пессимистическая автономная блокировка (Pessimistic Offline Lock, 445)** — позволяет выявлять потенциальные ошибки намного раньше, но его реализация сопряжена с гораздо большими трудностями, а практическое применение приводит к существенному снижению степени параллелизма системы.

Применяя любой из подходов, вы значительно упростите свою задачу; но не пытайтесь при этом вручную управлять блокировками всех объектов. **Блокировка с низкой степенью детализации (Coarse-Grained Lock, 457)**, например, позволяет контролировать параллельное функционирование целой *группы* объектов. Еще одно полезное решение — **неявная блокировка (Implicit Lock, 468)** — также облегчает жизнь разработчиков корпоративных приложений, поскольку устраняет необходимость прямого управления блокировками и вероятность возникновения ошибок из-за "забывчивости" (устранять такие ошибки очень трудно).

Довольно распространено мнение о том, что решение, касающееся схемы управления параллельными операциями, принимается после удовлетворения всех требований,

предъявляемых к системе, и носит сугубо технологический характер. Позвольте с этим не согласиться. Выбор оптимистической или пессимистической стратегии затрагивает *основополагающие* принципы взаимодействия пользователя с конкретной системой. Проектирование решения **оптимистическая автономная блокировка** должно быть подкреплено множеством разнообразных сведений о предметной области, полученных от будущих пользователей системы. Подобный высокий уровень осведомленности об особенностях домена необходим и при выборе удачных вариантов **блокировки с низкой степенью детализации**.

Реализация параллельных вычислений — одна из наиболее сложных проблем в области разработки программного обеспечения. Параллельный код очень трудно тестировать. Ошибки крайне сложно воспроизвести. Их причины трудно проследить и проанализировать. Названные типовые решения до сих пор как-то отвечали возлагаемым на них надеждам, но все это, скажем так, заповедная территория. И если вам придется на нее ступить, возьмите в помощь надежного проводника или хотя бы предварительно обратитесь к книгам, упомянутым в конце главы.

Параллельные операции и серверы приложений

До сих пор речь шла о параллельных операциях преимущественно в терминах множества сеансов, в процессе своего функционирования затрагивающих единый общий источник данных. Другая форма параллелизма — конкурирующие процессы, работающие под управлением сервера приложений. Каким образом сервер одновременно обслуживает множество запросов и как это может повлиять на проектирование приложения сервера? Наиболее серьезное отличие такой модели от всего, что обсуждалось выше, состоит в отсутствии транзакций, представленных в общепринятой форме.

Хорошее многопоточное приложение, корректно использующее механизмы синхронизации (скажем, критические секции и семафоры), создать довольно сложно. В то же время при написании такой программы очень *легко* допустить ошибки, которые трудно выявить и практически невозможно воспроизвести, что в результате позволяет гарантировать ее работоспособность максимум в 99 случаях из 100. Применять ее, по понятным причинам, опасно. Поэтому рекомендуется по возможности избегать использования инструментов явного создания потоков и механизмов управления ими.

Самый простой альтернативный путь связан с реализацией схемы *процесс на сеанс (process-per-session)*, когда каждый сеанс функционирует в рамках собственного процесса. Основное преимущество такого подхода состоит в полной изоляции процессов, так что прикладным программистам уже не приходится возиться с потоками. Кроме того, варианты старта нового процесса для обработки каждого запроса и использования одного процесса, "привязанного" к сеансу, который между периодами обслуживания запросов пребывает в состоянии ожидания, практически одинаково эффективны. Во многих ранних Web-системах, например, для обработки каждого запроса выделялся отдельный Perl-процесс.

Проблема практического применения схемы "процесс на сеанс" связана с расходованием излишних ресурсов. Для повышения эффективности системы можно организовать *пул* процессов, в котором каждый процесс в любой момент времени занят обработкой одного запроса, но способен последовательно обслуживать многие запросы, относящиеся

к различным сеансам. При использовании такой схемы — назовем ее *процесс на запрос (process-per-request)* — для поддержки заданного количества сеансов потребуется гораздо меньше процессов. Уровень изоляции также не пострадает, поскольку в применении потоков по-прежнему потребности нет. Основной недостаток модели "процесс на запрос" — необходимость тщательно следить за тем, чтобы любой ресурс, использованный для обработки запроса, корректно и своевременно освобождался. Подобная схема лежит в основе текущей версии модуля `mod-perl` Web-сервера Apache, а также множества серьезных крупномасштабных систем обработки транзакций.

Даже при использовании модели "процесс на запрос" для обслуживания некоторого разумного количества запросов понадобится слишком много процессов. Чтобы повысить пропускную способность системы еще больше, можно "поступиться принципами" и организовать процесс, инициирующий множество потоков. В случае применения подхода *поток на запрос (thread-per-request)* каждый запрос обрабатывается отдельным потоком, функционирующим в контексте единого процесса. Поскольку потоки потребляют гораздо меньше ресурсов сервера, чем процессы, для обслуживания большего числа запросов потребуется меньше аппаратных затрат, т.е. сервер будет использоваться более эффективно. Недостаток подхода очевиден: отсутствие изоляции между потоками и опасность их взаимовлияния.

По нашему мнению, большего внимания заслуживает модель "процесс на запрос". Хотя она менее эффективна, чем схема "поток на запрос", но зато настолько же пригодна к масштабированию. Но что гораздо важнее, модель "процесс на запрос" обладает большей надежностью. В частности, если из строя выходит один поток, он нарушает работоспособность процесса в целом, т.е. препятствует обработке *всех* запросов, а при использовании схемы "процесс на запрос" ущерб в подобной критической ситуации наносится только *одному* запросу. Возможность избежать возни с отладкой потоков — пусть даже ценой приобретения дополнительного аппаратного обеспечения — особенно оправданна в тех случаях, когда члены команды разработчиков недостаточно квалифицированны. Целесообразно, чтобы в ходе проектирования системы несколько человек занимались тестированием производительности ее прототипов и сравнением относительных значений стоимости вариантов "процесс на запрос" и "поток на запрос" (к сожалению, подобный подход к разработке приложения встречается крайне редко).

Некоторые среды разработки обеспечивают возможность применения компромиссного решения, предусматривающего наличие изолированной области памяти, назначаемой каждому потоку. Так, в технологии COM эта схема реализуется с помощью архитектурного элемента *однопоточный апартамент (single-threaded apartment)*, а на платформе J2EE — в компонентной модели Enterprise Java Beans. Если что-то подобное есть и в среде, которую используете вы, не погнушайтесь им и испытайте на практике.

Если вы остановили свой выбор на модели "поток на запрос", не забудьте о самом важном — обеспечить наличие в приложении некой изолированной "зоны", позволяющей пренебрегать аспектами многопоточности. Обычный способ достижения подобной цели — заставить каждый поток создавать нужные ему новые объекты в самом начале цикла обработки запроса и обеспечить гарантии того, что эти объекты не размещаются там, где их могут "видеть" другие потоки (например, в статических переменных). Таким образом вы сможете гарантировать полную изоляцию каждого объекта, поскольку лишите "сторонние" потоки возможности каким бы то ни было способом ссылаться на него.

Многие разработчики обеспокоены самой необходимостью создания объектов, так как им внушили, будто это слишком "ресурсоемкий" и потому "дорогой" процесс. Как следствие, они часто прибегают к модели пула объектов. Возникает проблема: необходимо каким-то образом синхронизировать доступ к объектам из пула. Следует отметить, что стоимость создания объекта существенно зависит от типа применяемых виртуальной машины и стратегий управления оперативной памятью. В современных вычислительных средах процесс создания объектов протекает, вообще говоря, очень быстро. (Позвольте один небольшой вопрос: сколько, по вашему мнению, объектов класса `Date` Java 1.3 можно создать в течение одной секунды на машине Мартина, оснащенной процессором Pentium III с частотой 600 МГц?[2]) Создание свежих копий объектов для каждого сеанса позволяет избежать массы хлопот и повысить возможность масштабирования.

Занимаясь многопоточным программированием, следует быть особенно осторожным со статическими переменными типа классов и глобальными переменными, поскольку любое обращение к ним требует соответствующей синхронизации. То же справедливо и в отношении одноэлементных множеств. Если вам необходим некий аналог глобальной памяти, воспользуйтесь **реестром (Registry, 495)**, который можно реализовать таким образом, чтобы он выглядел как статическая переменная, но в действительности использовал блоки памяти, отведенные конкретному потоку.

Если вам удается создавать объекты в контексте сеанса и поддерживать существование относительно безопасной изолированной зоны, конструирование некоторых объектов сопряжено с заведомо большими затратами и потому должно выполняться иначе — наиболее характерным примером служит объект, представляющий соединение с базой данных. Такие объекты необходимо располагать в пуле, откуда при необходимости они могут быть затребованы и куда затем возвращены. Подобные операции нуждаются в строгой синхронизации.

Дополнительные источники информации

Эта глава только касается поверхности бездонного омута технологий управления параллельными операциями. Чтобы погрузиться глубже, понадобится более серьезная экипировка; в этом вам помогут книги [8, 27, 36].

[2] Два миллиона.

Глава 6

Сеансы и состояния

Различия между системными и бизнес-транзакциями уже рассматривались при обсуждении параллельного выполнения заданий (см. главу 5). Эти различия не только влияют на те или иные аспекты параллелизма, но и обусловливают способы сохранения в контексте транзакции информации временного характера, которая до определенного момента не может фиксироваться в базе данных.

Те же принципиальные различия лежат в основе многих дискуссий о роли и месте так называемых *сеансов с сохранением промежуточного состояния* (или, коротко говоря, *сеансов "с состоянием"*) *(stateful sessions)* и *сеансов "без состояния" (stateless sessions)*. По этому поводу сломано немало копий, но, как мне кажется, основная проблема нередко оказывается скрытой за частоколом сугубо технических вопросов. Необходимо понимать, что "состояние" является неотъемлемой частью сеансов определенных категорий и дальнейшие решения должны приниматься только с учетом этого факта.

В чем преимущество отсутствия "состояния"

Что подразумевается под сервером "без состояния"? Если говорить об объектах, то главное их достоинство состоит в том, что они сочетают в себе *состояние* (данные) и *поведение* (функции). Объект, лишенный состояния, — это объект без полей данных. Подобные объекты встречаются довольно редко, поскольку считается, что их наличие — признак плохого проектирования.

С другой стороны, однако, это явно не то, *что* думают многие, когда говорят об "отсутствии состояния" в объектах распределенных корпоративных приложений. Под сервером без состояния понимается объект, который просто *не* сохраняет данные между циклами обработки отдельных запросов. Подобный объект вполне способен содержать поля, но когда вызывается метод сервера без состояния, значения этих полей трактуются как неопределенные.

Примером сервера без состояния может служить система, в ответ на запрос возвращающая Web-страницу с информацией о книге. Пользователь задает код ISBN книги и активизирует объект ASP-документа или сервлета. Извлекая из базы данных информацию об авторе книги, ее наименовании и т.п., объект сервера может временно сохранять ее во внутренних полях, чтобы во всеоружии подойти к фазе генерации текста HTML. Объект также способен реализовать некоторую бизнес-логику, связанную,

скажем, с определением того, какие дополнительные данные о книге следует предоставить пользователю. По завершении обработки запроса накопленная в полях объекта информация становится бесполезной. Очередной запрос с иным кодом ISBN — это, как говорится, совсем другая история. Поэтому поля такого объекта во избежание возможных ошибок инициируются заново.

Теперь вообразите, что ставится задача сбора сведений обо всех кодах ISBN, запрашиваемых клиентом с определенным IP-адресом. Информацию можно располагать в списке, поддерживаемом объектом сервера. В паузах между циклами обработки запросов список должен каким-то образом сохраняться, поэтому речь следует вести об объекте сервера "с состоянием". Переход от объекта без состояния к объекту с состоянием — это не просто замена предлога "без" предлогом "с". Многие воспринимают подобное требование едва ли не как катастрофу. Но почему?

Основная проблема связана с потребностью в ресурсах. Объекту сервера с состоянием необходимо сохранять все данные, образующие состояние, в период ожидания, пока пользователь предается тяжким раздумьям, "пялясь" в Web-страницу. Объект сервера без состояния, однако, в такой ситуации мог бы заняться обслуживанием запросов, инициируемых в других сеансах. Проведем далекий от реальности, тем не менее полезный мыслительный эксперимент. Представим, что сто человек интересуются книгами, упоминаемыми на страницах электронного каталога, и для обработки запроса по любой книге требуется одна секунда. Каждые 10 секунд каждый пользователь инициирует по одному запросу, и все запросы равномерно распределяются во времени. Если необходимо проследить историю запросов пользователей с помощью объектов сервера с состоянием, надлежит выделить по одному объекту для каждого пользователя, т.е. 100 объектов. Но 90% времени жизни объектов будет пропадать вхолостую. Отказавшись от намерений накопить сведения о читательских пристрастиях пользователей и решив применять объекты без состояния, способные просто обрабатывать запросы, можно обойтись всего десятью объектами сервера, которые будут заняты "делом" непрерывно.

Если в промежутках между вызовами методов данные не сохраняются, не имеет значения, каким именно объектом обслуживается запрос, но если состояние должно фиксироваться, обработкой запроса должен заниматься один и тот же объект. Отсутствие потребности в сохранении состояния дает возможность сформировать пул объектов, который позволит с меньшими затратами обслужить большее количество запросов. Чем больше пользователей-"тугодумов", тем выше ценность объектов сервера без состояния. Вполне очевидно, что особенно полезны объекты серверов без состояния, которые обслуживают Web-сайты с высоким уровнем трафика. Концепция объектов без состояния совершенно органично вписывается в модель Web, поскольку основной протокол Web, а именно HTTP, относится к категории протоколов без состояния.

Так что, никаких состояний — и точка, верно? Нет, если можно — пожалуйста. Но есть одна проблема: многие сценарии взаимодействия клиентов с сервером по своей природе предполагают сохранение состояния. Рассмотрим метафору "карты покупателя", лежащую в основе тысяч приложений электронной коммерции. В процессе общения с сайтом виртуального магазина (в данном случае книжного) посетитель формирует запросы и выбирает книги, которые желает приобрести. Карта покупателя должна сохраняться в течение всего пользовательского сеанса. По сути, речь идет о бизнес-транзакции с состоянием, которая может быть реализована посредством сеанса с состоянием. Если посетитель будет только пролистывать книги, но ничего так и не купит, его сеанс в этом частном

случае лишится состояния, но стоит ему выбрать хотя бы одну книгу, данные о ней и определят состояние сеанса. Можно попытаться избежать необходимости сохранения информации состояния, но тогда придется существенно обеднить приложение; при выборе схемы с состоянием, напротив, нужно решать, как именно ее использовать. Хорошая новость заключается в том, что для реализации сеанса с состоянием, оказывается, можно применять сервер без состояния. Еще более любопытно, что такое решение не всегда в достаточной мере привлекательно.

Состояние сеанса

Содержимое карты покупателя, о которой упоминалось в предыдущем разделе, представляет *состояние сеанса (session state)* — в том смысле, что данные, отображаемые в карте, имеют отношение только к конкретному сеансу. Это состояние действительно в контексте конкретной бизнес-транзакции, т.е. отделено от других сеансов и охватываемых ими бизнес-транзакций. (Здесь, как и прежде, подразумевается, что каждая бизнес-транзакция функционирует в контексте одного сеанса и в каждом сеансе в один и тот же момент времени выполняется всего одна бизнес-транзакция.) Состояние сеанса отличается от того, что принято называть *хранимыми данными (record data)*, т.е. от информации, которая размещается в базах данных и становится доступной для сеансов всех пользователей, наделенных соответствующими полномочиями. Чтобы информация состояния сеанса приобрела статус хранимых данных, она должна быть зафиксирована в базе данных.

Поскольку состояние сеанса существует в контексте определенной бизнес-транзакции, оно обладает многими свойствами (скажем, такими, как ACID — atomicity (атомарность), consistency (согласованность), isolation (изолированность) и durability (устойчивость)), которые обычно имеют в виду, говоря о транзакциях. Следствия этого факта не всегда воспринимаются верно.

Одно из любопытных следствий связано с эффектом согласованности. Например, в процессе редактирования информации полиса страхования текущее состояние полиса может быть некорректным. Пользователь изменяет некоторое значение, посредством запроса отсылает его системе, а последняя возвращает ответ, уведомляя об ошибке. Значение является частью состояния сеанса, но оно неверно. Состояние, таким образом, не всегда удовлетворяет заданным условиям в период, когда сеанс активен, — критерии достигаются только после фиксации результатов бизнес-транзакции.

Самая большая проблема, касающаяся состояния сеанса, связана с обеспечением изолированности. Во время редактирования того же полиса страхования могут произойти какие угодно события — слишком уж велика степень неопределенности ситуации. Наиболее очевидный пример — одновременное обращение двух пользователей к одному полису. Рассмотрим две записи: полис как таковой и сведения о клиенте. Запись полиса включает величину риска, которая частично зависит от значения почтового кода в записи клиента. Пользователь начинает редактировать полис и по прошествии 10 минут выполняет какое-то действие, которое приводит к открытию и отображению записи клиента. Допустим, что в то же время другой пользователь изменяет почтовый код и величину риска; это неминуемо приведет к ситуации несогласованного чтения.

Не все данные, хранимые на протяжении сеанса, трактуются как состояние сеанса. Во время протекания сеанса некоторая информация может кэшироваться, причем не в целях сохранения, а для повышения производительности системы. Поскольку данные кэш-памяти можно удалить без потери функциональности, они принципиально отличаются от состояния сеанса, которое подлежит обязательному сохранению в промежутках между циклами обработки запросов.

Способы сохранения состояния сеанса

Как же все-таки сохранять информацию состояния сеанса, если это действительно нужно делать? Можно предложить три основных, хотя и не вполне исключающих друг друга, решения.

Типовое решение **сохранение состояния сеанса на стороне клиента (Client Session State, 473)** предусматривает сохранение информации о состоянии сеанса на клиентской машине. Существует несколько вариантов достижения цели: кодирование данных для Web-представления, использование файлов cookie, сериализация информации в скрытых полях Web-формы и сохранение объектов в приложении толстого клиента.

Сохранение состояния сеанса на стороне сервера (Server Session State, 475) может быть, в частности, настолько простым, как размещение данных в памяти на период между циклами обработки запросов. Однако обычно используется механизм долговременного хранения информации в виде некоего сериализованного объекта. Объект сберегают в файловой системе сервера приложений или в источнике данных, допускающем совместный доступ, например в виде простой таблицы базы данных с двумя полями: первое — ключ сеанса, а второе — содержимое сериализованного объекта.

Решение **сохранение состояния сеанса в базе данных (Database Session State, 479)** также предусматривает использование носителей сервера, но с более тщательным структурированием информации по таблицам и полям базы данных.

Выбор подходящего варианта сопряжен с преодолением ряда препятствий. Прежде всего необходимо учесть возможности канала связи между клиентом и сервером. Решение **сохранение состояния сеанса на стороне клиента** предполагает активный обмен информацией о состоянии сеанса при обработке каждого запроса. Если речь идет всего о нескольких полях, это не проблема, но с увеличением объемов данных возрастает и нагрузка на канал. В одном из приложений, которым мне пришлось заниматься, порция передаваемой информации превышала мегабайт (или, как выразился коллега, размер трех пьес Шекспира). Следует признать, что клиент и сервер обменивались данными в формате XML, который нельзя отнести к самым компактным, но все равно информации было слишком много.

Конечно, без некоторых данных просто не обойтись, поскольку они, например, подлежат воспроизведению на уровне представления. Но при использовании решения **сохранение состояния сеанса на стороне клиента** с каждым запросом приходится передавать *все* данные, даже если они не будут визуализированы. Таким образом, это решение целесообразно применять только тогда, когда объем данных, определяющих состояние, достаточно мал. Помимо того, следует позаботиться об аспектах безопасности и целостности информации. Если вы не займетесь шифрованием данных, у вас будут все основания полагать, что любой злоумышленник сможет изменить состояние вашего сеанса.

Информация сеанса подлежит изоляции. В большинстве случаев все происходящее в пределах одного сеанса не должно влиять на прохождение остальных сеансов. Если вы

заказываете билет на самолет, ваши операции не должны воздействовать на сеансы, инициированные другими пользователями, и наоборот. Данные "сеанса" потому так и называются, что их не должен видеть никто посторонний. Проблема изоляции информации становится особенно острой (по вполне понятным причинам) при использовании типового решения **сохранение состояния сеанса в базе данных**.

Если пользователей слишком много, для повышения пропускной способности системы следует рассмотреть возможность кластеризации аппаратного обеспечения и подумать о необходимости *переноса сеанса (session migration)* с одного сервера на другой по мере завершения обработки одного запроса и поступления других. Противоположная схема — *привязка к серверу (server affinity)* — предполагает, что все запросы, инициируемые в рамках одного сеанса, обслуживаются конкретным сервером. Модель переноса сеанса позволяет достичь баланса производительности системы, особенно в тех случаях, когда сеансы длинны. Однако при использовании типового решения **сохранение состояния сеанса на стороне сервера** она может стать довольно неуклюжей, поскольку зачастую передача информации о состоянии сеанса с одного сервера на другой сопряжена с дополнительными трудностями. Впрочем, можно отыскать и компромиссные варианты, стирающие грани между решениями **сохранение состояния сеанса на стороне сервера** и **сохранение состояния сеанса в базе данных**.

Модель привязки к серверу также далеко не безгрешна — даже в большей мере, чем кажется на первый взгляд. Пытаясь гарантировать возможность привязки, кластерная система не всегда способна проследить за всеми вызовами, чтобы обнаружить, к каким сеансам они относятся. В результате привязка обеспечивается за счет того, что все вызовы со стороны конкретного клиента направляются одному и тому же серверу. Часто клиент распознается по IP-адресу. Но если клиент отделен от сервера приложений прокси-сервером, это значит, что тем же IP-адресом могут обладать и многие другие клиенты, и все они будут "привязаны" к тому же серверу приложений, что никак не назовешь удачным вариантом развития событий.

Если серверу необходимо обратиться к состоянию сеанса, последнее должно быть представлено в доступной форме. При использовании типового решения **сохранение состояния сеанса на стороне сервера** искомое состояние, как можно полагать, прямо "под рукой". Применяя решение **сохранение состояния сеанса на стороне клиента**, вам, вероятно, придется обеспечить преобразование данных в нужную форму. Решение **сохранение состояния сеанса в базе данных**, естественно, вынуждает обращаться за информацией о состоянии сеанса к базе данных (и, помимо того, нередко выполнять дополнительные преобразования). Каждый подход по-своему определяет показатели быстроты реагирования системы, которые напрямую зависят от объема и структурной сложности данных состояния.

Если речь идет, например, о системе электронной коммерции, каждый сеанс, вероятно, не должен охватывать слишком много данных, но зато вам придется иметь дело с большим количеством не очень активных пользователей. Поэтому удачным вариантом в аспекте производительности может оказаться решение **сохранение состояния сеанса в базе данных**. В лизинговой системе, напротив, вы рискуете передавать с каждым запросом слишком большие массивы информации. Это как раз тот случай, когда уместным окажется **сохранение состояния сеанса на стороне сервера**.

Одна из самых досадных проблем функционирования многих систем связана с нештатным прерыванием сеанса, когда клиент просто отключается от сервера, не "пообещав" ничего конкретного. В таком случае наиболее выигрышно выглядит решение **сохранение состояния сеанса на стороне клиента**, позволяющее серверу легко позабыть о неблагодарном клиенте. При использовании решения **сохранение состояния сеанса на стороне сервера** сеанс можно трактовать как прерванный только по истечении заданного лимита времени, но вам придется побеспокоиться об удалении информации о состоянии сеанса.

Заслуживает внимания и ситуация, связанная с выходом из строя системы в целом: сбоем приложения клиента, отказом сервера и/или разрывом сетевого соединения. Решение **сохранение состояния сеанса в базе данных** обычно позволяет поладить со всеми тремя неприятностями. "Выстоит" ли информация при условии **сохранения состояния сеанса на стороне сервера**, зависит от того, предусматривалось ли сбережение резервной копии данных объекта сеанса в энергонезависимом хранилище. Результаты **сохранения состояния сеанса на стороне клиента**, разумеется, "погибнут" вместе с клиентом, если такое вдруг случится, но, вероятно, останутся в неприкосновенности при других возможных авариях.

Собравшись применить любое из рассмотренных типовых решений, не забудьте об усилиях, которые придется приложить для его практического воплощения. Обычно проще реализовать **сохранение состояния сеанса на стороне сервера**, особенно в тех случаях, когда не нужно сохранять состояние в промежутках между циклами обработки запросов. Использование решений **сохранение состояния сеанса в базе данных** и **сохранение состояния сеанса на стороне клиента**, как правило, сопряжено с необходимостью дополнительного программирования процедур преобразования состояния из формата передачи по каналу или представления в базе данных в формат объекта. Вам вряд ли удастся выполнить работу настолько же быстро и полно, как в случае **сохранения состояния сеанса на стороне сервера**, особенно если структуры данных отличаются повышенной сложностью. Вариант **сохранения состояния сеанса в базе данных** на первый взгляд кажется вполне привлекательным (если вы к тому же определились с параметрами отображения объектов в реляционные структуры), но вам придется поднатужиться, чтобы изолировать данные сеанса от попыток несанкционированного обращения.

Как часто случается при выборе способов реализации различных архитектурных элементов корпоративных приложений, указанные подходы нельзя назвать взаимоисключающими. Для сохранения тех или иных фрагментов данных состояния сеанса можно применять две или даже все три схемы. Это, правда, приведет к заметному усложнению результата из-за опасности утраты контроля над тем, что, где и как сохраняется. Тем не менее, если используется решение, отличное от **сохранения состояния сеанса на стороне клиента**, в памяти клиента все равно придется сохранять по меньшей мере некий идентификатор сеанса.

Я отдал бы предпочтение решению **сохранение состояния сеанса на стороне сервера**, особенно если на случай поломки сервера резервная копия данных сохраняется отдельно. Мне по душе и вариант **сохранения состояния сеанса на стороне клиента**, предусматривающий хранение идентификатора сеанса и данных состояния небольшого объема. Но я не стал бы прибегать к **сохранению состояния сеанса в базе данных**, если только речь не идет о необходимости высокого уровня надежности, о кластеризации аппаратного обеспечения либо о невозможности сохранения резервных копий данных.

Глава 7

Стратегии распределенных вычислений

Объекты окружают нас уже долгое время — иногда кажется, что они были всегда. Многие, создавая объекты, сразу же задумываются о необходимости их "распределения". Впрочем, распределение объектов (как и любых других инструментов вычислений) сопряжено с гораздо большими сложностями, чем можно было бы предвидеть [38]. Прочитав эту главу, вы узнаете о некоторых сложных уроках распределенных вычислений и сможете избежать неприятностей, с которыми вам пришлось бы столкнуться, шагая вперед самостоятельно.

Соблазны модели распределенных объектов

Два-три раза в год мне доводится участвовать в одном и том же "шоу". Архитектор очередной объектно-ориентированной системы (допустим, приложения для обработки каких-то заказов) с гордостью выставляет на общее обозрение план распределения объектов (вполне возможно, что эскиз может выглядеть так, как показано на рис. 7.1): каждый программный компонент размещается в отдельном узле системы.

"Зачем все это?" — спрашиваю я.

"Производительность, вестимо, — отвечает архитектор, глядя на меня со слабо скрываемым превосходством. — Мы можем запустить каждый компонент на обработку в своем собственном блоке. Если мощности блока не хватит, мы запросто добавим еще парочку, чтобы сбалансировать нагрузку". Теперь он уже и не пытается утаить самолюбования вперемешку с удивлением по поводу того, что я вообще посмел открыть рот.

Между тем передо мной возникает любопытная дилемма: заявить парню все сразу и выставить за дверь либо не торопясь показать ему дорогу к светлому будущему. Последнее во всех смыслах выгоднее, но гораздо хлопотнее, поскольку архитектор обычно слишком пленен собственными иллюзиями и вряд ли легко с ними расстанется.

Эта книга (хотелось бы надеяться) поможет вам понять изъяны подобной распределенной архитектуры. Многие поставщики инструментальных средств программирования не устают твердить: основное достоинство технологий распределения объектов заключается именно в том, что вы можете взять "пригоршню" объектов и разбросать их по узлам сети как вашей душе угодно, а фирменное промежуточное программное обеспечение сделает прозрачными все взаимосвязи. Свойство прозрачности позволяет объектам вызывать друг друга внутри одного процесса, между процессами на одной машине или на разных машинах, не заботясь о местонахождении "собеседников".

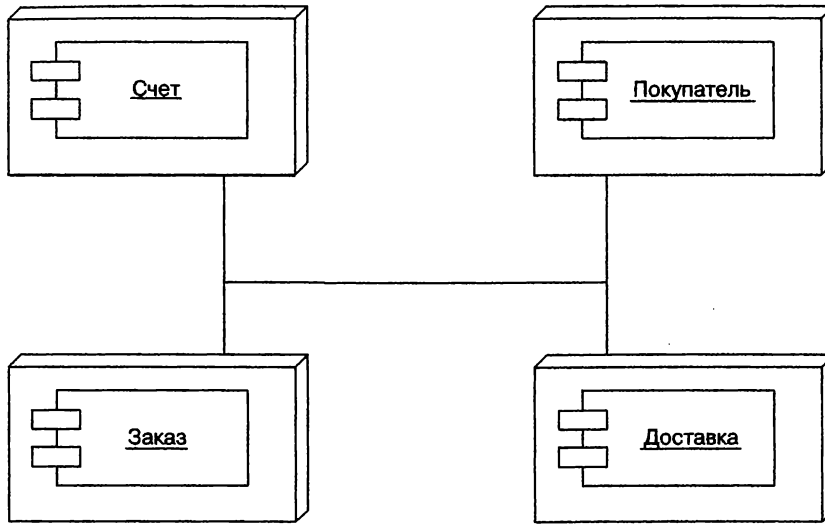


Рис. 7.1. Распределение объектов путем разнесения всех компонентов приложения по разным узлам (не рекомендуется!)

Безусловно, все это просто замечательно, но... Хотя многие стороны жизни распределенных объектов действительно приобретают искомую прозрачность, это явно не относится к аспектам производительности. Наш герой-архитектор осуществил распределение объектов, как ему казалось, исходя из соображений производительности, но на самом деле выбор подобной структуры наверняка снизит эффективность системы и существенно усложнит процессы ее разработки и практического внедрения.

Интерфейсы локального и удаленного вызова

Главная причина неудовлетворительной работоспособности модели распределения кода по "классовой" принадлежности связана с основополагающими аспектами функционирования компьютерных систем. Вызов процедуры в пределах одного процесса протекает чрезвычайно быстро. Вызов между двумя отдельными процессами, работающими на одном компьютере, обслуживается на несколько порядков медленнее. Активизируйте один из процессов на другой машине, и вы увеличите время обработки еще на пару порядков, в зависимости от сложности топологии конкретной сети.

Отсюда следует, что интерфейсы одного и того же объекта, предназначенные для локального и удаленного доступа, должны различаться.

На роль локального интерфейса более всего подходит интерфейс с высокой степенью детализации. Хороший интерфейс класса, представляющего почтовый адрес, например, должен включать отдельные методы для задания/считывания почтового кода, наименования города, названия улицы и т.п. Достоинство интерфейса с высокой степенью детализации состоит в том, что он следует общему принципу объектной ориентации, состоящему в применении множества небольших простых фрагментов кода, которые могут сочетаться и переопределяться разными способами для пополнения и изменения множества функций класса в будущем.

Детальный интерфейс, однако, утрачивает свои преимущества при использовании в режиме удаленного доступа. Когда вызов метода обрабатывается медленно, несомненно, целесообразнее получать значения почтового кода, наименования города и названия улицы сразу, а не по одному. Такой интерфейс, спроектированный не с учетом гибкости и возможности расширения, а в целях уменьшения количества вызовов, должен отличаться низкой степенью детализации. Он, очевидно, гораздо менее удобен в использовании, но зато более эффективен в смысле быстродействия.

Конечно, поставщики инструментальных средств наверняка будут убеждать вас в том, что различия в использовании их фирменного промежуточного программного обеспечения в локальном и удаленном режимах "практически" неощутимы. Если вызов локален, он осуществляется с максимальной скоростью. Если же речь идет об удаленном вызове, скорость снижается, но вы платите эту цену только тогда, когда без удаленного вызова действительно не обойтись. Утверждение во многом справедливо; однако нельзя обойти вниманием то принципиальное положение, что любой объект, допускающий удаленные вызовы, должен быть снабжен интерфейсом с низкой степенью детализации, а объект, адресуемый в локальном режиме, — интерфейсом с высокой степенью детализации. Для поддержки взаимодействия двух объектов следует выбрать подходящий тип интерфейса. Если объект допускает возможность вызова из контекста стороннего процесса, вам *придется* использовать менее детальный интерфейс и расплачиваться усложнением модели программирования и потерей гибкости. Все это имеет смысл, разумеется, только тогда, когда приобретения того стоят; в общем же случае взаимодействия между процессами, безусловно, следует избегать.

По этим причинам нельзя просто взять группу классов, спроектированных в расчете на использование в едином процессе, применить технологию CORBA или что-то подобное и заявить, что распределенное приложение готово к работе. Распределенная модель вычислений — это нечто большее. Основывая свою стратегию распределения на классах, вы неизбежно придете к системе, изобилующей удаленными вызовами и потому нуждающейся в реализации неповоротливых интерфейсов с низкой степенью детализации. В завершение вы с удивлением заметите, что удаленных вызовов все еще слишком много и любая незначительная модификация кода дается с большим трудом.

Вот мы и добрались до моего *Первого Закона Распределения Объектов*, который гласит: "Не распределяйте объекты!"

Хорошо, но как тогда эффективно использовать несколько процессоров? В большинстве случаев этого можно добиться с помощью механизма кластеризации (рис. 7.2). Разместите все классы в контексте одного процесса и активизируйте несколько копий процесса на разных узлах. Тогда каждый процесс будет пользоваться только локальными вызовами и достигнет цели значительно быстрее. Вы сможете применить во всех классах

интерфейсы с высокой степенью детализации, упростить модель программирования и облегчить задачу сопровождения системы.

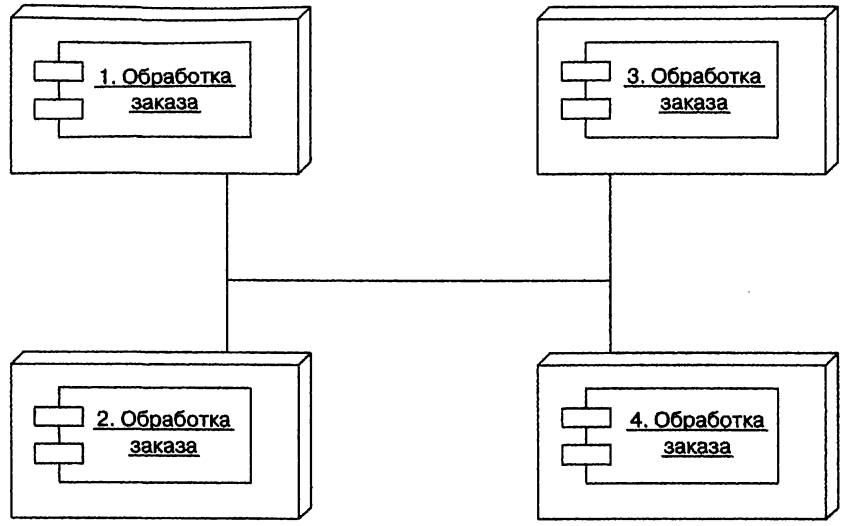


Рис. 7.2. Кластеризация предполагает размещение нескольких копий одного приложения на различных узлах

Когда без распределения не обойтись

Как следует из сказанного выше, границы распределения кода целесообразно свести к минимуму и попытаться в возможно большей степени использовать мощности узлов за счет их кластеризации. Такое решение, однако, приемлемо не всегда; встречаются ситуации, когда приходится применять отдельные процессы. За уменьшение их количества можно и побороться, но искоренить саму модель распределения вам, вероятно, не удастся.

- Наиболее очевидный пример связан с разделением кода традиционного клиента и сервера. Персональные компьютеры — это узлы, совместно пользующиеся централизованным хранилищем информации. Поскольку они физически разделены, взаимодействия процессов не избежать. Разделение кода клиента и сервера представляет собой типичный пример межпроцессного разделения вычислений.
- Вторая полоса разделения пролегает между сервером приложений и сервером базы данных. Проводить ее, как кажется, вовсе не обязательно, поскольку весь прикладной код можно, во всяком случае теоретически, выполнять в контексте процесса СУБД, используя механизмы поддержки хранимых процедур. Но часто такое решение непрактично, поэтому приходится отдавать предпочтение нескольким отдельным процессам. Правда, они могут выполняться на одной машине, но это лишь незначительно уменьшит те издержки, на которые потребуется пойти. К счастью, SQL спроектирован как интерфейс удаленного вызова, что способствует некоторому снижению затрат.

- Еще одна граница может разделять Web-сервер и сервер приложений. При прочих равных условиях лучше выполнять код обоих серверов в рамках одного процесса, но эти прочие условия, увы, не всегда оказываются равными.
- Иногда разделять процессы заставляет необходимость применения компонентов программного обеспечения от разных поставщиков. Код фирменного программного пакета часто выполняется в контексте собственного процесса, что вновь заставляет вспомнить о распределении объектов. В лучшем случае пакет может быть оснащен интерфейсом с низкой степенью детализации.
- Наконец, могут существовать и иные жизненно важные доводы в пользу расщепления кода сервера приложений. Если так, вам остается позаботиться о том, чтобы интерфейсы классов не оказались чрезмерно "детальными".

Сужение границ распределения

Проектируя программное приложение, необходимо пытаться сузить границы распределения кода до минимально возможных пределов, и там, где распределения не миновать, уделять этим границам самое пристальное внимание. Чтобы уменьшить количество удаленных вызовов, придется пересмотреть в системе очень многое.

Впрочем, можно все еще проектировать приложение так, будто оно должно функционировать в контексте единого процесса, и применять объекты с детализированными интерфейсами, но исключительно для внутренних целей, размещая на "границах" объекты с "огрубленными" интерфейсами, которые должны предоставить возможность доступа к "детальным" объектам. Подобный **интерфейс удаленного доступа (Remote Facade, 405)** не выполняет ничего иного, кроме обеспечения взаимодействия объектов с высокой степенью детализации интерфейса с внешней средой, и служит только целям распределения.

Типовое решение **интерфейс удаленного доступа** помогает избавиться от трудностей, сопровождающих попытки практической реализации модели интерфейсов с низкой степенью детализации. В этом случае огрубленными методами снабжаются только такие объекты, которые действительно нуждаются в службе удаленного доступа, и разработчикам становится ясно, за что именно они платят.

Оставляя за огрубленными методами роль интерфейса удаленного доступа, вы не запрещаете пользоваться "точными" методами, если заведомо ясно, что объект адресуется в контексте того же процесса. Это обстоятельство делает политику распределения более прозрачной. В тесном контакте с **интерфейсом удаленного доступа** существует **объект переноса данных (Data Transfer Object, 419)**: нужны ведь не только огрубленные методы, но и возможность передачи объектов с низкой степенью детализации интерфейса. Когда запрашивается почтовый адрес, подобная информация должна пересылаться одним блоком. Объект домена обычно нельзя посылать непосредственно, поскольку он является предметом множества локальных ссылок. Поэтому все необходимые клиенту сведения переправляются в виде специального **объекта переноса данных**. (Многие, кто занимается корпоративными Java-приложениями, для подобной цели используют термин *объект-значение (value object)*, но он до некоторой степени противоречит смыслу одноименного типового решения **объект-значение (Value Object, 500)**.) **Объект переноса данных** присутствует на обоих концах связи, поэтому важно понимать, что он не затрагивает ничего, что

не имело бы отношения к этой связи: действительно, один **объект переноса данных** способен адресовать только другие **объекты переноса данных**, а также объекты базовых типов, например строки.

Еще одна схема распределения предполагает использование *брокера (broker)*, обеспечивающего доставку объектов между процессами. Ее можно трактовать как менее традиционный вариант типового решения **загрузка по требованию (Lazy Load, 220)**, отличающийся от обычного тем, что вместо считывания информации из базы данных опосредованию подвергается процесс обмена объектами. Единственная сложность сопряжена с тем, справится ли брокер с массой заявок. Мне пока не попадались реальные примеры приложений, которые успешно действовали бы по такой схеме, но некоторые инструментальные средства отображения объектов в реляционные структуры (скажем, TOPLink) предлагают такие возможности, и есть уже положительные высказывания на их счет.

Интерфейсы распределения

Традиционно интерфейсы распределенных программных компонентов строились до сих пор на основе механизмов *удаленного вызова процедур (remote procedure call — RPC)*, реализуемых в виде глобальных процедур или методов объектов. Однако в последние несколько лет все чаще встречаются интерфейсы на базе XML и HTTP. Наиболее употребительной формой таких интерфейсов можно считать SOAP.

Коммуникации HTTP на основе XML удобны в нескольких отношениях. Они позволяют легко передавать большие порции структурированной информации, что вполне согласуется с требованием минимизации количества удаленных вызовов. Поскольку протокол HTTP носит универсальный характер, а формат XML поддерживается синтаксическими анализаторами, доступными на множестве платформ, в обмене данными могут принимать участие самые разные приложения. Текстовая природа XML упрощает контроль содержимого передаваемых сообщений, а HTTP облегчает прохождение пакетов данных через межсетевые экраны, когда по соображениям безопасности трудно открыть другие порты.

Объектно-ориентированные интерфейсы классов и методов, тем не менее, своего значения не теряют. Упаковка всех подлежащих перемещению данных в XML-структуры и строки в значительной степени отягощает удаленный вызов. Замена XML-интерфейсов прямыми вызовами позволяет существенно повысить производительность многих приложений. Если на обеих сторонах связи используется один и тот же бинарный механизм, XML-интерфейс оказывается просто иной, более красноречивой формой выражения. Если две взаимодействующие системы функционируют на одной платформе, лучше воспользоваться технологиями удаленного вызова уровня платформы. Однако Web-службы становятся особенно полезными в условиях, когда в "общении" заинтересованы системы, действующие на разных платформах. Я соглашаюсь с целесообразностью применения Web-служб на основе XML только тогда, когда не нахожу более прямолинейных подходов.

Разумеется, можно взять все лучшее из двух решений, если разместить слой HTTP-интерфейса "поверх" слоя объектно-ориентированного интерфейса, предусматривая, что все вызовы, обращенные к Web-серверу, должны транслироваться им в вызовы

методов нижележащих объектов. Да, вы получите "все лучшее", но ценой усложнения системы. Поэтому подобную совмещенную архитектуру следует использовать только тогда, когда вам действительно одновременно нужны HTTP-службы и объектно-ориентированный API-интерфейс удаленного доступа, обеспечивающий, например, поддержку политики безопасности или средства управления транзакциями.

Говоря о механизмах удаленного вызова процедур, я имел в виду *синхронные* RPC-интерфейсы. Впрочем, не думаю, что это всегда лучший выбор. Напротив, я все чаще отдаю предпочтение модели интерфейса, основанного на сообщениях, который по своей природе асинхронен. Типовые решения в архитектуре программных систем, основанных на сообщениях, — это глубокая тема, заслуживающая отдельной книги, и потому здесь я просто умываю руки. Надеюсь, что в скором будущем такая книга появится. Теперь же мне остается настоятельно рекомендовать вам именно такие — асинхронные и основанные на сообщениях — интерфейсы, поскольку они, по моему убеждению, позволяют наилучшим образом реализовать Web-службы.

Глава 8

Общая картина

В каждой из предыдущих глав рассматривался какой-либо один аспект системы и исследовались различные варианты его реализации. Теперь настало время объединить фрагменты мозаики в общую картину и перейти к поиску ответа на основной вопрос: какими типовыми решениями стоит пользоваться, занимаясь проектированием корпоративного программного приложения.

То, о чем пойдет здесь речь, во многом повторяет сказанное выше. Признаюсь, я неоднократно задавал себе вопрос, зачем вообще нужна эта глава, и наконец пришел к убеждению, что было бы неплохо вкратце обсудить все ранее сделанные выводы в едином контексте, чтобы у читателя сложилось цельное представление обо всех типовых решениях, упомянутых в книге.

Я полностью отдаю себе отчет в ограниченности своих рекомендаций. Фродо, один из персонажей киноленты "Властелин колец", помнится, как-то произнес: "Не ходите к эльфам за советом — они всё равно не скажут ни да, ни нет". Я ни в коем случае не утверждаю, что владею бессмертным трансцендентным знанием, и ясно понимаю, что любое слово может нести в себе опасность. Если вы обратились к этой книге в надежде, что она поможет предпринять верные шаги для реализации конкретного проекта, должен заметить, что вы осведомлены о своих нуждах намного лучше, чем я. Должен признаться, мне приходится испытывать чувство глубокого разочарования каждый раз, когда на конференциях или по электронной почте ко мне, "ученому мужу", обращаются за консультациями по поводу конкретных архитектурных или функциональных решений, а я за пять минут не в состоянии сказать ничего вразумительного (а о проблемах моих читателей я вообще ничего не знаю).

Поэтому воспринимайте эту главу именно так, как она преподносится. Мне неведомы все правильные ответы — мне даже не заданы все вопросы. Пользуйтесь материалом как подспорьем, но не воспринимайте его как готовую истину, наличие которой исключает потребность в творческом поиске. В конце концов, вам самому принимать решения и мириться с их результатами.

Впрочем, наши с вами решения — отнюдь не откровения Господни, высеченные в камне. Исправление архитектурных ошибок — дело хотя и трудное (причем заранее не известно, насколько), но *не* невозможное. Даже если вы не причисляете себя к сторонникам *экстремального программирования (extreme programming)* [5], советую серьезно присмотреться к трем техническим концепциям, связанным с *непрерывной интеграцией (continuous integration)* [19], *разработкой по результатам тестирования (test driven development)* [7] и *рефакторингом (refactoring)* [18]. Они хоть и не панацея, но способны облегчить вашу жизнь, коль скоро вы того пожелаете. А вы наверняка пожелаете, если только не относитесь к редкому числу счастливых всезнаек (мне такие не встречались).

Предметная область

Приступая к проекту, необходимо решить, какой из трех вариантов организации логики *предметной области* целесообразно применить в конкретной ситуации — **сценарий транзакции (Transaction Script, 133)**, **модель предметной области (Domain Model, 140)** или **модуль таблицы (Table Module, 148)**.

Как отмечалось в главе 2, "Организация бизнес-логики", главной движущей силой, обусловливающей подобный выбор, является сложность бизнес-логики — нечто, не поддающееся сколько-нибудь точной количественной (и даже качественной) оценке. Другие факторы (такие, как доступность соединения с базой данных) также играют определенную роль.

Самым простым является типовое решение **сценарий транзакции**, вполне согласующееся с процедурной моделью программирования, которая хорошо знакома практически всем. **Сценарий транзакции** прекрасно подходит для описания логики любой системной транзакции и легко реализуется в приложениях, взаимодействующих с реляционными базами данных. Основной недостаток решения состоит в том, что оно неприемлемо для описания сложной бизнес-логики и особенно восприимчиво к эффектам повторения фрагментов кода. Если речь идет, скажем, о простейшем приложении электронной коммерции, реализующем концепцию "карты покупателя" в условиях прозрачной ценовой модели, **сценарий транзакции** — это как раз то, что нужно. По мере усложнения логики трудности, сопровождающие применение **сценария транзакции**, возрастают по экспоненциальному закону.

На противоположном полюсе находится **модель предметной области**. Стойкие приверженцы парадигмы объектно-ориентированного программирования (такие, как я) не признают иных вариантов построения приложений. Если задача настолько проста, что для ее решения можно было бы написать **сценарий транзакции**, зачем тогда обращать на нее свое драгоценное внимание? Кроме того, мой достаточно обширный профессиональный опыт естественным образом велит заниматься действительно сложными вещами; и ничто не поможет совладать с пучиной бизнес-правил лучше, чем "богатая" **модель предметной области**. Если вы к ней привыкнете, то сможете эффективно применять ее и в более простых случаях.

Модель предметной области, однако, также страдает определенными изъянами. Прежде всего это сложность изучения и практического освоения. Поборники объектного стиля часто свысока взирают на "инакомыслящих", но это не мешает им прийти к объективному выводу о том, что аккуратное применение **модели предметной области** требует навыка, а небрежность здесь просто недопустима. Второе препятствие, осложняющее работу, — тесная связь **модели предметной области** с реляционной моделью данных. Конечно, самые фанатичные ревнители объектов ухитряются избавиться от проблемы путем перехода к объектным базам данных. Но по многим, большей частью нетехническим, причинам объектные базы данных нельзя считать разумной и просто реальной альтернативой реляционным системам, особенно в контексте корпоративных приложений. Так что объектные и реляционные модели хотя и сосуществуют, но уживаются с трудом, и это заметно усложняет типовые решения по отображению объектов в реляционные структуры.

Модуль таблицы представляет собой привлекательный островок вблизи экватора, разделяющего враждебные земли **сценария транзакции** и **модели предметной области**. Он лучше справляется с представлением бизнес-логики, нежели **сценарий транзакции**. Помимо того, не касаясь сложной бизнес-логики, **модуль таблицы** нормально соседствует с реляционными базами данных. Если ваша рабочая среда напоминает .NET, где многое сосредоточено вокруг всеобъемлющего **множества записей (Record Set, 523)**, тогда **модуль таблицы** ведет себя просто прекрасно, выгодно представляя бизнес-логику и удачно подчеркивая сильные стороны реляционной модели данных.

Это как раз тот случай, когда наличие инструментальных средств в той или иной мере определяет выбор архитектурных решений. Подчас наоборот — принципы архитектуры подсказывают, каким инструментам отдавать предпочтение, и, строго говоря, именно этим путем следует идти. На практике, однако, чаще приходится выбирать архитектуру с оглядкой на инструменты. Как раз таким решением и является **модуль таблицы** — его ценность во многом обусловлена доступностью подходящих средств разработки и исполнения кода.

Если вы читали рассуждения о бизнес-логике в главе 2, "Организация бизнес-логики", многое из того, о чем здесь идет речь, вам уже знакомо. Тем не менее я полагаю, что повторение оправданно, поскольку речь идет об основополагающем решении. Теперь перейдем к слою источника данных, не забывая, однако, о контексте выбранного варианта реализации слоя предметной области.

Источник данных

Избрав способ формирования слоя предметной области, необходимо определить, как подключить бизнес-логику к *источнику данных*. Принимаемые на этом этапе решения целиком зависят от типа слоя домена, а потому дальнейший материал целесообразно рассредоточить по соответствующим разделам.

Источник данных для сценария транзакции

Простейшие **сценарии транзакции (Transaction Script, 133)** могут содержать собственную логику взаимодействия с базой данных, но я избегал бы подобных решений даже в тривиальных случаях. Слой источника данных должен быть отделен от кода бизнес-логики, и я придерживаюсь этой позиции при проектировании приложений любой сложности. В данной ситуации можно применить одно из двух альтернативных типовых решений — **шлюз записи данных (Row Data Gateway, 175)** или **шлюз таблицы данных (Table Data Gateway, 167)**.

Выбор во многом зависит от особенностей рабочей платформы и возможных направлений развития разрабатываемой системы в будущем. **Шлюз записи данных** позволяет считывать каждую запись в объект с четко определенным интерфейсом. При использовании **шлюза таблицы данных** вам, возможно, удастся сократить объем кода, исключив необходимость явного извлечения данных, но интерфейс, реализующий доступ к структуре множества записей, лишится изрядной доли наглядности.

Ключевое решение определяется свойствами платформы. Например, наличие платформы, оснащенной разнообразными инструментами, которые поддерживают модель **множества записей (Record Set, 523)**, подталкивает к использованию **шлюза таблицы данных**.

В рассматриваемом контексте обычно нет нужды применять иные средства объектно-реляционного отображения. Проблемы структурного толка практически не возникают, поскольку структуры памяти хорошо отображаются в схему базы данных. Имеет смысл присмотреться к типовому решению **единица работы (Unit of Work, 205)**, но обычно проще следить за изменениями непосредственно в сценарии. Можно не беспокоиться и о большинстве проблем параллелизма, поскольку сценарий нередко в точности соответствует логике системной транзакции. Одно достаточно общее исключение связано с ситуацией, когда один запрос извлекает данные для редактирования, а другой предпринимает попытку сохранения внесенных изменений. В этом случае лучшим выбором наверняка окажется **оптимистическая автономная блокировка (Optimistic Offline Lock, 434)**, поскольку она проста в реализации и препятствует возникновению конфликтов, угрожающих взаимоблокировкой нескольких бизнес-транзакций.

Источник данных для модуля таблицы

Основной довод в пользу **модуля таблицы (Table Module, 148)** может быть связан с наличием хорошей инфраструктуры **множества записей**. В таком случае возникает потребность в типовом решении по объектно-реляционному отображению, и это обстоятельство неумолимо подводит нас к необходимости применения **шлюза таблицы данных**. Названные решения гармонично дополняют друг друга.

Больше, собственно говоря, ничего и не нужно. В своих лучших вариантах **множество записей** оснащено неким механизмом управления параллельными заданиями, что, по сути, превращает его в **единицу работы**, и это, безусловно, помогает разработчику сохранить нервные клетки и остатки шевелюры.

Источник данных для модели предметной области

Это уже любопытнее. Слабость **модели предметной области (Domain Model, 140)** во многом обусловлена усложнением аппарата взаимодействия с базой данных, и величина проблемы пропорциональна степени сложности модели.

Если **модель предметной области** проста и содержит, скажем, пару десятков классов, структура которых близка схеме базы данных, тогда имеет смысл прибегнуть к решению **активная запись (Active Record, 182)**. Если необходимо несколько ослабить зависимости, в этом поможет **шлюз таблицы данных** или **шлюз записи данных**.

По мере усложнения модели стоит обратиться к **преобразователю данных (Data Mapper, 187)**. Этот подход отвечает требованию обеспечения максимальной независимости **модели предметной области** от всех других слоев системы. Но следует иметь в виду, что **преобразователь данных** труднее реализовать на практике, нежели альтернативные типовые решения. Если у вас нет команды опытных профессионалов или вы не в состоянии отыскать некие упрощения модели, лучше воспользоваться готовыми инструментами объектно-реляционного отображения.

Как только будет отдано предпочтение **преобразователю данных**, в игру вступают многие другие типовые решения из области объектно-реляционного отображения. В частности, настоятельно советую обратить внимание на **единицу работы**, которая фокусирует в себе все функции управления параллельными операциями.

Слой представления

Слой представления относительно независим от выбора форм реализации нижележащих слоев системы. Первый вопрос касается того, какому типу интерфейса отдать предпочтение — толстому клиенту или HTML-обозревателю. Толстый клиент в состоянии обеспечить более привлекательную и разнообразную интерфейсную графику, но такой выбор сопряжен с дополнительными расходами по внедрению и сопровождению клиентских частей системы. Я выбрал бы интерфейс HTML-обозревателя, если бы его возможностей оказалось достаточно, и только в противном случае обратился бы к модели толстого клиента. Программирование интерфейса толстого клиента требует существенно больших усилий, причем это вызвано его изощренностью, а не какими-то внутренними технологическими трудностями.

В этой книге типовые решения из области проектирования интерфейсов толстого клиента не упоминаются. Поэтому, если вы сделаете именно такой выбор, помочь вам я не смогу.

Если же вы предпочтете вариант, связанный с HTML, вам придется решить, как структурировать приложение. В качестве основы я предложил бы типовое решение **модель–представление–контроллер (Model View Controller, 347)**. Тогда вам останется лишь определить формы контроллера и представления.

Решение во многом может быть обусловлено наличием в вашем распоряжении тех или иных инструментальных средств. Если, например, вы пользуетесь Visual Studio, простейший вариант действий — применить **контроллер страниц (Page Controller, 350)** вместе с **представлением по шаблону (Template View, 368)**. Если вы программируете на Java, изучите возможность использования соответствующих инструментальных оболочек. Сегодня особой популярностью пользуется Struts, и такой выбор подведет вас непосредственно к паре решений — **контроллер запросов (Front Controller, 362)** и **представление по шаблону**.

Если ваш выбор менее ограничен, а проектируемое приложение в большей степени ориентировано на представление документов, причем в виде смеси статических и динамических Web-страниц, я рекомендовал бы **контроллер страниц**. Более сложные условия навигации и повышенные требования к графическому интерфейсу предполагают использование **контроллера запросов**.

Выбор между **представлением по шаблону** и **представлением с преобразованием (Transform View, 379)** зависит от того, какой инструмент программирования — страницы сервера или XSLT — вы применяете. Сейчас более популярно **представление по шаблону**, хотя мне импонирует относительная простота тестирования, обеспечиваемая **представлением с преобразованием**. Если ваш сайт должен поддерживать возможности доступа с помощью разнообразных обозревателей, следует воспользоваться **двухэтапным представлением (Two Step View, 383)**.

Способ взаимодействия слоя представления с нижележащими слоями зависит от типа последних и от того, функционируют ли они в рамках единого процесса. Я предпочитаю, чтобы по мере возможности весь код работал в контексте одного процесса — это позволяет избежать трудностей, связанных со значительным замедлением обработки вызовов. Если же вы вынуждены использовать несколько процессов, поверх слоя предметной области следует расположить **интерфейс удаленного доступа (Remote Facade, 405)**, а для взаимообмена информацией с Web-сервером — применять **объекты переноса данных (Data Transfer Object, 419)**.

Платформы и инструменты

На страницах этой книги я все время пытаюсь продемонстрировать опыт выполнения проектов на множестве различных платформ. Навыки работы с Forte, CORBA и Smalltalk вполне применимы при использовании Java и .NET. Единственная причина, по которой я уделяю повышенное внимание средам Java и .NET, связана с тем, что они, по общему мнению, являются самыми многообещающими платформами, пригодными для разработки и внедрения корпоративных приложений сегодня и в будущем. (В этом ряду хотелось бы видеть и динамично развивающиеся современные языки сценариев, подобные Python и Ruby, и потому необходимо дать им шанс.)

Попытаемся рассмотреть те же проблемы с позиций двух упомянутых платформ. Впрочем, это не лишено определенной доли риска: технологии меняются значительно быстрее, нежели алгоритмы и типовые решения, поэтому не забывайте, что эти строки написаны в начале 2002 года.

Java и J2EE

Сегодня в сообществе Java самые серьезные споры посвящены тому, насколько ценной следует считать компонентную модель Enterprise Java Beans. После многочисленных "финальных" черновых вариантов (их, кажется, было не меньше, чем прощальных концертов группы *The Who*) спецификация EJB 2.0 наконец увидела свет. Но для конструирования хорошего J2EE-приложения применять компоненты EJB вовсе *не* обязательно, что бы вам ни говорили их поставщики. Вполне можно обойтись моделью POJO (Plain Old Java Objects) в сочетании с JDBC.

Проектные альтернативы для платформы J2EE варьируются в зависимости от применяемых типовых решений и во многом определяются способами реализации бизнес-логики.

Если поверх некоторого **шлюза (Gateway, 483)** функционирует **сценарий транзакции (Transaction Script, 133)**, общеупотребительным подходом, связанным с технологией EJB, в настоящий момент является такой: реализовать **сценарий транзакции** на основе *компонентов сеанса (session beans)*, а *компоненты сущностей (entity beans)* использовать как **шлюз записи данных (Row Data Gateway, 175)**. Если бизнес-логика достаточно проста, такую архитектуру следует считать наиболее разумной. Существует и одна проблема: от сервера EJB очень трудно избавиться, если решение перестает вас удовлетворять по технологическим (или финансовым) причинам. Подход "без EJB" — это объекты POJO для **сценария транзакции**, выполняемого поверх **шлюза записи данных** или **шлюза таблицы данных (Table Data Gateway, 167)**. Если более приемлемы множества записей формата JDBC 2.0, это повод для использования их в виде **множеств записей (Record Set, 523)** в контексте **шлюза таблицы данных**. Если вы еще не приняли окончательного решения о необходимости использования EJB, реализуйте подход без EJB, применив компоненты сеанса в качестве **интерфейса удаленного доступа (Remote Facade, 405)** к компонентам сущностей.

Если выбрана **модель предметной области (Domain Model, 140)**, современные "учения" указывают на необходимость применять компоненты сущностей. Если **модель предметной области** проста и в достаточной мере удачно соотносится со схемой базы данных, такой подход вполне разумен, а компоненты сущностей будут представлять собой **активные записи (Active Record, 182)**. Также неплохо окружать компоненты сущностей оболочками

в виде компонентов сеанса, действующих как **интерфейсы удаленного доступа**. Если, однако, **модель предметной области** более сложна, целесообразно обеспечить ее полную независимость от структуры компонентов EJB, чтобы получить возможность конструировать, выполнять и тестировать бизнес-логику без оглядки на причуды EJB-контейнера. В такой схеме для реализации **модели предметной области** лучше всего использовать объекты POJO с оболочками в форме компонентов сеанса, функционирующих в роли **интерфейсов удаленного доступа**. Если вы решили не пользоваться компонентами EJB, можно разместить все приложение на Web-сервере, чтобы избежать удаленных вызовов между слоями представления и бизнес-логики. Если **модель предметной области** основана на модели объектов POJO, последнюю можно применить и для реализации **преобразователей данных (Data Mapper, 187)** — либо с привлечением неких готовых инструментов объектно-реляционного отображения, либо с помощью доморощенных средств, разработка которых, возможно, оказалась бы оправданной и уместной.

Применяя компоненты сущностей в любом контексте, *не* снабжайте их удаленными интерфейсами. Я не вижу смысла в таких компонентах. Компоненты сущностей обычно применяются как **модели предметной области** или **шлюзы записи данных**. Чтобы они хорошо выполняли возложенные на них функции, следует снабдить их интерфейсами с высокой степенью детализации. Надеюсь, я уже прожужжал вам уши по поводу того, что удаленный интерфейс должен быть "огрубленным", с низкой степенью детализации, так что компоненты сущностей следует ориентировать только на локальное применение. (Единственное известное мне исключение — это типовое решение **составная сущность (Composite Entity)**, описанное в [3], которое представляет альтернативный способ использования компонентов сущностей и, откровенно говоря, мне не очень нравится.)

На данный момент типовое решение **модуль таблицы (Table Module, 148)** не завоевало общего признания в мире Java. Было бы интересно понаблюдать, что произойдет, если множество записей формата JDBC окружить дополнительными службами; тогда решение, вероятно, выглядело бы вполне жизнеспособным. В такой ситуации лучше других подошла бы модель POJO; кроме того, можно было бы заключить **модуль таблицы** в оболочку в виде компонентов сеанса, действующих как **интерфейсы удаленного доступа** и возвращающих **множества записей**.

.NET

Если присмотреться к .NET, Visual Studio и к истории развития инструментальных средств разработки от Microsoft в целом, становится понятно, что здесь преобладает типовое решение **модуль таблицы**, представляющее собой компромисс между **сценарием транзакции**, с одной стороны, и **моделью предметной области** — с другой, а также пополненное впечатляющим набором инструментов, которые позволяют воспользоваться всеми преимуществами вездесущей модели **множества записей**.

Итак, решение **модуль таблицы** на платформе Microsoft трактуется как выбор, предлагаемый по умолчанию. В самом деле, весомых причин для применения **сценариев транзакции** просто нет, за исключением самых простых случаев, но и здесь сценарии должны обрабатывать и возвращать **множества записей**.

Впрочем, это не значит, что можно слепо отказаться от **модели предметной области**. В .NET удается конструировать **модели предметной области** так же легко, как и в любой другой объектно-ориентированной среде. Однако в этом случае от инструментов разработки нельзя ожидать такой же безграничной помощи, какая доступна при реализации

модулей таблицы, поэтому, прежде чем бросаться к **модели предметной области**, следовало бы до какого-то момента мириться с теми потенциальными дополнительными трудностями, которые обусловлены выбором решения **модуль таблицы**.

Сейчас горячей темой в .NET является все, что имеет отношение к Web-службам. Но не стоит пользоваться Web-службами внутри приложения; лучше обращаться к ним, как в Java, в слое представления, чтобы облегчить возможную интеграцию кода. Реальных причин для разнесения кода Web-сервера и бизнес-логики по отдельным процессам в рамках .NET-приложения нет, так что применение **интерфейса удаленного доступа** в данном случае вряд ли оправданно.

Хранимые процедуры

Использовать хранимые процедуры или нет — этот вопрос никогда не терял актуальности. Хранимые процедуры зачастую оказываются оптимальным вариантом, поскольку выполняются в контексте процесса системы баз данных и потому позволяют избежать пресловутых неповоротливых удаленных вызовов. Однако большинство сред поддержки хранимых процедур не способно обеспечить приемлемых средств структуризации кода. Помимо того, перенос логики приложения в хранимые процедуры означает, что вы до определенной степени связываете себя с конкретным поставщиком СУБД. (Удачным примером решения подобных проблем служит подход компании Oracle, позволяющий выполнять Java-приложения в контексте процесса СУБД; это равнозначно размещению всего слоя предметной области в базе данных. На текущий момент зависимость от поставщика СУБД все еще сохраняется, но зато исключаются затраты на перенос кода.)

Ввиду низкой степени переносимости кода хранимых процедур и отсутствия надлежащих инструментов их структуризации многие разработчики просто избегают их использовать для описания бизнес-логики. Я склоняюсь к той же точке зрения, если только не вижу серьезных мотивов для повышения производительности (хотя таковые появляются, надо признать, довольно часто). В подобных ситуациях я беру метод слоя предметной области и благополучно перемещаю его в соответствующую хранимую процедуру, причем поступаю таким образом только из соображений повышения быстродействия приложения, трактуя принятое решение как оптимизационное, а не архитектурное. (В [30] приведены и другие аргументы в пользу более широкого применения аппарата хранимых процедур.)

Общий способ использования хранимых процедур сопряжен с контролем доступа к базе данных и реализацией некоего **шлюза таблицы данных**. По этому поводу у меня нет каких-то собственных сложившихся убеждений. Могу сказать одно: я предпочитаю изолировать доступ к базе данных с помощью одних и тех же типовых решений, независимо от того, каким образом он реализуется — посредством хранимых процедур или более традиционных конструкций SQL.

Web-службы

Во время работы над этой книгой среди законодателей мод в мире программного обеспечения бытовало общее мнение о том, что Web-службы вскоре превратят хрустальную мечту о повторном использовании кода в реальность и вытеснят бизнес системной интеграции с рынка, но в данном случае меня это не очень заботит. В контексте рассматриваемых типовых решений Web-службам отводится всего лишь вспомогательная роль:

они имеют отношение больше к системной интеграции, нежели к конструированию приложений. Вам не стоит без особой нужды пытаться расчленять цельное приложение на отдельные Web-службы, взаимодействующие друг с другом. Лучше спроектировать приложение и представить те или иные его части как Web-службы, трактуя их как **интерфейсы удаленного доступа**. И прежде всего позаботьтесь о том, чтобы все эти байки о "чрезвычайной простоте" Web-служб не заставили вас забыть *Первый Закон Распределения Объектов* (см. главу 7, стр. 113).

Хотя в большинстве опубликованных примеров кода, которые мне доводилось видеть, Web-службы используются синхронно, почти как вызовы XML RPC, мне кажется более предпочтительной асинхронная модель, основанная на сообщениях. У меня нет соответствующих типовых решений (эта книга и без того достаточно велика), но я ожидаю их появления уже в ближайшие несколько лет.

Другие модели слоев

Мое изложение строится на основе модели трех слоев, но это отнюдь не единственная модель, заслуживающая доверия. В других серьезных книгах по архитектуре программных систем предлагаются альтернативные схемы "расслоения" кода, и все они достойны внимания. Поэтому полезно с ними познакомиться и соотнести с тем, о чем говорится здесь. Возможно, для ваших приложений они окажутся более подходящими.

Прежде всего рассмотрим так называемую модель *Брауна* [9]. Модель охватывает пять слоев: *представление*, *контроллер/медиатор*, *домен* (*предметная область*, *бизнес-логика*), *отображение данных* и *источник данных* (табл. 8.1). Два дополнительных слоя, по существу, выполняют посреднические функции между базовыми слоями: контроллер/медиатор соединяет слои представления и домена, а слой отображения данных служит связующим звеном между предметной областью и источником данных.

Таблица 8.1. *Слои по Брауну*

Браун	Фаулер
Представление	Представление
Контроллер/медиатор	Представление (**контроллер приложения — Application Controller, 397**)
Домен	Домен
Отображение данных	Источник данных (**преобразователь данных — Data Mapper, 187**)
Источник данных	Источник данных

Я полагаю, что промежуточные слои полезны только иногда, поэтому в моей модели им отводится роль типовых решений: так, **контроллер приложения** — это посредник между представлением и бизнес-логикой, а **преобразователь данных** — прослойка между источником данных и доменом. **Контроллеру приложения** отведено место в главе 14,

"Типовые решения, предназначенные для представления данных в Web", а **преобразователь данных** описан в главе 10, "Архитектурные типовые решения источников данных".

По моему мнению, промежуточные слои — часто, но *не* заведомо полезные — надлежит воспринимать только как факультативное проектное решение. Я исповедую следующий подход: если какой-либо из трех базовых слоев переходит разумную грань сложности, модель *можно* пополнить дополнительным слоем, принимающим на себя избыток функций.

Хорошая схема расслоения кода приложений на платформе J2EE реализована в виде набора типовых решений *Core J2EE* [3]. Здесь различаются следующие слои: *клиент*, *представление*, *бизнес*, *интеграция* и *ресурсы* (табл. 8.2). Для слоев бизнеса и интеграции существуют прямые аналоги в нашей схеме. Слой ресурсов представляет внешние службы, с которыми соединен слой интеграции. Но основное отличие связано с расщеплением нашего слоя представления на две части между клиентом (слой клиента) и сервером (собственно слой представления). Это полезный ход, но и он, как нетрудно догадаться, далеко не всегда отвечает реальному положению вещей.

Таблица 8.2. *Слои Core J2EE*

Core J2EE	Фаулер
Клиент	Часть слоя представления, функционирующая в контексте клиента (например, в системах с толстыми клиентами)
Представление	Часть слоя представления, функционирующая на сервере (например, обработчики HTTP, страницы сервера)
Бизнес	Домен
Интеграция	Источник данных
Ресурсы	Внешние ресурсы, к которым обращается источник данных

В архитектуре *Microsoft DNA* [24] различают три слоя — *представление*, *бизнес* и *доступ к данным*, которые довольно точно отвечают трем слоям, рассматриваемым в книге (табл. 8.3). Наибольшее отличие состоит в способе передачи информации от слоя доступа к данным. В Microsoft DNA все слои имеют дело с множествами записей, возвращаемыми SQL-запросами, которые инициировались кодом слоя доступа к данным. Отсюда следует, что слои бизнеса и представления непосредственно "осведомлены" о существовании базы данных.

Таблица 8.3. *Слои Microsoft DNA*

Microsoft DNA	Фаулер
Представление	Представление
Бизнес	Домен
Доступ к данным	Источник данных

Это можно объяснить следующим образом: множество записей DNA действует как **объект переноса данных (Data Transfer Object, 419)** между слоями. Слой бизнеса способен модифицировать множества записей на пути их следования к слою представления и даже создавать собственные множества (впрочем, последнее случается редко). Хотя подобная форма коммуникаций довольно громоздка, она обладает тем немаловажным преимуществом, что позволяет слою представления пользоваться интерфейсными элементами управления, содержащими актуальную информацию, в том числе и измененную кодом слоя бизнеса.

Слой домена в этом случае структурируется в форме **модулей таблицы (Table Module, 148)**, а источник данных приобретает вид **шлюзов таблицы данных (Table Data Gateway, 167)**.

Схема *Маринеску* [28] содержит пять слоев (табл. 8.4). Слой представления расщеплен на два слоя, отображающих структуру **контроллера приложения**. Домен также подвергся расщеплению: на основе **модели предметной области (Domain Model, 140)** сконструирован **слой служб (Service Layer, 156)**, что соответствует обычной практике деления бизнес-логики на две части. Это общий подход, подкрепленный ограничениями модели EJB как **модели предметной области** (за подробностями обращайтесь к с. 140).

Таблица 8.4. *Слои по Маринеску*

Маринеску	Фаулер
Представление	Представление
Приложение	Представление (**контроллер приложения**)
Службы	Домен (**слой служб**)
Домен	Домен (**модель предметной области**)
Сохранение данных	Источник данных

Идея вычленения слоя служб из слоя предметной области основана на подходе, предполагающем возможность отмежевания логики процесса от "чистой" бизнес-логики. Уровень служб обычно охватывает логику, которая относится к конкретному варианту использования системы или обеспечивает взаимодействие с другими инфраструктурами (например, с помощью механизма сообщений). Стоит ли иметь отдельные слои служб и предметной области — вопрос, достойный обсуждения. Я склоняюсь к мысли о том, что подобное решение *может* оказаться полезным, хотя и не всегда, но некоторые уважаемые мною коллеги эту точку зрения не разделяют.

В [30] предложена самая сложная схема расслоения программной системы (табл. 8.5). *Нильссон* предусматривает активное использование хранимых процедур и поощряет размещение в них фрагментов бизнес-логики для повышения производительности приложения. Возможность вынесения бизнес-логики в хранимые модули меня не прельщает, поскольку она чревата усложнением процедур сопровождения кода. Впрочем, иногда такой подход совершенно оправдан и полезен. Слои хранимых процедур Нильссона содержат как источники данных, так и бизнес-логику.

Таблица 8.5. *Слои по Нильссону*

Нильссон	Фаулер
Потребитель	Представление
Вспомогательный слой потребителя	Представление (**контроллер приложения**)
Приложение	Домен (**слой служб**)
Домен	Домен (**модель предметной области**)
Доступ к хранимым данным	Источник данных
Общедоступные хранимые процедуры	Источник данных (возможно, с частью бизнес-логики)
Приватные хранимые процедуры	Источник данных (возможно, с частью бизнес-логики)

Как и Маринеску, для описания бизнес-логики Нильссон использует отдельные слои приложения и домена. Он предлагает возможность отказа от слоя домена в малых системах, это вполне созвучно моему мнению о том, что для небольших систем **модель предметной области** несколько теряет свою ценность.

ЧАСТЬ II

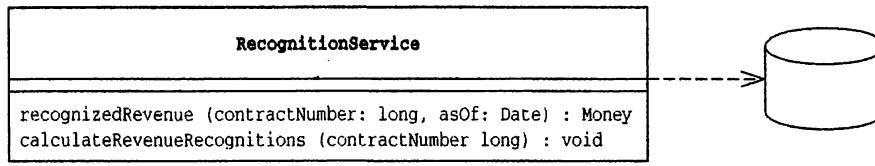
Типовые решения

Глава 9

Представление бизнес-логики

Сценарий транзакции (Transaction Script)

Способ организации бизнес-логики по процедурам, каждая из которых обслуживает один запрос, инициируемый слоем представления



Многие бизнес-приложения могут восприниматься как последовательности транзакций. Одна транзакция способна модифицировать данные, другая — воспринимать их в структурированном виде и т.д. Каждый акт взаимодействия клиента с сервером описывается определенным фрагментом логики. В одних случаях задача оказывается настолько же простой, как отображение части содержимого базы данных. В других могут предусматриваться многочисленные вычислительные и контрольные операции.

Сценарий транзакции организует логику вычислительного процесса преимущественно в виде единой процедуры, которая обращается к базе данных напрямую или при посредничестве кода тонкой оболочки. Каждой транзакции ставится в соответствие собственный **сценарий транзакции** (общие подзадачи могут быть вынесены в подчиненные процедуры).

Принцип действия

При использовании типового решения **сценарий транзакции** логика предметной области распределяется по транзакциям, выполняемым в системе. Если, например, пользователю необходимо заказать номер в гостинице, соответствующая процедура должна предусматривать действия по проверке наличия подходящего номера, вычислению суммы оплаты и фиксации заказа в базе данных.

Простые случаи не требуют особых объяснений. Разумеется, как и при написании иных программ, структурировать код по модулям следует осмысленно. Это не должно вызывать затруднений, если только транзакция не оказывается слишком сложной. Одно из основных преимуществ **сценария транзакции** заключается в том, что вам не приходится беспокоиться о наличии и вариантах функционирования других параллельных транзакций. Ваша задача — получить входную информацию, опросить базу данных, сделать выводы и сохранить результаты.

Где расположить **сценарий транзакции**, зависит от организации слоев системы. Этим местом может быть страница сервера, сценарий CGI или объект распределенного сеанса. Я предпочитаю обособлять **сценарии транзакции** настолько строго, насколько это возможно. В самом крайнем случае можно размещать их в различных подпрограммах, а лучше — в классах, отличных от тех, которые относятся к слоям представления и источника данных. Помимо того, следует избегать вызовов, направленных из **сценариев транзакции** к коду логики представления; это облегчит тестирование **сценариев транзакции** и их возможную модификацию.

Существует два способа разнесения кода **сценариев транзакции** по классам. Наиболее общий, прямолинейный и удобный во многих ситуациях — использование одного класса для реализации нескольких **сценариев транзакции**. Второй, следующий схеме типового решения **команда (Command)** [20], связан с разработкой собственного класса для каждого **сценария транзакции** (рис. 9.1): определяется тип, базовый по отношению ко всем **командам**, в котором предусматривается некий метод выполнения, удовлетворяющий логике **сценария транзакции**. Преимущество такого подхода — возможность манипулировать экземплярами сценариев как объектами в период выполнения, хотя в системах, где бизнес-логика организована с помощью **сценариев транзакции**, подобная потребность возникает сравнительно редко. Разумеется, во многих языках модель классов можно полностью игнорировать, полагаясь, скажем, только на глобальные функции. Однако вполне очевидно, что аппарат создания объектов помогает преодолевать проблемы потоков вычислений и облегчает изоляцию данных.

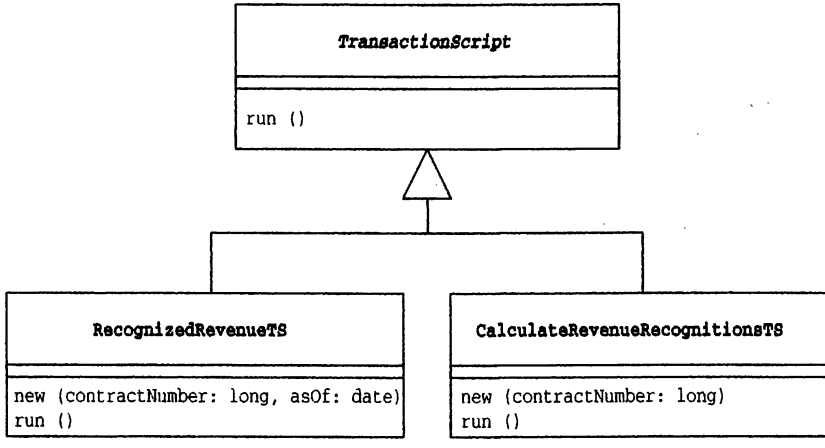


Рис. 9.1. Пример иерархии наследования классов, реализующих **сценарии транзакции**

Термин **сценарий транзакции** выбран потому, что в большинстве случаев приходится иметь дело с одним сценарием для каждой транзакции уровня системы баз данных. Пусть такой исход нельзя гарантировать на все сто процентов, но в первом приближении это верно.

Назначение

Главным достоинством типового решения **сценарий транзакции** является простота. Именно такой вид организации логики, эффективный с точки зрения восприятия и производительности, весьма характерен и естествен для небольших приложений.

По мере усложнения бизнес-логики становится все труднее содержать ее в хорошо структурированном виде. Одна из достойных внимания проблем связана с повторением фрагментов кода. Поскольку каждый **сценарий транзакции** призван обслуживать одну транзакцию, все общие порции кода неизбежно приходится воспроизводить вновь и вновь.

Проблема может быть частично решена за счет тщательного анализа кода, но наличие более сложной бизнес-логики требует применять **модель предметной области (Domain Model, 140)**. Последняя предлагает гораздо больше возможностей структурирования кода, повышения степени его удобочитаемости и уменьшения повторяемости.

Определить количественные критерии выбора конкретного типового решения довольно сложно, особенно если одни решения знакомы вам в большей степени, нежели другие. Проект, основанный на **сценарии транзакции**, с помощью техники рефакторинга (refactoring) вполне возможно преобразовать в реализацию **модели предметной области**, но дело слишком хлопотно, чтобы им стоило заниматься. Поэтому, чем раньше вы расставите все точки над *i*, тем лучше. Однако каким бы ярым приверженцем объектных технологий вы ни становились, не отбрасывайте **сценарий транзакции**: существует множество простых проблем, и их решения также должны быть простыми.

Задача определения зачтенного дохода

Рассматривая различные типовые решения по организации бизнес-логики, удобно воспользоваться одним и тем же примером. Чтобы не повторять формулировку проблемы несколько раз, я приведу ее только здесь.

Определение зачтенного дохода (revenue recognition) является довольно типичной задачей во многих отраслях бизнеса. Речь, по существу, идет о том, когда и как именно следует учитывать доходы от продажи товаров и услуг. Если я продаю вам чашку кофе, все просто: вы получаете продукт, я беру деньги и мгновенно фиксирую их в кассовом аппарате. Однако во многих случаях дело обстоит значительно сложнее. Предположим, вы платите мне гонорар за услуги, оказываемые на протяжении года. Каким бы образом плата ни вносилась (даже ежедневно), я, вероятно, не смогу ее учесть в своих бухгалтерских книгах непосредственно, поскольку договор, как мы условились, подписан на год. Можно, например, вести зачет, отображая двенадцатую часть суммы в каждом месяце.

Правила определения зачтенного дохода многочисленны, разнообразны и нестабильны. Одни устанавливаются законом, другие регламентируются профессиональными стандартами, третьи вытекают из корпоративной политики. Поэтому проблема в общей постановке может быть довольно сложной.

Но я не собираюсь вдаваться в детали. Представим компанию, которая продает три типа программных продуктов — текстовые процессоры, СУБД и электронные таблицы. Допустим, что в соответствии с правилами при продаже текстового процессора следует учитывать сразу всю сумму; если речь идет об электронных таблицах, третья часть суммы может быть отображена в бухгалтерской отчетности в день продажи, вторая треть — 60-ю днями позже, а оставшаяся треть — через 90 дней; если продана СУБД, схема зачета такова: треть суммы сегодня, треть по истечении 30 дней, а треть — через 60 дней. (Иных оснований, кроме моего больного воображения, у этого регламента нет, хотя реальные правила подчас ничуть не лучше.) Упрощенная схема определения зачтенного дохода представлена на рис. 9.2.

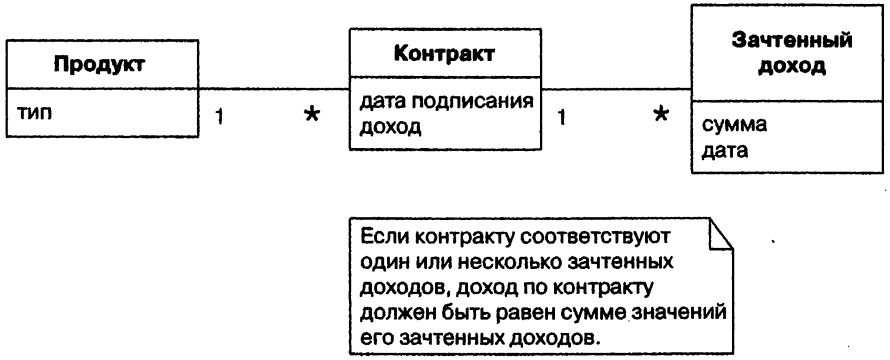


Рис. 9.2. Упрощенная схема определения зачтенного дохода: общий зачтенный доход по контракту складывается из нескольких значений

Пример: определение зачтенного дохода (Java)

В примере используются два **сценария транзакции**: один для вычисления зачтенного дохода по контракту, а второй — для определения части дохода по контракту, зачтенной к заданной дате. Схема базы данных содержит описания трех таблиц, предназначенных для хранения сведений о продуктах (`products`), контрактах (`contracts`) и зачтенных доходах (`revenueRecognitions`).

```
CREATE TABLE products (ID int primary key, name varchar,
↳type varchar)
CREATE TABLE contracts (ID int primary key, product int,
↳revenue decimal, dateSigned date)
CREATE TABLE revenueRecognitions (contract int, amount decimal,
↳recognizedOn date, PRIMARY KEY (contract, recognizedOn))
```

Первый сценарий вычисляет доход, зачтенный на определенную дату, путем поиска соответствующих записей в таблице `revenueRecognitions` и суммирования значений поля `amount`.

Во многих случаях **сценарии транзакции** содержат SQL-код, оперирующий информацией в базе данных напрямую. Здесь же в качестве оболочки SQL-запросов используется простой **шлюз таблицы данных (Table Data Gateway, 167)**. Поскольку пример слишком

прост, вместо отдельных шлюзов для каждой таблицы применяется один общий шлюз. Ниже показано, как может выглядеть метод поиска данных в шлюзе.

```
class Gateway...
   public ResultSet findRecognitionsFor(long contractID,
↳MfDate asof) throws SQLException {
      PreparedStatement stmt = db.prepareStatement(
↳findRecognitionsStatement);
      stmt.setLong(1,contractID);
      stmt.setDate(2, asof.toSqlDate());
      ResultSet result = stmt.executeQuery();
      return result;
   }
   private static final String findRecognitionsStatement =
      "SELECT amount " +
      "FROM revenueRecognitions " +
      "WHERE contract = ?AND recognizedOn <= ?";
   private Connection db;
```

Сценарий суммирования значений `amount`, возвращенных методом объекта шлюза, представлен ниже.

```
class RecognitionService...
   public Money recognizedRevenue(long contractNumber,
↳MfDate asOf) {
      Money result = Money.dollars(0);
      try {
         ResultSet rs = db.findRecognitionsFor(contractNumber,
↳asOf);
         while (rs.next()) {
            result = result.add(Money.dollars(rs.getBigDecimal(
↳"amount")));
         }
         return result;
      } catch (SQLException e) {throw
↳new ApplicationException(e);
      }
   }
```

Если процесс вычисления настолько прост, можно заменить сценарий, обрабатывающий данные в памяти, вызовом SQL-запроса с агрегирующей функцией, обеспечивающей суммирование значений `amount`.

Для вычисления зачтенного дохода по существующему контракту я применяю схожий вариант расщепления кода. Сценарий сосредоточивает в себе необходимую бизнес-логику.

```
class RecognitionService...
   public void calculateRevenueRecognitions(
↳long contractNumber) {
      try {
         ResultSet contracts = db.findContract(contractNumber);
         contracts.next();
```

```
        Money totalRevenue =
↳Money.dollars(contracts.getBigDecimal("revenue"));
        MfDate recognitionDate = new MfDate(contracts.getDate(
↳"dateSigned"));
        String type = contracts.getString("type");
        if (type.equals("S")) {
           Money[] allocation = totalRevenue.allocate(3);
           db.insertRecognition
              (contractNumber, allocation[0],
↳recognitionDate);
           db.insertRecognition
              (contractNumber, allocation[1],
↳recognitionDate.addDays(60));
           db.insertRecognition
              (contractNumber, allocation[2],
↳recognitionDate.addDays(90));
        } else if (type.equals("W")) {
           db.insertRecognition(contractNumber, totalRevenue,
↳recognitionDate);
        } else if (type.equals("D")) {
           Money[] allocation = totalRevenue.allocate(3);
           db.insertRecognition
              (contractNumber, allocation[0],
↳recognitionDate);
           db.insertRecognition
              (contractNumber, allocation[1],
↳recognitionDate.addDays(30));
           db.insertRecognition
              (contractNumber, allocation[2],
↳recognitionDate.addDays(60));
        }
     } catch (SQLException e) {throw
↳new ApplicationException(e);
     }
  }
```

Для создания объектов, представляющих денежные величины, используется типовое решение **деньги (Money, 502)**, ведь при делении денежной суммы так легко потерять копейку из-за ошибок округления!

Поддержка SQL обеспечивается с помощью **шлюза таблицы данных**. Ниже приведен метод поиска контракта.

```
class Gateway...
   public ResultSet findContract(long contractID)
↳throws SQLException {
      PreparedStatement stmt = db.prepareStatement(
↳findContractStatement);
      stmt.setLong(1, contractID);
      ResultSet result = stmt.executeQuery();
      return result;
   }
private static final String findContractStatement =
   "SELECT *" +
```

```
    "FROM contracts c, products p " +
    "WHERE ID = ?AND c.product = p.ID";
```

Далее приведена функция-оболочка SQL-команды вставки.

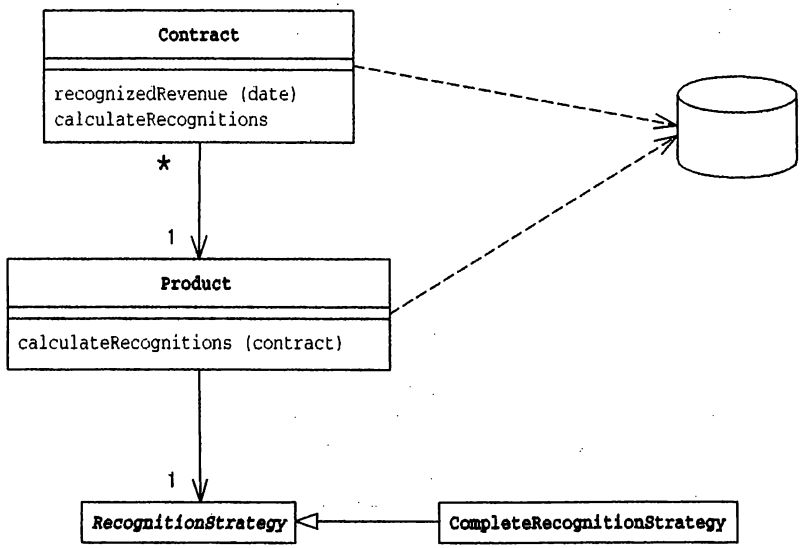
```
class Gateway...
    public void insertRecognition (long contractID,
↳Money amount, MfDate asof) throws SQLException {
        PreparedStatement stmt = db.prepareStatement(
↳insertRecognitionStatement);
        stmt.setLong(1, contractID);
        stmt.setBigDecimal(2, amount.amount());
        stmt.setDate(3, asof.toSqlDate());
        stmt.executeUpdate();
    }
    private static final String insertRecognitionStatement =
        "INSERT INTO revenueRecognitions VALUES (?, ?, ?)";
```

В системе Java служба определения зачтенного дохода ("recognition service") может быть обычным классом или компонентом сеанса (session bean).

Сопоставив все это с соответствующим примером, иллюстрирующим **модель предметной области**, вы убедитесь в том, что подход, основанный на **сценарии транзакции**, намного проще. Гораздо труднее представить, *что* произойдет при усложнении бизнес-логики. Правила бухгалтерского учета доходов весьма запутанны и часто зависят не только от типа продукта, но и от даты подписания контракта. По мере расширения набора правил и их усложнения код **сценария транзакции** становится все труднее содержать в порядке, и потому адепты объектного подхода (такие, как я) в подобных обстоятельствах предпочитают обращаться к **модели предметной области**.

Модель предметной области (Domain Model)

Объектная модель домена, охватывающая поведение (функции) и свойства (данные)



В своих наихудших проявлениях бизнес-логика бывает чрезвычайно сложной, с множеством правил и условий, оговаривающих различные варианты использования и особенности поведения системы. Для облегчения именно таких трудностей и предназначены объекты. Типовое решение **модель предметной области** предусматривает создание сети взаимосвязанных объектов, каждый из которых представляет некую осмысленную сущность — либо такую крупную, как промышленная корпорация, либо настолько мелкую, как строка формы заказа.

Принцип действия

Реализация **модели предметной области** означает пополнение приложения целым слоем объектов, описывающих различные стороны определенной области бизнеса. Одни объекты призваны имитировать элементы данных, которыми оперируют в этой области, а другие должны формализовать те или иные бизнес-правила. Функции тесно сочетаются с данными, которыми они манипулируют.

Объектно-ориентированная модель предметной области часто напоминает схему соответствующей базы данных, хотя между ними все еще остается множество различий. В **модели предметной области** смешиваются данные и функции, допускаются многозначные атрибуты, создаются сложные сети ассоциаций и используются связи наследования.

В сфере корпоративных программных приложений можно выделить две разновидности **моделей предметной области**. "Простая" во многом походит на схему базы данных

и содержит, как правило, по одному объекту домена в расчете на каждую таблицу. "Сложная" модель может отличаться от структуры базы данных и содержать иерархии наследования, стратегии и иные [20] типовые решения, а также сложные сети мелких взаимосвязанных объектов. Сложная модель более адекватно представляет запутанную бизнес-логику, но труднее поддается отображению в реляционную схему базы данных. В простых моделях подчас достаточно применять варианты тривиального типового решения **активная запись (Active Record, 182)**, в то время как в сложных без замысловатых **преобразователей данных (Data Mapper, 187)** порой просто не обойтись.

Бизнес-логика обычно подвержена частым изменениям, поэтому весьма важна возможность простой модификации и тестирования этого слоя кода. Отсюда следует настоятельная необходимость снижать степень зависимости **модели предметной области** от других слоев системы. Более того, как вы сможете убедиться, именно это требование является основополагающим аспектом многих типовых решений, имеющих отношение к "расслоению" системы.

С **моделью предметной области** связано большое количество различных контекстов. Простейший вариант — однопользовательское приложение, где единый граф объектов считывается из дискового файла и располагается в оперативной памяти. Такой стиль работы присущ настольным программам, но менее характерен для многоуровневых приложений, поскольку в них намного больше объектов. Размещение каждого объекта в памяти сопряжено с чрезмерными затратами ресурсов памяти и времени. Прелесть объектно-ориентированных систем баз данных заключается в том, что они создают впечатление, будто объекты пребывают в памяти постоянно.

Без такой системы заботиться о создании объектов вам придется самому. Обычно в ходе выполнения сеанса в память загружается полный граф объектов, хотя речь вовсе не идет обо *всех* объектах и, может быть, классах. Если, например, ведется поиск множества контрактов, достаточно считать информацию только о таких продуктах, которые упоминаются в этих контрактах. Если же в вычислениях участвуют объекты контрактов и зачтенных доходов, объекты продуктов, возможно, создавать вовсе не нужно. Точный перечень данных, загружаемых в память, определяется параметрами объектно-реляционного отображения.

Если в продолжение процесса обработки нескольких вызовов необходим один и тот же граф объектов, состояние сервера следует каким-то образом сохранять (подобным вопросам посвящена глава 6, "Сеансы и состояния").

Одна из типичных проблем бизнес-логики связана с чрезмерным увеличением объектов. Занимаясь конструированием интерфейсного экрана, позволяющего манипулировать заказами, вы наверняка заметите, что только некоторые функции отличаются сугубо специфическим характером и узким назначением. Возлагая на единственный класс заказа всю полноту ответственности, вы рискуете раздуть его до непомерной величины. Чтобы избежать подобного, можно выделить общие характеристики "заказов" и сосредоточить их в одноименном классе, а все остальные функции вынести во вспомогательные классы **сценариев транзакции (Transaction Script, 133)** или даже слоя представления.

При этом, однако, возникает опасность повторения фрагментов кода. Функции, не относящиеся к категории общих, отыскать довольно трудно, и многие предпочитают этим просто не заниматься, соглашаясь с дублированием кода. Повторение часто приводит к усложнению и несогласованности, хотя, по моему мнению, эффекты излишнего увеличения размеров классов наблюдаются значительно реже, чем можно было ожидать.

Если такое действительно происходит, результаты вполне очевидны и легко поправимы. Поэтому советую размещать весь родственный код в пределах одного класса и заниматься его разделением по нескольким классам только тогда, когда это в самом деле целесообразно.

Особенности Java-реализации

Замечено, что, когда начинается обсуждение разработки **модели предметной области** средствами J2EE, происходят резкие колебания орбиты Земли. Многие учебники и вводные курсы, посвященные тематике J2EE, для реализации **моделей предметной области** рекомендуют использовать компоненты сущностей (entity beans), хотя подобный подход, как известно, грешит некоторыми серьезными проблемами (это характерно по меньшей мере для текущей версии (2.0) спецификации Java).

Компоненты сущностей особенно полезны при использовании технологий управления хранением данных на уровне контейнера (Container Managed Persistence — CMP). Я сказал бы больше: вне контекста CMP применение компонентов сущностей практически лишено смысла. Однако CMP представляет собой слишком ограниченную форму объектно-реляционного отображения и не в состоянии обеспечить поддержку многих типовых решений, дополняющих сложную **модель предметной области**.

Компоненты сущностей не являются реентерабельными, т.е. не поддерживают возможность повторного вхождения. Иными словами, если компонент сущностей вызывает другой объект, этот (или следующий в цепочке ссылок) объект не в состоянии адресовать исходный компонент. В сложных **моделях предметной области** свойство реентерабельности используется довольно часто, поэтому отсутствие такового у компонентов сущностей представляет серьезное препятствие. Что еще хуже, образцы реентерабельного поведения бывает трудно распознать. Поэтому некоторые просто ограничиваются тем, что не позволяют одним компонентам сущностей вызывать другие. Такой подход исключает проблему реентерабельности, но во многом снижает достоинства **модели предметной области** как таковой.

В **модели предметной области** должны использоваться объекты и интерфейсы высокого уровня детализации. Компоненты сущностей могут вызываться в удаленном режиме (в соответствии с более ранними спецификациями Java такое условие было обязательным). Наличие удаленных объектов, снабженных детальными интерфейсами, значительно снижает производительность системы. Для преодоления проблемы достаточно использовать компоненты сущностей только в локальном режиме.

Для функционирования компонентам сущностей необходимы контейнер и подключение к базе данных. Выполнение этих условий требует дополнительного времени. Помимо того, код компонентов сущностей с большим трудом поддается отладке.

Приемлемой альтернативой являются обычные объекты Java, хотя это часто служит источником недоразумений — многие считают, будто такие объекты Java не способны работать в контексте EJB-контейнера. Полагаю, о традиционных объектах Java забывают только потому, что для них не придумали какого-то броского названия. Готовясь к докладу на конференции в 2000 году, Ребекка Парсонз (Rebecca Parsons), Джошуа Маккензи (Josh Mackenzie) и я таковое

изобрели: POJO — plain old Java objects. **Модель предметной области** на основе объектов POJO проста в реализации, независима от EJB и допускает возможность функционирования и тестирования вне EJB-контейнера (именно поэтому поставщики EJB не заинтересованы в успехе POJO).

Мое мнение таково: использование компонентов сущностей как инструментов реализации **модели предметной области** оправданно, если бизнес-логика в достаточной мере скромна. В этой ситуации можно построить модель, наделенную простыми связями с базой данных, и предусмотреть, скажем, по одному классу компонента сущностей на каждую таблицу. Если же бизнес-логика основана на механизме наследования, стратегиях и иных замысловатых типовых решениях, лучше прибегнуть к объектам POJO в сочетании с **преобразователями данных**, воспользовавшись коммерческими инструментами или доморощенным слоем кода.

Занимаясь сложной **моделью предметной области**, хочется оставаться в возможно более полной независимости от среды реализации, но технологии EJB требуют слишком серьезного внимания.

Назначение

Если вопросы *как*, касающиеся **модели предметной области**, трудны потому, что предмет чересчур велик, вопрос *когда* сложен ввиду неопределенности ситуации. Все зависит от степени сложности поведения системы. Если вам приходится иметь дело с изощренными и часто меняющимися бизнес-правилами, включающими проверки, вычисления и ветвления, вполне вероятно, что для их описания вы предпочтете объектную модель. Если, напротив, речь идет о паре сравнений значения с нулем и нескольких операциях сложения, проще прибегнуть к **сценарию транзакции (Transaction Script, 133)**.

Существует еще один фактор, который нельзя обойти вниманием: насколько комфортно чувствует себя команда разработчиков, манипулируя объектами домена. Изучение способов проектирования и применения **модели предметной области** — урок крайне сложный, пробудивший к жизни целый информационный пласт о "смене парадигмы". Чтобы привыкнуть к модели, нужны практика и советы профессионала, но зато среди тех, кто дошел до цели, мне почти не встречались такие, кто хотел бы вновь вернуться к **сценарию транзакции**, — разве только в самых простых случаях.

При необходимости взаимодействия с базой данных в контексте **модели предметной области** прежде всего я обратился бы к **преобразователю данных**. Это типовое решение поможет сохранить независимость бизнес-модели от схемы базы данных и обеспечить наилучшие возможности изменения их в будущем.

Встречаются ситуации, когда **модель предметной области** целесообразно снабдить более отчетливым интерфейсом API, и для этого можно порекомендовать типовое решение **слой служб (Service Layer, 156)**.

Дополнительные источники информации

Почти каждая книга об объектно-ориентированном проектировании так или иначе затрагивает **модели предметной области**, так как именно они зачастую находятся в центре внимания разработчиков прикладных систем.

Если вам нужно введение в объектные технологии, лучшим на сегодня, по моему мнению, является пособие [26]. За примерами **моделей предметной области** обращайтесь

к книге [17]. С проблематикой реляционной модели вас близко познакомит [21]. Чтобы сконструировать хорошую **модель предметной области**, необходимо правильно воспринимать объекты на концептуальном уровне. В этом смысле лучше других поможет книга [29]. Если хотите разобраться с типовыми решениями, примыкающими к **модели предметной области**, а также многими другими, используемыми в объектных системах различного назначения, непременно прочитайте классический труд [20].

Эрик Эванс (Eric Evans) занимается подготовкой книги [15], специально посвященной созданию **моделей предметной области**. Пока я видел только черновик рукописи, и выглядит она весьма многообещающе.

Пример: определение зачтенного дохода (Java)

Самые большие трудности в описании **модели предметной области** связаны с тем, что любые примеры должны быть достаточно просты, чтобы их можно было легко понять; но подобная простота не позволяет продемонстрировать сильные стороны модели — вы сможете ими воспользоваться только при решении по-настоящему сложной проблемы.

Но, если пример и не в состоянии представить убедительные доводы в пользу **модели предметной области**, он по крайней мере способен дать вам почувствовать, на что она может и должна быть похожа. Здесь я обращусь к той же задаче, связанной с определением зачтенного дохода.

Сразу замечу, что каждый класс в этом маленьком примере (рис. 9.3) содержит и функции и данные: например, скромный класс `RevenueRecognition` обладает двумя полями, конструктором и парой методов.

```
class RevenueRecognition...

   private Money amount;
   private MfDate date;
   public RevenueRecognition(Money amount, MfDate date) {
      this.amount = amount;
      this.date = date;
   }
   public Money getAmount() {
      return amount;
   }
   boolean isRecognizableBy(MfDate asOf) {
      return asOf.after(date) || asOf.equals(date);
   }
```

Для вычисления величины зачтенного дохода на определенную дату требуются классы, представляющие контракт и зачтенный доход.

```
class Contract...

   private List revenueRecognitions = new ArrayList();
   public Money recognizedRevenue(MfDate asOf) {
      Money result = Money.dollars(0);
      Iterator it = revenueRecognitions.iterator();
      while (it.hasNext()) {
         RevenueRecognition r = (RevenueRecognition)it.next();
         if (r.isRecognizableBy(asOf))
```

```
            result = result.add(r.getAmount());
        }
        return result;
    }
```

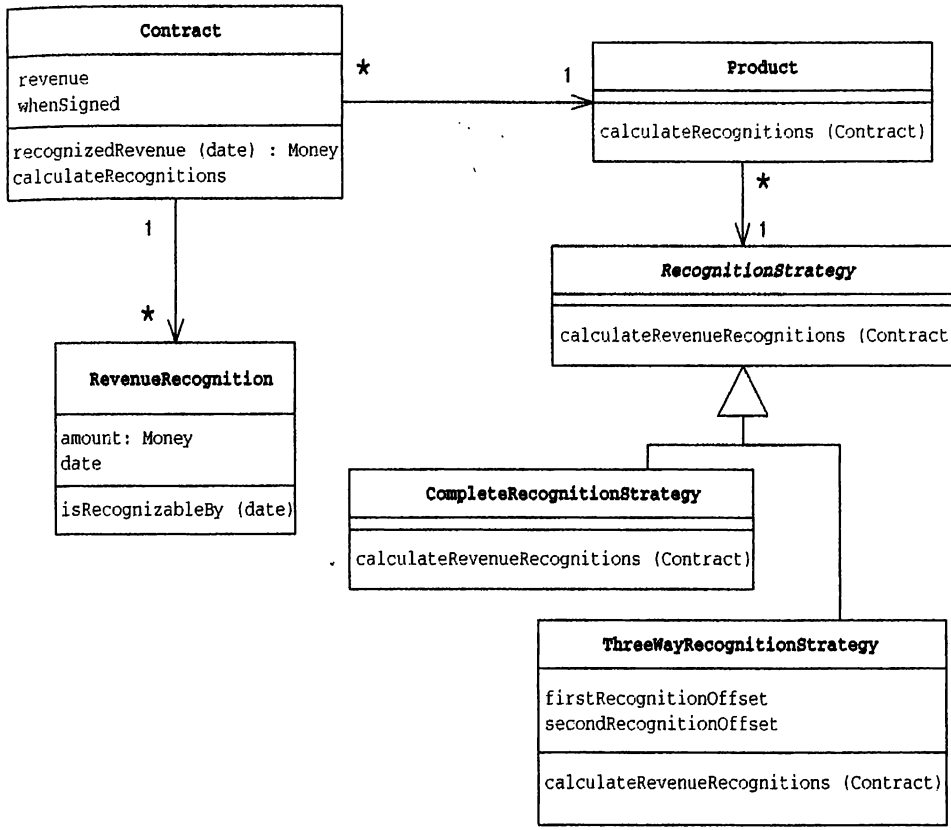


Рис. 9.3. Диаграмма классов для примера **модели предметной области**

Одна общая проблема, возникающая при использовании **моделей предметной области**, связана с тем, как многочисленным классам следует взаимодействовать при выполнении различных — даже простейших — операций. Часто высказывается небезосновательное недовольство по поводу того, что в процессе создания объектно-ориентированного приложения немало времени уходит на поиски и отслеживание подобных связей. Сосредоточив все родственные функции в пределах одного класса, можно уменьшить степень взаимозависимости объектов, пусть ценой повторения отдельных фрагментов кода.

Как видно из диаграммы на рис. 9.3, вычисление зачтенного дохода сопряжено с созданием множества мелких объектов: от "контракта" и "продукта" до иерархии стратегий определения зачтенного дохода. Типовое решение **стратегия (Strategy)** [20] — широко известный объектно-ориентированный подход, который позволяет распределить группу операций по иерархии небольших классов. Каждый экземпляр продукта соединяется с одним экземпляром стратегии вычисления, соответствующей определенному алгоритму

поиска зачтенного дохода. В нашем случае есть два производных класса стратегий. Структура кода может выглядеть, как показано ниже.

```
class Contract...

   private Product product;
   private Money revenue;
   private MfDate whenSigned;
   private Long id;
   public Contract(Product product, Money revenue,
↳MfDate whenSigned) {
      this.product = product;
      this.revenue = revenue;
      this.whenSigned = whenSigned;
   }

class Product...

   private String name;
   private RecognitionStrategy recognitionStrategy;
   public Product(String name, RecognitionStrategy
↳recognitionStrategy) {
      this.name = name;
      this.recognitionStrategy = recognitionStrategy;
   }
   public static Product newWordProcessor(String name) {
      return new Product(name, new
↳CompleteRecognitionStrategy());
   }
   public static Product newSpreadsheet(String name) {
      return new Product(name, new ThreeWayRecognitionStrategy(
↳60,90));
   }
   public static Product newDatabase(String name) {
      return new Product(name, new ThreeWayRecognitionStrategy(
↳30,60));
   }

class RecognitionStrategy...

   abstract void calculateRevenueRecognitions(Contract
↳contract);

class CompleteRecognitionStrategy...

   void calculateRevenueRecognitions(Contract contract) {
      contract.addRevenueRecognition(new RevenueRecognition(
↳contract.getRevenue(), contract.getWhenSigned()));
   }

class ThreeWayRecognitionStrategy...

   private int firstRecognitionOffset;
   private int secondRecognitionOffset;
```

```
   public ThreeWayRecognitionStrategy(int firstRecognitionOffset,
↳int secondRecognitionOffset)
   {
      this.firstRecognitionOffset = firstRecognitionOffset;
      this.secondRecognitionOffset = secondRecognitionOffset;
   }
   void calculateRevenueRecognitions(Contract contract) {
      Money[] allocation = contract.getRevenue().allocate(3);
      contract.addRevenueRecognition(new RevenueRecognition
↳(allocation[0], contract.getWhenSigned()));
      contract.addRevenueRecognition(new RevenueRecognition
↳(allocation[1],
↳contract.getWhenSigned().addDays(firstRecognitionOffset)));
      contract.addRevenueRecognition(new RevenueRecognition
↳(allocation[2],
↳contract.getWhenSigned().addDays(secondRecognitionOffset)));
   }
```

Ценность стратегий заключается в том, что они удачным образом обеспечивают возможности роста приложения. Например, для добавления очередного алгоритма определения зачтенного дохода достаточно создать новый производный класс и переопределить метод `calculateRevenueRecognitions`.

Когда конструируется объект продукта, к нему привязывается соответствующий объект стратегии. Для тестирования я применяю приведенный ниже код.

```
class Tester...

   private Product word = Product.newWordProcessor(
↳"Thinking Word");
   private Product calc = Product.newSpreadsheet(
↳"Thinking Calc");
   private Product db = Product.newDatabase("Thinking DB");
```

Как только значения заданы, вычисление удельных доходов уже не требует знаний о производных классах стратегий.

```
class Contract...

   public void calculateRecognitions() {
      product.calculateRevenueRecognitions(this);
   }

class Product...

   void calculateRevenueRecognitions(Contract contract) {
     recognitionStrategy.calculateRevenueRecognitions(contract);
   }
```

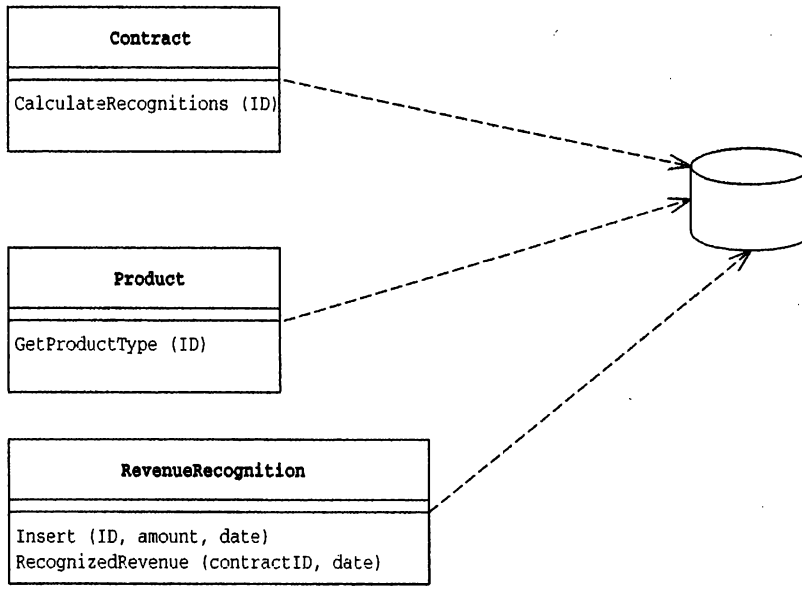
Прием, связанный с последовательной передачей управления от объекта к объекту, позволяет сосредоточить функции поведения системы в объекте, наиболее подходящем для этой цели, и во многом устраняет необходимость использования условных конструкций. Вы наверняка обратили внимание, что приведенный фрагмент кода вообще

не содержит условных операторов — направление вычислений задается в момент создания продукта. **Модели предметной области** особенно хороши, когда существует множество схожих условий протекания процесса, которые могут быть отображены в структуре объектов как таковых. В этом случае сложность алгоритмов вычислений перемещается на уровень связей между объектами. Чем более близка логика, тем с большей вероятностью различные части кода системы должны пользоваться одними и теми же связями. Любому алгоритму вычислений соответствует определенная сеть объектов.

Заметьте, что здесь ничего не говорилось о том, каким образом объекты создаются на основе содержимого базы данных и в ней сохраняются, и тому есть ряд причин. Во-первых, отображение **модели предметной области** в схему базы данных всегда является в той или иной мере сложной задачей. Во-вторых, назначение **модели предметной области** во многом состоит в сокрытии базы данных от верхних слоев системы.

Модуль таблицы (Table Module)

Объект, охватывающий логику обработки всех записей хранимой или виртуальной таблицы базы данных



Одна из ключевых предпосылок объектной модели — сочетание элементов данных и пользующихся ими функций. Традиционный объектно-ориентированный подход основан на концепции объектов с идентификационными признаками в совокупности с требованиями **модели предметной области (Domain Model, 140)**. Если, например, речь идет о классе, представляющем сущность "служащий", любой экземпляр класса соответствует определенному служащему; коль скоро есть ссылка на объект, отвечающий

служащему, с ним легко выполнять все необходимые операции, собирать информацию и следовать в направлении связей с другими объектами.

Одна из проблем **модели предметной области** заключается в сложности создания интерфейсов к реляционным базам данных; последние в подобной ситуации приобретают роль эдаких бедных родственников, с которыми никто не желает иметь дела. Поэтому считывание информации из базы данных и запись ее с необходимыми преобразованиями превращается в прихотливую игру ума.

Типовое решение **модуль таблицы** предусматривает создание по одному классу на каждую таблицу базы данных, и единственный экземпляр класса содержит всю логику обработки данных таблицы. Основное отличие **модуля таблицы** от **модели предметной области** состоит в том, что если, например, приложение обслуживает множество заказов, в соответствии с **моделью предметной области** придется сконструировать по одному объекту на каждый заказ, а при использовании **модуля таблицы** понадобится всего один объект, представляющий одновременно все заказы.

Принцип действия

Сильная сторона решения **модуль таблицы** заключается в том, что оно позволяет сочетать данные и функции для их обработки и в то же время эффективно использовать ресурсы реляционной базы данных. На первый взгляд **модуль таблицы** во многом напоминает обычный объект, но отличается тем, что не содержит какого бы то ни было упоминания об идентификационном признаке объекта. Если, скажем, требуется получить адрес служащего, для этого применяется метод `anEmployeeModule.GetAddress (long employeeID)`. В том случае, когда необходимо выполнить операцию, касающуюся определенного служащего, соответствующему методу следует передать ссылку на идентификатор, значение которого зачастую совпадает с первичным ключом служащего в таблице базы данных.

Модулю таблицы, как правило, отвечает некая табличная структура данных. Подобная информация обычно является результатом выполнения SQL-запроса и сохраняется в виде **множества записей (Record Set, 523)**. **Модуль таблицы** предоставляет обширный арсенал методов ее обработки. Объединение функций и данных обеспечивает многие преимущества модели инкапсуляции.

Нередко для решения общей задачи необходимо создать несколько **модулей таблицы** и, более того, позволить им манипулировать одним и тем же **множеством записей** (рис. 9.4).

Наиболее очевидный пример связан с использованием отдельных **модулей таблицы** для каждой (хранимой) таблицы базы данных. Кроме того, можно сконструировать **модули таблицы** для любых достойных SQL-запросов и виртуальных таблиц.

Конкретный **модуль таблицы** может принимать вид экземпляра класса или набора статических методов. Достоинство первого варианта состоит в том, что он позволяет инициировать модуль данными существующего множества записей, чаще всего получаемого в результате обработки SQL-запроса. Для манипуляции записями применяются методы класса. Не исключается и возможность создания иерархии наследования, когда, скажем, определяются базовый класс с описанием контракта общего вида и целое семейство производных классов, отвечающих частным разновидностям контрактов.

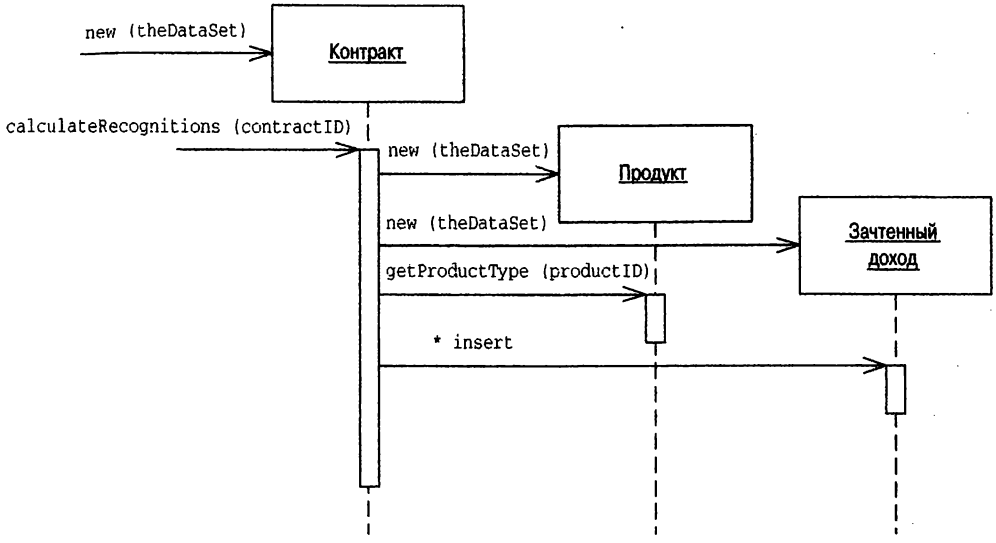


Рис. 9.4. Несколько **модулей таблицы** *способны обращаться к одному и тому же* **множеству записей**

Модуль таблицы способен содержать методы-оболочки, представляющие запросы к базе данных. Альтернативой служит **шлюз таблицы данных (Table Data Gateway, 167)**. Недостаток последнего обусловлен необходимостью конструирования дополнительного класса, а преимущество заключается в возможности применения единого **модуля таблицы** для данных из различных источников, поскольку каждому отвечает собственный **шлюз таблицы данных**.

Шлюз таблицы данных позволяет структурировать информацию в виде **множества записей**, которое затем передается конструктору **модуля таблицы** в качестве аргумента (рис. 9.5). Если необходимо использовать несколько **модулей таблицы**, все они могут быть созданы на основе одного и того же **множества записей**. Затем каждый **модуль таблицы** применяет к **множеству записей** функции бизнес-логики и передает измененное **множество записей** слою представления для отображения и редактирования информации средствами графических табличных элементов управления. Последние не "осведомлены", откуда поступили данные — непосредственно от реляционной СУБД или от промежуточного **модуля таблицы**, который успел осуществить их предварительную обработку. По завершении редактирования информация возвращается **модулю таблицы** для проверки перед сохранением в базе данных. Одно из преимуществ подобного стиля — возможность тестирования **модуля таблицы** путем "искусственного" создания **множества записей** в памяти без обращения к реальной таблице базы данных.

Слово "таблица" в названии типового решения подчеркивает, что в приложении предусматривается по одному **модулю таблицы** для каждой хранимой таблицы базы данных. Это правда, но не вся. Полезно также иметь **модули таблицы** для общеупотребительных виртуальных таблиц и запросов, поскольку структура **модуля таблицы** на самом деле не зависит напрямую от структуры физических таблиц базы данных, а определяется в основном характеристиками виртуальных таблиц и запросов, используемых в приложении.

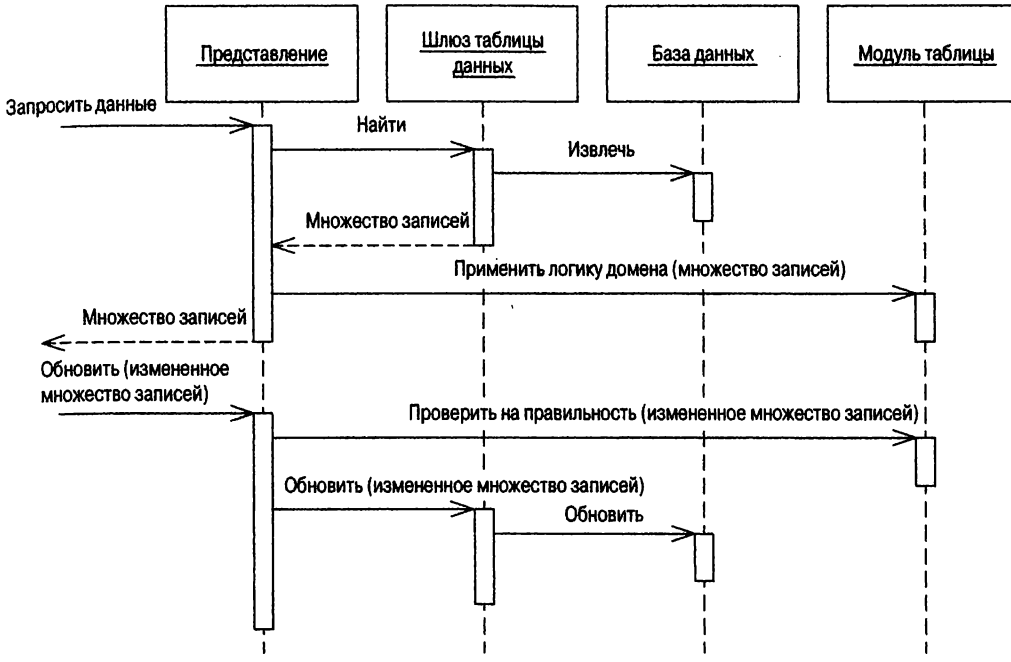


Рис. 9.5. Схема взаимодействия слоев кода с **модулем таблицы**

Назначение

Типовое решение **модуль таблицы** во многом основывается на табличной структуре данных и потому допускает очевидное применение в ситуациях, где доступ к информации обеспечивается при посредничестве **множеств записей**. Структуре данных отводится центральная роль, так что методы обращения к данным в структуре должны быть прямолинейными и эффективными.

Модуль таблицы, однако, не позволяет воспользоваться всей мощью объектного подхода к организации сложной бизнес-логики. Вам не удастся создать прямые связи от объекта к объекту, да и механизм полиморфизма действует в этих условиях не безукоризненно. Поэтому для реализации особо изощренной логики предпочтительнее **модель предметной области**. По существу, проблема сводится к поиску компромисса между способностью **модели предметной области** к эффективному отображению бизнес-логики и простотой интеграции кода приложения и реляционных структур данных, обеспечиваемой **модулем таблицы**.

Если объекты **модели предметной области** относительно точно отвечают таблицам базы данных, возможно, целесообразнее применить **модель предметной области** совместно с **активной записью (Active Record, 182)**. **Модуль таблицы**, однако, ведет себя лучше, чем комбинация **модели предметной области** и **активной записи**, если разные части приложения основаны на общей табличной структуре данных. Впрочем, в среде Java, например, **модуль таблицы** пока не пользуется популярностью, хотя с распространением модели множеств записей ситуация, возможно, и изменится.

Чаще всего образцы использования **модуля таблицы** приходится видеть в проектах на основе архитектуры Microsoft COM. В технологии COM (и .NET) **множество записей** представляет собой основное хранилище данных, с которыми оперирует приложение. **Множества записей** могут передаваться графическим элементам управления для воспроизведения информации на экране. Добротный механизм доступа к реляционным данным, представленным в виде **множеств записей**, реализован в семействе библиотек Microsoft ADO. В подобных случаях **модуль таблицы** позволяет описать бизнес-логику в хорошо структурированном виде.

Пример: определение зачтенного дохода (C#)

Настал черед вновь обратиться к задаче определения зачтенного дохода (см. раздел 9.1.3), но на сей раз в контексте **модуля таблицы**. Напомню, что цель состоит в вычислении зачтенного дохода ("revenue recognitions") по контракту ("contract") в зависимости от упомянутого в нем типа продукта ("product"). Здесь будем различать три типа продуктов — текстовый процессор (ниже в примере кода будем ссылаться на него с помощью значения `WP` (от word processor) перечислимого типа `ProductType`), электронная таблица (`SS` — spreadsheet) и СУБД (`DB` — database).

Модуль таблицы основывается на некой структуре данных, обычно реляционной (в будущем, вероятно, на эту роль сможет претендовать модель XML). В нашем случае реляционную схему допустимо изобразить так, как показано на рис. 9.6.

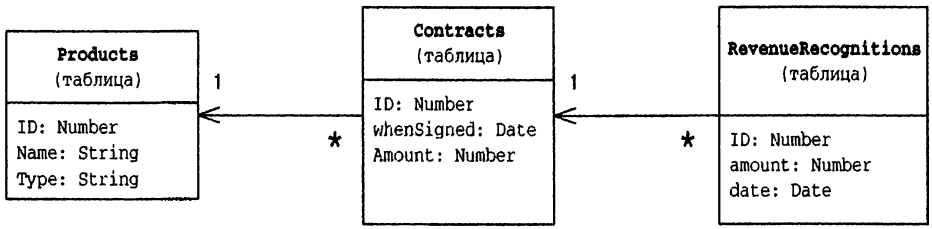


Рис. 9.6. Схема базы данных о контрактах, продуктах и зачтенном доходе

Классы, манипулирующие этими данными, следуют единой форме: каждой таблице отвечает определенный класс **модуля таблицы**. В архитектуре .NET можно создать объект множества данных, обеспечивающий представление структуры базы данных в памяти. Поэтому имеет смысл конструировать классы, способные оперировать множеством данных. Каждый класс **модуля таблицы** содержит элемент данных системного .NET-типа `DataTable`, который соответствует одной из таблиц множества данных. Способность считывать табличную информацию свойственна всем **модулям таблицы**, ею может обладать даже **супертип слоя (Layer Supertype, 491)**.

```
class TableModule...

   protected DataTable table;
   protected TableModule(DataSet ds, String tableName) {
      table = ds.Tables[tableName];
   }
```

Конструктор производного класса вызывает конструктор суперкласса и передает наименование конкретной таблицы.

```
class Contract...

   public Contract(DataSet ds) : base(ds, "Contracts") {}
```

Это позволяет создать новый модуль таблицы, просто передав множество данных соответствующему конструктору, синтаксис которого приведен ниже, и отделить код, конструирующий множество данных, от **модулей таблицы**, что отвечает рекомендациям спецификации ADO.NET.

```
contract = new Contract(dataset);
```

Очень удобен инструмент индексации, реализованный в C#, который позволяет "добраться" до определенной записи таблицы данных по значению первичного ключа.

```
class Contract...
   public DataRow this[long key ] {
      get {
         String filter = String.Format("ID ={0}", key);
         return table.Select(filter)[0];
      }
   }
```

Функции первой группы осуществляют вычисление значений зачтенного дохода по контракту и обновление соответствующей таблицы. Получаемые значения зависят от типа продукта. Поскольку эти операции основаны преимущественно на данных из таблицы контрактов, я решил добавить в класс `Contract` надлежащий метод.

```
class Contract...

   public void CalculateRecognitions(long contractID) {
      DataRow contractRow = this[contractID];
      Decimal amount = (Decimal)contractRow["amount"];
      RevenueRecognition rr = new RevenueRecognition(
↳table.DataSet);
      Product prod = new Product(table.DataSet);
      long prodID = GetProductId(contractID);
      if (prod.GetProductType(prodID) == ProductType.WP) {
         rr.Insert(contractID, amount, (DateTime)GetWhenSigned(
↳contractID));
      } else if (prod.GetProductType(prodID) ==
↳ProductType.SS) {
         Decimal[] allocation = allocate(amount, 3);
         rr.Insert(contractID,
↳allocation[0],(DateTime)GetWhenSigned(contractID));
         rr.Insert(contractID, allocation[1], (DateTime)
            GetWhenSigned(contractID).AddDays(60));
         rr.Insert(contractID, allocation[2], (DateTime)
            GetWhenSigned(contractID).AddDays(90));
      } else if (prod.GetProductType(prodID) ==
↳ProductType.DB) {
```

```
            Decimal[] allocation = allocate(amount, 3);
            rr.Insert(contractID, allocation[0],
↳(DateTime)GetWhenSigned(contractID));
            rr.Insert(contractID, allocation[1], (DateTime)
               GetWhenSigned(contractID).AddDays(30));
            rr.Insert(contractID, allocation[2], (DateTime)
               GetWhenSigned(contractID).AddDays(60));
         } else throw new Exception("invalid product id");
      }
      private Decimal[] allocate(Decimal amount, int by) {
         Decimal lowResult = amount / by;
         lowResult = Decimal.Round(lowResult, 2);
         Decimal highResult = lowResult + 0.01m;
         Decimal[] results = new Decimal[by];
         int remainder = (int)amount % by;
         for (int i = 0; i < remainder; i++) results[i] =
↳highResult;
         for (int i = remainder; i < by; i++) results[i] =
↳lowResult;
         return results;
      }
```

Здесь можно было бы применить типовое решение **деньги (Money, 502)**, но для разнообразия я показал, как работать с типом `Decimal`. Метод `allocate` схож с тем, который используется в контексте решения **деньги**.

Другие классы также необходимо снабдить соответствующими методами. Класс продуктов должен "уметь" сообщать о типе конкретного продукта. Этого можно добиться посредством соответствующих перечислимого типа и метода.

```
   public enum ProductType {WP, SS, DB};

class Product...

   public ProductType GetProductType(long id) {
      String typeCode = (String)this[id ]["type"];
      return (ProductType)Enum.Parse(typeof(ProductType),
↳typeCode);
   }
```

Метод `GetProductType` инкапсулирует информацию таблицы типа `DataTable`. Имеет смысл обеспечить инкапсуляцию всех столбцов данных, а не обращаться к ним напрямую, как я поступал с полем `amount` таблицы контрактов (хотя использовать модель инкапсуляции, вообще говоря, неплохо, в данном случае я от нее отказался, поскольку она вступала в противоречие с предположением о том, что различные части системы способны адресовать множество данных непосредственно). Использовать отдельные функции доступа к столбцам имеет смысл только тогда, когда необходимо выполнять дополнительные операции (например, преобразование строки в значение типа `ProductType`).

Здесь уместно напомнить и о том, что предпочтительнее пользоваться строго типизированными множествами данных .NET, хотя в этом случае я применил нетипизированное множество.

Еще одна функция связана со вставкой новой записи, описывающей зачтенный доход.

```
class RevenueRecognition...

   public long Insert(long contractID, Decimal amount,
↳DateTime date) {
      DataRow newRow = table.NewRow();
      long id = GetNextID();
      newRow["ID"] = id;
      newRow["contractID"] = contractID;
      newRow["amount"] = amount;
      newRow["date"] = String.Format("{0:s}", date);
      table.Rows.Add(newRow);
      return id;
   }
```

И вновь значение метода не столько в инкапсуляции записи данных, сколько в том, что он структурирует строки кода, позволяя избежать их неоднократного повторения.

Функции второй группы связаны с суммированием значений всех зачтенных доходов по контракту, полученных на определенную дату. Поскольку вычисления затрагивают таблицу зачтенного дохода, целесообразно определить подходящий метод в классе, отвечающем этой таблице.

```
class RevenueRecognition...

   public Decimal RecognizedRevenue(long contractID,
↳DateTime asOf) {
      String filter = String.Format("ContractID = {0} AND
↳date <= #{1:d}#", contractID, asOf);
      DataRow[] rows = table.Select(filter);
      Decimal result = 0m;
      foreach (DataRow row in rows) {
         result += (Decimal)row["amount"];
      }
      return result;
   }
```

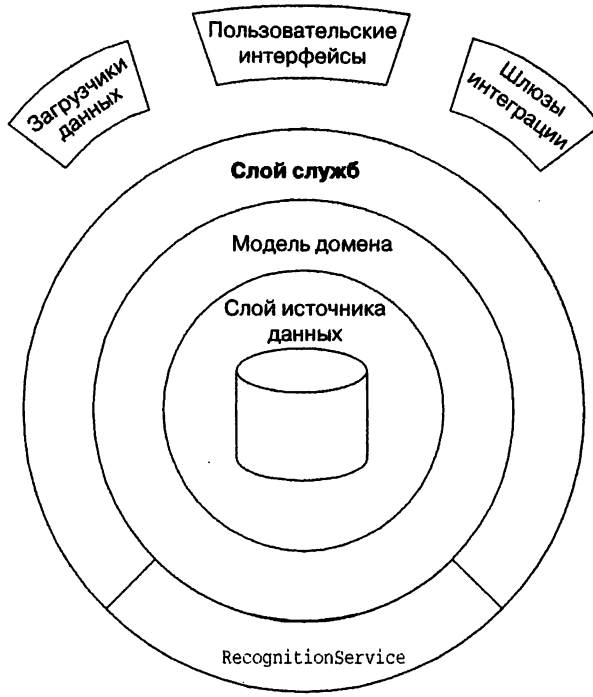
Этот фрагмент демонстрирует прекрасное средство ADO.NET, позволяющее определять строку предложения `WHERE` и выбирать необходимое подмножество записей таблицы типа `DataTable`. Ничто не мешает пойти еще дальше и воспользоваться агрегирующей функцией.

```
class RevenueRecognition...
   public Decimal RecognizedRevenue2 (long contractID,
↳DateTime asOf) {
      String filter = String.Format("ContractID = {0} AND
↳date <= #{1:d}#", contractID, asOf);
      String computeExpression = "sum(amount)";
      Object sum = table.Compute(computeExpression, filter);
      return (sum is System.DBNull) ? 0 : (Decimal)sum;
   }
```

Слой служб (Service Layer)

Рэнди Стаффорд

Схема определения границ приложения посредством слоя служб, который устанавливает множество доступных действий и координирует отклик приложения на каждое действие



Корпоративные приложения обычно подразумевают применение разного рода интерфейсов к хранимым данным и реализуемой логике — загрузчиков данных, интерфейсов пользователя, шлюзов интеграции и т.д. Несмотря на различия в назначении, подобные интерфейсы часто нуждаются в одних и тех же функциях взаимодействия с приложением для манипулирования данными и выполнения бизнес-логики. Функции могут быть весьма сложными и способны включать транзакции, охватывающие многочисленные ресурсы, а также операции по координации реакций на действия. Описание логики взаимодействия в каждом отдельно взятом интерфейсе сопряжено с многократным повторением одних и тех же фрагментов кода.

Слой служб определяет границы приложения [12] и множество операций, предоставляемых им для интерфейсных клиентских слоев кода. Он инкапсулирует бизнес-логику приложения, управляет транзакциями и координирует реакции на действия.

Принцип действия

Слой служб может быть реализован несколькими способами, удовлетворяющими всем упомянутым выше условиям. Различия проявляются в методах распределения ответственности вне интерфейса **слоя служб**. Прежде чем окунуться в особенности альтернативных реализаций, позвольте осветить некоторые основополагающие аспекты.

Разновидности "бизнес-логики"

Подобно **сценарию транзакции (Transaction Script, 133)** и **модели предметной области (Domain Model, 140)**, **слой служб** представляет собой типовое решение по организации бизнес-логики. Многие проектировщики, и я в том числе, любят разносить бизнес-логику по двум категориям: *логика домена (domain logic)* имеет дело только с предметной областью как таковой (примером могут служить стратегии вычисления зачтенного дохода по контракту), а *логика приложения (application logic)* описывает сферу ответственности приложения [11] (скажем, уведомляет пользователей и сторонние приложения о протекании процесса вычисления доходов). Логику приложения часто называют также "логикой рабочего процесса", несмотря на то что под "рабочим процессом" часто понимаются совершенно разные вещи.

Модель предметной области более предпочтительна в сравнении со **сценарием транзакции**, поскольку исключает возможность дублирования бизнес-логики и позволяет бороться со сложностью с помощью классических проектных решений. Но размещение логики приложения в "чистых" классах домена чревато нежелательными последствиями. Во-первых, классы домена допускают меньшую вероятность повторного использования, если они реализуют специфическую логику приложения и зависят от тех или иных прикладных инструментальных пакетов. Во-вторых, смешивание логики обеих категорий в контексте одних и тех же классов затрудняет возможность новой реализации логики приложения с помощью специфических инструментальных средств, если необходимость такого шага становится очевидной. По этим причинам **слой служб** предусматривает распределение "разной" логики по отдельным слоям, что обеспечивает традиционные преимущества расслоения, а также большую степень свободы применения классов домена в разных приложениях.

Варианты реализации

Двумя базовыми вариантами реализации **слоя служб** являются создание *интерфейса доступа к домену (domain facade)* и конструирование *сценария операции (operation script)*. При использовании подхода, связанного с интерфейсом доступа к домену, **слой служб** реализуется как набор "тонких" интерфейсов, размещенных "поверх" **модели предметной области**. В классах, реализующих интерфейсы, никакая бизнес-логика отражения не находит — она сосредоточена исключительно в контексте **модели предметной области**. Тонкие интерфейсы устанавливают границы и определяют множество операций, посредством которых клиентские слои взаимодействуют с приложением, обнаруживая тем самым характерные свойства **слоя служб**.

Создавая сценарий операции, вы реализуете **слой служб** как множество более "толстых" классов, которые непосредственно воплощают в себе логику приложения, но за бизнес-логикой обращаются к классам домена. Операции, предоставляемые клиентам **слоя служб**, реализуются в виде сценариев, создаваемых группами в контексте классов, каждый из которых определяет некоторый фрагмент соответствующей логики. Подобные классы, расширяющие **супертип слоя (Layer Supertype, 491)** и уточняющие объявленные в нем абстрактные характеристики поведения и сферы ответственности, формируют "службы" приложения (в названиях служебных типов принято употреблять суффикс "Service"). **Слой служб** и заключает в себе эти прикладные классы.

Быть или не быть удаленному доступу

Интерфейс любого класса **слоя служб** обладает низкой степенью детализации почти по определению, поскольку в нем объявляется набор прикладных операций, открытых для внешних интерфейсных клиентских слоев. Поэтому классы **слоя служб** хорошо приспособлены для удаленного доступа.

За удаленные вызовы приходится платить, применяя технологии распределения объектов. Чтобы обеспечить возможность совмещения методов **слоя служб** с контекстом **объектов переноса данных (Data Transfer Object, 419)**, требуется приложить немалые усилия. Не стоит недооценивать затраты на выполнение этой работы, особенно при использовании сложной **модели предметной области** и насыщенных интерфейсов редактирования данных для "навороченных" вариантов использования! Это весьма трудоемкое и хлопотное дело — по крайней мере второе по степени сложности после объектно-реляционного отображения. Да, и не забывайте о *Первом Законе Распределения Объектов* (см. главу 7, с.113).

Советую начинать со **слоя служб**, адресуемого в локальном режиме, и включать возможность удаленного вызова только при необходимости (если таковая вообще возникнет), размещая "поверх" **слоя служб** соответствующие **интерфейсы удаленного доступа (Remote Facade, 405)** или снабжая нужными интерфейсами объекты **слоя служб** как таковые. Если приложение обладает графическим Web-интерфейсом или использует шлюзы для интеграции, основанные на Web-службах, я не могут назвать правило, которое заставляло бы располагать бизнес-логику в контексте процесса, отделенного от страниц сервера или Web-служб. Отдавая предпочтение режиму совместного выполнения, вы сэкономите много сил и времени, не утратив возможность масштабирования.

Определение необходимых служб и операций

Установить, какие операции должны быть размещены в **слое служб**, отнюдь не сложно. Это определяется нуждами клиентов **слоя служб**, первой (и наиболее важной) из которых обычно является пользовательский интерфейс. Как известно, он предназначен для поддержки вариантов использования приложения, поэтому в качестве отправной точки для определения набора операций **слоя служб** должны рассматриваться модель вариантов использования и пользовательский интерфейс приложения.

Как это ни грустно, большинство вариантов использования корпоративных приложений составляют банальные операции CRUD (create, read, update, delete — создать, считать, обновить, удалить) над объектами домена: создать экземпляр того, считать коллекцию этих или обновить что-нибудь еще. На практике варианты использования CRUD практически всегда полностью соответствуют операциям **слоя служб**.

Несмотря на сказанное выше, обязанности приложения по обработке вариантов использования никак нельзя назвать банальными. Создание, обновление или удаление в приложении объекта домена, не говоря уже о проверке правильности его содержимого, требует отправки уведомлений пользователям или другим интегрированным приложениям. Координацией откликов и отправлением их в рамках атомарных транзакций и занимаются операции **слоя служб**.

К сожалению, определить, в какие абстракции **слоя служб** необходимо сгруппировать родственные операции, далеко не просто. Универсального рецепта для решения этой проблемы не существует. Небольшому приложению вполне может хватить одной абстракции, названной по имени самого приложения. Более крупные приложения обычно разбиваются на несколько подсистем, каждая из которых включает в себя вертикальный срез всех имеющихся архитектурных слоев. В этом случае я предпочитаю создавать по одной одноименной абстракции для каждой подсистемы. В качестве альтернатив можно предложить создание абстракций, отражающих основные составляющие модели предметной области, если таковые отличаются от подсистем (например, `ContractService`, `ProductsService`) или абстракций, названных в соответствии с типами поведения (например, `RecognitionService`).

Особенности Java-реализации

Оба подхода, упомянутых в предыдущем разделе, — создание интерфейса доступа к домену и построение сценария операции — допускают реализацию класса **слоя служб** в виде объекта POJO или же компонента сеанса, не имеющего состояний. К сожалению, какое бы решение вы не выбрали, вам обязательно придется чем-то пожертвовать — либо простотой тестирования, либо легкостью управления транзакциями. Так, объекты POJO легче тестировать, поскольку для их функционирования не нужны EJB-контейнеры. С другой стороны, **слою служб**, реализованному в виде объекта POJO, будет труднее работать со службами распределенных транзакций, управляемых на уровне контейнера, особенно при обмене данными между разными службами. Компоненты сеанса, напротив, хорошо справляются с распределенными транзакциями, управляемыми на уровне контейнера, однако могут быть протестированы и запущены только при наличии EJB-контейнера. Проще говоря, из двух зол приходится выбирать меньшее.

Я предпочитаю реализовать **слой служб** с использованием локальных интерфейсов и сценария операции в виде компонентов сеанса EJB 2.0, обращающихся за логикой домена к классам объектов домена POJO. Что ни говори, а **слой служб** очень удобно реализовать в виде компонента сеанса, не меняющего состояния, потому что в EJB предусмотрена поддержка распределенных транзакций, управляемых на уровне контейнера. Кроме того, с помощью локальных интерфейсов, появившихся в EJB 2.0, **слой служб** может использовать весьма ценные службы транзакций и в то же время избежать всех неприятностей, связанных с вопросами распределения объектов.

Говоря о Java, хочу обратить ваше внимание на отличие **слоя служб** от **интерфейса доступа посредством компонентов сеанса (Session Facade)** — типового решения для J2EE, описанного в [3, 28]. Разработчики **интерфейса доступа посредством компонентов сеанса** намеревались избежать падения производительности, связанной с чрезмерным количеством удаленных вызовов компонентов

сущностей; поэтому в данном типовом решении доступ к компонентам сущностей происходит с помощью компонентов сеанса. В отличие от этого, **слой служб** направлен на определение как можно большего количества общих операций, для того чтобы избежать дублирования кода и обеспечить возможность повторного использования; это архитектурное решение, которое выходит за рамки технологии. Действительно, типовое решение **граница приложения (Application Boundary)**, описанное в [12] и ставшее прототипом **слоя служб**, появилось примерно на три года раньше, чем EJB. **Интерфейс доступа посредством компонентов сеанса** имеет много общего со **слоем служб**, однако на данный момент его имя, назначение и позиционирование принципиально иные.

Назначение

Преимуществом использования **слоя служб** является возможность определения набора общих операций, доступных для применения многими категориями клиентов, и координация откликов приложения на выполнение каждой операции. В сложных случаях отклики могут включать в себя логику приложения, передаваемую в рамках атомарных транзакций с использованием нескольких ресурсов. Таким образом, если у бизнес-логики приложения есть более одной категории клиентов, а отклики на варианты использования передаются через несколько ресурсов транзакций, использование **слоя служб** с транзакциями, управляемыми на уровне контейнера, становится просто необходимым, даже если архитектура приложения не является распределенной.

Гораздо легче ответить на вопрос, когда **слой служб** *не нужно* использовать. Скорее всего, вам не понадобится **слой служб**, если у логики приложения есть только одна категория клиентов, например пользовательский интерфейс, отклики которого на варианты использования не охватывают несколько ресурсов транзакций. В этом случае управление транзакциями и выбор откликов можно возложить на **контроллеры страниц (Page Controller, 350)**, которые будут обращаться непосредственно к слою источника данных.

Тем не менее, как только у вас появится вторая категория клиентов или начнет использоваться второй ресурс транзакции, вам неизбежно придется ввести **слой служб**, что потребует полной переработки приложения.

Дополнительные источники информации

Слой служб имеет не так уж много прототипов. В его основу легло типовое решение **граница приложения (Application Boundary)**, описанное Алистером Кокберном (Alistair Cockburn) в [12]. В источнике [2], посвященном удаленным службам, обсуждается роль интерфейсов доступа в распределенных системах. Сравните эти концепции с описаниями типового решения **интерфейс доступа посредством компонентов сеанса (Session Facade)**, содержащимися в [3, 28]. Рассмотрение вариантов использования в соответствии с типами поведения, приведенное в справочнике [11], может помочь в определении обязанностей приложения, координацию которых должен проводить **слой служб**. Похожие идеи под названием "системных операций" представлены в более раннем справочнике [13], посвященном технологии Fusion.

Пример: определение зачтенного дохода (Java)

Продолжим рассмотрение примера определения зачтенного дохода, который был начат в разделах, посвященных типовым решениям **сценарий транзакции** и **модель предметной области**. На сей раз я продемонстрирую использование **слоя служб**, реализовав его операцию в виде сценария, который инкапсулирует в себе логику приложения и обращается за логикой домена к классам домена. Для реализации **слоя служб** (вначале с помощью объектов POJO, а затем компонентов EJB) будет использован подход сценария операции.

Чтобы продемонстрировать работу **слоя служб**, расширим нашу задачу, добавив в нее немного логики приложения. Пусть варианты использования нашего приложения предполагают, что при вычислении зачтенного дохода по контракту приложение должно отослать отчет администратору контракта и опубликовать сообщение посредством промежуточного программного обеспечения для уведомления интегрированных приложений.

Для начала немного изменим класс `RecognitionService` из примера определения зачтенного дохода для **сценария транзакции**. Расширим его **супертип слоя** и добавим несколько **шлюзов (Gateway, 483)**, инкапсулирующих в себе логику приложения. Полученная схема классов показана на рис. 9.7. Класс `RecognitionService` станет POJO-реализацией службы из **слоя служб**, а его методы будут представлять две операции этого слоя.

Методы класса `RecognitionService` представляют логику приложения в виде сценариев, обращающихся за логикой домена к классам объектов домена, описанным в примере для **модели предметной области**.

```
public class ApplicationService {
   protected EmailGateway getEmailGateway(){
      // возвращает экземпляр объекта EmailGateway
   }
   protected IntegrationGateway getIntegrationGateway(){
      // возвращает экземпляр объекта IntegrationGateway
   }
}
public interface EmailGateway {
   void sendEmailMessage(String toAddress, String subject,
↳String body);
}
public interface IntegrationGateway {
   void publishRevenueRecognitionCalculation(Contract contract);
}
public class RecognitionService
extends ApplicationService {
   public void calculateRevenueRecognitions(long
↳contractNumber) {
      Contract contract =Contract.readForUpdate(contractNumber);
      contract.calculateRecognitions();
      getEmailGateway().sendEmailMessage(
         contract.getAdministratorEmailAddress(),
         "RE: Contract #" + contractNumber,
         contract + " has had revenue recognitions calculated.");
      getIntegrationGateway().
↳publishRevenueRecognitionCalculation(contract);
   }
```

```
    public Money recognizedRevenue(long contractNumber,
↳Date asOf) {
        return Contract.read(contractNumber).recognizedRevenue(
↳asOf);
    }
}
```

Для упрощения примера я опустил детали хранения данных. Достаточно сказать, что класс `Contract` реализует статические методы, предназначенные для считывания из источника данных контрактов с заданными номерами. Один из этих методов (а именно `readForUpdate`) считывает контракты для последующего обновления, что позволяет **преобразователю данных (Data Mapper, 187)**, используемому в приложении, регистрировать считываемый объект или объекты, скажем, в **единице работы (Unit of Work, 205)**.

В этом примере были опущены и детали управления транзакциями. Метод `calculateRevenueRecognitions()` по своей сути является транзакционным, поскольку во время его выполнения обновляются постоянные объекты контракта путем добавления к ним значения зачтенного дохода, ставятся в очередь публикуемые сообщения и по электронной почте отправляются отчеты. Все эти отклики должны быть переданы посредством атомарных транзакций, потому что отправку отчета и публикацию сообщений следует выполнять только в том случае, если обновление объектов базы данных прошло успешно.

В среде J2EE EJB-контейнеры могут управлять распределенными транзакциями, реализуя службы приложений (и **шлюзы**) в виде компонентов сеанса, не меняющих своего состояния и использующих ресурсы транзакций. На рис. 9.8 изображена схема реализации класса `RecognitionService`, использующего локальные интерфейсы EJB 2.0 и идиому "бизнес-интерфейса". В этой реализации также используется **супертип слоя**, обеспечивающий стандартную реализацию методов классов компонентов, необходимых EJB-контейнерам, в дополнение к методам, специфичным для данного приложения. Кроме того, если рассматривать интерфейсы `EmailGateway` и `IntegrationGateway` как бизнес-интерфейсы соответствующих компонентов сеанса, то управление распределенными транзакциями может выполняться путем присвоения методам `calculateRevenue Recognitions`, `sendEmailMessage` и `publishRevenueRecognitionCalculation` статуса транзакционных. В этом случае методы класса `RecognitionService`, рассмотренные в примере с реализацией объекта POJO, "плавно переходят" в класс `Recognition-ServiceBeanImpl` без каких-либо изменений.

Важно отметить, что для координации транзакционных откликов на выполнение операций **слоя служб** используются и сценарии операции, и классы объектов домена. Рассмотренный ранее метод `calculateRevenueRecognitions` реализует сценарий логики приложения для выполнения откликов на варианты использования, однако обращается за логикой предметной области к классам домена. Здесь также продемонстрировано несколько приемов, представляющих собой попытку избежать дублирования логики в сценариях операций **слоя служб**. Обязанности по выполнению откликов выносятся в различные объекты (например, в **шлюзы**), которые могут быть повторно использованы посредством делегирования. Доступ к этим объектам удобно получать через **супертип слоя**.

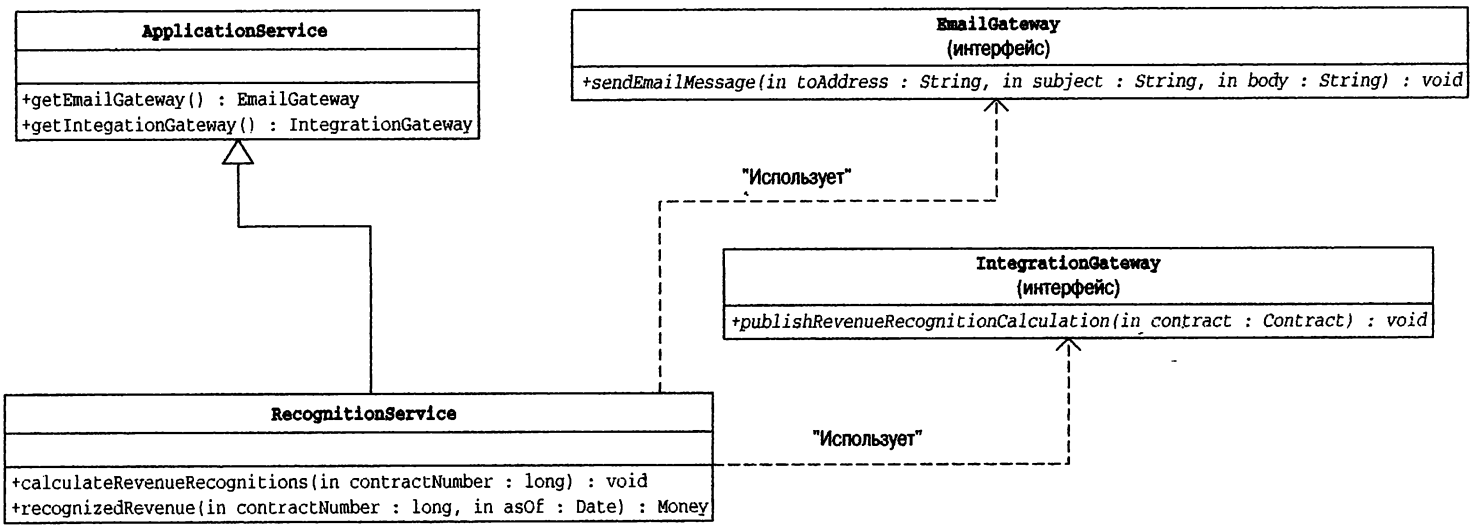


Рис. 9.7. Схема связей POJO-класса RecognitionService

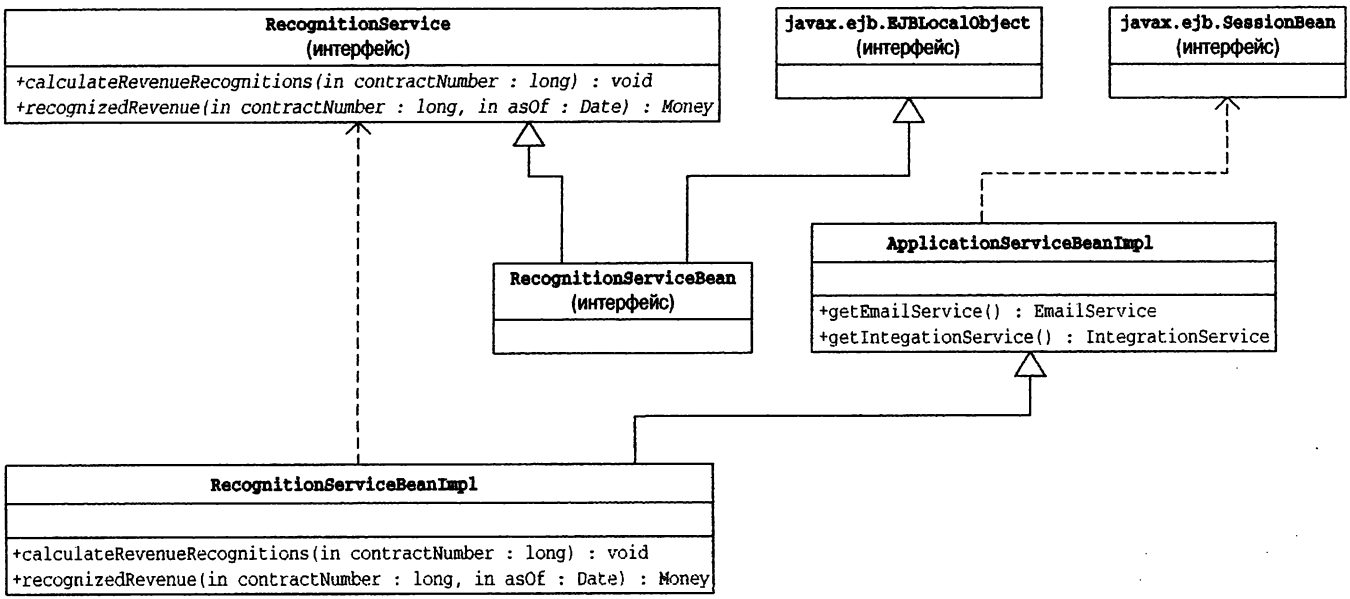


Рис. 9.8. Схема связей EJB-класса RecognitionService

Некоторые могут не согласиться с моим выбором, предложив реализовать сценарий операции посредством типового решения **наблюдатель (Observer)**, представленного в [20]. Конечно, это более эффектное решение, однако реализовать его в многопоточном, не имеющем состояний **слое служб** было бы весьма затруднительно. Как мне кажется, открытый код сценария операции гораздо понятнее и проще в использовании.

Можно было бы поспорить и о размещении логики приложения. Думаю, некоторые предпочли бы реализовать ее в методах объектов домена, таких, как `Contract.calculateRevenueRecognitions()`, или вообще в слое источника данных, что позволило бы обойтись без отдельного **слоя служб**. Тем не менее подобное размещение логики приложения кажется мне весьма нежелательным, и вот почему. Во-первых, классы объектов домена, которые реализуют логику, специфичную для приложения (и зависят от **шлюзов** и других объектов, специфичных для приложения), менее подходят для повторного использования другими приложениями. Это должны быть модели частей предметной области, представляющих интерес для данного приложения, поэтому подобные объекты вовсе не обязаны описывать возможные отклики на *все* варианты использования приложения. Во-вторых, инкапсуляция логики приложения на более высоком уровне (каковым не является слой источника данных) облегчает изменение реализации этого слоя, возможно, посредством некоторых специальных инструментальных средств.

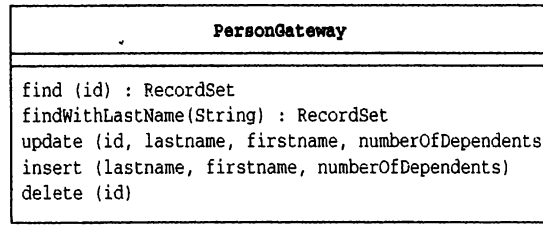
Слой служб как типовое решение для организации слоя логики корпоративного приложения сочетает в себе применение сценариев и классов объектов домена, используя преимущества тех и других. Реализация **слоя служб** допускает некоторые варианты, например использование интерфейсов доступа к домену или сценариев операций, объектов POJO или компонентов сеанса либо сочетание их обоих. Кроме того, **слой служб** может быть предназначен для локальных вызовов, удаленных вызовов или и тех и других. Вне зависимости от способа реализации, и это наиболее важно, данное типовое решение служит основой для инкапсулированной реализации бизнес-логики приложения и последовательных обращений к этой логике ее многочисленными клиентами.

Глава 10

Архитектурные типовые решения источников данных

Шлюз таблицы данных (Table Data Gateway)

Объект, выполняющий роль **шлюза (Gateway, 483)** *к базе данных*



Использование SQL в логике приложений может быть связано с некоторыми проблемами. Не все разработчики владеют языком SQL или хорошо в нем разбираются. В свою очередь, администраторы СУБД должны иметь удобный доступ к командам SQL для настройки и расширения своих баз данных.

Типовое решение **шлюз таблицы данных** содержит в себе все команды SQL, необходимые для извлечения, вставки, обновления и удаления данных из таблицы или представления. Методы этого типового решения используются другими объектами для взаимодействия с базой данных.

Принцип действия

Интерфейс **шлюза таблицы данных** чрезвычайно прост. Обычно он включает в себя несколько методов поиска, предназначенных для извлечения данных, а также методы обновления, вставки и удаления. Каждый метод передает входные параметры вызову соответствующей команды SQL и выполняет ее в контексте установленного соединения с базой данных. Как правило, это типовое решение не имеет состояний, поскольку всего лишь передает данные в таблицу и из таблицы.

Пожалуй, наиболее интересная особенность **шлюза таблицы данных** — это то, как он возвращает результат выполнения запроса. Даже простой запрос типа "найти данные с указанным идентификатором" может возвратить несколько записей. Это не составляет проблемы для сред разработки, допускающих множественные результаты, однако большинство классических языков программирования позволяют возвращать только одно значение.

В качестве альтернативы можно отобразить таблицу базы данных в какую-нибудь простую структуру наподобие коллекции. Это позволит работать с множественными результатами, однако потребует копирования данных из результирующего множества записей базы данных в упомянутую коллекцию. На мой взгляд, этот способ не слишком хорош, поскольку не подразумевает выполнения проверки времени компиляции и не предоставляет явного интерфейса, что приводит к многочисленным опечаткам программистов, ссылающихся на содержимое коллекции. Более удачным решением является использование универсального **объекта переноса данных (Data Transfer Object, 419)**.

Вместо всего перечисленного результат выполнения SQL-запроса может быть возвращен в виде **множества записей (Record Set, 523)**. Вообще говоря, это не совсем корректно, поскольку объект, расположенный в оперативной памяти, не должен "знать" об SQL-интерфейсе. Кроме того, если вы не можете создавать множества записей в собственном коде, это вызовет определенные трудности при замене базы данных файлом. Тем не менее этот способ весьма эффективен во многих средах разработки, широко использующих **множество записей**, например в таких, как .NET. В этом случае **шлюз таблицы данных** хорошо сочетается с **модулем таблицы (Table Module, 148)**. Если все обновления таблиц выполняются через **шлюз таблицы данных**, результирующие данные могут быть основаны на виртуальных, а не на реальных таблицах, что уменьшает зависимость кода от базы данных.

Если вы используете **модель предметной области (Domain Model, 140)**, методы **шлюза таблицы данных** могут возвращать соответствующий объект домена. Следует, однако, иметь в виду, что это подразумевает двунаправленные зависимости между объектами домена и шлюзом. И те и другие тесно связаны между собой, поэтому необходимость создания таких зависимостей не слишком усложняет дело, однако мне это все равно не нравится.

Как правило, для каждой таблицы базы данных создается собственный **шлюз таблицы данных**. Впрочем, в наиболее простых случаях можно ограничиться разработкой одного **шлюза таблицы данных**, который будет включать в себя все методы для всех таблиц. Кроме того, отдельные **шлюзы таблицы данных** могут быть созданы для представлений (виртуальных таблиц) и даже для некоторых запросов, не хранящихся в базе данных в форме представлений. Конечно же, **шлюз таблицы данных** для представления не сможет обновлять данные и поэтому не будет обладать соответствующими методами. Тем не менее, если вы можете сами обновлять таблицы, инкапсуляция процедур обновления в методах типового решения **шлюз таблицы данных** — прекрасный выбор.

Назначение

Принимая решение об использовании **шлюза таблицы данных**, как, впрочем, и **шлюза записи данных (Row Data Gateway, 175)**, необходимо подумать о том, следует ли вообще обращаться к **шлюзу** и если да, то к какому именно.

На мой взгляд, **шлюз таблицы данных** — это наиболее простое типовое решение интерфейса базы данных, поскольку оно замечательно отображает таблицы или записи баз данных на объекты. Кроме того, **шлюз таблицы данных** естественным образом инкапсулирует точную логику доступа к источнику данных. Я крайне редко использую это типовое решение с **моделью предметной области**, потому что гораздо большей изолированности **модели предметной области** от источника данных можно добиться с помощью **преобразователя данных (Data Mapper, 187)**.

Типовое решение **шлюз таблицы данных** особенно хорошо сочетается с **модулем таблицы**. Методы **шлюза таблицы данных** возвращают структуры данных в виде множеств записей, с которыми затем работает **модуль таблицы**. На самом деле другой подход отображения базы данных для **модуля таблицы** придумать просто невозможно (по крайней мере, мне так кажется).

Подобно **шлюзу записи данных**, **шлюз таблицы данных** прекрасно подходит для использования в **сценариях транзакции (Transaction Script, 133)**. В действительности выбор одного из нескольких типовых решений зависит только от того, как они обрабатывают множественные результаты. Некоторые предпочитают осуществлять передачу данных посредством **объекта переноса данных**, однако мне это решение кажется более трудоемким (если только этот объект уже не был реализован где-нибудь в другом месте вашего проекта). Рекомендую использовать **шлюз таблицы данных** в том случае, если его представление результирующего множества данных подходит для работы со **сценарием транзакции**.

Что интересно, **шлюзы таблицы данных** могут выступать в качестве посредника при обращении к базе данных **преобразователей данных**. Правда, когда весь код пишется вручную, это не всегда нужно, однако данный прием может быть весьма эффективен, если для реализации **шлюза таблицы данных** используются метаданные, а реальное отображение содержимого базы данных на объекты домена выполняется вручную.

Одно из преимуществ использования **шлюза таблицы данных** для инкапсуляции доступа к базе данных состоит в том, что этот интерфейс может применяться и для обращения к базе данных с помощью средств языка SQL, и для работы с хранимыми процедурами. Более того, хранимые процедуры зачастую сами организованы в виде **шлюзов таблицы данных**. В этом случае хранимые процедуры, предназначенные для вставки и обновления данных, инкапсулируют реальную структуру таблицы. В свою очередь, процедуры поиска могут возвращать представления, что позволяет скрыть фактическую структуру используемой таблицы.

Дополнительные источники информации

В [3] рассматривается аналог **шлюза таблицы данных** под именем **объект доступа к данным (Data Access Object)**. В нем результаты выполнения запросов возвращаются в виде коллекций **объектов переноса данных**. Мне не совсем понятно, может ли это типовое решение быть построено только на основе таблицы. Концепции и идеи, изложенные в книге, применимы как к **шлюзу таблицы данных**, так и к **шлюзу записи данных**.

В моей книге это типовое решение получило другое имя — **шлюз таблицы данных**. Во-первых, мне хотелось, чтобы имя типового решения отображало его связь с более общей концепцией **шлюза**. Кроме того, термин *объект доступа к данным (Data Access Object)* и его аббревиатура *DAO* уже давно применяются компанией Microsoft для обозначения собственной технологии.

Пример: класс PersonGateway (C#)

Типовое решение **шлюз таблицы данных** — это стандартная технология доступа к базам данных в мире Windows, поэтому в качестве языка программирования для иллюстрации следующего примера был выбран C#. Однако следует отметить, что классическая реализация **шлюза таблицы данных** немного не вписывается в среду .NET, поскольку не использует всех преимуществ объектов `DataSet` библиотеки ADO.NET; вместо этого считывание записей базы данных осуществляется в потоковом режиме посредством курсороподобного интерфейса (объекты `DataReader`). Концепция потокового режима очень удобна при работе с большими объемами информации, поскольку избавляет от необходимости помещать их в оперативную память.

В этом примере рассматривается класс `PersonGateway`, предназначенный для доступа к таблице `Person`. Ниже приведен код метода поиска, возвращающего результат выполнения SQL-команды `SELECT` в виде объекта ADO.NET `DataReader`, который предоставляет доступ к отобранным данным.

```
class PersonGateway...

      public IDataReader FindAll() {
         String sql = "select * from person";
         return new OleDbCommand(sql,
↳DB.Connection).ExecuteReader();
      }
      public IDataReader FindWithLastName(String lastName) {
         String sql = "SELECT * FROM person WHERE lastname = ?";
         IDbCommand comm = new OleDbCommand(sql, DB.Connection);
         comm.Parameters.Add (new OleDbParameter("lastname",
↳lastName));
         return comm.ExecuteReader();
      }
      public IDataReader FindWhere(String whereClause) {
         String sql = String.Format("select * from
↳person where {0}", whereClause);
         return new OleDbCommand(sql,
↳DB.Connection).ExecuteReader();
      }
```

В большинстве случаев будем извлекать группу строк посредством объекта `DataReader`. Тем не менее иногда вам может понадобиться извлечь отдельную строку, для чего применяется метод, показанный ниже.

```
class PersonGateway...

      public Object [] FindRow (long key) {
         String sql = "SELECT * FROM person WHERE id = ?";
         IDbCommand comm = new OleDbCommand(sql, DB.Connection);
         comm.Parameters.Add(new OleDbParameter("key", key));
         IDataReader reader = comm.ExecuteReader();
         reader.Read();
         Object [] result = new Object [reader.FieldCount];
         reader.GetValues(result);
         reader.Close();
```

```
        return result;
    }
```

Методы обновления и вставки получают новые данные в качестве аргументов и вызывают соответствующие команды SQL.

```
class PersonGateway...

    public void Update (long key, String lastname, String
↳firstname, long numberOfDependents) {
        String sql = @"
            UPDATE person
                SET lastname = ?, firstname = ?,
↳numberOfDependents = ?
                WHERE id = ?";
        IDbCommand comm = new OleDbCommand(sql, DB.Connection);
        comm.Parameters.Add(new OleDbParameter ("last",
↳lastname));
        comm.Parameters.Add(new OleDbParameter ("first",
↳firstname));
        comm.Parameters.Add(new OleDbParameter ("numDep",
↳numberOfDependents));
        comm.Parameters.Add(new OleDbParameter ("key", key));
        comm.ExecuteNonQuery();
    }

class PersonGateway...

    public long Insert(String lastName, String firstName,
↳long numberOfDependents) {
        String sql = "INSERT INTO person VALUES (?,?,?,?)";
        long key = GetNextID();
        IDbCommand comm = new OleDbCommand(sql, DB.Connection);
        comm.Parameters.Add(new OleDbParameter ("key", key));
        comm.Parameters.Add(new OleDbParameter ("last",
↳lastName));
        comm.Parameters.Add(new OleDbParameter ("first",
↳firstName));
        comm.Parameters.Add(new OleDbParameter ("numDep",
↳numberOfDependents));
        comm.ExecuteNonQuery();
        return key;
    }
```

И наконец, метод удаления принимает в качестве аргумента только значение ключа.

```
class PersonGateway...

    public void Delete (long key) {
        String sql = "DELETE FROM person WHERE id = ?";
        IDbCommand comm =new OleDbCommand(sql, DB.Connection);
        comm.Parameters.Add(new OleDbParameter ("key", key));
        comm.ExecuteNonQuery();
    }
```

Пример: использование объектов ADO.NET DataSet (C#)

Универсальное типовое решение **шлюз таблицы данных** подходит практически для любой платформы, так как представляет собой не более чем оболочку для операторов SQL. Для доступа к базам данных в среде .NET чаще используются объекты `DataSet`, однако **шлюз таблицы данных** может быть применен и здесь (правда, в несколько другой форме).

Для загрузки и обновления данных в объектах `DataSet` применяются объекты `DataAdapter`. Мне показалось удобным создать для объектов `DataSet` и `DataAdapter` некий "диспетчер" (holder), который затем будет использован шлюзом для их хранения (рис. 10.1). Большинство предлагаемых ниже методов универсальны и могут быть реализованы в суперклассе.

Объект `DataSetHolder` индексирует объекты `DataSet` и `DataAdapter` по именам таблиц.

```
class DataSetHolder...

      public DataSet Data = new DataSet();
      private Hashtable DataAdapters = new Hashtable();
```

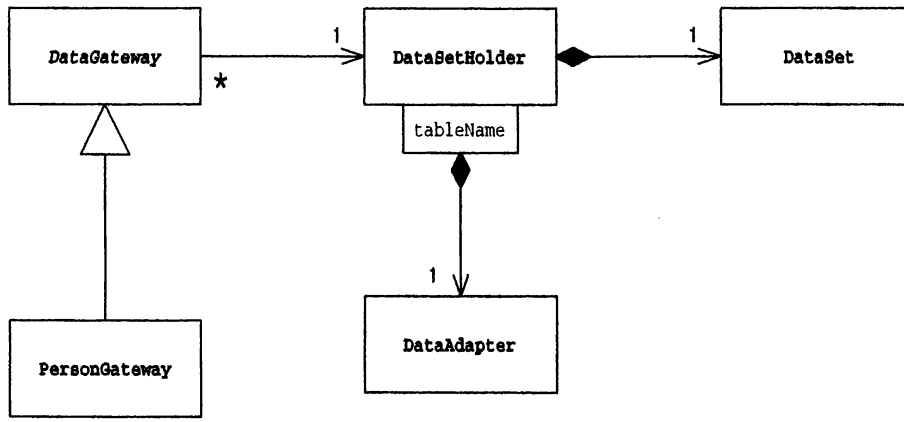


Рис. 10.1. Схема классов для **шлюза таблицы данных**, *ориентированного на доступ к данным с помощью объектов DataSet и вспомогательного объекта DataSetHolder*

Шлюз сохраняет объект `DataSetHolder` и предоставляет доступ к содержимому соответствующего объекта `DataSet` своим клиентам.

```
class DataGateway...

      public DataSetHolder Holder;
      public DataSet Data {
         get {return Holder.Data;}
      }
```

Шлюз может работать с существующим объектом `DataSetHolder` или же создать новый.

```
class DataGateway...

      protected DataSetGateway() {
         Holder =new DataSetHolder();
      }
      protected DataSetGateway(DataSetHolder holder) {
         this.Holder = holder;
      }
```

В данном примере поиск записей выполняется немного иначе. Объект `DataSet` представляет собой контейнер таблично-ориентированных данных и может содержать в себе данные нескольких таблиц. Поэтому данные лучше загрузить в объект `DataSet`.

```
class DataGateway...

      public void LoadAll() {
         String commandString = String.Format(
↳"select * from {0}", TableName);
         Holder.FillData(commandString, TableName);
      }
      public void LoadWhere(String whereClause) {
         String commandString = String.Format("select * from {0}
↳where {1}", TableName, whereClause);
         Holder.FillData(commandString, TableName);
      }
      abstract public String TableName {get;}

class PersonGateway...

      public override String TableName {
         get {return "Person";}
      }

class DataSetHolder...

      public void FillData(String query, String tableName) {
         if (DataAdapters.Contains(tableName)) throw new
↳MutlipleLoadException();
         OleDbDataAdapter da = new OleDbDataAdapter(query,
↳DB.Connection);
         OleDbCommandBuilder builder = new
↳OleDbCommandBuilder(da);
         da.Fill(Data, tableName);
         DataAdapters.Add(tableName, da);
      }
```

Обновление данных осуществляется путем выполнения соответствующих операций над объектом `DataSet` непосредственно в клиентском коде.

```
         person.LoadAll();
         person [key ]["lastname"] = "Odell";
         person.Holder.Update();
```

Для облегчения доступа к конкретным строкам таблицы шлюз можно оснастить индексатором.

```
class PersonGateway...

      public DataRow this [long key ] {
         get {
            String filter = String.Format("id = {0}", key);
            return Table.Select(filter)[0];
         }
      }
      public override DataTable Table {
         get {return Data.Tables [TableName];}
      }
```

Обновление данных происходит с помощью метода `Update` объекта `DataSetHolder`.

```
class DataSetHolder...

      public void Update() {
      foreach (String table in DataAdapters.Keys)
         ((OleDbDataAdapter)DataAdapters[table]).Update(
↳Data, table);
      }
      public DataTable this[String tableName] {
         get {return Data.Tables[tableName];}
      }
```

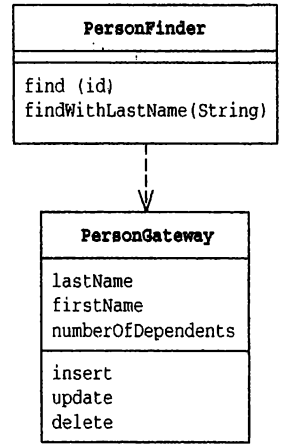
Вставка данных может быть выполнена примерно таким же способом: получить объект `DataSet`, вставить в таблицу новую строку и заполнить каждое поле новой записи. Впрочем, метод обновления способен выполнить вставку за один вызов.

```
class PersonGateway...

      public long Insert(String lastName, String firstname,
↳int numberOfDependents) {
         long key = PersonGatewayDS().GetNextID();
         DataRow newRow = Table.NewRow();
         newRow ["id"] = key;
         newRow ["lastName"] = lastName;
         newRow ["firstName"] = firstname;
         newRow ["numberOfDependents"] = numberOfDependents;
         Table.Rows.Add(newRow);
         return key;
      }
```

Шлюз записи данных (Row Data Gateway)

Объект, выполняющий роль **шлюза (Gateway, 483)** *к отдельной записи источника данных. Каждой строке таблицы базы данных соответствует свой экземпляр* **шлюза записи данных**



Реализация доступа к базам данных в объектах, расположенных в оперативной памяти, имеет ряд недостатков. Прежде всего, если этим объектам присуща собственная бизнес-логика, добавление кода доступа к базе данных значительно повышает их сложность. Кроме того, это серьезно усложняет тестирование. Если объекты, расположенные в оперативной памяти, связаны с базой данных, тестирование выполняется крайне медленно из-за проблем, вызванных необходимостью доступа к базе данных. Особенно раздражает, когда приходится осуществлять доступ к нескольким базам данных, имеющим небольшие (но крайне досадные!) расхождения в реализации SQL.

Типовое решение **шлюз записи данных** предоставляет в ваше распоряжение объекты, которые полностью аналогичны записям базы данных, однако могут быть доступны с помощью обычных механизмов используемого языка программирования. Все детали доступа к источнику данных скрыты за интерфейсом.

Принцип действия

Шлюз записи данных выступает в роли объекта, полностью повторяющего одну запись, например одну строку таблицы базы данных. Каждому столбцу таблицы соответствует поле записи. Обычно **шлюз записи данных** должен выполнять все возможные преобразования типов источника данных в типы, используемые приложением, однако эти преобразования весьма просты. Рассматриваемое типовое решение содержит все данные о строке, поэтому клиент имеет возможность непосредственного доступа к **шлюзу записи данных**. Шлюз выступает в роли интерфейса к строке данных и прекрасно подходит для применения в **сценариях транзакции (Transaction Script, 133)**.

При реализации **шлюза записи данных** возникает вопрос: куда "пристроить" методы поиска, генерирующие экземпляр данного типового решения? Разумеется, можно воспользоваться статическими методами поиска, однако они исключают возможность полиморфизма (что могло бы пригодиться, если понадобится определить разные методы поиска для различных источников данных). В подобной ситуации часто имеет смысл создать отдельные объекты поиска, чтобы у каждой таблицы реляционной базы данных был один класс для проведения поиска и один класс шлюза для сохранения результатов этого поиска (рис. 10.2).

Иногда **шлюз записи данных** трудно отличить от **активной записи (Active Record, 182)**. В этом случае следует обратить внимание на наличие какой-либо логики домена; если она есть, значит, это **активная запись**. Реализация **шлюза записи данных** должна включать в себя только логику доступа к базе данных и никакой логики домена.

Как и другие формы табличной инкапсуляции, **шлюз записи данных** можно применять не только к таблице, но и к представлению или запросу. Конечно же, в последних случаях существенно осложняется выполнение обновлений, поскольку приходится обновлять таблицы, на основе которых были созданы соответствующие представления или запросы. Кроме того, если с одними и теми же таблицами работают два **шлюза записи данных**, обновление второго шлюза может аннулировать обновление первого. Универсального способа предотвратить эту проблему не существует; разработчикам просто необходимо следить за тем, как построены виртуальные **шлюзы записи данных**. В конце концов, это же может случиться и с обновляемыми представлениями. Разумеется, вы можете вообще не реализовать методы обновления.

Шлюзы записи данных довольно трудоемки в написании. Тем не менее генерацию их кода можно значительно облегчить посредством типового решения **отображение метаданных (Metadata Mapping, 325)**. В этом случае весь код, описывающий доступ к базе данных, может быть автоматически сгенерирован в процессе сборки проекта.

Назначение

Принимая решение об использовании **шлюза записи данных**, необходимо подумать о двух вещах: следует ли вообще использовать шлюз, и если да, то какой именно — **шлюз записи данных** или **шлюз таблицы данных (Table Data Gateway, 167)**.

Как правило, я использую **шлюз записи данных**, когда у меня есть **сценарий транзакции**. В этом случае доступ к базе данных легко реализовать таким образом, чтобы соответствующий код мог повторно использоваться другими **сценариями транзакции**.

Я не использую **шлюз записи данных** с **моделью предметной области (Domain Model, 140)**. Если отображение на объекты домена достаточно простое, его можно реализовать и с помощью **активной записи**, не добавляя дополнительный слой кода. Если же отображение сложное, для его реализации рекомендуется применить **преобразователь данных (Data Mapper, 187)**. Последний лучше справляется с отделением структуры данных от объектов домена, потому что объектам домена не нужно знать о структуре базы данных. Конечно же, **шлюз записи данных** можно использовать, чтобы скрыть структуру базы данных от объектов домена. Это очень удобно, если вы собираетесь изменить структуру базы данных и не хотите менять логику домена. Тем не менее в этом случае у вас появится три различных представления данных: одно в бизнес-логике, одно в **шлюзе записи данных** и еще одно в базе данных. Для крупномасштабных систем это слишком много. Поэтому я обычно использую **шлюзы записи данных**, отражающие структуру базы данных.

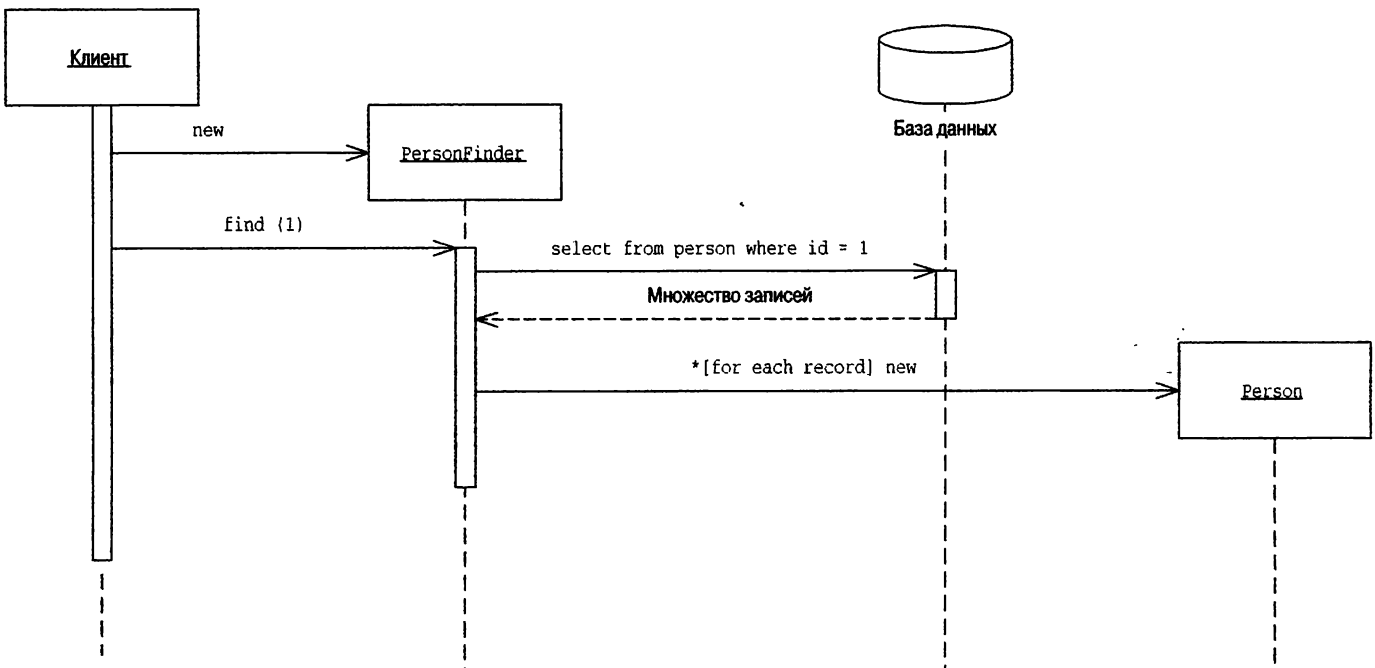


Рис. 10.2. Взаимодействия со **шлюзом записи данных** *для поиска нужной строки*

Интересно, что **шлюз записи данных** и **преобразователь данных** вполне могут сосуществовать. Несмотря на то что с первого взгляда это кажется излишней тратой сил, сочетание **шлюза записи данных** и **преобразователя данных** может оказаться весьма эффективным в том случае, если первый автоматически генерируется на основе метаданных, а второй создается вручную.

Если **шлюз записи данных** используется со **сценарием транзакции**, вы можете заметить, что в различных сценариях повторяется одна и та же бизнес-логика, которую можно было бы реализовать в **шлюзе записи данных**. Перенос этой логики в **шлюз записи данных** превратит его в **активную запись**. Это весьма удачное решение, поскольку позволяет избежать дублирования элементов бизнес-логики.

Пример: запись о сотруднике (Java)

Ниже приведен пример реализации **шлюза записи данных** для простой таблицы сотрудников под именем `people`.

```
create table people (ID int primary key, lastname varchar,
↳firstname varchar, number_of_dependents int)
```

У данной таблицы есть шлюз `PersonGateway`. Код этого класса начинается с описаний полей данных и *функций доступа (accessors)*.

```
class PersonGateway...

   private String lastName;
   private String firstName;
   private int numberOfDependents;
   public String getLastName() {
      return lastName;
   }
   public void setLastName(String lastName) {
      this.lastName = lastName;
   }
   public String getFirstName() {
      return firstName;
   }
   public void setFirstName(String firstName) {
      this.firstName = firstName;
   }
   public int getNumberOfDependents() {
      return numberOfDependents;
   }
   public void setNumberOfDependents(int numberOfDependents) {
      this.numberOfDependents = numberOfDependents;
   }
```

Класс шлюза включает в себя собственные методы обновления и вставки данных[1].

[1] Хранение идентификаторов следует поручить **супертипу слоя (Layer Supertype, 491) шлюзов записи данных**.

```
class PersonGateway...

   private static final String updateStatementString =
         "UPDATE people " +
         " set lastname = ?, firstname = ?,
↳number_of_dependents = ? " +
         " where id = ?";
   public void update() {
      PreparedStatement updateStatement = null;
      try {
         updateStatement = DB.prepare(updateStatementString);
         updateStatement.setString(1, lastName);
         updateStatement.setString(2, firstName);
         updateStatement.setInt(3, numberOfDependents);
         updateStatement.setInt(4, getID().intValue());
         updateStatement.execute();
      } catch (Exception e) {
         throw new ApplicationException(e);
      } finally {DB.cleanUp(updateStatement);
      }
   }
   private static final String insertStatementString =
         "INSERT INTO people VALUES (?,?,?,?)";
   public Long insert() {
      PreparedStatement insertStatement = null;
      try {
         insertStatement = DB.prepare(insertStatementString);
         setID(findNextDatabaseId());
         insertStatement.setInt(1, getID().intValue());
         insertStatement.setString(2, lastName);
         insertStatement.setString(3, firstName);
         insertStatement.setInt(4, numberOfDependents);
         insertStatement.execute();
         Registry.addPerson(this);
         return getID();
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {DB.cleanUp(insertStatement);
      }
   }
```

Для извлечения из базы данных записей о сотрудниках применяется специальный класс `PersonFinder`. Он используется в сочетании с классом `PersonGateway` для создания новых объектов шлюза.

```
class PersonFinder...

   private final static String findStatementString =
         "SELECT id, lastname, firstname,
↳number_of_dependents " +
         "from people " +
         "WHERE id = ?";
    public PersonGateway find(Long id) {
       PersonGateway result = (PersonGateway)
```

```
↳Registry.getPerson(id);
       if (result != null) return result;
       PreparedStatement findStatement = null;
       ResultSet rs = null;
       try {
          findStatement = DB.prepare(findStatementString);
          findStatement.setLong(1, id.longValue());
          rs = findStatement.executeQuery();
          rs.next();
          result = PersonGateway.load(rs);
          return result;
       } catch (SQLException e) {
          throw new ApplicationException(e);
       } finally {DB.cleanUp(findStatement,rs);
       }
   }
   public PersonGateway find(long id) {
      return find(new Long(id));
   }

class PersonGateway...

   public static PersonGateway load(ResultSet rs)
↳throws SQLException {
         Long id = new Long(rs.getLong(1));
         PersonGateway result = (PersonGateway)
↳Registry.getPerson(id);
         if (result != null) return result;
         String lastNameArg = rs.getString(2);
         String firstNameArg = rs.getString(3);
         int numDependentsArg = rs.getInt(4);
         result = new PersonGateway(id, lastNameArg,
↳firstNameArg, numDependentsArg);
         Registry.addPerson(result);
         return result;
   }
```

Для извлечения данных о нескольких сотрудниках на основе заданного критерия можно реализовать удобный метод поиска.

```
class PersonFinder...

   private static final String findResponsibleStatement =
         "SELECT id, lastname, firstname,
↳number_of_dependents " +
         "from people "+
         "WHERE number_of_dependents > 0";
   public List findResponsibles() {
      List result = new ArrayList();
      PreparedStatement stmt = null;
      ResultSet rs = null;
      try {
         stmt = DB.prepare(findResponsibleStatement);
         rs = stmt.executeQuery();
```

```
            while (rs.next()) {
                result.add(PersonGateway.load(rs));
            }
            return result;
        } catch (SQLException e) {
            throw new ApplicationException(e);
        } finally {DB.cleanUp(stmt,rs);
        }
    }
```

Данный метод использует **реестр (Registry, 495)** для хранения **коллекций объектов (Identity Map, 216)**.

Теперь мы можем применить шлюзы в **сценарии транзакции**.

```
    PersonFinder finder = new PersonFinder();
    Iterator people = finder.findResponsibles().iterator();
    StringBuffer result = new StringBuffer();
    while (people.hasNext()) {
        PersonGateway each = (PersonGateway) people.next();
        result.append(each.getLastName());
        result.append("");
        result.append(each.getFirstName());
        result.append("");
        result.append(String.valueOf(
➥each.getNumberOfDependents()));
        result.append("
    }
    return result.toString();
```

Пример: использование диспетчера данных для объекта домена (Java)

В большинстве случаев я использую **шлюз записи данных** со **сценарием транзакции**. Если же применять **шлюз записи данных** с **моделью предметной области**, объектам домена нужно получать данные из шлюза. Вместо копирования данных в объект домена, можно использовать **шлюз записи данных** в качестве диспетчера данных для объекта домена.

```
class Person...

    private PersonGateway data;
    public Person(PersonGateway data) {
        this.data = data;
    }
```

Теперь функции доступа логики домена могут обращаться к шлюзу за необходимыми данными.

```
class Person...

    public int getNumberOfDependents() {
        return data.getNumberOfDependents();
    }
```

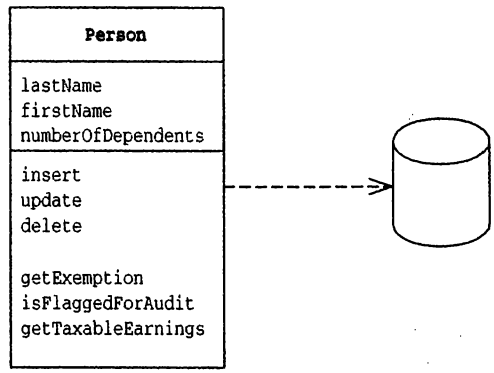
Логика домена использует get-методы для извлечения данных из шлюза.

```
class Person...

   public Money getExemption() {
      Money baseExemption = Money.dollars(1500);
      Money dependentExemption = Money.dollars(750);
      return baseExemption.add(dependentExemption.multiply(
↳this.getNumberOfDependents()));
   }
```

Активная запись (Active Record)

Объект, выполняющий роль оболочки для строки таблицы или представления базы данных. Он инкапсулирует доступ к базе данных и добавляет к данным логику домена



Этот объект охватывает и данные и поведение. Большая часть его данных является постоянной и должна храниться в базе данных. В типовом решении **активная запись** используется наиболее очевидный подход, при котором логика доступа к данным включается в объект домена. В этом случае все знают, как считывать данные из базы данных и как их записывать в нее.

Принцип действия

В основе типового решения **активная запись** лежит **модель предметной области (Domain Model, 140)**, классы которой повторяют структуру записей используемой базы данных. Каждая **активная запись** отвечает за сохранение и загрузку информации в базу данных, а также за логику домена, применяемую к данным. Это может быть вся бизнес-логика приложения. Впрочем, иногда некоторые фрагменты логики домена содержатся в **сценариях транзакции (Transaction Script, 133)**, а общие элементы кода, ориентированные на работу с данными, — в **активной записи**.

Структура данных **активной записи** должна в точности соответствовать таковой в таблице базы данных: каждое поле объекта должно соответствовать одному столбцу таблицы. Значения полей следует оставлять такими же, какими они были получены в результате выполнения SQL-команд; никакого преобразования на этом этапе делать не нужно. При необходимости вы можете применить **отображение внешних ключей (Foreign Key Mapping, 258)**, однако это не обязательно. **Активная запись** может применяться к таблицам или представлениям (хотя в последнем случае реализовать обновления будет значительно сложнее). Использование представлений особенно удобно при составлении отчетов.

Как правило, типовое решение **активная запись** включает в себя методы, предназначенные для выполнения следующих операций:

- создание экземпляра **активной записи** на основе строки, полученной в результате выполнения SQL-запроса;
- создание нового экземпляра **активной записи** для последующей вставки в таблицу;
- статические методы поиска, выполняющие стандартные SQL-запросы и возвращающие **активные записи**;
- обновление базы данных и вставка в нее данных из **активной записи**;
- извлечение и установка значений полей (get- и set-методы);
- реализация некоторых фрагментов бизнес-логики.

Методы извлечения и установки значений полей могут выполнять и другие действия, например преобразование типов SQL в типы, используемые приложением. Кроме того, get-метод может возвращать соответствующую **активную запись** таблицы, с которой связана текущая таблица (путем просмотра первой), даже если для структуры данных не было определено **поле идентификации (Identity Field, 237)**.

Классы **активной записи** довольно удобны с точки зрения разработчиков, однако не позволяют им полностью абстрагироваться от реляционной базы данных. Впрочем, это не так уж плохо, поскольку дает возможность использовать меньше типовых решений, предназначенных для отображения объектной модели на базу данных.

Активная запись очень похожа на **шлюз записи данных (Row Data Gateway, 175)**. Принципиальное отличие между ними состоит в том, что **шлюз записи данных** содержит только логику доступа к базе данных, в то время как **активная запись** содержит и логику доступа к данным, и логику домена. Как это часто бывает в мире программного обеспечения, граница между упомянутыми типовыми решениями весьма приблизительна, однако игнорировать ее все-таки не следует.

Поскольку **активная запись** тесно привязана к базе данных, рекомендую использовать в этом типовом решении статические методы поиска. Однако я не вижу смысла выделять методы поиска в отдельный класс, как это делалось в **шлюзе записи данных** (здесь это не нужно, да и тестировать будет легче).

Как и другие типовые решения, предназначенные для работы с таблицами, **активную запись** можно применять не только к таблицам, но и к представлениям или запросам.

Назначение

Активная запись хорошо подходит для реализации не слишком сложной логики домена, в частности операций создания, считывания, обновления и удаления. Кроме того, она прекрасно справляется с извлечением и проверкой на правильность отдельной записи.

Как уже отмечалось, при разработке **модели предметной области** основная проблема заключается в выборе между **активной записью** и **преобразователем данных (Data Mapper, 187)**. Преимуществом **активной записи** является простота ее реализации. Недостаток же состоит в том, что **активные записи** хороши только тогда, когда точно отображаются на таблицы базы данных (изоморфная схема). Если бизнес-логика приложения достаточно сложна, вам наверняка захочется использовать имеющиеся отношения, коллекции, наследование и т.п. Все это не слишком хорошо отображается на **активную запись**, а добавление этих элементов "по частям" приведет к страшной неразберихе. В подобных ситуациях лучше воспользоваться **преобразователем данных**.

Еще одним недостатком использования **активной записи** является тесная зависимость структуры ее объектов от структуры базы данных. В этом случае изменить структуру базы данных или **активной записи** довольно сложно, а ведь по мере развития проекта подобная необходимость возникает очень и очень часто.

Активную запись хорошо сочетать со **сценарием транзакции**, особенно если вас смущает постоянное повторение одного и того же кода и сложность обновления таблиц и сценариев; подобные ситуации нередко сопровождают использование **сценариев транзакции**. В этом случае вы можете приступить к созданию **активных записей**, постепенно перенося в них повторяющуюся логику. В качестве еще одного полезного приема могу порекомендовать создать для таблицы оболочку в виде **шлюза (Gateway, 483)** и затем постепенно перенести в нее логику, превращая шлюз в **активную запись**.

Пример: простой класс Person (Java)

Рассмотрим простой (даже слишком простой) пример, иллюстрирующий основные аспекты применения **активной записи**. Для начала создадим класс `Person`.

```
class Person...

   private String lastName;
   private String firstName;
   private int numberOfDependents;
```

Помимо этих полей, класс `Person` наследует от своего суперкласса поле `ID`.

База данных имеет аналогичную структуру.

```
create table people (ID int primary key, lastname varchar,
↳firstname varchar, number_of_dependents int)
```

Для поиска и загрузки данных в объект применяются статические методы класса `Person`.

```
class Person...

   private final static String findStatementString =
```

```
        "SELECT id, lastname, firstname,
↳number_of_dependents" +
        "FROM people" +
        "WHERE id = ?";
   public static Person find(Long id) {
      Person result = (Person) Registry.getPerson(id);
      if (result != null) return result;
      PreparedStatement findStatement = null;
      ResultSet rs = null;
      try {
         findStatement = DB.prepare(findStatementString);
         findStatement.setLong(1, id.longValue());
         rs = findStatement.executeQuery();
         rs.next();
         result = load(rs);
         return result;
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {
         DB.cleanUp(findStatement,rs);
      }
   }
    public static Person find(long id) {
       return find(new Long(id));
    }
    public static Person load(ResultSet rs)
↳throws SQLException {
       Long id = new Long(rs.getLong(1));
       Person result = (Person) Registry.getPerson(id);
       if (result != null) return result;
       String lastNameArg = rs.getString(2);
       String firstNameArg = rs.getString(3);
       int numDependentsArg = rs.getInt(4);
       result = new Person(id, lastNameArg, firstNameArg,
↳numDependentsArg);
       Registry.addPerson(result);
       return result;
   }
```

Для обновления объекта применяется простой метод экземпляра.

```
class Person...

   private final static String updateStatementString =
         "UPDATE people" +
         "set lastname = ?, firstname = ?,
↳number_of_dependents = ?" +
         "where id = ?";
   public void update() {
      PreparedStatement updateStatement = null;
      try {
         updateStatement = DB.prepare(updateStatementString);
         updateStatement.setString(1, lastName);
         updateStatement.setString(2, firstName);
```

```
            updateStatement.setInt(3, numberOfDependents);
            updateStatement.setInt(4, getID().intValue());
            updateStatement.execute();
        } catch (Exception e) {
            throw new ApplicationException(e);
        } finally {
            DB.cleanUp(updateStatement);
        }
    }
```

Процедура вставки тоже не слишком сложна.

```
class Person...

    private final static String insertStatementString =
            "INSERT INTO people VALUES (?,?,?,?)";
    public Long insert() {
        PreparedStatement insertStatement = null;
        try {
            insertStatement = DB.prepare(insertStatementString);
             setID(findNextDatabaseId());
             insertStatement.setInt(1, getID().intValue());
             insertStatement.setString(2, lastName);
             insertStatement.setString(3, firstName);
             insertStatement.setInt(4, numberOfDependents);
             insertStatement.execute();
             Registry.addPerson(this);
             return getID();
        } catch (Exception e) {
            throw new ApplicationException(e);
        } finally {
            DB.cleanUp(insertStatement);
        }
    }
```

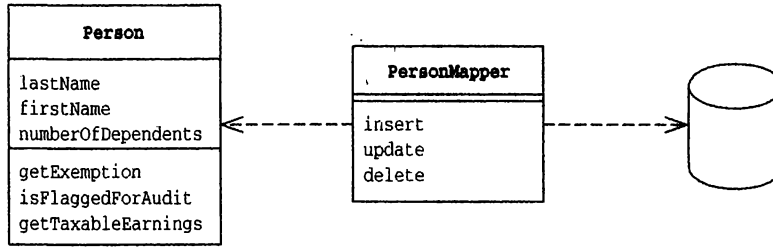
Вся бизнес-логика, например определение суммы вычетов при расчете налогов, должна быть реализована непосредственно в классе `Person`.

```
class Person...

    public Money getExemption() {
    Money baseExemption = Money.dollars(1500);
    Money dependentExemption = Money.dollars(750);
    return baseExemption.add(dependentExemption.multiply(
↳this.getNumberOfDependents()));
    }
```

Преобразователь данных (Data Mapper)

Слой **преобразователей (Mapper, 489)**, *который осуществляет передачу данных между объектами и базой данных, сохраняя последние независимыми друг от друга и от самого преобразователя*



Объекты и реляционные СУБД используют разные механизмы структурирования данных. В реляционных базах данных не отображаются многие характеристики объектов, в частности коллекции и наследование. При построении объектной модели с большим объемом бизнес-логики эти механизмы позволяют лучше организовать данные и соответствующее им поведение. С другой стороны, использование подобных механизмов приводит к несовпадению объектной и реляционной схем.

Объектная модель и реляционная СУБД должны обмениваться данными. Несовпадение схем делает эту задачу крайне сложной. Если объект "знает" о структуре реляционной базы данных, изменение одного из них приводит к необходимости изменения другого.

Типовое решение **преобразователь данных** представляет собой слой программного обеспечения, которое отделяет объекты, расположенные в оперативной памяти, от базы данных. В функции **преобразователя данных** входит передача данных между объектами и базой данных и изоляция их друг от друга. Благодаря использованию этого типового решения объекты, расположенные в оперативной памяти, могут даже не "подозревать" о самом факте присутствия базы данных. Им не нужен SQL-интерфейс и тем более схема базы данных. (В свою очередь, схема базы данных никогда не "знает" об объектах, которые ее используют.) Более того, поскольку **преобразователь данных** является разновидностью **преобразователя**, он полностью скрыт от уровня домена.

Принцип действия

Основной функцией **преобразователя данных** является отделение домена от источника данных, однако реализовать эту функцию можно лишь с учетом множества деталей. Кроме того, конструкция слоев отображения также может быть реализована по-разному. Поэтому многие приведенные ниже советы носят достаточно общий характер — я пытался найти как можно более универсальное решение, чтобы научить вас отделять рыбу от костей.

Для начала рассмотрим пример элементарного **преобразователя данных**. Структура этого слоя слишком проста и, возможно, не стоит тех усилий, которые могут быть потрачены на ее реализацию. Кроме того, простая структура преобразователя влечет за собой применение более простых (и поэтому лучших) типовых решений, что вряд ли

подойдет для реальных систем. Тем не менее начинать объяснение новых идей лучше именно на простых примерах.

В этом примере у нас есть классы `Person` и `PersonMapper`. Для загрузки данных в объект `Person` клиент вызывает метод поиска класса `PersonMapper` (рис. 10.3). Преобразователь использует **коллекцию объектов (Identity Map, 216)** для проверки, загружены ли данные о запрашиваемом лице; если нет, он их загружает.

Выполнение обновлений показано на рис. 10.4. Клиент указывает преобразователю на необходимость сохранить объект домена. Преобразователь извлекает данные из объекта домена и отсылает их в базу данных.

При необходимости весь слой **преобразователя данных** может быть заменен, например, чтобы провести тестирование или чтобы использовать один и тот же уровень домена для работы с несколькими базами данных.

Рассмотренный нами элементарный **преобразователь данных** всего лишь отображает таблицу базы данных на эквивалентный ей объект по принципу "столбец-на-поле". Конечно же, в реальной жизни не все так просто. Преобразователи должны уметь обрабатывать классы, поля которых объединяются одной таблицей, классы, соответствующие нескольким таблицам, классы с наследованием, а также справляться со связыванием загруженных объектов. Все типовые решения объектно-реляционного отображения, рассмотренные в этой книге, так или иначе направлены на решение подобных проблем. Как правило, эти решения легче создавать на основе **преобразователя данных**, чем с помощью каких-либо других средств.

Для выполнения вставки и обновления данных слой отображения должен знать, какие объекты были изменены, какие созданы, а какие уничтожены. Кроме того, все эти действия нужно каким-то образом "уместить" в рамки транзакции. Хороший способ организовать механизм обновления — использовать типовое решение **единица работы (Unit of Work, 205)**.

В схеме, показанной на рис. 10.3, предполагалось, что один вызов метода поиска приводит к выполнению одного SQL-запроса. Это не всегда так. Например, загрузка данных о заказе, состоящем из нескольких пунктов, может включать в себя загрузку каждого пункта заказа. Обычно запрос клиента приводит к загрузке целого графа связанных между собой объектов. В этом случае разработчик преобразователя должен решить, как много объектов можно загрузить за один раз. Поскольку число обращений к базе данных должно быть как можно меньшим, методам поиска должно быть известно, как клиенты используют объекты, чтобы определить оптимальное количество загружаемых данных.

Описанный пример подводит нас к ситуации, когда в результате выполнения одного запроса преобразователь загружает несколько классов объектов домена. Если вы хотите загрузить заказы и пункты заказов, это можно сделать с помощью одного запроса, применив операцию соединения к таблицам заказов и заказываемых товаров. Полученное результирующее множество записей применяется для загрузки экземпляров класса заказов и класса пунктов заказа (см. стр. 220).

Как правило, объекты тесно связаны между собой, поэтому на каком-то этапе загрузку данных следует прерывать. В противном случае выполнение одного запроса может привести к загрузке всей базы данных! Для решения этой проблемы и одновременной минимизации влияния на объекты, расположенные в памяти, слой отображения использует **загрузку по требованию (Lazy Load, 220)**. По этой причине объекты приложения не могут совсем ничего не "знать" о слое отображения. Скорее всего, они должны быть "осведомлены" о методах поиска и некоторых других механизмах.

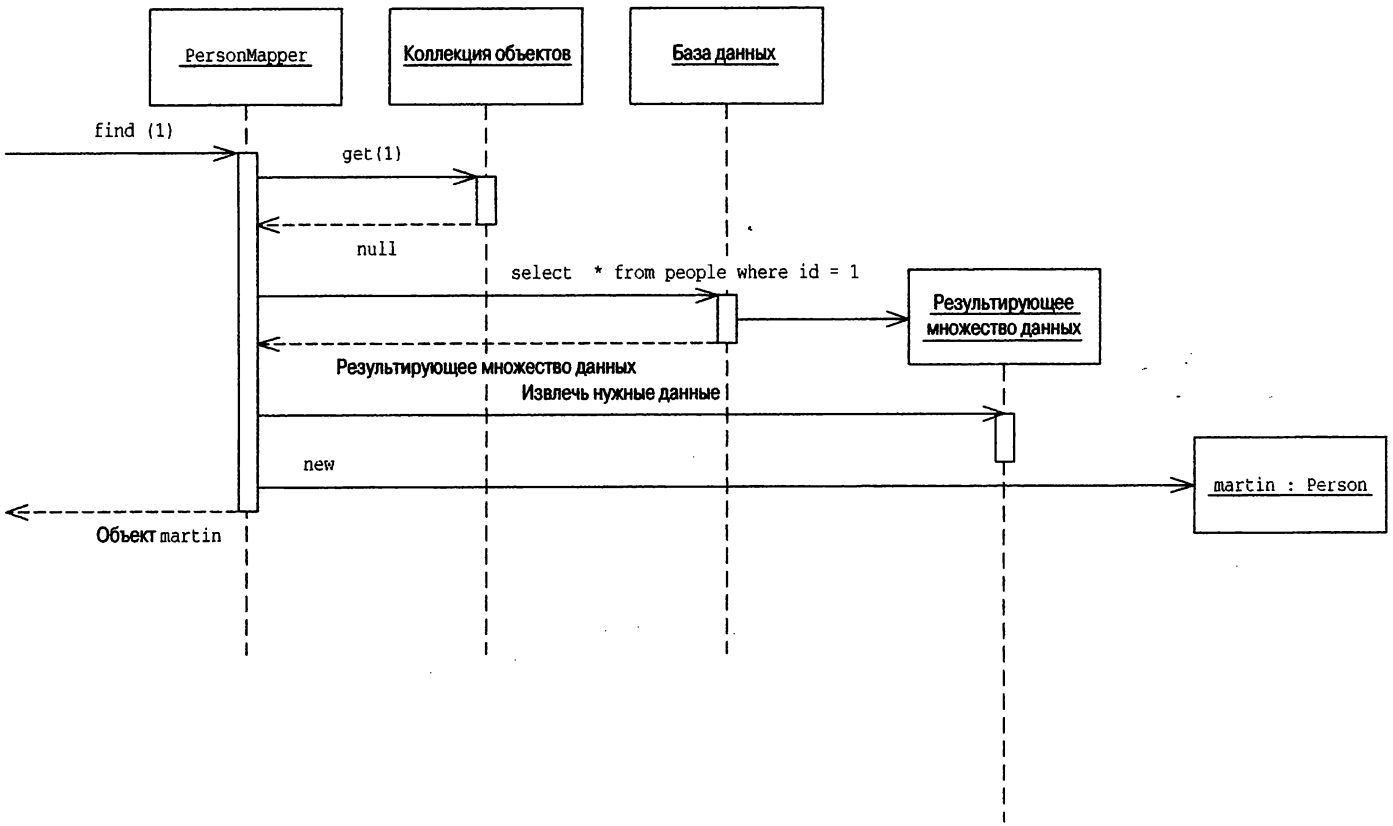


Рис. 10.3. Извлечение данных из базы данных

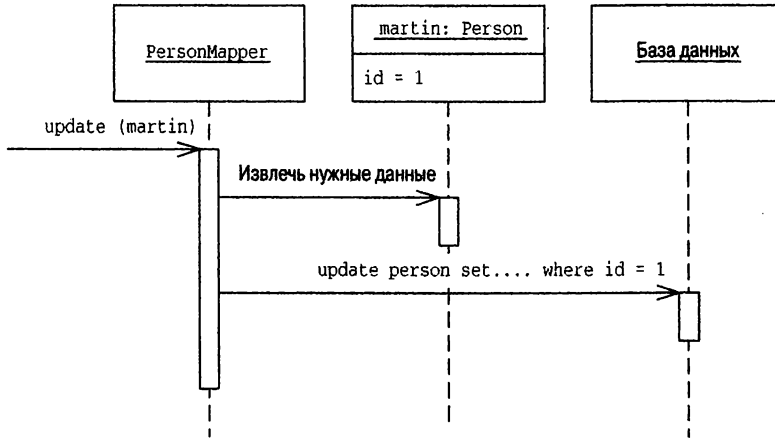


Рис. 10.4. Обновление данных

В приложении может быть один или несколько **преобразователей данных**. Если код для преобразователей пишется вручную, рекомендую создать по одному преобразователю для каждого класса домена или для корневого класса в иерархии доменов. Если же вы используете **отображение метаданных (Metadata Mapping, 325)**, можете ограничиться и одним классом преобразователя. В последнем случае все зависит от количества методов поиска. Если приложение достаточно большое, методов поиска может оказаться слишком много для одного преобразователя, поэтому разумнее будет разбить их на несколько классов, создав отдельный класс для каждого класса домена или корневой класс в иерархии доменов. У вас появится масса небольших классов с методами поиска, однако теперь разработчику будет гораздо проще найти то, что ему нужно.

Как и все другие поисковые механизмы баз данных, упомянутые методы поиска должны использовать **коллекцию объектов**, чтобы отслеживать, какие записи уже были считаны из базы данных. Для этого можно создать **реестр (Registry, 495) коллекций объектов** либо назначить каждому методу поиска свою **коллекцию объектов** (при условии, что в течение одного сеанса каждый класс использует только один метод поиска).

Обращение к методам поиска

Чтобы получить возможность работать с объектом, его нужно загрузить из базы данных. Как правило, этот процесс запускается слоем представления, который выполняет загрузку некоторых начальных объектов, после чего передает управление слою домена. Слой домена последовательно загружает объект за объектом, используя установленные между ними ассоциации. Этот прием весьма эффективен при условии, что у слоя домена есть все объекты, которые должны быть загружены в оперативную память, или же что слой домена загружает дополнительные объекты только по мере необходимости путем применения **загрузки по требованию**.

Иногда объектам домена может понадобиться обратиться к методам поиска, определенным в **преобразователе данных**. Как показывает практика, удачная реализация **загрузки по требованию** позволяет полностью избежать подобных ситуаций. Конечно же, при разработке простых приложений применение ассоциаций и **загрузки по требованию** может не оправдать затраченных усилий, однако добавлять лишнюю зависимость объектов домена от **преобразователя данных** тоже не следует.

В качестве решения этой дилеммы можно предложить использование **отделенного интерфейса (Separated Interface, 492)**. В этом случае все методы поиска, применяемые кодом домена, можно вынести в интерфейсный класс и поместить его в пакет домена.

Отображение данных на поля объектов домена

Преобразователи должны иметь доступ к полям объектов домена. Зачастую это вызывает трудности, поскольку предполагает наличие методов, открытых для преобразователей, чего в бизнес-логике быть не должно. (Я исхожу из предположения, что вы не совершили страшную ошибку, оставив поля объектов домена открытыми (public).) Универсального решения этой проблемы не существует. Вы можете применить более низкий уровень видимости, поместив преобразователи "поближе" к объектам домена (например, в одном пакете, как это делается в Java), однако подобное решение крайне запутает глобальную картину зависимостей, потому что другие части системы, которые "знают" об объектах домена, не должны "знать" о преобразователях. Вы можете использовать механизм отражения, который зачастую позволяет обойти правила видимости конкретного языка программирования. Это довольно медленный метод, однако он может оказаться гораздо быстрее выполнения SQL-запроса. И наконец, вы можете использовать открытые методы, предварительно снабдив их полями состояния, генерирующими исключение при попытке использовать эти методы не для загрузки данных преобразователем. В этом случае назовите методы так, чтобы их по ошибке не приняли за обычные get- и set-методы.

Описанная проблема тесно связана с вопросом создания объекта. Последнее можно осуществить двумя способами. Первый состоит в том, чтобы создать объект с помощью *конструктора с инициализацией (rich constructor)*, который сразу же заполнит новый объект всеми необходимыми данными. Второй заключается в создании пустого объекта и последующем заполнении его данными. Обычно я предпочитаю первый вариант — так приятно, когда объект уже "укомплектован". Вдобавок ко всему это дает возможность легко сделать какое-нибудь поле неизменяемым — достаточно проследить, чтобы ни один метод не изменял значение этого поля.

При использовании конструктора с инициализацией возникает проблема, связанная с наличием циклических ссылок. Если у вас есть два объекта, ссылающихся друг на друга, попытка загрузки первого объекта приведет к загрузке второго объекта, что, в свою очередь, снова приведет к загрузке первого объекта и так до тех пор, пока не произойдет переполнение стека. Возможный выход — описать **частный случай (Special Case, 511)**. Обычно это делается с использованием типового решения **загрузка по требованию**. Написание кода для **частного случая** — задача далеко не из легких, поэтому рекомендую попробовать что-нибудь другое, например воспользоваться конструктором без аргументов для создания *пустого объекта (empty object)*. Создайте пустой объект и сразу же поместите его в **коллекцию объектов**. Теперь, если загружаемые объекты окажутся связанными циклической ссылкой, **коллекция объектов** возвратит нужное значение для прекращения "рекурсивной" загрузки.

Как правило, заполнение пустого объекта осуществляется посредством set-методов. Некоторые из них могут устанавливать значения полей, которые после загрузки данных должны оставаться неизменяемыми. Чтобы предотвратить случайное изменение этих полей после загрузки объекта, присвойте set-методам специальные имена и, возможно, оснастите их полями проверки состояния. Кроме того, вы можете выполнить загрузку данных с использованием механизма отражения.

Отображения на основе метаданных

Разрабатывая корпоративное приложение для взаимодействия с базой данных, необходимо решить, как поля объектов домена будут отображаться на столбцы таблиц базы данных. Самый простой (и часто самый лучший) способ выполнить это — явно описать отображение в коде, что требует создания по одному классу преобразователя на каждый объект домена. Преобразователь выполняет отображение путем присвоения значений и содержит в себе SQL-команды для доступа к базе данных (обычно они хранятся в виде текстовых констант). В качестве альтернативы этому способу можно предложить использование **отображения метаданных**, которое сохраняет метаданные таблиц в виде обыкновенных данных, в классе или в отдельном файле. Огромное преимущество метаданных заключается в том, что они позволяют вносить изменения в преобразователь путем генерации кода или применения отражающего программирования (reflective programming) без написания дополнительного кода вручную.

Назначение

В большинстве случаев **преобразователь данных** применяется для того, чтобы схема базы данных и объектная модель могли изменяться независимо друг от друга. Как правило, подобная необходимость возникает при использовании **модели предметной области**. Основным преимуществом **преобразователя данных** является возможность работы с **моделью предметной области** без учета структуры базы данных как в процессе проектирования, так и во время сборки и тестирования проекта. В этом случае объектам домена ничего не известно о структуре базы данных, поскольку все отображения выполняются преобразователями.

Применение **преобразователей данных** помогает и в написании кода, поскольку позволяет работать с объектами домена без необходимости понимать принцип хранения соответствующей информации в базе данных. Изменение **модели предметной области** не требует изменения структуры базы данных и наоборот, что крайне важно при наличии сложных отображений, особенно при использовании уже существующих баз данных.

Разумеется, за все удобства нужно платить. "Ценой" использования **преобразователя данных** является необходимость реализации дополнительного слоя кода, чего можно избежать, применив, скажем, **активную запись (Active Record, 182)**. Поэтому основным критерием выбора того или иного типового решения является сложность бизнес-логики. Если бизнес-логика довольно проста, ее, скорее всего, можно реализовать и без применения **модели предметной области** или **преобразователя данных**. В свою очередь, реализация более сложной логики невозможна без использования **модели предметной области** и, как следствие этого, **преобразователя данных**.

Я бы не стал использовать **преобразователь данных** без **модели предметной области**. Но можно ли использовать **модель предметной области** без **преобразователя данных**? Если модель довольно проста, а ее разработчики сами контролируют изменения структуры базы данных, доступ объектов домена к базе данных можно осуществлять непосредственно с помощью **активной записи**. В этом случае роль преобразователя с успехом выполняют сами объекты домена. Тем не менее по мере усложнения логики домена функции доступа к базе данных лучше вынести в отдельный слой.

Хочу также обратить ваше внимание на то, что вам не понадобится разрабатывать полнофункциональный слой отображения базы данных на объекты домена с "нуля". Это весьма сложно, да и на рынке программного обеспечения существует масса подобных

продуктов. В большинстве случаев рекомендую приобрести готовый преобразователь, вместо того чтобы заниматься его разработкой самому.

Пример: простой преобразователь данных (Java)

Рассмотрим один из вариантов использования **преобразователя данных**. Я специально подобрал такой простой пример, чтобы помочь вам лучше разобраться в базовой структуре этого типового решения. В нашем случае используется класс `Person`, предназначенный для загрузки изоморфной ему таблицы `people`.

```
class Person...

   private String lastName;
   private String firstName;
   private int numberOfDependents;
```

Схема базы данных выглядит следующим образом:

```
create table people (ID int primary key, lastname varchar,
↳firstname varchar, number_of_dependents int)
```

Здесь рассматривается простой случай, когда метод поиска и **коллекция объектов** реализованы прямо в классе `PersonMapper`. Впрочем, я добавил типовое решение **супертип слоя (Layer Supertype, 491)**, представленное классом `AbstractMapper`, чтобы показать, какие общие методы могут быть вынесены в суперкласс. При выполнении загрузки объект `AbstractMapper` проверяет, нет ли запрашиваемых данных в **коллекции объектов**, и в случае отрицательного ответа извлекает запрошенные данные из базы данных.

Выполнение поиска начинается в классе `PersonMapper`, который передает вызов соответствующему абстрактному методу поиска класса `AbstractMapper`. Поиск записи выполняется по ее идентификатору.

```
class PersonMapper...

   protected String findStatement() {
      return "SELECT " + COLUMNS +
         " FROM people" +
         " WHERE id = ?";
   }
   public static final String COLUMNS = "id, lastname,
↳firstname, number_of_dependents ";
   public Person find(Long id) {
      return (Person) abstractFind(id);
   }
   public Person find(long id) {
      return find(new Long(id));
   }

class AbstractMapper...

   protected Map loadedMap = new HashMap();
   abstract protected String findStatement();
```

```
   protected DomainObject abstractFind(Long id) {
      DomainObject result = (DomainObject) loadedMap.get(id);
      if (result != null) return result;
      PreparedStatement findStatement = null;
      try {
         findStatement = DB.prepare(findStatement());
         findStatement.setLong(1, id.longValue());
         ResultSet rs = findStatement.executeQuery();
         rs.next();
         result = load(rs);
         return result;
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {
         DB.cleanUp(findStatement);
      }
   }
```

Метод поиска вызывает метод загрузки, выполнение которого разбито между классами `AbstractMapper` и `PersonMapper`. Объект `AbstractMapper` проверяет идентификатор запрашиваемой записи, извлекая его из базы данных и регистрируя новый объект в **коллекции объектов**.

```
class AbstractMapper...

   protected DomainObject load(ResultSet rs)
throws SQLException {
      Long id = new Long(rs.getLong(1));
      if (loadedMap.containsKey(id)) return
(DomainObject) loadedMap.get(id);
      DomainObject result = doLoad(id,rs);
      loadedMap.put(id,result);
      return result;
   }
   abstract protected DomainObject doLoad(Long id,
ResultSet rs) throws
      SQLException;

class PersonMapper...

   protected DomainObject doLoad(Long id, ResultSet rs)
throws SQLException {
      String lastNameArg = rs.getString(2);
      String firstNameArg = rs.getString(3);
      int numDependentsArg = rs.getInt(4);
      return new Person(id, lastNameArg, firstNameArg,
numDependentsArg);
   }
```

Обратите внимание, что **коллекция объектов** проверяется дважды — методом `abstractFind` и методом `load`. На это есть свои причины.

Вначале **коллекцию объектов** проверяет метод поиска. Если искомый объект уже есть в коллекции, это избавляет от необходимости обращения к базе данных — зачем же проделывать такой длинный путь, когда без него можно обойтись? Впрочем, такую же проверку должен выполнять и метод загрузки, поскольку некоторые запросы могут не быть полностью разрешены путем обращения к **коллекции объектов**. Предположим, я хочу найти в базе данных всех сотрудников, чьи фамилии удовлетворяют некоторому критерию поиска. Я не уверен, что все необходимые мне записи уже были загружены в память, поэтому должен обратиться к базе данных и выполнить запрос.

```
class PersonMapper...

   private static String findLastNameStatement =
         "SELECT " + COLUMNS +
         " FROM people " +
         " WHERE UPPER(lastname) like UPPER(?)" +
         " ORDER BY lastname";
   public List findByLastName(String name) {
      PreparedStatement stmt = null;
      ResultSet rs = null;
      try {
         stmt = DB.prepare(findLastNameStatement);
         stmt.setString(1, name);
         rs = stmt.executeQuery();
         return loadAll(rs);
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {
         DB.cleanUp(stmt,rs);
      }
   }

class AbstractMapper...

   protected List loadAll(ResultSet rs) throws SQLException {
      List result = new ArrayList();
      while (rs.next())
         result.add(load(rs));
      return result;
   }
```

Выполняя этот запрос, я могу извлечь строки, которые соответствуют уже загруженным записям. Чтобы избежать дублирования, я вынужден еще раз проверить **коллекцию объектов**.

Подобная реализация метода поиска в каждом производном классе связана с написанием несложных, но постоянно повторяющихся фрагментов кода. Этого можно избежать, добавив к суперклассу общий метод.

```
class AbstractMapper...

   public List findMany(StatementSource source) {
      PreparedStatement stmt = null;
      ResultSet rs = null;
```

```
        try {
            stmt = DB.prepare(source.sql());
            for (int i = 0;i < source.parameters().length; i++)
                stmt.setObject(i+1, source.parameters()[i]);
            rs = stmt.executeQuery();
            return loadAll(rs);
        } catch (SQLException e) {
            throw new ApplicationException(e);
        } finally {
            DB.cleanUp(stmt,rs);
        }
    }
```

Для работы этого метода мне нужен интерфейс, который бы выполнял роль оболочки для строк с SQL-выражениями и для параметров, которые должны быть загружены в эти выражения.

```
interface StatementSource...

    String sql();
    Object [] parameters();
```

Теперь этот интерфейс можно реализовать посредством вложенного класса.

```
class PersonMapper...

    public List findByLastName2(String pattern) {
        return findMany(new FindByLastName(pattern));
    }
    static class FindByLastName implements StatementSource {
        private String lastName;
        public FindByLastName(String lastName) {
            this.lastName = lastName;
        }
        public String sql() {
            return
                "SELECT " + COLUMNS +
                " FROM people " +
                " WHERE UPPER(lastname) like UPPER(?)" +
                " ORDER BY lastname";
        }
        public Object[] parameters() {
            Object[] result = {lastName};
            return result;
        }
    }
```

Подобные действия по реализации интерфейса можно выполнить и в других местах, где встречаются повторяющиеся вызовы SQL-выражений. Я постарался сделать этот пример более понятным, чтобы вам было легче применить его в своих проектах. Если вам постоянно приходится повторять одни и те же фрагменты кода, подумайте о реализации чего-нибудь подобного.

Обновление полей выполняется с учетом типа каждого из них.

```
class PersonMapper...

   private static final String updateStatementString =
         "UPDATE people " +
         " SET lastname = ?, firstname = ?,
↳number_of_dependents = ?" +
         " WHERE id = ?";
   public void update(Person subject) {
      PreparedStatement updateStatement = null;
      try {
         updateStatement = DB.prepare(updateStatementString);
         updateStatement.setString(1, subject.getLastName());
         updateStatement.setString(2, subject.getFirstName());
         updateStatement.setInt(3,
↳subject.getNumberOfDependents());
         updateStatement.setInt(4, subject.getID().intValue());
         updateStatement.execute();
      } catch (Exception e) {
         throw new ApplicationException(e);
      } finally {
         DB.cleanUp(updateStatement);
      }
   }
```

Часть операций по выполнению вставки может быть вынесена в **супертип слоя**.

```
class AbstractMapper...

   public Long insert(DomainObject subject) {
      PreparedStatement insertStatement = null;
      try {
         insertStatement = DB.prepare(insertStatement());
         subject.setID(findNextDatabaseId());
         insertStatement.setInt(1, subject.getID().intValue());
         doInsert(subject, insertStatement);
         insertStatement.execute();
         loadedMap.put(subject.getID(), subject);
         return subject.getID();
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {
         DB.cleanUp(insertStatement);
      }
   }
   abstract protected String insertStatement();
   abstract protected void doInsert(DomainObject subject,
↳PreparedStatement insertStatement)
         throws SQLException;

class PersonMapper...

   protected String insertStatement() {
```

```
    return "INSERT INTO people VALUES (?,?,?,?)";
}
protected void doInsert(
    DomainObject abstractSubject,
    PreparedStatement stmt)
    throws SQLException
{
    Person subject = (Person) abstractSubject;
    stmt.setString(2, subject.getLastName());
    stmt.setString(3, subject.getFirstName());
    stmt.setInt(4, subject.getNumberOfDependents());
}
```

Пример: отделение методов поиска (Java)

Чтобы объекты домена могли обращаться к методам поиска, можно воспользоваться **отделенным интерфейсом**, который отделит методы поиска от преобразователей (рис. 10.5). Интерфейсы поиска можно поместить в отдельный пакет, "видимый" для слоя домена, или же, как показано на рис. 10.5, непосредственно в слой домена.

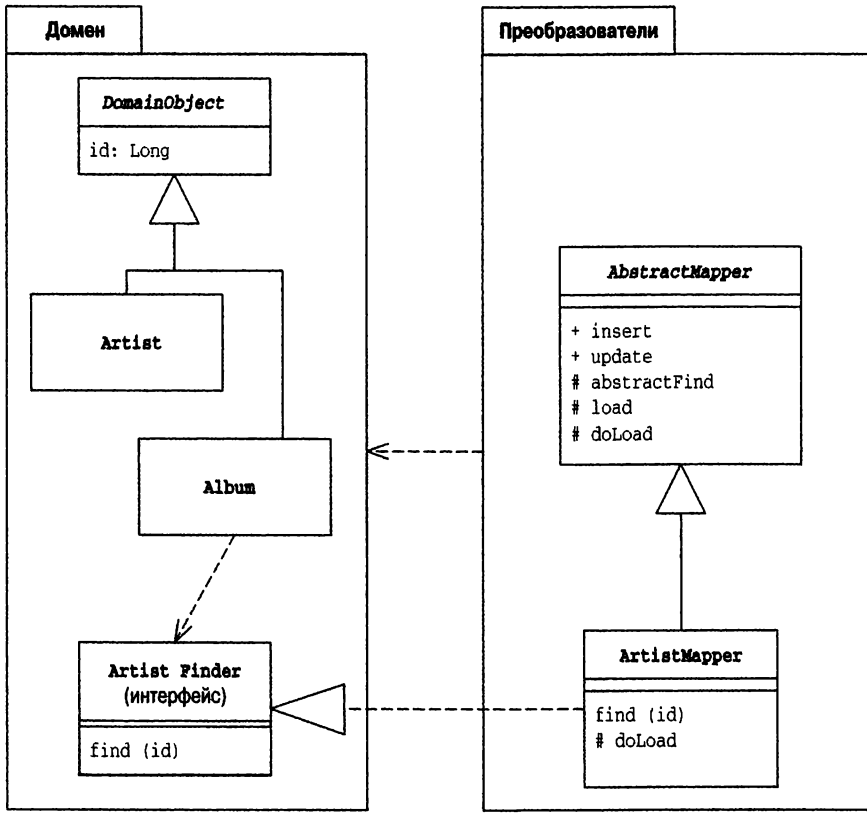


Рис. 10.5. Определение интерфейса поиска в пакете домена

Довольно часто поиск объекта выполняется по искусственному идентификатору (surrogate ID), образованному на основе нескольких первичных ключей поиска. Большая часть кода подобных методов достаточно универсальна, поэтому ее удобно вынести в **супертип слоя**. Все, что для этого нужно, — создать **супертип слоя** для объектов домена, который бы "знал" об идентификаторах последних.

Объявление методов поиска содержится в интерфейсе поиска. Как правило, их лучше не делать универсальными, поскольку нам нужно знать тип возвращаемых значений.

```
interface ArtistFinder...

  Artist find(Long id);
  Artist find(long id);
```

Интерфейс поиска рекомендуется объявлять в пакете домена, а сами методы размещать в **реестре (Registry, 495)**. В нашем примере интерфейс поиска реализован в классе преобразователя.

```
class ArtistMapper implements ArtistFinder...

   public Artist find(Long id) {
      return (Artist) abstractFind(id);
   }
   public Artist find (long id) {
      return find(new Long(id));
   }
```

Основную работу по выполнению поиска берет на себя **супертип слоя**, который проверяет **коллекцию объектов**, чтобы узнать, нет ли запрошенного объекта в оперативной памяти. Если объекта нет, **супертип слоя** подставляет в SQL-выражение, переданное объектом `ArtistMapper`, нужные параметры и выполняет его.

```
class AbstractMapper...

   abstract protected String findStatement();
   protected Map loadedMap = new HashMap();
   protected DomainObject abstractFind(Long id) {
      DomainObject result = (DomainObject) loadedMap.get(id);
      if (result != null) return result;
      PreparedStatement stmt = null;
      ResultSet rs = null;
      try {
         stmt = DB.prepare(findStatement());
         stmt.setLong(1, id.longValue());
         rs = stmt.executeQuery();
         rs.next();
         result = load(rs);
         return result;
      }catch (SQLException e) {
        throw new ApplicationException(e);
      } finally {cleanUp(stmt,rs);
      }
   }
```

```
class ArtistMapper...

   protected String findStatement() {
      return "select " + COLUMN_LIST + " from artists art
↳where ID =?";
   }
   public static String COLUMN_LIST = "art.ID, art.name";
```

Метод поиска проверяет, какой объект нужно загрузить — новый или уже существующий. В свою очередь, метод загрузки извлекает из базы данных требующиеся данные и помещает их в новый объект.

```
class AbstractMapper...

   protected DomainObject load(ResultSet rs)
↳throws SQLException {
      Long id = new Long(rs.getLong("id"));
      if (loadedMap.containsKey(id))
↳return (DomainObject) loadedMap.get(id);
      DomainObject result = doLoad(id, rs);
      loadedMap.put(id, result);
      return result;
   }
   abstract protected DomainObject doLoad(Long id, ResultSet rs)
↳throws SQLException;

class ArtistMapper...

   protected DomainObject doLoad(Long id, ResultSet rs)
↳throws SQLException {
      String name = rs.getString("name");
      Artist result = new Artist(id, name);
      return result;
   }
```

Обратите внимание, что метод загрузки также выполняет проверку **коллекции объектов**. Вообще-то в нашем примере это лишнее, однако в реальных ситуациях метод загрузки может быть вызван другими методами поиска, которые не проводили подобной проверки. В этом случае разработчику производного класса требуется всего лишь реализовать метод `doLoad`, чтобы тот загружал необходимые данные, а также возвратить нужное SQL-выражение посредством метода `findStatement`.

Поиск можно проводить и на основе запроса. Предположим, у нас есть база данных альбомов и композиций и нам нужно написать метод поиска, который бы возвращал список всех композиций заданного альбома. Как и прежде, объявление методов поиска выполняется в соответствующем интерфейсе.

```
interface TrackFinder...

   Track find(Long id);
   Track find(long id);
   List findForAlbum(Long albumID);
```

Поскольку речь идет о конкретном методе поиска, его следует реализовать не в **супертипе слоя**, а в специализированном классе `TrackMapper`. Как и раньше, нам понадобится реализовать два метода: один возвращает готовое SQL-выражение, а второй подставляет в него параметры и выполняет запрос.

```
class TrackMapper...

   public static final String findForAlbumStatement =
         "SELECT ID, seq, albumID, title " +
         " FROM tracks " +
         " WHERE albumID = ? ORDER BY seq";
   public List findForAlbum(Long albumID) {
      PreparedStatement stmt = null;
      ResultSet rs = null;
      try {
         stmt = DB.prepare(findForAlbumStatement);
         stmt.setLong(1, albumID.longValue());
         rs = stmt.executeQuery();
         List result = new ArrayList();
         while (rs.next())
            result.add(load(rs));
         return result;
      }catch (SQLException e) {
        throw new ApplicationException(e);
      } finally {cleanUp(stmt,rs);
      }
   }
```

Метод поиска вызывает метод загрузки для каждой строки результирующего множества данных. Метод загрузки создает в оперативной памяти новый объект и заполняет его данными. Как и в предыдущем примере, некоторая часть работы, включая проверку **коллекции объектов**, может быть вынесена в **супертип слоя**.

Пример: создание пустого объекта (Java)

Существует два способа загрузки объекта. Один из них заключается в создании полностью инициализированного объекта с помощью конструктора, как в предыдущих примерах. В этом случае код метода загрузки выглядит приблизительно так, как показано ниже.

```
class AbstractMapper...

   protected DomainObject load(ResultSet rs)
↳throws SQLException {
      Long id = new Long(rs.getLong(1));
      if (loadedMap.containsKey(id))
↳return (DomainObject) loadedMap.get(id);
      DomainObject result = doLoad(id, rs);
      loadedMap.put(id, result);
      return result;
   }
   abstract protected DomainObject doLoad(Long id, ResultSet rs)
```

```
throws SQLException;

class PersonMapper...

   protected DomainObject doLoad(Long id, ResultSet rs)
throws SQLException {
      String lastNameArg = rs.getString(2);
      String firstNameArg = rs.getString(3);
      int numDependentsArg = rs.getInt(4);
      return new Person(id, lastNameArg, firstNameArg,
numDependentsArg);
   }
```

Еще один способ загрузки — создать пустой объект, после чего заполнить его данными, присваивая значения атрибутам объекта с помощью set-методов.

```
class AbstractMapper...

   protected DomainObjectEL load(ResultSet rs)
throws SQLException {
      Long id = new Long(rs.getLong(1));
      if (loadedMap.containsKey(id))
return (DomainObjectEL) loadedMap.get(id);
      DomainObjectEL result = createDomainObject();
      result.setID(id);
      loadedMap.put(id, result);
      doLoad (result, rs);
      return result;
   }
   abstract protected DomainObjectEL createDomainObject();
   abstract protected void doLoad(DomainObjectEL obj,
ResultSet rs) throws SQLException;

class PersonMapper...

   protected DomainObjectEL createDomainObject() {
      return new Person();
   }
   protected void doLoad(DomainObjectEL obj, ResultSet rs)
throws SQLException {
      Person person = (Person) obj;
      person.dbLoadLastName(rs.getString(2));
      person.setFirstName(rs.getString(3));
      person.setNumberOfDependents(rs.getInt(4));
   }
```

Обратите внимание, что здесь для объектов домена был выбран другой **супертип слоя**. Это необходимо для того, чтобы получить контроль над использованием set-методов. Для чего это может понадобиться? Предположим, я хочу, чтобы фамилия сотрудника была неизменяемым полем. В этом случае значение данного поля не должно изменяться после загрузки объекта, поэтому к объекту домена добавляется специальное поле состояния.

```
class DomainObjectEL...

   private int state = LOADING;
   private static final int LOADING = 0;
   private static final int ACTIVE = 1;
   public void beActive() {
      state = ACTIVE;
   }
```

Теперь я могу проверить значение этого поля в процессе загрузки.

```
class Person...

   public void dbLoadLastName(String lastName) {
      assertStateIsLoading();
      this.lastName = lastName;
   }
class DomainObjectEL...

   void assertStateIsLoading() {
     Assert.isTrue(state = LOADING);
   }
```

Описанный метод имеет один недостаток: он содержится в интерфейсе, недоступном большинству клиентов класса `Person`. Поэтому стоит подумать об установке значения поля с помощью отражения, что позволит полностью обойти механизмы защиты Java.

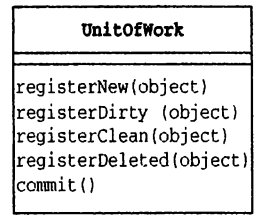
Стоит ли создавать поле состояния? Вообще-то я не уверен. С одной стороны, это позволит перехватывать ошибки, вызванные случайным применением методов обновления. С другой стороны, являются ли эти ошибки настолько серьезными, чтобы ради них имело смысл создавать целый механизм проверки? У меня еще нет окончательного мнения на этот счет.

Глава 11

Объектно-реляционные типовые решения, предназначенные для моделирования поведения

Единица работы (Unit of Work)

Содержит список объектов, охватываемых бизнес-транзакцией, координирует запись изменений в базу данных и разрешает проблемы параллелизма



Извлекая данные из базы данных или записывая в нее обновления, необходимо отслеживать, что именно было изменено; в противном случае сделанные изменения не будут сохранены в базе данных. Точно так же созданные объекты необходимо вставлять, а удаленные — уничтожать.

Разумеется, изменения в базу данных можно вносить при каждом изменении содержимого объектной модели, однако это неизбежно выльется в гигантское количество мелких обращений к базе данных, что значительно снизит производительность. Более того, транзакция должна оставаться открытой на протяжении всего сеанса взаимодействия с базой данных, что никак не применимо к реальной бизнес-транзакции, охватывающей

множество запросов. Наконец, вам придется отслеживать считываемые объекты для того, чтобы не допустить несогласованности данных.

Типовое решение **единица работы** позволяет контролировать все действия, выполняемые в рамках бизнес-транзакции, которые так или иначе связаны с базой данных. По завершении всех действий оно определяет окончательные результаты работы, которые и будут внесены в базу данных.

Принцип действия

Базы данных нужны для того, чтобы вносить в них изменения: добавлять новые объекты или же удалять или обновлять уже существующие. **Единица работы** — это объект, который отслеживает все подобные действия. Как только вы начинаете делать что-нибудь, что может затронуть содержимое базы данных, вы создаете **единицу работы**, которая должна контролировать все выполняемые изменения. Каждый раз, создавая, изменяя или удаляя объект, вы сообщаете об этом **единице работы**. Кроме того, следует сообщать, какие объекты были считаны из базы данных, чтобы не допустить их несогласованности (для чего **единица работы** проверяет, не были ли запрошенные объекты изменены во время считывания).

Когда вы решаете зафиксировать сделанные изменения, **единица работы** определяет, что ей нужно сделать. Она сама открывает транзакцию, выполняет всю необходимую проверку на наличие параллельных операций (с помощью **пессимистической автономной блокировки (Pessimistic Offline Lock, 445)** или **оптимистической автономной блокировки (Optimistic Offline Lock, 434)**) и записывает изменения в базу данных. Разработчики приложений никогда явно не вызывают методы, выполняющие обновления базы данных. Таким образом, им не приходится отслеживать, что было изменено, или беспокоиться о том, в каком порядке необходимо выполнить нужные действия, чтобы не нарушить целостность на уровне ссылок, — **единица работы** сделает это за них.

Разумеется, чтобы **единица работы** действительно вела себя подобным образом, ей должно быть известно, за какими объектами необходимо следить. Об этом ей может сообщить оператор, выполняющий изменение объекта, или же сам объект.

При *регистрации посредством вызывающего оператора (caller registration)* (рис. 11.1) пользователь объекта, который будет подвергнут изменениям, должен зарегистрировать его в **единице работы**. В противном случае изменения объекта зафиксированы в базе данных не будут. Хотя это и допускает появление случайных ошибок со стороны некоторых чересчур забывчивых разработчиков, но позволяет выполнять в оперативной памяти изменения, которые не должны быть записаны в базу данных. Впрочем, я считаю, что применение данного способа может внести слишком много путаницы, поэтому гораздо лучше создать явную копию объекта и проводить свои эксперименты над ней.

При *регистрации посредством изменяемого объекта (object registration)* бремя регистрации перекладывается на методы самого объекта (рис. 11.2). Загружая объект из базы данных, метод загрузки регистрирует его как "достоверный" (clean). В свою очередь, set-методы, изменяющие значения полей объекта, регистрируют его как "измененный" (dirty). Для применения этого способа регистрации **единицу работы** необходимо передать объекту в качестве аргумента или же сохранить в хорошо известном месте. Передача **единицы работы** изменяемому объекту может оказаться заданием не из легких, поэтому гораздо проще поместить ее в какой-нибудь объект сеанса.

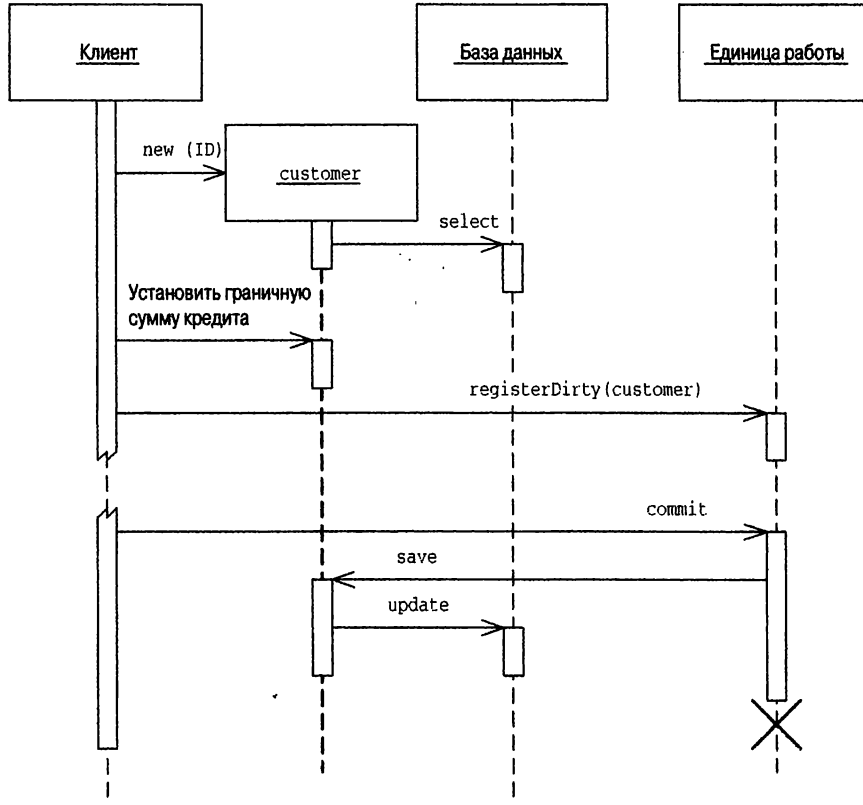


Рис. 11.1. Регистрация изменяемого объекта вызывающим оператором

Даже регистрация изменяемого объекта посредством его методов накладывает на разработчика определенные обязательства; в этом случае он должен не забыть разместить вызовы функций регистрации в нужных методах. Постепенно это входит в привычку, однако от забывчивости не застрахован никто.

Для добавления вызовов функций регистрации вполне естественно применить метод автоматической генерации кода, однако это может быть сделано только тогда, когда сгенерированный код легко отделить от написанного вручную. Данная проблема прекрасно решается средствами аспектно-ориентированного программирования. Кроме того, для выполнения этой задачи можно произвести последующую обработку (post-processing) объектных файлов, как поступил и я. В примере, который рассматривается несколько ниже, процессор последующей обработки проанализировал все файлы Java с расширением `.classes`, отобрал все необходимые методы и вставил в их байт-код вызовы функций регистрации. Конечно же, этот способ не слишком красив, однако он позволяет отделить код базы данных от обычного кода. Аспектно-ориентированный подход дает возможность выполнить это более аккуратно с использованием исходного кода, и я думаю, что по мере распространения средств аспектно-ориентированного программирования данная стратегия войдет в широкое употребление.

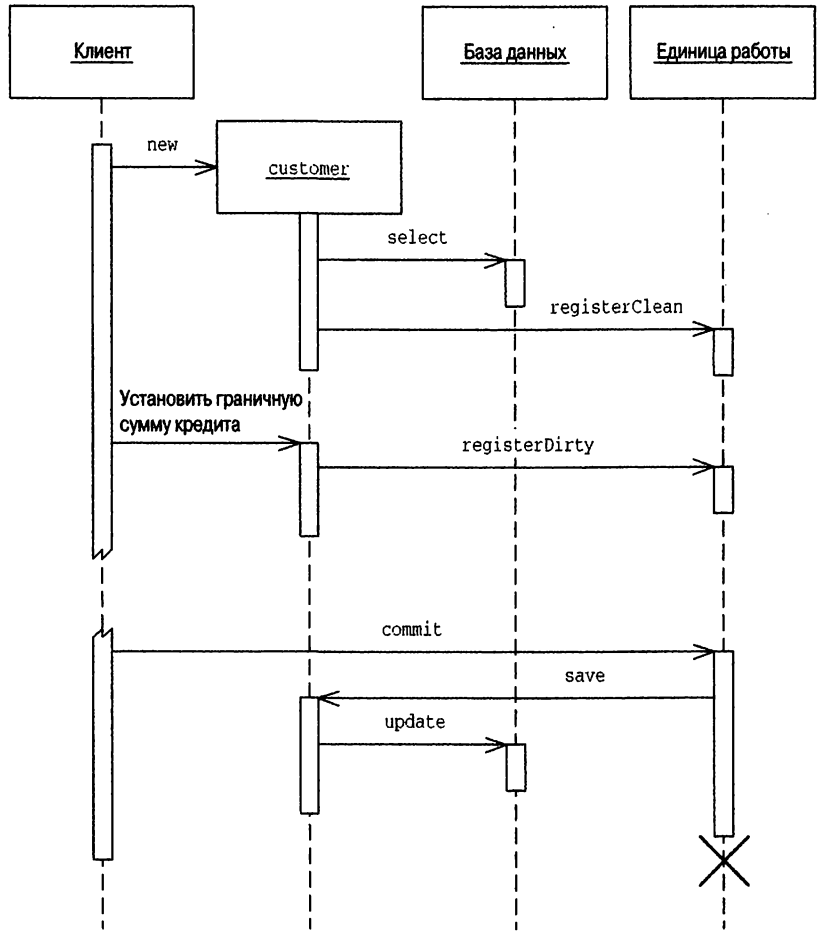


Рис. 11.2. Регистрация изменяемого объекта его же методами

Еще одним распространенным приемом является применение *контроллера единицы работы (unit of work controller)*, который используется в TOPLink (рис. 11.3). Здесь **единица работы** обрабатывает все процедуры считывания из базы данных и регистрирует достоверные объекты. Регистрации измененных объектов не происходит. Вместо этого **единица работы** создает копию объекта во время его считывания и сравнивает измененный объект с исходной копией во время фиксации обновлений. Хотя этот способ требует дополнительных расходов на создание копий, он позволяет проводить выборочное обновление только тех полей, которые действительно были изменены, и обходиться без вызовов функций регистрации в объектах домена. Существует и смешанный подход, при котором **единица работы** создает копии только изменяемых объектов. Это требует регистрации, однако допускает выборочное обновление и значительно уменьшает издержки, связанные с созданием копий, если большинство объектов извлекаются из базы данных только для чтения.

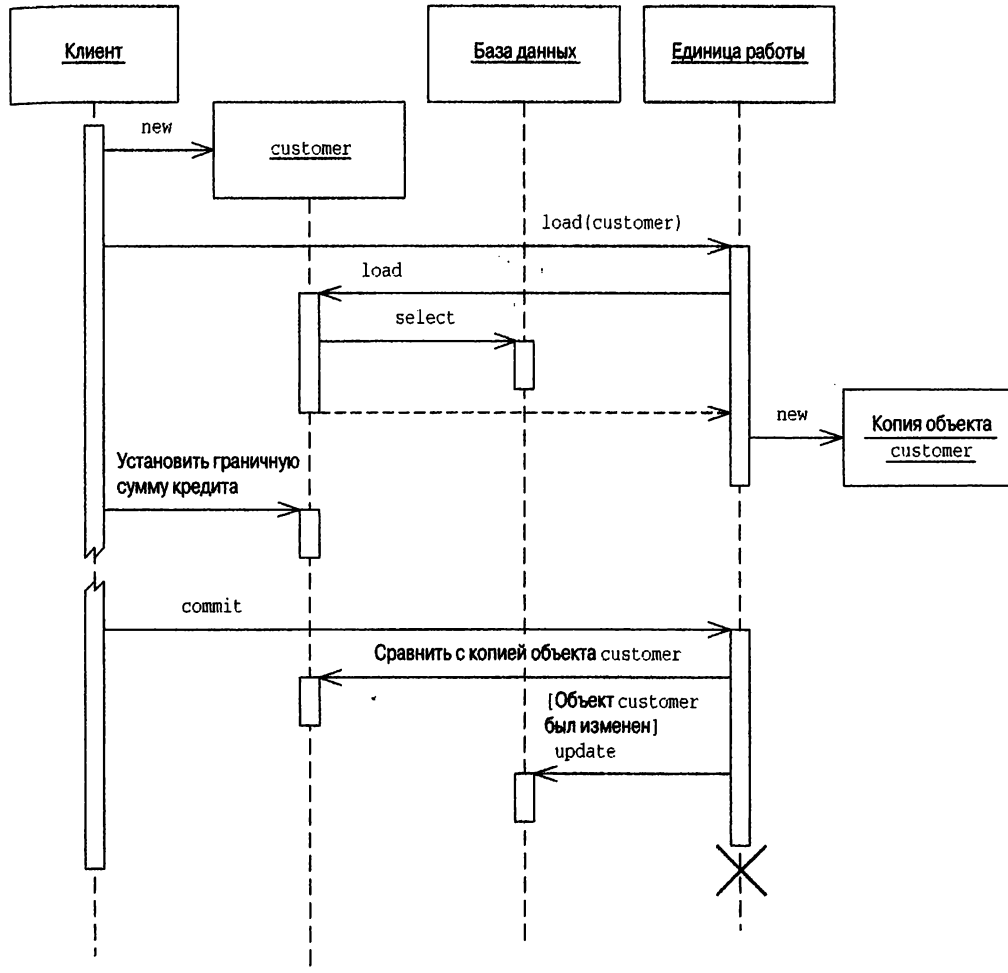


Рис. 11.3. Использование **единицы работы** *в качестве контроллера доступа к базе данных*

Регистрация изменяемого объекта посредством вызывающего оператора может оказаться весьма удобной при создании объектов. Как известно, иногда объекты создаются только на короткое время. Хорошим примером является тестирование объектов домена, которое выполняется гораздо быстрее при отсутствии записи обновлений в базу данных. Это легко осуществить, применив регистрацию посредством вызывающего оператора. Впрочем, существуют и другие решения, например предоставление специального конструктора для создания временных объектов, который не регистрирует объекты в **единице работы**, или, что еще лучше, реализация **частного случая (Special Case, 511) единицы работы**, который при фиксации изменений не выполнял бы никаких действий.

Еще одной областью применения **единицы работы** может стать организация порядка выполнения обновлений, если СУБД использует проверку целостности на уровне ссылок. В большинстве случаев проблемы порядка выполнения обновлений можно избежать;

для этого достаточно убедиться, что проверка целостности на уровне ссылок выполняется только в момент завершения транзакции, а не при каждом вызове SQL-команды. Большинство современных СУБД допускают такую возможность, и в этом случае об упомянутой проблеме можно забыть. Если же это не так, для организации порядка выполнения обновлений удобно воспользоваться **единицей работы**. В небольших системах это можно реализовать вручную, явно определив последовательность обновления таблиц на основе зависимостей между внешними ключами. В более крупных приложениях для определения порядка обновлений лучше воспользоваться метаданными. Дальнейшее рассмотрение этого вопроса выходит за рамки настоящей книги. Обычно для реализации данного подхода используются готовые коммерческие продукты. Если же это придется делать самому, рекомендую отталкиваться от топологической сортировки (во всяком случае, мне это советовали).

Похожий прием может использоваться и для снижения риска возникновения взаимоблокировок. Этого можно добиться, если каждая транзакция будет выполнять обновление таблиц в одном и том же порядке. Зафиксировав этот порядок в **единице работы**, вы гарантируете, что запись обновлений в таблицы всегда будет происходить строго в одной и той же последовательности.

Объектам должно быть известно, где найти текущую **единицу работы**. Для этого удобно воспользоваться **реестром (Registry, 495)**, глобальным по отношению к потоку (thread-scoped). В качестве другого возможного решения можно предложить передачу **единицы работы** нуждающимся в ней объектам — либо в вызовах соответствующих методов, либо при создании объекта. И в том и в другом случае убедитесь, что доступ к **единице работы** может получить только один поток, иначе начнется настоящий кошмар.

Единицу работы хорошо применять и при пакетных обновлениях данных. Суть *пакетного обновления (batch update)* заключается в отправке нескольких SQL-команд как единого целого, чтобы их можно было обработать посредством одного удаленного вызова. Это особенно удобно, когда операции обновления, вставки и удаления идут нескончаемым потоком, одна за другой. Разные среды разработки предусматривают различные уровни поддержки пакетных обновлений. Например, в JDBC есть средство, позволяющее организовывать в пакеты отдельные SQL-выражения. Если подобного средства у вас нет, его можно сымитировать, создав строку путем конкатенации нескольких SQL-выражений и затем использовав ее как одно выражение. Пример выполнения такой операции для платформ Microsoft описывается в [30]. Тем не менее, если вы решитесь на подобную манипуляцию, убедитесь, что она не противоречит правилам предварительной компиляции выражений.

Единица работы может применяться с любыми ресурсами транзакций, а не только с базами данных, поэтому ее можно использовать для манипулирования очередями сообщений и мониторами транзакций.

Особенности .NET-реализации

В .NET для реализации **единицы работы** используется объект `DataSet`, лежащий в основе отсоединенной модели доступа к данным. Последнее обстоятельство немного отличает его от остальных разновидностей этого типового решения. Большинство попадавшихся мне **единиц работы** регистрируют считываемые объекты и отслеживают их изменения. В отличие от этого, в среде .NET данные загружаются из базы данных в объект `DataSet`, дальнейшие изменения

которого происходят в автономном режиме. Объект `DataSet` состоит из таблиц (объекты `DataTable`), которые, в свою очередь, состоят из столбцов (объекты `DataColumn`) и строк (объекты `DataRow`). Таким образом, объект `DataSet` представляет собой "зеркальную" копию множества данных, полученного в результате выполнения одного или нескольких SQL-запросов. У каждой строки `DataRow` есть версии (`Current`, `Original`, `Proposed`) и состояния (`Unchanged`, `Added`, `Deleted`, `Modified`). Наличие последних, а также тот факт, что объектная модель `DataSet` в точности повторяет структуру базы данных, значительно упрощает запись изменений обратно в базу данных.

Назначение

Основным назначением **единицы работы** является отслеживании действий, выполняемых над объектами домена, для дальнейшей синхронизации данных, хранящихся в оперативной памяти, с содержимым базы данных. Если вся работа выполняется в рамках системной транзакции, следует беспокоиться только о тех объектах, которые вы изменяете. Конечно же, для этого лучше воспользоваться **единицей работы**, однако существуют и другие решения.

Пожалуй, наиболее простая альтернатива — явно сохранять объект после каждого изменения. Недостатком этого подхода является необходимость большого количества обращений к базе данных; например, если на протяжении выполнения одного метода объект был изменен трижды, вам придется выполнять три обращения к базе данных, вместо того чтобы ограничиться одним обращением по окончании всех изменений.

Во избежание многократных обращений к базе данных запись всех изменений можно отложить до окончания работы. В этом случае вам придется отслеживать все изменения, которые были внесены в содержимое объектов. Для реализации подобной стратегии в код можно ввести дополнительные переменные, однако учтите: если переменных слишком много, они становятся неуправляемыми. Переменные хорошо использовать со **сценарием транзакции (Transaction Script, 133)**, но они совсем не подходят для **модели предметной области (Domain Model, 140)**.

Вместо того чтобы хранить измененные объекты в переменных, для каждого объекта можно создать флаг, который будет указывать на состояние объекта — изменен или не изменен. В этом случае по окончании транзакции необходимо отобрать все измененные объекты и записать их содержимое в базу данных. Удобство этого метода зависит от того, насколько просто находить измененные объекты. Если все объекты находятся в одной иерархии, вы можете последовательно просмотреть всю иерархию и записать в базу данных все обнаруженные изменения. В свою очередь, более общие структуры объектов, в частности **модель предметной области**, просматривать гораздо труднее.

Огромным преимуществом **единицы работы** является то, что она хранит все данные об изменениях в одном месте. Поэтому вам не придется запоминать море информации, чтобы не упустить из виду все изменения объектов. Кроме того, **единица работы** может послужить прекрасной основой для разрешения более сложных ситуаций, таких, как обработка бизнес-транзакций, охватывающих несколько системных транзакций, посредством **оптимистической автономной блокировки** и **пессимистической автономной блокировки**.

Пример: регистрация посредством изменяемого объекта (Java)

Дейвид Райс

Рассмотрим создание **единицы работы**, которая отслеживает все изменения в рамках заданной бизнес-транзакции и затем фиксирует их в базе данных. В нашем примере у слоя домена есть **супертип слоя (Layer Supertype, 491)**, представленный классом `Domain Object`, с которым будет взаимодействовать **единица работы**. Для хранения множества изменений создадим три списка: новые объекты, измененные объекты и удаленные объекты.

```
class UnitOfWork...

   private List newObjects =new ArrayList();
   private List dirtyObjects =new ArrayList();
   private List removedObjects =new ArrayList();
```

Состояние этих списков поддерживается методами регистрации. Последние должны выполнять несколько базовых видов проверки, например проверку того, что идентификатор объекта не равен значению `NULL` или что измененный объект не был зарегистрирован как новый.

```
class UnitOfWork...

   public void registerNew(DomainObject obj) {
      Assert.notNull("id not null", obj.getId());
      Assert.isTrue("object not dirty",
↳!dirtyObjects.contains(obj));
      Assert.isTrue("object not removed",
↳!removedObjects.contains(obj));
      Assert.isTrue("object not already registered new",
↳!newObjects.contains(obj));
      newObjects.add(obj);
   }
   public void registerDirty(DomainObject obj) {
      Assert.notNull("id not null", obj.getId());
      Assert.isTrue("object not removed",
↳!removedObjects.contains(obj));
      if (!dirtyObjects.contains(obj) &&
↳!newObjects.contains(obj)) {
            dirtyObjects.add(obj);
      }
   }
   public void registerRemoved(DomainObject obj) {
      Assert.notNull("id not null", obj.getId());
      if (newObjects.remove(obj)) return;
      dirtyObjects.remove(obj);
      if (!removedObjects.contains(obj)) {
          removedObjects.add(obj);
      }
   }
   public void registerClean(DomainObject obj) {
      Assert.notNull("id not null", obj.getId());
}
```

Обратите внимание, что в приведенном фрагменте кода метод `registerClean()` не выполняет никаких действий. Это связано с тем, что в **единицу работы** часто помещают **коллекцию объектов (Identity Map, 216)**. Использование **коллекции объектов** необходимо практически во всех случаях, когда состояние объекта домена нужно хранить в памяти, поскольку появление многочисленных копий одного и того же объекта может привести к непредсказуемым результатам. Если бы в классе `UnitOfWork` была определена **коллекция объектов**, метод `registerClean()` записывал бы в нее зарегистрированные объекты, загруженные из базы данных. Аналогичным образом метод `registerNew` помещал бы в коллекцию новые объекты, а метод `registerRemoved` удалял бы из нее объекты, подлежащие уничтожению. Поскольку в нашем случае **коллекции объектов** нет, без метода `registerClean()` можно обойтись. Я встречал реализации этого метода, которые удаляли некоторые объекты из списка измененных и помещали их в список неизмененных, однако частичное отклонение изменений всегда запутывает дело. Будьте крайне внимательны, меняя состояние объекта в списке изменений!

Метод `commit()` определяет **преобразователь данных (Data Mapper, 187)**, соответствующий каждому из объектов, и вызывает нужный метод отображения. Методы `updateDirty()` и `deleteRemoved()` здесь не показаны. Их поведение аналогично поведению метода `insertNew()`, текст которого приведен ниже.

```
class UnitOfWork...

   public void commit() {
      insertNew();
      updateDirty();
      deleteRemoved();
   }
   private void insertNew() {
      for (Iterator objects = newObjects.iterator();
↳objects.hasNext();) {
         DomainObject obj = (DomainObject) objects.next();
         MapperRegistry.getMapper(obj.getClass()).insert(obj);
      }
   }
```

В коде нашего класса `UnitOfWork` не было реализовано отслеживание считываемых объектов, которые следует проверять на наличие ошибок несогласованного чтения. Обсудим это немного позднее, при рассмотрении **оптимистической автономной блокировки**.

Перейдем к выполнению регистрации. Вначале каждый объект домена должен найти **единицу работы**, которая обслуживает текущую бизнес-транзакцию. Поскольку **единица работы**, скорее всего, будет нужна всей модели предметной области, передавать ее от объекта к объекту в качестве параметра было бы нецелесообразно. Как известно, каждая бизнес-транзакция выполняется в рамках одного потока, поэтому можно связать **единицу работы** с запущенным в данный момент потоком, используя класс `java.lang.ThreadLocal`. Чтобы не слишком усложнять поставленную задачу, реализуем ее посредством статических методов класса `UnitOfWork`. Если с потоком выполнения бизнес-транзакции уже связан какой-либо объект сеанса, текущую **единицу работы** следует поместить именно в этот объект — зачем создавать дополнительные расходы, связанные с отображением на другой поток? Кроме того, с логической точки зрения **единица работы** принадлежит данному сеансу.

```
class UnitOfWork...

   private static ThreadLocal current = new ThreadLocal();
   public static void newCurrent() {
      setCurrent(new UnitOfWork());
   }
   public static void setCurrent(UnitOfWork uow) {
      current.set(uow);
   }
   public static UnitOfWork getCurrent() {
      return (UnitOfWork) current.get();
   }
```

Теперь можно добавить к абстрактному классу `DomainObject` методы, позволяющие объекту домена зарегистрироваться в текущей **единице работы**.

```
class DomainObject...

   protected void markNew() {
      UnitOfWork.getCurrent().registerNew(this);
   }
   protected void markClean() {
      UnitOfWork.getCurrent().registerClean(this);
   }
   protected void markDirty() {
      UnitOfWork.getCurrent().registerDirty(this);
   }
   protected void markRemoved() {
      UnitOfWork.getCurrent().registerRemoved(this);
   }
```

Попав в соответствующую ситуацию, конкретные объекты домена должны помечаться как новые или измененные.

```
class Album...

   public static Album create(String name) {
      Album obj = new Album(IdGenerator.nextId(), name);
      obj.markNew();
      return obj;
   }
   public void setTitle(String title) {
      this.title = title;
      markDirty();
   }
```

Я опустил регистрацию удаляемых объектов, которую можно реализовать в рамках метода `remove()` абстрактного класса `DomainObject`. Кроме того, если вы реализовали метод `registerClean()`, **преобразователи данных** должны регистрировать все только что загруженные объекты как достоверные (clean).

Последнее, что от вас требуется, — создать текущую **единицу работы** и зафиксировать изменения в базе данных. Как описывать управление **единицей работы** в явном виде, показано ниже.

```
class EditAlbumScript...

   public static void updateTitle(Long albumId, String title) {
      UnitOfWork.newCurrent();
      Mapper mapper = MapperRegistry.getMapper(Album.class);
      Album album = (Album) mapper.find(albumId);
      album.setTitle(title);
      UnitOfWork.getCurrent().commit();
   }
```

Подобный принцип подходит только самым простым приложениям. В более крупных системах управление **единицей работы** рекомендуется реализовать в неявном виде, чтобы избежать утомительного повторения одних и тех же фрагментов кода. Ниже приведен код **супертипа слоя** сервлета, который создает текущую **единицу работы** и вызывает ее метод `commit`. Вместо того чтобы переопределять метод `doGet()`, производные классы **супертипа слоя** реализуют метод `handleGet()`, которому будет передана нужная **единица работы**.

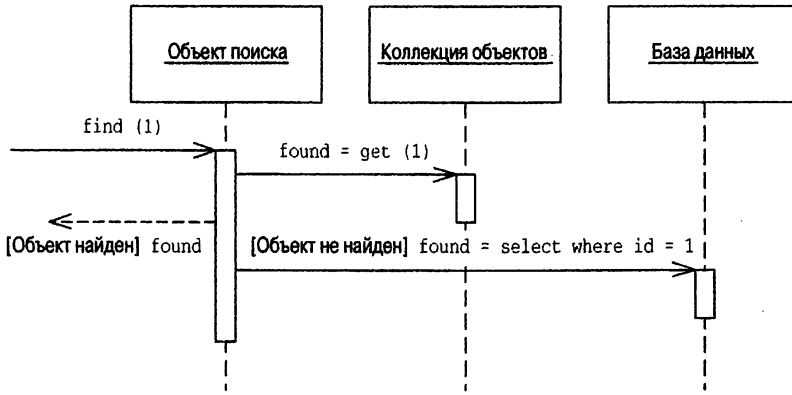
```
class UnitOfWorkServlet...

   final protected void doGet(HttpServletRequest request,
↳HttpServletResponse response)
         throws ServletException, IOException {
      try {
         UnitOfWork.newCurrent();
         handleGet(request, response);
         UnitOfWork.getCurrent().commit();
      } finally {
         UnitOfWork.setCurrent(null);
      }
   }
   abstract void handleGet(HttpServletRequest request,
↳HttpServletResponse response)
         throws ServletException, IOException;
```

Возможно, приведенный пример сервлета слишком прост, так как в нем не выполняется никаких операций по управлению транзакциями. Если бы вы использовали **контроллер запросов (Front Controller, 362)**, то, скорее всего, передали бы управление **единицей работы** его командам, а не методу `doGet()`. Это же относится практически к любому другому контексту выполнения.

Коллекция объектов (Identity Map)

Гарантирует, что каждый объект будет загружен из базы данных только один раз, сохраняя загруженный объект в специальной коллекции. При получении запроса просматривает коллекцию в поисках нужного объекта



Старое изречение гласит: "Человек, который носит две пары часов, никогда не знает, сколько времени". Впрочем, две пары часов — это еще не так серьезно, как загрузка объектов из базы данных. Если вы не будете предельно внимательны, то вполне можете загрузить одни и те же данные в два разных объекта. Легко представить, какая неразбериха начнется, когда вы попытаетесь одновременно внести в базу данных обновления этих объектов...

Описанная ситуация тесно связана и с проблемой производительности. Если одни и те же данные будут загружены несколько раз, это приведет к чрезмерным (и, что самое печальное, совершенно ненужным) расходам на выполнение удаленных вызовов функций. Таким образом, предупреждение повторной загрузки данных не только помогает избежать ошибок, но и значительно ускоряет производительность работы приложения.

Типовое решение **коллекция объектов** хранит в себе данные обо всех объектах, загруженных из базы данных в пределах одной бизнес-транзакции. Теперь, как только вам понадобится какая-нибудь информация из базы данных, вы можете обратиться к **коллекции объектов**, чтобы посмотреть, нет ли там того, что вы ищете?

Принцип действия

Толчком для разработки типового решения **коллекция объектов** послужила идея создать набор коллекций, содержащих объекты, которые были извлечены из базы данных. В случае простой изоморфной схемы для каждой таблицы базы данных создается своя коллекция. Прежде чем загружать объект из базы данных, необходимо проверить, нет ли его в **коллекции объектов**? Если в ней обнаружится объект, в точности соответствующий тому, что вам нужен, вы возвращаете найденный объект. Если же подходящего объекта не оказалось, вы загружаете объект из базы данных, помещая его в коллекцию для дальнейшего использования.

С реализацией **коллекции объектов** связано множество спорных моментов. Кроме того, поскольку применение **коллекций объектов** затрагивает вопрос управления параллельными операциями, рекомендую подумать об использовании **оптимистической автономной блокировки (Optimistic Offline Lock, 434)**.

Выбор ключей

Прежде всего при реализации данного типового решения необходимо подумать о выборе ключа для поиска объектов. Конечно же, наиболее очевидным является использование первичного ключа соответствующей таблицы базы данных. Это очень удобно, если первичный ключ состоит из одного поля и является неизменяемым. В эту же концепцию вписывается и применение искусственного первичного ключа (surrogate primary key). В большинстве случаев значение ключа принадлежит какому-нибудь простому типу данных, поэтому реализовать сравнение будет несложно.

Явная или универсальная?

Коллекция объектов может быть явной (explicit) или универсальной (generic). Доступ к явной **коллекции объектов** осуществляется посредством специализированных методов, соответствующих конкретному типу искомого объекта, например `findPerson(1)`. В универсальной **коллекции объектов** для доступа ко всем типам объектов используется общий метод (возможно, принимающий аргумент, который указывает на тип объекта, например `find("Person", 1)`). Очевидное преимущество универсальной **коллекции объектов** — возможность ее поддержки с помощью универсального повторно используемого объекта. Совсем несложно создать повторно используемый **реестр (Registry, 495)**, который будет применим ко всем типам объектов и не потребует обновления при добавлении в него новой коллекции.

Несмотря на сказанное выше, я предпочитаю создавать явные **коллекции объектов**. Во-первых, это дает возможность проводить проверку времени компиляции в строго типизированных языках. Во-вторых, такой объект обладает всеми преимуществами явных интерфейсов: гораздо легче увидеть, какие коллекции доступны на данный момент и как они называются. Конечно же, при добавлении новой коллекции вам придется добавлять новый метод, однако наличие явного интерфейса того стоит.

На выбор типа **коллекции объектов** влияет тип используемого ключа. Универсальная коллекция применима только тогда, когда все объекты имеют ключи одного и того же типа. Кстати, это хороший аргумент в пользу инкапсуляции различных типов ключей баз данных в одном объекте ключа (более подробно об этом речь идет в разделе, посвященном **полю идентификации (Identity Field, 237)**).

Сколько нужно коллекций?

Существует два варианта создания коллекций — по одной на каждый класс или по одной на сеанс. Последний вариант можно использовать только тогда, когда первичные ключи уникальны в пределах всей базы данных (преимущества этой концепции обсуждаются в разделе, посвященном **полю идентификации**). Конечно же, иметь единственную **коллекцию объектов** очень удобно: вам больше не нужно искать информацию в разных местах или задумываться о наследовании.

При наличии нескольких коллекций наиболее простым решением является создание по одной коллекции на каждый класс или каждую таблицу. Это хороший выбор, если схемы базы данных и объектной модели совпадают. Если же они различны, количество

коллекций рекомендуется привязывать к объектам, а не к таблицам, поскольку объекты, вообще говоря, не должны "знать" никаких деталей отображения.

Определение количества коллекций крайне затрудняет наследование. Если в базе данных есть машины, являющиеся производным типом средств передвижения, сколько нужно создать коллекций — одну общую или по одной на каждое средство передвижения? Размещение их в разных коллекциях может крайне усложнить полиморфные ссылки, поскольку каждому средству поиска нужно будет знать обо всех существующих коллекциях. Поэтому я предпочитаю иметь одну общую коллекцию для каждой иерархии объектов, однако в таком случае ключи объектов должны быть уникальными в пределах своей иерархии, что весьма затруднительно при использовании **наследования с таблицами для каждого конкретного класса (Concrete Table Inheritance, 313)**.

Преимуществом использования единой **коллекции объектов** является то, что при добавлении новых таблиц не требуется добавлять новые коллекции. Впрочем, привязка коллекций к **преобразователям данных (Data Mapper, 187)** совсем несложна (см. ниже).

Куда их поместить?

Коллекции объектов нужно поместить туда, где их будет легко найти. Кроме того, они должны быть привязаны к контексту текущего процесса. Каждый сеанс должен иметь свой экземпляр **коллекции объектов**, полностью изолированный от экземпляров других сеансов. Таким образом, **коллекцию объектов** следует поместить в объект, специфичный для сеанса. Прекрасным кандидатом для этого может стать **единица работы (Unit of Work, 205)**, поскольку она хранит в себе сведения обо всех объектах, извлеченных, обновленных, созданных и удаленных на протяжении текущего сеанса. Если же вы не используете **единицу работы**, рекомендую поместить **коллекцию объектов** в **реестр**, привязанный к сеансу.

Как уже отмечалось, каждому сеансу должна соответствовать своя **коллекция объектов**; в противном случае придется обеспечить для нее транзакционную защиту, за что не возьмется ни один здравомыслящий разработчик. Впрочем, существует несколько исключений. Наиболее значительное из них — это использование объектной базы данных в качестве кэша транзакций, даже если данные при этом записываются в реляционную базу данных. Пока что я не встречал никаких независимых исследований на тему производительности кэша транзакций, однако думаю, что на это стоит обратить внимание. Целый ряд весьма уважаемых мною людей — страстные приверженцы использования кэша транзакций как способа увеличения производительности.

Вторым исключением является использование объектов, доступных только для чтения. Если объект не может быть изменен, вам нет необходимости беспокоиться о том, что он будет присутствовать в нескольких сеансах. В очень загруженных системах наличие общей коллекции весьма удобно, поскольку позволяет однократно загрузить все данные, необходимые только для чтения, и затем использовать их на протяжении всего процесса. В этом случае **коллекции объектов**, доступных только для чтения, будут использоваться в контексте процесса, а доступных для обновления — в контексте сеанса. Это же применимо и к объектам, которые изменяются так редко, что их можно безбоязненно поместить в **коллекцию объектов**, доступную на протяжении всего процесса, а если изменения все-таки произойдут, то можно обновить коллекцию, отправив соответствующий запрос серверу.

Даже если вы твердо решили создать только одну **коллекцию объектов**, ее можно разбить на две части: только для чтения и для обновления. Чтобы клиенты не догадались,

какая часть коллекции соответствует каким объектам, необходимо предоставить интерфейс, который будет проверять обе коллекции.

Назначение

Как правило, **коллекция объектов** применяется для управления любыми объектами, которые были загружены из базы данных и затем подверглись изменениям. Основное назначение **коллекции объектов** — не допустить возникновения ситуации, когда два разных объекта приложения будут соответствовать одной и той же записи базы данных, поскольку изменение этих объектов может происходить несогласованно и, следовательно, вызывать трудности с отображением на базу данных.

Еще одним преимуществом **коллекции объектов** является возможность использования ее в качестве кэша записей, считываемых из базы данных. Это избавляет от необходимости повторно обращаться к базе данных, если снова понадобится какой-нибудь объект.

Скорее всего, **коллекция объектов** не понадобится для хранения неизменяемых объектов. Если объект не может быть изменен, вам не нужно беспокоиться о потенциальных конфликтах, связанных с его обновлением. Поскольку **объекты-значения (Value Object, 500)** являются неизменяемыми, из этого следует, что для работы с ними **коллекция объектов**, по большому счету, не нужна. Несмотря на это, она может пригодиться и здесь — прежде всего в качестве кэша. Помимо этого, наличие **коллекции объектов** позволяет предупредить использование некорректного метода проверки на равенство — проблема, весьма распространенная в Java, где нельзя переопределить действие оператора ==.

Коллекция объектов не нужна и при использовании **отображения зависимых объектов (Dependent Mapping, 283)**, поскольку все обращения к зависимым объектам контролируются объектами-владельцами. Впрочем, она может понадобиться, если вы хотите осуществлять доступ к зависимому объекту с помощью ключа базы данных. Разумеется, в этом случае **колекция объектов** будет представлять собой не более чем индекс (поэтому весьма сомнительно, можно ли ее в таком случае вообще называть коллекцией).

Коллекция объектов помогает избежать конфликтов обновления, возникающих в пределах одного сеанса, однако она бессильна что-либо сделать в случае межсеансовых проблем. Это довольно сложный вопрос, к которому мы вернемся в разделах, посвященных **оптимистической автономной блокировке (Optimistic Offline Lock, 434)** и **пессимистической автономной блокировке (Pessimistic Offline Lock, 445)**.

Пример: методы для работы с коллекцией объектов (Java)

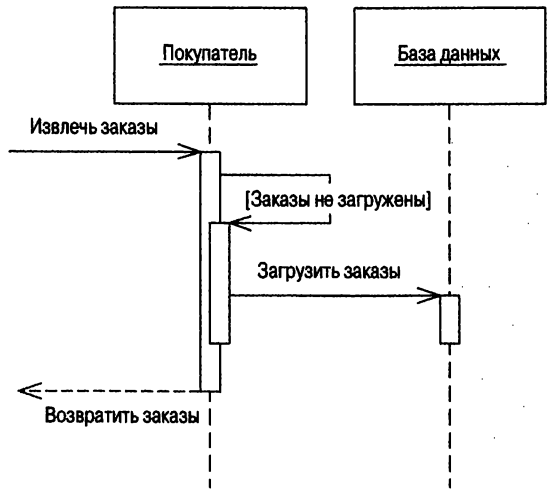
Каждая **коллекция объектов** имеет поле, содержащее коллекцию, и методы доступа.

```
private Map people = new HashMap();
public static void addPerson(Person arg) {
   soleInstance.people.put(arg.getID(), arg);
}
public static Person getPerson(Long key) {
   return (Person) soleInstance.people.get(key);
}
public static Person getPerson(long key) {
   return getPerson(new Long(key));
}
```

Одна из неприятных особенностей Java связана с тем, что значение типа `long` не является объектом, а следовательно, не может использоваться в качестве индекса коллекции. К счастью, в нашей ситуации это не так критично, поскольку не предполагается выполнять над индексами никаких арифметических действий. Это действительно могло бы понадобиться лишь тогда, когда объект пытаются извлечь путем явного указания его ключа в виде номера. В реальном коде приложения это вряд ли понадобится, а вот при тестировании фрагментов кода встречается сплошь и рядом. Чтобы облегчить тестирование, я добавил к **коллекции объектов** get-метод, который принимает в качестве аргумента значение типа `long`.

Загрузка по требованию (Lazy Load)

Объект, который не содержит все требующиеся данные, однако может загрузить их в случае необходимости



Загрузку данных в оперативную память следует организовать таким образом, чтобы при загрузке интересующего вас объекта из базы данных автоматически извлекались и другие связанные с ним объекты. Это значительно облегчает жизнь разработчика, которому в противном случае пришлось бы прописывать загрузку всех связанных объектов вручную.

К сожалению, подобный процесс может принять устрашающие формы, если загрузка одного объекта повлечет за собой загрузку огромного количества связанных с ним объектов — крайне нежелательная ситуация, особенно когда вам были нужны только несколько конкретных записей.

Типовое решение **загрузка по требованию** прерывает процесс загрузки, оставляя соответствующую метку в структуре объектов. Это позволяет загрузить необходимые данные только тогда, когда они действительно понадобятся. Подобные ситуации бывают

и в обычной жизни: иногда так ленишься что-то делать[1], но зато как приятно, если оказывается, что делать это было вовсе не нужно.

Принцип действия

Существует четыре основных способа реализации **загрузки по требованию**: инициализация по требованию, виртуальный прокси-объект, диспетчер значения и фиктивный объект.

Самый простой из них — *инициализация по требованию (lazy initialization)* [6]. Основная идея данного подхода заключается в том, что при каждой попытке доступа к полю выполняется проверка, не содержит ли оно значение NULL. Если поле содержит NULL, метод доступа загружает значение поля и лишь затем его возвращает. Это может быть реализовано только в том случае, если поле является самоинкапсулированным, т.е. если доступ к такому полю осуществляется только посредством get-метода (даже в пределах самого класса).

Использовать значение NULL в качестве признака незагруженного поля очень удобно. Исключение составляют лишь те ситуации, когда NULL является допустимым значением загруженного поля. В этом случае необходимо выбрать какой-нибудь другой признак того, что поле не загружено, или же определить **частный случай (Special Case, 511)** для поля, значение которого может быть равно NULL.

Принцип инициализации по требованию очень прост, хотя и нуждается в наличии дополнительной зависимости между объектом и базой данных. Поэтому его хорошо применять с **активной записью (Active Record, 182)**, **шлюзом таблицы данных (Table Data Gateway, 167)** или **шлюзом записи данных (Row Data Gateway, 175)**. Если же вы используете **преобразователь данных (Data Mapper, 187)**, то понадобится реализовать дополнительный слой непрямого доступа, который можно получить посредством *виртуального прокси-объекта (virtual proxy)* [20]. Виртуальный прокси-объект имитирует объект, являющийся значением поля, однако в действительности ничего в себе не содержит. В этом случае загрузка реального объекта будет выполнена только тогда, когда будет вызван один из методов виртуального прокси-объекта.

Преимуществом виртуального прокси-объекта является то, что он "выглядит" точь-в-точь как реальный объект, который должен находиться в данном поле. Тем не менее это не настоящий объект, поэтому вы легко можете столкнуться с досадной проблемой идентификации. Более того, одному и тому же реальному объекту может соответствовать сразу несколько виртуальных прокси-объектов. Все они будут иметь разные идентификаторы объекта, однако, по сути, представлять собой один и тот же объект. Поэтому вам придется, как минимум, переопределить метод проверки на равенство, используя равенство объектов по значениям полей, а не по идентификаторам. Без этого, а также при невнимательном отношении к виртуальным прокси-объектам, вы можете наделать массу ошибок, которые будет весьма трудно отследить.

В некоторых средах разработки использование описанного подхода может привести к еще одной проблеме, связанной с появлением слишком большого количества виртуальных прокси-объектов (по одному на каждый "замещаемый" класс). Этого легко избежать в динамически типизированных языках, однако в статически типизированных языках появление большого количества виртуальных прокси-объектов способно привести

---

[1] Lazy в переводе с английского означает "ленивый". — *Прим. пер.*

к страшной неразберихе. Даже когда в самой платформе реализованы соответствующие вспомогательные средства (например, прокси-объекты Java), это может вызвать еще какие-нибудь сложности.

Подобных проблем не возникает, если виртуальные прокси-объекты используются для замены классов коллекций, например списков. Поскольку элементы коллекций являются **объектами-значениями (Value Object, 500)**, их идентификаторы не играют роли. Кроме того, виртуальные прокси-объекты придется создавать всего лишь для нескольких классов коллекций.

При использовании классов домена описанных проблем можно избежать, применив *диспетчер значения (value holder)*, с концепцией которого я столкнулся еще во время программирования на Smalltalk. Диспетчер значения — это объект, который выполняет роль оболочки для какого-нибудь другого объекта. Чтобы добраться к значению базового объекта, необходимо обратиться за ним к диспетчеру значения. При первом обращении диспетчер значения извлекает необходимую информацию из базы данных. Этот способ имеет ряд недостатков. Во-первых, класс должен знать о наличии диспетчера значения, а во-вторых, теряются преимущества строгой типизации. Чтобы избежать проблем идентификации, необходимо гарантировать, что диспетчер значения никогда не будет передаваться методам за пределами его класса-владельца.

И наконец, *фиктивный объект (ghost)* — это реальный объект с неполным состоянием. Когда подобный объект загружается из базы данных, он содержит только свой идентификатор. При первой же попытке доступа к одному из его полей объект загружает значения всех остальных полей. Для большей наглядности фиктивный объект можно представить себе в виде объекта, все поля которого одновременно инициализируются по требованию, или объекта, который является собственным виртуальным прокси-объектом. Разумеется, все данные не обязательно загружать за один раз; для большего удобства их можно разбить на группы полей, которые часто используются вместе. Загрузив фиктивный объект, вы можете сразу же поместить его в **коллекцию объектов (Identity Map, 216)**. Это позволит сохранить идентификатор объекта и избежать проблем чтения, связанных с наличием циклических ссылок.

Виртуальный прокси-объект или фиктивный объект не обязательно должны быть напрочь лишены содержимого. Если реальный объект включает в себя быстро загружаемые или часто используемые данные, их можно загрузить вместе с прокси-объектом или фиктивным объектом (подобные объекты иногда называют "облегченными").

Многие проблемы реализации **загрузки по требованию** связаны с наследованием. Если вы собираетесь создавать фиктивные объекты, вам нужно знать, какого они должны быть типа. Однако зачастую это можно узнать только тогда, когда объект уже будет загружен. В статически типизированных языках подобными проблемами чревато и использование виртуальных прокси-объектов.

Помимо всего прочего, использование **загрузки по требованию** может повлечь за собой чрезмерное количество обращений к базе данных. Хорошим примером подобной *"волнообразной" загрузки (ripple loading)* является ситуация, когда вы применили **загрузку по требованию**, чтобы заполнить объектами некоторую коллекцию, и затем пытаетесь поочередно просмотреть ее содержимое. В этом случае вам придется обращаться к базе данных каждый раз при переходе к очередному объекту, вместо того чтобы загрузить все объекты сразу. Как я уже убедился, подобный способ загрузки значительно снижает производительность приложения. Разумеется, чтобы гарантировать отсутствие подобной

проблемы, достаточно никогда не создавать коллекцию объектов, извлеченных путем **загрузки по требованию**. Вместо этого **загрузку по требованию** можно применить к самой коллекции таким образом, чтобы содержимое последней загружалось целиком. Данный прием не следует применять тогда, когда коллекция слишком большая, например все IP-адреса мира. Как правило, такие объекты не связаны ассоциациями в объектной модели, поэтому подобные ситуации не будут возникать слишком часто, однако если они все же возникнут, рекомендую воспользоваться **обработчиком списка значений (Value List Handler)** [3].

Загрузку по требованию хорошо применять в аспектно-ориентированном программировании. Ее поведение можно выделить в отдельный аспект, что позволит изменять стратегию загрузки независимо от самого объекта и освободить разработчиков домена от необходимости заниматься ее проблемами. Кроме того, я видел прозрачную реализацию **загрузки по требованию**, полученную путем последующей обработки байт-кода Java.

Зачастую разные варианты использования приложения требуют загрузки разного объема данных. Нередки ситуации, когда в одних случаях необходимо загрузить одно подмножество графа объектов, а в других — другое. Таким образом, максимальная эффективность приложения достигается при условии, что для нужного варианта использования загружается нужное подмножество графа объектов.

Для реализации описанного принципа необходимо, чтобы для каждого варианта использования применялись свои объекты взаимодействия с базой данных. Например, если вы применяете **преобразователь данных** для загрузки заказов, попробуйте создать два объекта преобразователей: один, который сразу же загружает все пункты запрошенного заказа, и второй, который загружает их по требованию. В зависимости от варианта использования, приложение будет выбирать тот или иной преобразователь.

Теоретически вам может понадобиться целое множество вариантов загрузки, однако на практике обычно используются только два: загрузка всего объема данных и загрузка части данных, достаточной для идентификации объектов. Добавление еще каких-либо вариантов загрузки значительно усложняет приложение и совершенно не стоит затраченных усилий.

Назначение

Принимая решение об использовании **загрузки по требованию**, необходимо обдумать следующее: сколько данных требуется извлекать при загрузке одного объекта и сколько обращений к базе данных для этого понадобится. Бессмысленно использовать **загрузку по требованию**, чтобы извлечь значение поля, хранящегося в той же строке, что и остальное содержимое объекта; в большинстве случаев загрузка данных за одно обращение не приведет к дополнительным расходам, даже если речь идет о полях большого размера, таких, как **крупный сериализованный объект (Serialized LOB, 292)**. Вообще говоря, о применении **загрузки по требованию** следует думать только в том случае, когда доступ к значению поля требует дополнительного обращения к базе данных.

В плане производительности использование **загрузки по требованию** влияет на то, в какой момент времени будут получены необходимые данные. Зачастую все данные удобно извлекать посредством одного обращения к базе данных, в частности если это соответствует одному взаимодействию с пользовательским интерфейсом. Таким образом, к **загрузке по требованию** лучше всего обращаться тогда, когда для извлечения

дополнительных данных необходимо отдельное обращение к базе данных и когда извлекаемые данные не используются вместе с основным объектом.

Применение **загрузки по требованию** существенно усложняет приложение, поэтому я стараюсь прибегать к ней только тогда, когда без нее действительно не обойтись.

Пример: инициализация по требованию (Java)

Суть инициализации по требованию описывается примерно следующим образом:

```
class Supplier...

   public List getProducts() {
      if (products == null) products =
↳Product.findForSupplier(getID());
      return products;
   }
```

В этом примере первая попытка доступа к полю `products` приводит к загрузке его значения из базы данных.

Пример: виртуальный прокси-объект (Java)

Для обеспечения работы виртуального прокси-объекта необходимо создать класс, сходный с используемым реальным классом, однако представляющий собой всего лишь простую оболочку последнего. В нашем примере список товаров, заказанных у поставщика, будет храниться в обычном поле типа `List`.

```
class SupplierVL...

   private List products;
```

Самое сложное в создании прокси-объекта для списка — настроить его так, чтобы реальный список создавался только тогда, когда он будет затребован. Для этого создаваемому экземпляру виртуального списка необходимо передать код, применяемый для создания реального списка. В Java это лучше осуществить путем определения интерфейса загрузчика.

```
public interface VirtualListLoader {
   List load();
}
```

Теперь можно создать экземпляр виртуального списка с помощью загрузчика, который будет вызывать соответствующий метод отображения.

```
class SupplierMapper...

   public static class ProductLoader implements
↳VirtualListLoader {
      private Long id;
      public ProductLoader(Long id){
         this.id = id;
```

```
    }
    public List load(){
       return ProductMapper.create().findForSupplier(id);
    }
}
```

Во время выполнения загрузки экземпляр загрузчика товаров передается создаваемому экземпляру виртуального списка, который затем присваивается полю `products`.

```
class SupplierMapper...

   protected DomainObject doLoad(Long id, ResultSet rs)
↳throws SQLException {
      String nameArg = rs.getString(2);
      SupplierVL result = new SupplierVL(id, nameArg);
      result.setProducts(new VirtualList(new
↳ProductLoader(id)));
      return result;
   }
```

Поле `source` класса `VirtualList` является самоинкапсулированным. При первом обращении к реальному списку оно вызывает метод загрузки для извлечения списка заказанных товаров из базы данных.

```
class VirtualList...

   private List source;
   private VirtualListLoader loader;
   public VirtualList(VirtualListLoader loader) {
      this.loader = loader;
   }
   private List getSource() {
      if (source == null) source = loader.load();
      return source;
   }
```

Все методы работы с обычным списком реализуются в классе `VirtualList` для списка `source`.

```
class VirtualList...

   public int size() {
      return getSource().size();
   }
   public boolean isEmpty() {
      return getSource().isEmpty();
   }
   //...и т.п. для всех остальных методов списка
```

Теперь класс домена ничего не будет "знать" о том, как класс преобразователя выполняет **загрузку по требованию**. Более того, класс домена вообще не узнает о том, что она есть.

Пример: использование диспетчера значения (Java)

Диспетчер значения может быть использован в качестве универсальной **загрузки по требованию**. В этом случае классу домена известно об использовании **загрузки по требованию**, поскольку поле `products` имеет тип `ValueHolder`. Данный факт может быть скрыт от клиентов поставщика посредством get-метода.

```
class SupplierVH...

   private ValueHolder products;
   public List getProducts() {
      return (List)products.getValue();
   }
```

Объект `ValueHolder` сам выполняет функции, возложенные на **загрузку по требованию**. Чтобы держатель значения смог загрузить требующиеся данные, ему необходимо передать соответствующий код. Это можно сделать путем определения интерфейса загрузчика.

```
class ValueHolder...

   private Object value;
   private ValueLoader loader;
   public ValueHolder(ValueLoader loader) {
      this.loader =loader;
   }
   public Object getValue() {
      if (value == null) value = loader.load();
      return value;
   }
public interface ValueLoader {
   Object load();
}
```

Преобразователь может установить диспетчер значения, создав реализацию загрузчика и поместив ее в объект поставщика.

```
class SupplierMapper...

   protected DomainObject doLoad(Long id, ResultSet rs)
↳throws SQLException {
      String nameArg = rs.getString(2);
      SupplierVH result = new SupplierVH(id, nameArg);
      result.setProducts(new ValueHolder(new
↳ProductLoader(id)));
      return result;
   }
   public static class ProductLoader implements ValueLoader {
      private Long id;
      public ProductLoader(Long id) {
         this.id = id;
      }
      public Object load() {
```

```
        return ProductMapper.create().findForSupplier(id);
    }
}
```

Пример: использование фиктивных объектов (C#)

Большая часть логики по превращению объектов в фиктивные может быть реализована в **супертипах слоя (Layer Supertype, 491)**. Как следствие этого, если вы начнете применять фиктивные объекты, они будут использоваться везде. Я начну изучение фиктивных объектов с рассмотрения **супертипа слоя** объектов домена. Каждому объекту домена известно, является он фиктивным или нет.

```
class DomainObject...

    LoadStatus Status;
    public DomainObject (long key) {
        this.Key = key;
    }
    public Boolean IsGhost {
        get {return Status == LoadStatus.GHOST;}
    }
    public Boolean IsLoaded {
        get {return Status == LoadStatus.LOADED;}
    }
    public void MarkLoading() {
        Debug.Assert(IsGhost);
        Status = LoadStatus.LOADING;
    }
    public void MarkLoaded() {
        Debug.Assert(Status == LoadStatus.LOADING);
        Status =LoadStatus.LOADED;
    }
enum LoadStatus {GHOST, LOADING, LOADED};
```

Объекты домена могут иметь три состояния: фиктивный (ghost), загружаемый (loading) и загруженный (loaded). Я предпочитаю помещать информацию о состоянии в поля, доступные только для чтения, и выполнять изменение их значений посредством соответствующих явных методов.

Что самое утомительное при работе с фиктивными объектами, это необходимость изменить каждый метод доступа. Последний должен проверить, не является ли объект фиктивным, и, если это так, запустить метод загрузки его содержимого.

```
class Employee...

    public String Name {
        get {
            Load();
            return _name;
        }
        set {
            Load();
            _name =value;
```

```
        }
    }
    String _name;

class Domain Object...

    protected void Load() {
        if (IsGhost)
            DataSource.Load(this);
    }
```

Для любителей аспектно-ориентированного программирования подобная необходимость изменения методов доступа — замечательный повод прибегнуть к последующей обработке байт-кода.

Для выполнения загрузки содержимого объект домена должен вызвать нужный преобразователь. Между тем, согласно моим правилам "видимости", код объекта домена не может "видеть" код преобразователя. Чтобы избежать подобной зависимости, необходимо реализовать интересную комбинацию **реестра (Registry, 495)** и **отделенного интерфейса (Separated Interface, 492)**, как показано на рис. 11.4. В **реестр** домена будут помещены методы для работы с источником данных.

```
class DataSource...

    public static void Load (DomainObject obj) {
        instance.Load(obj);
    }
```

Экземпляр источника данных определяется с помощью интерфейса.

```
class DataSource...

    public interface IDataSource {
        void Load (DomainObject obj);
    }
```

Интерфейс источника данных реализуется в **реестре** преобразователей, определенном в слое источника данных. В нашем примере преобразователи помещены в словарь, индексированный по типам домена. Метод загрузки находит нужный преобразователь и указывает ему на необходимость загрузки соответствующего объекта домена.

```
class MapperRegistry : IDataSource...

    public void Load (DomainObject obj) {
        Mapper(obj.GetType()).Load (obj);
    }
    public static Mapper Mapper(Type type) {
        return (Mapper) instance.mappers[type];
    }
    IDictionary mappers = new Hashtable();
```

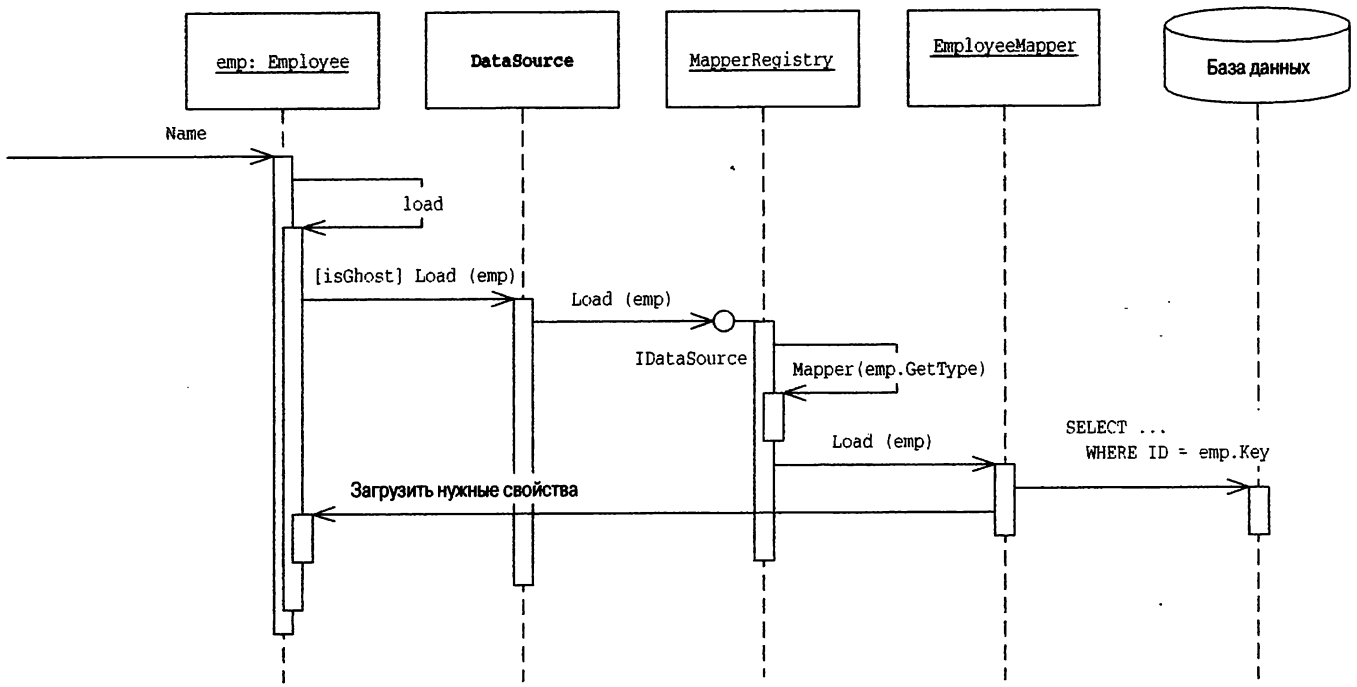


Рис. 11.5. Последовательность загрузки фиктивного объекта

В приведенном фрагменте кода показано, как объекты домена взаимодействуют с источником данных. Логика источника данных использует **преобразователи данных (Data Mapper, 187)**. Логика обновления, реализованная в преобразователях, точно такая же, как и без использования фиктивных объектов. В данном случае гораздо больший интерес представляют методы поиска и загрузки.

Конкретные классы преобразователей имеют собственные методы поиска, которые вызывают абстрактный метод поиска и приводят полученное значение к типу своего класса.

```
class EmployeeMapper...

      public Employee Find (long key) {
         return (Employee) AbstractFind(key);
      }

class Mapper...

      public DomainObject AbstractFind (long key) {
         DomainObject result;
         result = (DomainObject) loadedMap[key];
         if (result == null) {
            result = CreateGhost(key);
            loadedMap.Add(key, result);
         }
         return result;
      }
      IDictionary loadedMap = new Hashtable();
      public abstract DomainObject CreateGhost(long key);

class EmployeeMapper...

      public override DomainObject CreateGhost(long key) {
         return new Employee(key);
      }
```

Как видите, метод поиска возвращает фиктивный объект. Реальное содержимое объекта не загружается из базы данных до тех пор, пока не будет осуществлена попытка доступа к одному из свойств объекта домена, в результате чего метод доступа запустит выполнение загрузки.

```
class Mapper...

      public void Load (DomainObject obj) {
         if (!obj.IsGhost) return;
         IDbCommand comm = new OleDbCommand(findStatement(),
↳DB.connection);
         comm.Parameters.Add(new OleDbParameter("key",
↳obj.Key));
         IDataReader reader = comm.ExecuteReader();
         reader.Read();
         LoadLine (reader, obj);
         reader.Close();
      }
```

```
      protected abstract String findStatement();
      public void LoadLine (IDataReader reader,
↳DomainObject obj) {
         if (obj.IsGhost) {
            obj.MarkLoading();
            doLoadLine (reader, obj);
            obj.MarkLoaded();
         }
      }
      protected abstract void doLoadLine (IDataReader reader,
↳DomainObject obj);
```

Как и в предыдущих примерах, **супертип слоя** выполняет все действия, вынесенные в абстрактный класс, после чего вызывает абстрактный метод для конкретного производного класса. В этом примере я воспользовался объектом `DataReader` — моделью поточного считывания данных с помощью курсора, которая сегодня применяется на различных платформах. При желании можете расширить данный пример на использование объекта `DataSet`, более популярного в среде .NET.

В нашем примере у объекта `Employee` ("сотрудник") есть три типа свойств: поле `Name` ("имя"), имеющее простое значение типа `String`, поле `Department` ("отдел"), значение которого является ссылкой на другой объект, и поле `TimeRecords` ("трудовой стаж"), значением которого является коллекция. Загрузку всего этого содержимого выполняет реализация метода `doLoadLine()` в производном классе (рис. 11.5).

```
class EmployeeMapper...

      protected override void doLoadLine (IDataReader reader,
↳DomainObject obj) {
         Employee employee = (Employee) obj;
         employee.Name = (String) reader["name"];
         DepartmentMapper depMapper =
↳(DepartmentMapper) MapperRegistry.Mapper(typeof(Department));
         employee.Department =
↳depMapper.Find((int)reader["departmentID"]);
         loadTimeRecords(employee);
      }
```

Значение поля `Name` загружается путем простого считывания значения соответствующего столбца, на которое указывает курсор объекта `DataReader`. Значение поля `Department` считывается с помощью метода поиска объекта `DepartmentMapper`. В результате применения этого метода значением поля `Department` станет фиктивный объект; настоящие данные об отделе будут считаны только тогда, когда кто-то попытается осуществить доступ к самому объекту `Department`.

С коллекцией дело обстоит немного сложнее. Чтобы избежать волнообразной загрузки, все записи о стаже работы должны быть извлечены посредством одного запроса. Для этого понадобится особая реализация списка, которая будет выступать в роли фиктивного списка. Последний является всего лишь оболочкой реального списка и передает ему управление всеми действиями, затрагивающими содержимое списка. Единственное, что делает фиктивный список, — это следит за тем, чтобы любая попытка доступа к списку приводила к выполнению загрузки.

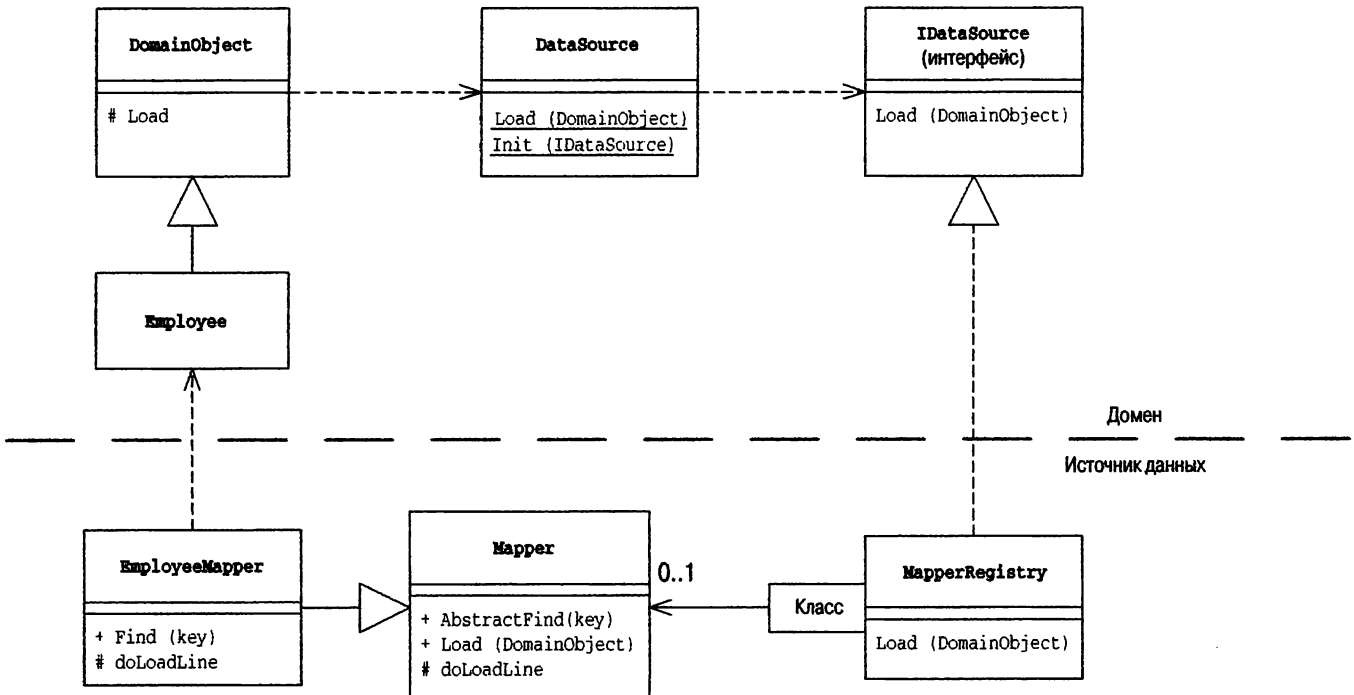


Рис. 11.4. Классы, принимающие участие в загрузке фиктивного объекта

```
class DomainList...

      IList data {
         get {
            Load();
            return _data;
         }
         set {_data = value;}
      }
      IList _data = new ArrayList();
      public int Count {
          get {return data.Count;}
      }
```

Класс `DomainList` используется объектами домена и является частью слоя домена. Методу, выполняющему загрузку, необходимо иметь доступ к SQL-командам, поэтому я воспользовался делегатом для определения функции загрузки, которая может быть предоставлена слоем отображения (рис. 11.6).

```
class DomainList...

      public void Load () {
         if (IsGhost) {
            MarkLoading();
            RunLoader(this);
            MarkLoaded();
         }
      }
      public delegate void Loader(DomainList list);
      public Loader RunLoader;
```

Для большей наглядности делегат можно представить себе в качестве особой разновидности **отделенного интерфейса** для одной функции. Действительно, вместо использования делегата можно было бы объявить интерфейс, содержащий единственную функцию.

У самого загрузчика `ListLoader` есть свойства, задающие SQL-выражение, параметры этого выражения и преобразователь, которые следует использовать для загрузки записей о трудовом стаже. Объект `EmployeeMapper` устанавливает значения полей объекта `ListLoader` при загрузке объекта `employee`.

```
class EmployeeMapper...

      void loadTimeRecords(Employee employee) {
         ListLoader loader = new ListLoader();
         loader.Sql = TimeRecordMapper.FIND_FOR_EMPLOYEE_SQL;
         loader.SqlParams.Add(employee.Key);
         loader.Mapper = MapperRegistry.Mapper(
↳typeof(TimeRecord));
         loader.Attach((DomainList) employee.TimeRecords);
      }

class ListLoader...
```

```
public String Sql;
public IList SqlParams = new ArrayList();
public Mapper Mapper;
```

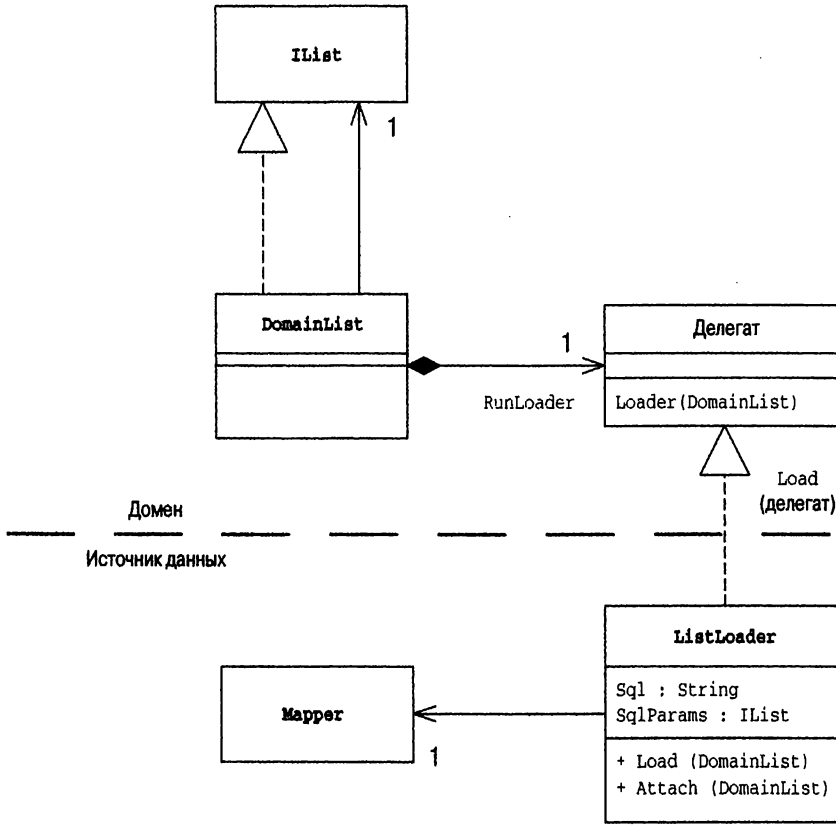


Рис. 11.6. Классы, необходимые для создания фиктивного списка. Сегодня в языке UML еще нет стандарта для изображения делегатов, поэтому я придумал собственный способ

Поскольку синтаксически присвоение делегата описывается довольно сложно, я снабдил объект `ListLoader` методом `Attach()`.

```
class ListLoader...

    public void Attach (DomainList list) {
       list.RunLoader = new DomainList.Loader(Load);
    }
```

После загрузки объекта employee коллекция записей о трудовом стаже будет находиться в фиктивном состоянии до тех пор, пока какой-нибудь из методов доступа не запустит загрузчик. В этом случае загрузчик выполнит SQL-запрос и заполнит список необходимыми данными.

```
class ListLoader...

      public void Load (DomainList list) {
         list.IsLoaded = true;
         IDbCommand comm = new OleDbCommand(Sql, DB.connection);
         foreach (Object param in SqlParams)
            comm.Parameters.Add(new OleDbParameter(
↳param.ToString(), param));
         IDataReader reader = comm.ExecuteReader();
         while (reader.Read()) {
            DomainObject obj = GhostForLine(reader);
            Mapper.LoadLine(reader, obj);
            list.Add(obj);
         }
         reader.Close();
      }
      private DomainObject GhostForLine(IDataReader reader) {
         return Mapper.AbstractFind((System.Int32)
↳reader[Mapper.KeyColumnName]);
      }
```

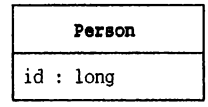
Использование фиктивных списков, продемонстрированное в данном примере, существенно уменьшает присутствие волнообразной загрузки. Конечно же, при желании можно было сделать так, чтобы она не появлялась совсем, например построить процесс отображения таким образом, чтобы данные об отделе загружались вместе с именем сотрудника. Впрочем, совместная загрузка всех элементов коллекции и так помогает избежать наиболее сложных случаев.

Глава 12

Объектно-реляционные типовые решения, предназначенные для моделирования структуры

Поле идентификации (Identity Field)

Сохраняет идентификатор записи базы данных для поддержки соответствия между объектом приложения и строкой базы данных



В реляционных базах данных для идентификации записей таблиц используются ключи, в частности первичные. Между тем объектам, расположенным в оперативной памяти, ключи не нужны, поскольку идентификация объекта выполняется по значениям его полей (или, как в C++, по адресу объекта в памяти). Для считывания данных идентификаторы не нужны, однако, чтобы иметь возможность записывать данные в базу данных, ее содержимое необходимо каким-то образом привязать к системе объектов приложения.

Суть типового решения **поле идентификации** до смешного проста: требуется лишь сохранить первичный ключ таблицы реляционной базы данных в полях объекта.

Принцип действия

Хотя основная идея **поля идентификации** крайне проста, в ее реализации существует множество спорных моментов.

Выбор ключа

Первой и наиболее важной проблемой в реализации **поля идентификации** является выбор ключа. Конечно же, это не всегда приходится делать самому. Довольно часто корпоративные приложения разрабатываются для готовой базы данных, у которой уже есть ключи. Вопрос выбора ключей широко обсуждается в литературе и источниках, посвященных созданию баз данных. Тем не менее необходимость отображать записи данных на объекты наталкивает на определенные размышления.

Первое, о чем следует задуматься при выборе ключа, — это какие ключи использовать: значащие или незначащие. *Значащий ключ (meaningful key)* — это нечто наподобие номера социального страхования или идентификационного кода, т.е. данные, несущие в себе определенный смысл. *Незначащий ключ (meaningless key)* — это случайное число, предназначенное исключительно для внутреннего использования в базе данных и не имеющее никакого смысла в обычной жизни. Как ни странно, на практике выбор значащих ключей не всегда удачен. Чтобы действительно стать ключами, они должны быть уникальными; чтобы действительно стать *хорошими* ключами, они должны быть неизменяемыми. Теоретически так оно и есть, однако человеческие ошибки зачастую сводят все на нет. Например, если вы неправильно наберете номер социального страхования моей жены, полученная запись не будет ни уникальной, ни неизменяемой (потому что вы, скорее всего, захотите исправить ошибку). Разумеется, СУБД должна среагировать на нарушение уникальности, однако это может произойти только тогда, когда запись уже попадет в базу данных (и разумеется, только в том случае, если в базе данных обнаружится еще один человек с таким же номером социального страхования). Итак, значащим ключам доверять нельзя. Они могут подойти для использования в небольших и/или очень стабильных системах, однако в большинстве случаев рекомендую отдавать предпочтение незначащим ключам.

Следующий спорный момент — это выбор между простыми и составными ключами. *Простой ключ (simple key)* состоит из одного поля базы данных, а *составной (compound)* — из нескольких. Составной ключ удобно использовать в ситуациях, когда одна таблица имеет смысл в контексте другой таблицы. В качестве примера можно привести ситуацию с заказами и пунктами заказов, когда в качестве составного ключа пунктов заказов можно использовать номер заказа в сочетании с порядковым номером этого пункта в данном заказе. Несмотря на то что составные ключи несут в себе больше смысла, простые ключи более универсальны. Если простые ключи используются во всех таблицах базы данных, их можно обрабатывать посредством общих, абстрактных фрагментов кода. Использование составных ключей требует специальной обработки в конкретных производных классах. (Разумеется, эта проблема успешно решается автоматической генерацией кода.) Кроме того, как уже отмечалось, составные ключи несут в себе определенную долю смысла, поэтому при их использовании необходимо тщательно следить за уникальностью и особенно за неизменяемостью ключевых полей.

Выбору подлежит и тип ключа. Основная операция, которая выполняется над ключами, — это проверка на равенство, поэтому в качестве ключей следует выбирать значения такого типа, для которого эта операция выполняется быстро. Еще одна распространенная операция — вычисление следующего значения ключа. Таким образом, в качестве ключа идеально подходят значения типа `long integer`. Можно использовать и строковые значения, однако в этом случае проверка на равенство будет выполняться медленнее, а вычисление следующего ключа окажется немного сложнее. Скорее всего, решающее слово в выборе типа ключей останется за администратором базы данных.

(Остерегайтесь применять в качестве ключей значения дат и времени. Они не только являются значащими, но и могут привести к проблемам с переносимостью и согласованностью. Особенно это касается дат, которые часто измеряются с точностью до различных долей секунды. Подобный способ измерения может легко нарушить синхронизацию данных и, как следствие, привести к проблемам идентификации.)

Используемые ключи могут быть уникальны по отношению к таблице или по отношению ко всей базе данных. Значения *ключа, уникального в пределах таблицы (table-unique key)*, должны быть уникальны для каждой строки текущей таблицы (обязательное условие для того, чтобы поле могло стать ключевым), но могут повторяться в других таблицах. В свою очередь, значения ключа, *уникального в пределах базы данных (database-unique key)*, должны быть уникальны для каждой строки каждой таблицы всей базы данных. В большинстве случаев можно обойтись и ключом, уникальным в пределах таблицы, однако использовать ключ, уникальный в пределах базы данных, значительно проще, а кроме того, это позволяет применять общую **коллекцию объектов (Identity Map, 216)**. Не беспокойтесь: современные базы данных умеют работать с большими числами, поэтому значения ключей вряд ли закончатся. Разумеется, при желании вы можете написать простой сценарий, который будет присваивать новым записям ключи удаленных объектов и таким образом "экономить" значения ключей, хотя для запуска такого сценария вам может понадобиться отсоединить приложение от базы данных. Впрочем, если вы используете 64-битовые ключи (что далеко не редкость), это вряд ли понадобится.

Используя ключи, уникальные в пределах таблицы, следует быть особенно осторожным с наследованием. Если вы применяете **наследование с таблицами для каждого конкретного класса (Concrete Table Inheritance, 313)** или **наследование с таблицами для каждого класса (Class Table Inheritance, 305)**, лучше выбирать ключи, уникальные в пределах всей иерархии объектов. Впрочем, я буду использовать термин "уникальный в пределах таблицы" даже тогда, когда формально это будет означать нечто наподобие "уникальный в пределах графа наследования".

Размер ключа может влиять на производительность, в частности по отношению к индексам. В большинстве случаев это зависит от используемой СУБД и/или количества строк в базе данных, однако, прежде чем принимать окончательное решение по поводу формы ключей, рекомендую выполнить "черновую" проверку.

Представление поля идентификации в объекте

Самая простая форма **поля идентификации** — это поле, тип которого соответствует типу ключа базы данных. Таким образом, если вы используете простой числовой ключ, вам вполне должно хватить поля какого-нибудь стандартного числового типа данных.

С составными ключами дело обстоит несколько сложнее. В этом случае рекомендую создать класс ключа. Универсальный класс ключа может содержать в себе последовательность объектов, являющихся элементами ключа. Поэтому ключевым методом класса ключа (к сожалению, приходится мириться с подобной игрой слов!) должно стать выполнение проверки на равенство. Кроме того, рекомендую реализовать метод, который будет возвращать части ключа при отображении объекта на базу данных.

Если все ключи имеют одну и ту же базовую структуру, все операции по обработке ключей можно вынести в **супертип слоя (Layer Supertype, 491)**. Сюда можно поместить все стандартное поведение, которое будет применяться в большинстве случаев, а обработку исключительных случаев реализовать в конкретных производных классах.

Для обработки составных ключей можно воспользоваться единственным классом, который будет содержать в себе общий список объектов, образующих ключ, или же создать по одному классу ключа на каждый класс домена с явными полями, соответствующими частям ключа. Обычно я предпочитаю явные решения, однако в данном случае не уверен, стоит ли так делать. Использование "явного" подхода приводит к появлению множества небольших классов, не представляющих собой ничего интересного. Основным преимуществом таких классов является возможность избежать ошибок, которые совершают пользователи, когда помещают элементы ключа в неправильном порядке, однако на практике это не представляет собой сколько-нибудь серьезной проблемы.

Если вы собираетесь импортировать данные между различными экземплярами баз данных, вам потребуется реализовать какую-нибудь схему, чтобы отделить ключи различных баз данных во избежание конфликтов между ключами. Для решения этой проблемы можно применить миграцию ключей импортируемых данных, однако подобный метод способен крайне запутать ситуацию.

Вычисление нового значения ключа

Каждому новому объекту нужен ключ. Теоретически это просто, однако на практике вычисление нового значения ключа может превратиться в проблему. Существует три способа получить новое значение ключа: автоматически сгенерировать его средствами базы данных, воспользоваться глобальным уникальным идентификатором (Globally Unique Identifier — GUID) или же сгенерировать ключ самому.

Наиболее простой способ — автоматическая генерация значения ключа с помощью средств базы данных. В этом случае каждый раз, когда в базу данных вставляются новые записи, она самостоятельно генерирует для них уникальный первичный ключ. От вас при этом не требуется *ничего*. Звучит слишком хорошо, чтобы быть правдой, не так ли? К сожалению, вы не ошиблись. Не все базы данных генерируют ключи по одному принципу; многие из них используются такими методами, которые усложняют объектно-реляционное отображение.

Самый распространенный принцип автоматической генерации ключа — объявить одно *автоматически генерируемое поле (auto-generated field)*, которому при вставке новой записи будет присваиваться новое значение. Данный подход имеет существенный недостаток: определить, какое именно значение было сгенерировано для ключа, крайне сложно. Например, если вы захотите вставить в базу данных новый заказ и указать несколько пунктов этого заказа, вам понадобится значение ключа нового заказа, чтобы указать его в качестве внешнего ключа пунктов заказов. Более того, это значение должно быть известно еще до окончания транзакции, чтобы все изменения можно было сохранить в рамках одной транзакции. К сожалению, получить эту информацию из базы данных обычно нельзя, поэтому описанный способ автоматической генерации не может применяться к таблицам, содержащим связанные объекты.

Альтернативный способ автоматической генерации ключа — использование *счетчика базы данных (database counter)*, который применяется в Oracle. В этом случае для вычисления значения ключа в базу данных отправляется SQL-запрос `SELECT`, содержащий ссылку на некоторую последовательность. СУБД выполняет запрос и возвращает результирующее множество данных, содержащее очередной член последовательности. В качестве шага последовательности можно указать любое целое число, что позволит получать несколько значений ключа одновременно. Запрос на вычисление значения ключа

автоматически помещается в отдельную транзакцию, поэтому обработка запроса не блокирует выполняющуюся в то же время вставку других записей. Подобный счетчик прекрасно подходит для наших нужд, однако это нестандартное решение, а кроме того, оно реализовано далеко не во всех базах данных.

Глобальный уникальный идентификатор (Globally Unique Identifier — GUID) представляет собой число, сгенерированное на некотором компьютере, которое гарантированно является уникальным во все времена для всех компьютеров мира. Многие платформы предоставляют интерфейсы API для генерации глобальных уникальных идентификаторов. Для вычисления нового идентификатора применяется чрезвычайно интересный алгоритм, который использует MAC-адрес Ethernet-карты, текущее время в наносекундах, идентификатор чипсета и, возможно, точное количество волосков на вашей левой руке (шутка). Умелое сочетание этих величин приводит к появлению действительно уникального (и поэтому безопасного) номера. Единственным недостатком глобального уникального идентификатора является его размер (а он, разумеется, очень большой). Зачастую возникают ситуации, когда пользователю нужно ввести значение ключа в поле окна или указать его в SQL-выражении, и, если оно окажется слишком длинным, это чревато массой опечаток. Кроме того, использование ключей такого большого размера может привести к резкому падению производительности (особенно это касается индексов).

Если ни один из перечисленных методов вам не подходит, значение ключа придется сгенерировать самому. В небольших системах это можно сделать следующим образом: с помощью SQL-функции `MAX()` определить максимальное значение ключа текущей таблицы и затем увеличить полученное значение на единицу. К сожалению, подобный запрос блокирует доступ ко всей таблице на время выполнения вставки. Это не составит проблемы, если вставка выполняется нечасто, однако при наличии множества параллельных вставок и обновлений применение подобного метода может привести к падению производительности. Кроме того, вам понадобится обеспечить полную изоляцию транзакций; в противном случае несколько транзакций могут получить одно и то же значение ключа.

Гораздо более удачным решением является применение *таблицы ключей (key table)*. Как правило, такая таблица состоит из двух столбцов: имя таблицы базы данных и следующее доступное значение ключа. Если ключи уникальны в пределах базы данных, таблица ключей будет содержать всего лишь одну строку. Если же ключи уникальны в пределах таблиц, то таблица ключей будет содержать по одной строке на каждую таблицу базы данных. Чтобы воспользоваться таблицей ключей, необходимо считать нужную строку, извлечь из нее значение ключа, увеличить его на заданный шаг и записать обратно в строку. При необходимости можно получить несколько ключей одновременно, увеличив значение в строке на достаточно большую величину. Это позволит сократить количество обращений к базе данных и количество параллельных обращений к таблице ключей.

Таблицу ключей хорошо спроектировать таким образом, чтобы доступ к ней осуществлялся в рамках отдельной транзакции, отличной от той, которая применяется для обновления основной таблицы данных. Предположим, я хочу вставить новый заказ в таблицу заказов. Для этого мне необходимо запретить запись в строку таблицы ключей, соответствующую таблице заказов (потому что я собираюсь обновлять эту строку). Блокировка строки будет действительна до окончания моей транзакции, поэтому всем, кто захочет получить значение этого же ключа, будет отказано в доступе. При использовании

ключей, уникальных в пределах таблицы, доступ к таблице ключей будет запрещен для всех, кто собирается выполнять вставку в таблицу заказов, а при использовании ключей, уникальных в пределах базы данных, — для всех, кто собирается выполнять вставку в любую из таблиц.

Если доступ к таблице ключей выполняется в отдельной транзакции, соответствующая строка будет заблокирована на гораздо меньшее время. Данный метод имеет небольшой недостаток: при откате транзакции, в рамках которой была выполнена вставка, ключ, полученный из таблицы ключей, будет утерян. К счастью, доступных номеров предостаточно, поэтому потерять один из них не так уж страшно. Кроме того, использование отдельной транзакции позволяет получить новый идентификатор сразу же после создания объекта, что зачастую происходит раньше открытия системной транзакции для завершения бизнес-транзакции.

Использование таблицы ключей влияет на то, какой ключ следует выбрать — уникальный в пределах таблицы или в пределах всей базы данных. Если ключи уникальны в пределах таблицы, в таблицу ключей придется добавлять новую строку каждый раз при добавлении в базу данных новой таблицы. Это прибавит работы, зато параллельных обращений к соответствующей строке таблицы ключей будет гораздо меньше. Конечно, если доступ к таблице ключей осуществляется в рамках отдельной транзакции, наличие большого числа параллельных обращений не слишком критично, особенно если за один запрос возвращается целый диапазон новых значений. Однако, если обновление таблицы ключей не удается вынести в отдельную транзакцию, использование ключа, уникального в пределах базы данных, станет большим неудобством.

Напоследок отмечу еще один важный момент. Код, с помощью которого вычисляется значение нового ключа, хорошо поместить в отдельный класс — так будет легче создать **фиктивную службу (Service Stub, 519)** для тестирования.

Назначение

Поле идентификации используется тогда, когда необходимо построить отображение между объектами, расположенными в оперативной памяти, и строками таблиц базы данных. Как правило, такая необходимость возникает при использовании **модели предметной области (Domain Model, 140)** или **шлюза записи данных (Row Data Gateway, 175)**. Подобное отображение не нужно, если вы используете **сценарий транзакции (Transaction Script, 133)**, **модуль таблицы (Table Module, 148)** или **шлюз таблицы данных (Table Data Gateway, 167)**.

Для небольшого объекта с семантикой значения (например, для объекта, представляющего денежную величину или диапазон дат), которому не соответствует отдельная таблица, вместо **поля идентификации** лучше воспользоваться **внедренным значением (Embedded Value, 288)**. В то же время для сложного графа объектов, к которому не нужно осуществлять запросов в пределах реляционной СУБД, добиться более удобного обновления и лучшей производительности можно за счет использования **крупного сериализованного объекта (Serialized LOB, 292)**.

В качестве альтернативы **полю идентификации** можно предложить расширение **коллекции объектов**. Это может пригодиться тогда, когда вы не хотите хранить **поле идентификации** в объекте приложения. В этом случае в **коллекции объектов** необходимо реализовать два метода поиска: объекта по ключу и ключа по объекту. Впрочем, к такому способу обращаются не слишком часто, поскольку хранить ключ в объекте обычно проще.

Дополнительные источники информации

Несколько приемов генерации ключей рассмотрены в [28].

Пример: числовой ключ (C#)

Самая простая форма **поля идентификации** — это числовое поле базы данных, которое отображается на числовое поле объекта приложения.

```
class DomainObject...

      public const long PLACEHOLDER_ID = -1;
      public long Id = PLACEHOLDER_ID;
      public Boolean isNew() {return Id == PLACEHOLDER_ID;}
```

У объекта, который был создан в оперативной памяти, но не был сохранен в базе данных, не будет значения ключа. Это может стать проблемой для объектов .NET, поскольку в .NET поля не могут иметь значение `NULL`. Поэтому идентификатору создаваемого объекта присвоено значение-заменитель `PLACEHOLDER_ID`.

Наличие у объекта ключа необходимо в двух случаях: при поиске и при вставке. Для выполнения поиска необходимо сформировать SQL-запрос, указав значение ключа в предложении `WHERE`. В .NET для проведения поиска множество строк базы данных можно загрузить в объект `DataSet` и затем с помощью метода поиска выбрать из них нужную.

```
class CricketerMapper...

      public Cricketer Find(long id) {
         return (Cricketer) AbstractFind(id);
      }

class Mapper...

      protected DomainObject AbstractFind(long id) {
         DataRow row = FindRow(id);
         return (row == null) ? null : Find(row);
      }
      protected DataRow FindRow(long id) {
         String filter = String.Format("id = {0}", id);
         DataRow[] results = table.Select(filter);
         return (results.Length == 0) ? null : results[0];
      }
      public DomainObject Find (DataRow row) {
         DomainObject result = CreateDomainObject();
         Load(result, row);
         return result;
      }
      abstract protected DomainObject CreateDomainObject();
```

Большая часть этого кода может быть вынесена в **супертип слоя (Layer Supertype, 491)**. Реализация методов поиска в производных классах нужна только затем, чтобы инкапсулировать обратное приведение результата к нужному типу. Естественно, это не касается языков, которые не используют типизацию во время компиляции.

Для простого числового **поля идентификации** выполнение вставки также может быть вынесено в **супертип слоя**.

```
class Mapper...

      public virtual long Insert (DomainObject arg) {
         DataRow row = table.NewRow();
         arg.Id = GetNextID();
         row["id"] = arg.Id;
         Save(arg, row);
         table.Rows.Add(row);
         return arg.Id;
      }
```

Суть вставки заключается в создании новой строки и присвоении ей следующего доступного значения ключа. После этого в новую строку останется скопировать содержимое соответствующего объекта.

Пример: использование таблицы ключей (Java)

Мэттью Фоммел и Мартин Фаулер

Если используемая СУБД поддерживает счетчик базы данных и вас не беспокоит зависимость от специфической реализации SQL, обязательно воспользуйтесь счетчиком. Даже если вы не хотите быть привязанными к конкретной СУБД, все равно подумайте об использовании счетчика: поскольку код генерации ключа будет хорошо инкапсулирован, вы всегда сможете переделать его в легкопереносимый алгоритм, если это понадобится. Вы даже можете разработать стратегию [20], чтобы использовать счетчики, когда они есть, и создавать свои счетчики, когда их нет.

Предположим, что все приходится делать вручную. Вначале необходимо создать таблицу ключей базы данных.

```
CREATE TABLE keys (name varchar primary key, nextID int)
INSERT INTO keys VALUES ('orders', 1)
```

Эта таблица будет содержать по одной строке на каждый счетчик базы данных. В нашем примере поле `nextID` инициализировано значением `1`. Если вы применяете эту таблицу к уже существующей базе данных, вместо единицы необходимо указать следующее доступное значение ключа. Если ключи уникальны в пределах базы данных, вам понадобится только одна строка, если они уникальны в пределах таблиц — по одной строке на каждую таблицу.

Весь код, касающийся генерации значений ключа, можно поместить в отдельный класс. Так его будет легче использовать в приложении и между приложениями. Кроме того, при наличии отдельного класса процесс резервирования ключа будет легче выделить в отдельную транзакцию.

В нашем примере построим генератор ключа, который будет использовать собственное соединение с базой данных и получать информацию о том, сколько ключей следует сгенерировать за один раз.

```
class KeyGenerator...

   private Connection conn;
   private String keyName;
   private long nextId;
   private long maxId;
   private int incrementBy;
   public KeyGenerator(Connection conn, String keyName,
↳int incrementBy) {
      this.conn = conn;
      this.keyName = keyName;
      this.incrementBy = incrementBy;
      nextId = maxId = 0;
      try {
         conn.setAutoCommit(false);
      } catch(SQLException exc) {
         throw new ApplicationException("Unable to turn off
↳autocommit", exc);
      }
   }
```

Необходимо гарантировать, чтобы фиксация транзакции не выполнялась автоматически, так как операции извлечения и обновления *должны* осуществляться в рамках одной транзакции.

Когда направляется запрос на вычисление нового значения ключа, генератор проверяет, не осталось ли в его кэше неиспользованных значений, извлеченных им в прошлый раз.

```
class KeyGenerator...

   public synchronized Long nextKey() {
      if (nextId == maxId) {
         reserveIds();
      }
      return new Long(nextId++);
   }
```

Если кэшированных значений уже не осталось, генератор вынужден обратиться к базе данных.

```
class KeyGenerator...

   private void reserveIds() {
      PreparedStatement stmt = null;
      ResultSet rs = null;
      long newNextId;
      try {
         stmt = conn.prepareStatement("SELECT nextID FROM keys
↳WHERE name = ? FOR UPDATE");
         stmt.setString(1, keyName);
         rs = stmt.executeQuery();
         rs.next();
         newNextId = rs.getLong(1);
      }
```

```
      catch (SQLException exc) {
         throw new ApplicationException("Unable to generate
↳ids", exc);
      }
      finally {
         DB.cleanUp(stmt, rs);
      }
      long newMaxId = newNextId + incrementBy;
      stmt = null;
      try {
         stmt = conn.prepareStatement("UPDATE keys SET
↳nextID = ? WHERE name = ?");
         stmt.setLong(1, newMaxId);
         stmt.setString(2, keyName);
         stmt.executeUpdate();
         conn.commit();
         nextId = newNextId;
         maxId = newMaxId;
      }
      catch (SQLException exc) {
         throw new ApplicationException("Unable to generate
↳ids", exc);
      }
      finally {
         DB.cleanUp(stmt);
      }
   }
```

В этом примере использован оператор `SELECT... FOR UPDATE`, чтобы указать базе данных на необходимость заблокировать запись в таблице ключей. Синтаксис данного оператора специфичен для Oracle, поэтому в других СУБД это SQL-выражение может выглядеть несколько иначе. Когда вы не можете установить блокировку записи с помощью оператора `SELECT`, ваша транзакция может быть прервана, если кто-то решит обновить значение ключа немного раньше вас. Впрочем, в подобных случаях достаточно просто перезапускать метод `reserveIds` до тех пор, пока вы не получите "неиспорченный" набор ключей.

Пример: использование составного ключа (Java)

Использование простого числового ключа — прекрасное решение, однако в некоторых случаях без составных ключей просто не обойтись.

Класс ключа

Как только ключ становится чем-то более сложным, нежели простой порядковый номер, его рекомендуется выделить в отдельный класс ключа. Последний должен хранить в себе элементы, образующие ключ, и выполнять проверку ключей на равенство.

```
class Key...

   private Object[] fields;
   public boolean equals(Object obj) {
      if (!(obj instanceof Key)) return false;
```

```
      Key otherKey = (Key) obj;
      if (this.fields.length != otherKey.fields.length)
↳return false;
      for (int i = 0; i < fields.length; i++)
         if (!this.fields[i].equals(otherKey.fields[i]))
↳return false;
      return true;
}
```

Самый простой способ создать экземпляр класса ключа — передать его конструктору в качестве аргумента массив полей, образующих ключ.

```
class Key...

   public Key(Object[] fields) {
      checkKeyNotNull(fields);
      this.fields = fields;
   }
   private void checkKeyNotNull(Object[] fields) {
      if (fields == null) throw new IllegalArgumentException(
↳"Cannot have a null key");
      for (int i = 0; i < fields.length; i++)
         if (fields[i] == null)
            throw new IllegalArgumentException("Cannot have a
↳null element of key");
   }
```

Если вам постоянно приходится создавать ключи, состоящие из конкретных элементов, к классу ключей можно добавить еще несколько конструкторов, принимающих те или иные аргументы. Точный набор конструкторов зависит от того, ключи какого типа используются в вашем приложении.

```
class Key...

   public Key(long arg) {
      this.fields = new Object[1];
      this.fields[0] = new Long(arg);
   }
   public Key(Object field) {
      if (field == null) throw new IllegalArgumentException(
↳"Cannot have a null key");
      this.fields = new Object[1];
      this.fields[0] = field;
   }
   public Key(Object arg1, Object arg2) {
      this.fields = new Object[2];
      this.fields[0] = arg1;
      this.fields[1] = arg2;
      checkKeyNotNull(fields);
   }
```

Не бойтесь создавать побольше подобных конструкторов. В конце концов, они очень удобны, а это весьма важно для всех, кто работает с ключами.

Аналогичным образом к классу ключа можно добавить всевозможные функции доступа, которые будут применяться для извлечения различных элементов ключа. Подобные функции необходимы приложению для выполнения отображений.

```
class Key...

   public Object value(int i) {
      return fields[i];
   }
   public Object value() {
      checkSingleKey();
      return fields[0];
   }
   private void checkSingleKey() {
      if (fields.length > 1)
         throw new IllegalStateException("Cannot
↳take value on composite key");
   }
   public long longValue() {
      checkSingleKey();
      return longValue(0);
   }
   public long longValue(int i) {
      if (!(fields[i] instanceof Long))
         throw new IllegalStateException("Cannot
↳take longValue on non long key");
      return ((Long) fields[i]).longValue();
   }
```

В этом примере выполняется отображение объектов на таблицы заказов (`orders`) и пунктов заказов (`line_items`). Таблица заказов имеет простой числовой ключ, а таблица пунктов заказов — составной ключ, представляющий собой совокупность первичного ключа заказа и порядкового номера пункта заказа в данном заказе.

```
CREATE TABLE orders (ID int primary key, customer varchar)
CREATE TABLE line_items (orderID int, seq int, amount int,
↳product varchar, primary key (orderID,seq))
```

Супертип слоя объектов домена должен содержать в себе поле ключа.

```
class DomainObjectWithKey...

   private Key key;
   protected DomainObjectWithKey(Key ID) {
      this.key = ID;
   }
   protected DomainObjectWithKey() {
   }
   public Key getKey() {
      return key;
   }
   public void setKey(Key key) {
      this.key = key;
   }
```

Чтение

Как и в других примерах этой книги, я решил разбить поведение преобразователей на методы поиска (которые находят нужную строку базы данных) и методы загрузки (которые загружают данные из этой строки в объект домена). И те и другие используют в своей работе объект ключа.

Основное различие между этим и остальными примерами данной книги (в которых применяются простые числовые ключи) состоит в необходимости вынесения в абстрактный класс элементов поведения, которые позднее будут переопределены в производных классах, имеющих более сложные ключи. В этом примере я исхожу из предположения, что в большей части таблиц используются простые числовые ключи. Тем не менее некоторые таблицы применяют ключи другого типа, поэтому поведение для простых числовых ключей было принято в качестве стандартного и вынесено в **супертип слоя** преобразователя. Одной из таблиц, использующих простые числовые ключи, является таблица заказов. Ниже приведен код стандартного метода поиска.

```
class OrderMapper...

   public Order find(Key key) {
      return (Order) abstractFind(key);
   }
   public Order find(Long id) {
      return find(new Key(id));
   }
   protected String findStatementString() {
      return "SELECT id, customer from orders WHERE id = ?";
   }

class AbstractMapper...

   abstract protected String findStatementString();
   protected Map loadedMap = new HashMap();
   public DomainObjectWithKey abstractFind(Key key) {
      DomainObjectWithKey result = (DomainObjectWithKey)
↳loadedMap.get(key);
      if (result != null) return result;
      ResultSet rs = null;
      PreparedStatement findStatement = null;
      try {
         findStatement = DB.prepare(findStatementString());
         loadFindStatement(key, findStatement);
         rs = findStatement.executeQuery();
         rs.next();
         if (rs.isAfterLast()) return null;
         result = load(rs);
         return result;
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {
         DB.cleanUp(findStatement, rs);
      }
   }
   // метод загрузки параметров для ключей,
```

```
   // не являющихся простыми числовыми
   protected void loadFindStatement(Key key, PreparedStatement
↳finder) throws SQLException {
      finder.setLong(1, key.longValue());
   }
```

В приведенном коде не было реализовано построение SQL-выражения, потому что в зависимости от типа ключа в него нужно передавать разное количество параметров. Таблица пунктов заказов имеет составной ключ, поэтому метод `findStatementString` должен быть переопределен в классе `LineItemMapper`.

```
class LineItemMapper...

   public LineItem find(long orderID, long seq) {
      Key key = new Key(new Long(orderID), new Long(seq));
      return (LineItem) abstractFind(key);
   }
   public LineItem find(Key key) {
      return (LineItem) abstractFind(key);
   }
   protected String findStatementString() {
      return
         "SELECT orderID, seq, amount, product " +
         "    FROM line_items " +
         "    WHERE (orderID = ?) AND (seq = ?)";
   }
   // переопределение метода загрузки
   // параметров для составных ключей
   protected void loadFindStatement(Key key, PreparedStatement
↳finder) throws SQLException {
      finder.setLong(1, orderID(key));
      finder.setLong(2, sequenceNumber(key));
   }
   // вспомогательные методы для извлечения элементов ключа
   private static long orderID(Key key) {
      return key.longValue(0);
   }
   private static long sequenceNumber(Key key) {
      return key.longValue(1);
   }
```

Помимо предоставления SQL-выражения и определения интерфейса для методов поиска, в производном классе следует переопределить метод загрузки параметров SQL-запроса, чтобы в соответствующее выражение можно было передать два параметра. Кроме того, я написал два вспомогательных метода, предназначенных для извлечения частей ключа. Это сделано для того, чтобы не описывать в теле метода `loadFindStatement()` явный доступ к элементам ключа с указанием их номеров в массиве полей. Использование явных номеров — признак дурного тона.

Выполнение загрузки также разбито на две части: стандартное поведение для простых числовых ключей реализовано в **супертипе слоя** и переопределено в производных классах с более сложными ключами. В нашем примере загрузка объекта заказа выглядит, как показано далее.

```
class AbstractMapper...

   protected DomainObjectWithKey load(ResultSet rs)
↳throws SQLException {
      Key key = createKey(rs);
      if (loadedMap.containsKey(key)) return
↳(DomainObjectWithKey) loadedMap.get(key);
      DomainObjectWithKey result = doLoad(key, rs);
      loadedMap.put(key, result);
      return result;
   }
   abstract protected DomainObjectWithKey doLoad(Key id,
↳ResultSet rs) throws SQLException;
   // метод загрузки параметров для ключей,
   // не являющихся простыми числовыми
   protected Key createKey(ResultSet rs) throws SQLException {
      return new Key(rs.getLong(1));
   }

class OrderMapper...

   protected DomainObjectWithKey doLoad(Key key, ResultSet rs)
↳throws SQLException {
      String customer = rs.getString("customer");
      Order result = new Order(key, customer);
      MapperRegistry.lineItem().loadAllLineItemsFor(result);
      return result;
   }
```

Классу `LineItemMapper` нужно переопределить метод `createKey()`, чтобы создавать ключ, состоящий из двух полей.

```
class LineItemMapper...

   protected DomainObjectWithKey doLoad(Key key, ResultSet rs)
↳throws SQLException {
      Order theOrder = MapperRegistry.order().find(
↳orderID(key));
      return doLoad(key, rs, theOrder);
   }
   protected DomainObjectWithKey doLoad(Key key, ResultSet rs,
↳Order order) throws SQLException
   {
      LineItem result;
      int amount = rs.getInt("amount");
      String product = rs.getString("product");
      result = new LineItem(key, amount, product);
      order.addLineItem(result); // ссылается на заказ
      return result;
   }
   // переопределяет стандартное поведение
   protected Key createKey(ResultSet rs) throws SQLException {
      Key key = new Key(new Long(rs.getLong("orderID")),
```

```
↳new Long(rs.getLong("seq")));
      return key;
   }
```

Кроме того, у класса `LineItemMapper` есть специальный метод, предназначенный для загрузки всех пунктов заданного заказа.

```
class LineItemMapper...

   public void loadAllLineItemsFor(Order arg) {
      PreparedStatement stmt = null;
      ResultSet rs = null;
      try {
         stmt = DB.prepare(findForOrderString);
         stmt.setLong(1, arg.getKey().longValue());
         rs = stmt.executeQuery();
         while (rs.next())
            load(rs, arg);
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally { DB.cleanUp(stmt, rs);
      }
   }
   private final static String findForOrderString =
         "SELECT orderID, seq, amount, product " +
            "FROM line_items " +
            "WHERE orderID = ?";
   protected DomainObjectWithKey load(ResultSet rs, Order order)
↳throws SQLException {
      Key key = createKey(rs);
      if (loadedMap.containsKey(key)) return
↳(DomainObjectWithKey) loadedMap.get(key);
      DomainObjectWithKey result = doLoad(key, rs, order);
      loadedMap.put(key, result);
      return result;
   }
```

Выполнение данного метода связано с определенными трудностями, потому что объект заказа помещается в **коллекцию объектов** только после своего создания. Эту проблему можно решить путем создания пустого объекта и вставки его прямо в **поле идентификации** (см. стр. 237).

Вставка

Как и операция чтения, операция вставки разбита на несколько частей: в абстрактном классе описано стандартное поведение для простых числовых ключей, переопределенное в производных классах с более сложными ключами. В **супертипе слоя** преобразователя я определил метод, который будет выступать в качестве интерфейса, а также шаблонный метод, выполняющий вставку нового объекта.

```
class AbstractMapper...

   public Key insert(DomainObjectWithKey subject) {
```

```
      try {
         return performInsert(subject,
↳findNextDatabaseKeyObject());
      } catch (SQLException e) {
         throw new ApplicationException(e);
      }
   }
   protected Key performInsert(DomainObjectWithKey subject,
↳Key key) throws SQLException {
      subject.setKey(key);
      PreparedStatement stmt = DB.prepare(
↳insertStatementString());
      insertKey(subject, stmt);
      insertData(subject, stmt);
      stmt.execute();
      loadedMap.put(subject.getKey(), subject);
      return subject.getKey();
   }
   abstract protected String insertStatementString();

class OrderMapper...

   protected String insertStatementString() {
      return "INSERT INTO orders VALUES(?,?)";
   }
```

Содержимое объекта передается методу `performInsert` посредством двух методов, отделяющих элементы ключа от основных данных. Это позволит описать стандартную реализацию вставки ключа, применимую для всех классов, использующих простой числовой ключ (например, для класса `Order`).

```
class AbstractMapper...

   protected void insertKey(DomainObjectWithKey subject,
↳PreparedStatement stmt) throws SQLException
   {
      stmt.setLong(1, subject.getKey().longValue());
   }
```

Оставшаяся часть данных, передаваемых методу `performInsert`, специфична для конкретного производного класса, поэтому в суперклассе соответствующее поведение описано как абстрактное.

```
class AbstractMapper...

   abstract protected void insertData(DomainObjectWithKey
↳subject, PreparedStatement stmt) throws SQLException;

class OrderMapper...

   protected void insertData(DomainObjectWithKey
↳abstractSubject, PreparedStatement stmt) {
      try {
```

```
            Order subject = (Order) abstractSubject;
            stmt.setString(2, subject.getCustomer());
        } catch (SQLException e) {
            throw new ApplicationException(e);
        }
    }
```

Класс `LineItemMapper` переопределяет оба приведенных метода. Здесь для вставки ключа из объекта извлекаются два значения.

```
class LineItemMapper...

    protected String insertStatementString() {
        return "INSERT INTO line_items VALUES (?, ?, ?, ?)";
    }
    protected void insertKey(DomainObjectWithKey subject,
↳PreparedStatement stmt)
        throws SQLException
    {
        stmt.setLong(1, orderID(subject.getKey()));
        stmt.setLong(2, sequenceNumber(subject.getKey()));
    }
```

Кроме того, класс `LineItemMapper` предоставляет собственную реализацию метода `insertData` для вставки основного содержимого объекта.

```
class LineItemMapper...

    protected void insertData(DomainObjectWithKey subject,
↳PreparedStatement stmt) throws SQLException
    {
        LineItem item = (LineItem) subject;
        stmt.setInt(3, item.getAmount());
        stmt.setString(4, item.getProduct());
    }
```

Подобная реализация вставки данных имеет смысл только в том случае, если большинство классов используют для хранения ключа одно и то же поле. Если же обработка значений ключа в производных классах слишком различна, гораздо проще вставить все данные посредством одной команды.

Вычисление следующего значения ключа также можно разбить на стандартное и переопределяемое поведение. В стандартном случае можно воспользоваться таблицей ключей, которая упоминалась в предыдущем примере. К сожалению, в ситуации с пунктами заказов все обстоит гораздо сложнее. Ключ таблицы пунктов заказов является составным и содержит в себе ключ таблицы заказов. Между тем у класса `LineItem` нет ссылки на класс `Order`, поэтому объект `LineItem` не может быть вставлен в базу данных, если ему не предоставят нужное значение ключа заказа. Это приводит к необходимости использования весьма неприятного подхода, связанного с генерацией исключения неподдерживаемой операции (unsupported operation exception).

```
class LineItemMapper...

   public Key insert(DomainObjectWithKey subject) {
      throw new UnsupportedOperationException
            ("Must supply an order when inserting a line item");
   }
   public Key insert(LineItem item, Order order) {
      try {
         Key key = new Key(order.getKey().value(),
↳getNextSequenceNumber(order));
         return performInsert(item, key);
      } catch (SQLException e) {
         throw new ApplicationException(e);
      }
   }
```

Разумеется, этого можно было избежать, создав обратную ссылку класса пунктов заказов на класс заказов (вследствие чего связь между классами превратилась бы в двунаправленную). Тем не менее в данном примере я решил продемонстрировать, что следует делать, если обратной ссылки нет.

Передав методу `insertKey` объект заказа, из него легко извлечь значение ключа, тем самым получив первый элемент ключа для пункта заказа. Но как узнать порядковый номер нового пункта заказа? Для этого необходимо определить следующий доступный номер пункта заказа, применив SQL-функцию `MAX` или же просмотрев пункты данного заказа, хранящиеся в оперативной памяти. В этом примере используется второй вариант.

```
class LineItemMapper...

   private Long getNextSequenceNumber(Order order) {
      loadAllLineItemsFor(order);
      Iterator it = order.getItems().iterator();
      LineItem candidate = (LineItem) it.next();
      while (it.hasNext()) {
         LineItem thisItem = (LineItem) it.next();
         if (thisItem.getKey() == null) continue;
         if (sequenceNumber(thisItem) > sequenceNumber(
↳candidate)) candidate = thisItem;
      }
      return new Long(sequenceNumber(candidate) + 1);
   }
   private static long sequenceNumber(LineItem li) {
      return sequenceNumber(li.getKey());
   }
   // метод сравнения может не сработать из-за несохраненных
   // значений ключа, равных null
   protected String keyTableRow() {
      throw new UnsupportedOperationException();
   }
```

Этот алгоритм был бы гораздо лучше, если бы я воспользовался методом `Collections.max`, однако, поскольку у нас может быть (и будет), как минимум, одно значение ключа, равное `NULL`, данный метод не подойдет.

Обновление и удаление

После всех перенесенных мучений операции обновления и удаления кажутся просто безобидными. Опять же, у нас есть абстрактный метод для стандартных случаев и переопределенный метод для частных случаев.

Обновление данных выполняется следующим образом:

```
class AbstractMapper...

   public void update(DomainObjectWithKey subject) {
      PreparedStatement stmt = null;
      try {
         stmt = DB.prepare(updateStatementString());
         loadUpdateStatement(subject, stmt);
         stmt.execute();
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {
         DB.cleanUp(stmt);
      }
   }
   abstract protected String updateStatementString();
   abstract protected void loadUpdateStatement(
↳DomainObjectWithKey subject, PreparedStatement stmt)
      throws SQLException;

class OrderMapper...

   protected void loadUpdateStatement(
↳DomainObjectWithKey subject, PreparedStatement stmt)
      throws SQLException
   {
      Order order = (Order) subject;
      stmt.setString(1, order.getCustomer());
      stmt.setLong(2, order.getKey().longValue());
   }
   protected String updateStatementString() {
      return "UPDATE orders SET customer = ? WHERE id = ?";
   }

class LineItemMapper...

   protected String updateStatementString() {
      return
         "UPDATE line_items " +
         "    SET amount = ?, product = ? " +
         "    WHERE orderId = ? AND seq = ?";
   }
   protected void loadUpdateStatement(
↳DomainObjectWithKey subject, PreparedStatement stmt)
```

```
            throws SQLException
    {
        stmt.setLong(3, orderID(subject.getKey()));
        stmt.setLong(4, sequenceNumber(subject.getKey()));
        LineItem li = (LineItem) subject;
        stmt.setInt(1, li.getAmount());
        stmt.setString(2, li.getProduct());
    }
```

А вот как происходит удаление:

```
class AbstractMapper...

    public void delete(DomainObjectWithKey subject) {
        PreparedStatement stmt = null;
        try {
            stmt = DB.prepare(deleteStatementString());
            loadDeleteStatement(subject, stmt);
            stmt.execute();
        } catch (SQLException e) {
            throw new ApplicationException(e);
        } finally {
            DB.cleanUp(stmt);
        }
    }
    abstract protected String deleteStatementString();
    protected void loadDeleteStatement(DomainObjectWithKey
↳subject, PreparedStatement stmt)
            throws SQLException
    {
        stmt.setLong(1, subject.getKey().longValue());
    }

class OrderMapper...

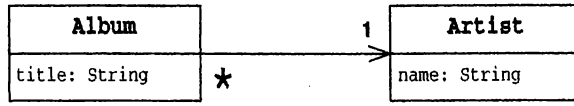
    protected String deleteStatementString() {
        return "DELETE FROM orders WHERE id = ?";
    }

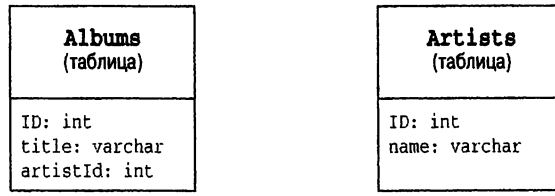
class LineItemMapper...

    protected String deleteStatementString() {
        return "DELETE FROM line_items WHERE orderid = ? AND
↳seq = ?";
    }
    protected void loadDeleteStatement(DomainObjectWithKey
↳subject, PreparedStatement stmt)
            throws SQLException
    {
        stmt.setLong(1, orderID(subject.getKey()));
        stmt.setLong(2, sequenceNumber(subject.getKey()));
    }
```

Отображение внешних ключей (Foreign Key Mapping)

Отображает ассоциации между объектами на ссылки внешнего ключа между таблицами базы данных





Объекты могут ссылаться непосредственно друг на друга с помощью объектных ссылок (object references). Даже самая простая объектно-ориентированная система обязательно содержит группу объектов, связанных между собой всеми возможными и невозможными способами. Разумеется, при сохранении объектов в базе данных необходимо позаботиться и о сохранении всех ссылок. К сожалению, поскольку содержимое объектов специфично для конкретного экземпляра запущенной программы, сохранение значений "в чистом виде" ничего не даст. Данную проблему еще более усложняет тот факт, что объекты могут содержать коллекции ссылок на другие объекты. Подобная структура нарушает определение первой нормальной формы реляционных баз данных.

Типовое решение **отображение внешних ключей** отображает объектную ссылку на внешний ключ базы данных.

Принцип действия

Первая мысль, возникающая по поводу данной проблемы, — воспользоваться **полем идентификации (Identity Field, 237)**. Действительно, каждому объекту соответствует определенное значение ключа таблицы базы данных. Если два объекта связаны между собой некоторой ассоциацией, в базе данных ее можно заменить внешним ключом. Например, если вы сохраняете в базе данных запись об альбоме, в эту же строку можно поместить и идентификатор исполнителя, связанного с этим альбомом (рис. 12.1).

Рассмотренный случай довольно прост. А как же быть, если речь идет о ссылке на коллекцию объектов? В базе данных нельзя сохранить коллекцию ключей, поэтому направление ссылки придется изменить. Таким образом, имея коллекцию композиций альбома, необходимо поместить внешний ключ альбома в запись о каждой композиции (рис. 12.2 и 12.3). Основные сложности подобных ситуаций связаны с обновлением. В рамках обновления в коллекции композиций альбома могут произойти изменения,

связанные с удалением или добавлением новых объектов. Как определить, какие изменения должны быть внесены в базу данных? Для этого можно воспользоваться тремя способами: удалением и вставкой, добавлением обратного указателя и операцией сравнения.

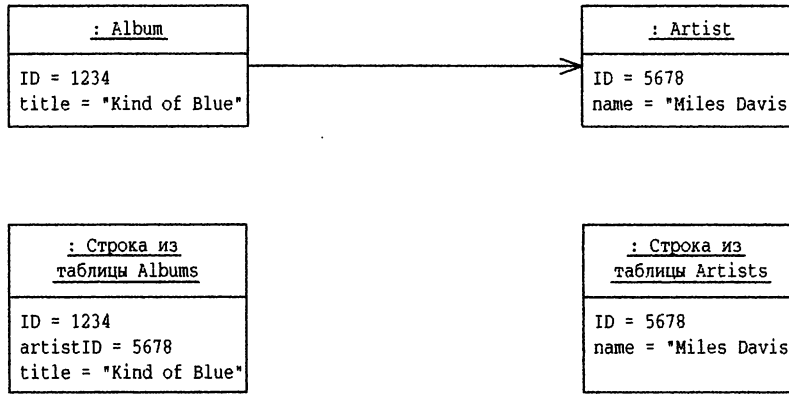


Рис. 12.1. Отображение коллекции на внешний ключ

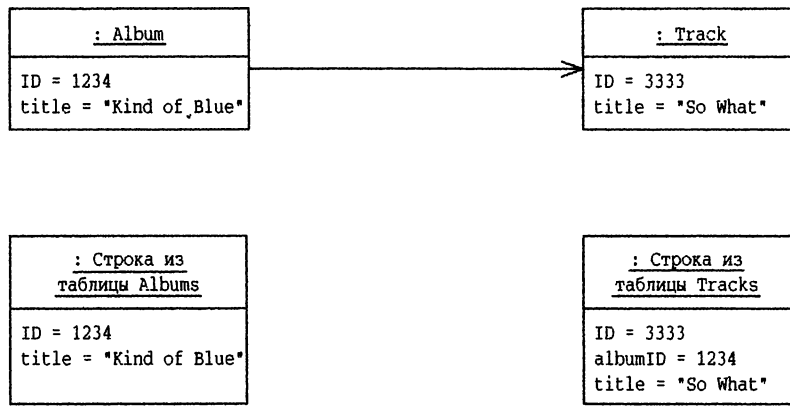


Рис. 12.2. Отображение коллекции на внешний ключ

Первый, и наиболее простой, способ — удалить из базы данных все композиции, ссылающиеся на заданный альбом, и затем вставить в нее те композиции, которые на данный момент находятся в альбоме. На первый взгляд это звучит не очень умно, особенно если набор композиций на самом деле не изменился. Тем не менее эта логика проста в реализации и, кроме того, работает гораздо надежнее других способов. Следует также отметить, что данный способ можно применить только в том случае, если композиции связаны **отображениями зависимых объектов (Dependent Mapping, 283)**, а значит, являются полностью зависимыми от объекта-альбома и на них нельзя ссылаться извне.

Добавление обратного указателя подразумевает создание ссылки композиции на альбом, в результате чего связь между этими объектами становится двунаправленной. Разумеется, это изменяет объектную модель, однако позволяет значительно упростить обновление данных путем простого обновления однозначных полей "с другой стороны" ссылки.

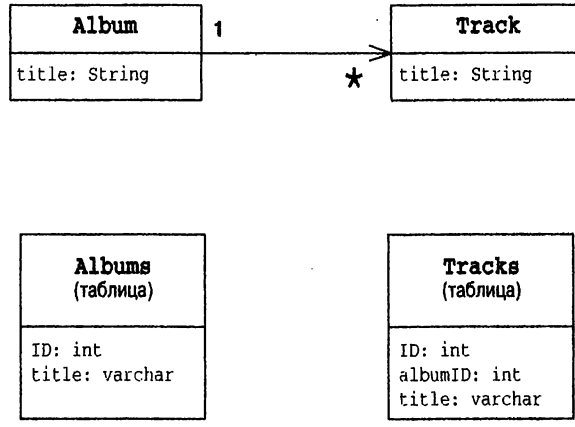


Рис. 12.3. Классы и таблицы, связанные многозначной ссылкой

Если ни один из перечисленных способов вам не подойдет, попробуйте провести сравнение. В качестве эталона можно воспользоваться текущим состоянием базы данных или же множеством данных, которое было считано при первой загрузке объекта. В первом случае от вас требуется еще раз считать из базы данных коллекцию композиций, входящих в альбом, и затем сравнить их с текущим содержимым альбома. Все композиции считанного множества данных, которых нет в альбоме, должны быть удалены из базы данных, а все композиции альбома, которых нет в считанном множестве данных, должны быть добавлены в базу данных.

Чтобы сравнить текущее состояние коллекции с множеством данных, которое было считано при первой загрузке объекта, необходимо реализовать сохранение считываемых данных. Это более предпочтительный способ, поскольку он избавляет от необходимости дополнительного обращения к базе данных. Впрочем, обращение к базе данных может понадобиться, если вы используете **оптимистическую автономную блокировку (Optimistic Offline Lock, 434)**.

В более общем случае каждый объект, добавляемый в коллекцию, должен быть проверен на предмет того, является ли он новым объектом. Это можно установить по наличию ключа; если у добавленного объекта нет ключа, значит, он новый и его следует вставить в базу данных. Для реализации подобной проверки хорошо воспользоваться **единицей работы (Unit of Work, 205)**, поскольку она автоматически вставляет в базу данных новые объекты. В любом случае после выполнения этих действий в базе данных необходимо найти строку, соответствующую добавленной композиции, и обновить ее внешний ключ так, чтобы он указывал на текущий альбом.

При удалении композиции из содержимого альбома необходимо знать, была ли она перемещена в другой альбом, оставлена без альбома или окончательно удалена из базы данных. Если композиция была перемещена в другой альбом, ее внешний ключ должен быть обновлен при обновлении альбома. Если композиция была оставлена без альбома, ее внешнему ключу необходимо присвоить значение `NULL`. И наконец, если композиция была удалена, ее необходимо удалить вместе со всеми другими удаляемыми записями. Выполнять удаление значительно проще, если наличие обратной ссылки обязательно (как в нашем примере, когда каждая композиция должна принадлежать какому-нибудь

альбому). В этом случае вам не придется определять, какие композиции были удалены из альбома, поскольку значения их внешних ключей будут обновлены при обновлении альбомов, в которые эти композиции были перемещены.

Обратная ссылка может быть неизменяемой (т.е. композиция не может быть перемещена в другой альбом). В этом случае добавление композиции в альбом всегда означает вставку строки в базу данных, а удаление из альбома — удаление соответствующей строки из базы данных. Это еще более упрощает обновление коллекций.

Работая с базой данных, необходимо следить за наличием циклических ссылок. Предположим, нужно загрузить заказ, ссылающийся на покупателя (который тоже будет загружен). Каждому покупателю соответствует множество платежей (сведения о котором тоже будут загружены), а каждый платеж ссылается на оплачиваемые им заказы. Последнее множество может включать в себя упомянутый заказ. Таким образом, будет загружен и этот заказ, после чего все повторяется снова и снова.

Во избежание подобной мешанины можно воспользоваться одним из двух способов создания объектов. Многие предпочитают применять конструкторы с аргументами, чтобы создаваемые объекты были полностью загружены данными. В этом случае придется прибегнуть к **загрузке по требованию (Lazy Load, 220)**, чтобы цикл загрузки прерывался в нужных местах. Если этого не сделать, вам грозит переполнение стека. Впрочем, если результаты тестирования оказались вполне приличными, без применения **загрузки по требованию** можно обойтись.

Второй возможный вариант — создать пустой объект и сразу же поместить его в **коллекцию объектов (Identity Map, 216)**. В этом случае при повторной ссылке на объект **коллекция объектов** сообщит, что он уже был загружен, и цикл будет прерван. Разумеется, объекты, создаваемые подобным образом, не заполнены данными, однако по окончании загрузки они будут содержать в себе все, что нужно. Это позволяет избежать описания частных случаев **загрузки по требованию**, что было бы крайне утомительно, если нужно всего лишь корректно загрузить объект.

Назначение

Отображение внешних ключей может применяться для моделирования практически всех видов связей между классами. Наиболее распространенный случай, когда **отображение внешних ключей** применить нельзя, — это связи типа "многие ко многим". Внешние ключи являются одномерными значениями, а из определения первой нормальной формы следует, что в одном поле нельзя хранить множественные значения внешних ключей. В этом случае вместо **отображения внешних ключей** необходимо воспользоваться **отображением с помощью таблицы ассоциаций (Association Table Mapping, 269)**.

Если у вас есть поле коллекции без обратного указателя, подумайте о том, не сделать ли "множественную сторону" ссылки **отображением зависимых объектов**. Это значительно упростит обработку коллекции.

Если связанный объект является **объектом-значением (Value Object, 500)**, вместо **отображения внешних ключей** следует воспользоваться **внедренным значением (Embedded Value, 288)**.

Пример: однозначная ссылка (Java)

Рассмотрим наиболее простой случай, когда объект-альбом ссылается на своего исполнителя.

```
class Artist...

   private String name;
   public Artist(Long ID, String name) {
      super(ID);
      this.name = name;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }

class Album...

   private String title;
   private Artist artist;
   public Album(Long ID, String title, Artist artist) {
      super(ID);
      this.title = title;
      this.artist = artist;
   }
   public String getTitle() {
      return title;
   }
   public void setTitle(String title) {
      this.title = title;
   }
   public Artist getArtist() {
      return artist;
   }
   public void setArtist(Artist artist) {
      this.artist = artist;
   }
```

Процесс загрузки альбома показан на рис. 12.4. Когда преобразователь `AlbumMapper` получает указание загрузить альбом, он выполняет запрос к базе данных и возвращает результирующее множество данных, соответствующее заданному альбому. Затем преобразователь выполняет запрос к результирующему множеству данных и извлекает оттуда значение внешнего ключа исполнителя. Теперь преобразователь может загрузить объект альбома найденными значениями. Если объект исполнителя уже был загружен в память, он будет извлечен из кэша; в противном случае он будет загружен из базы данных так, как это делалось для альбома.

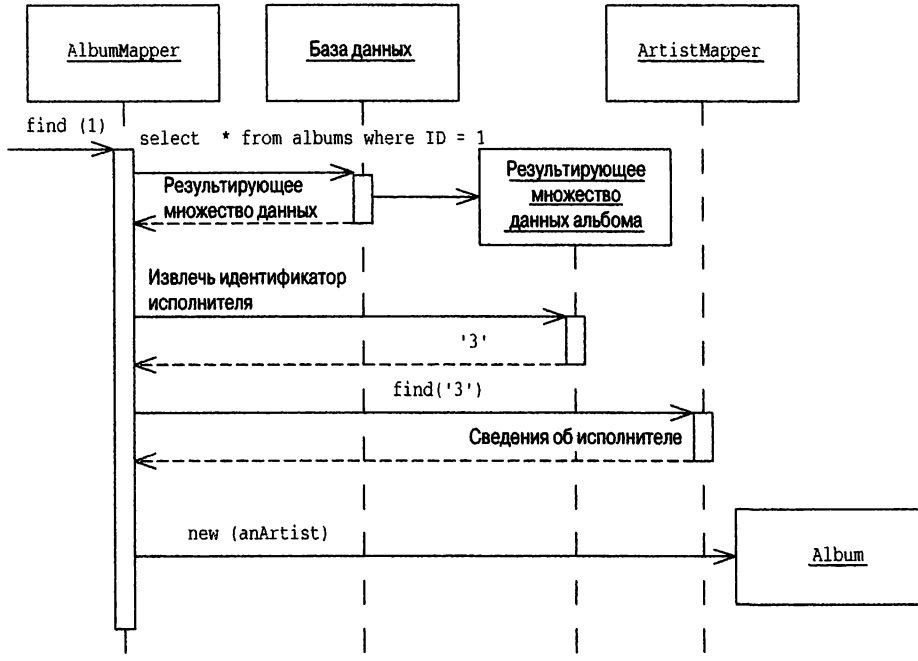


Рис. 12.4. Процесс загрузки однозначного поля

Для работы с **коллекцией объектов** метод поиска применяет абстрактное поведение.

```
class AlbumMapper...

   public Album find(Long id) {
      return (Album) abstractFind(id);
   }
   protected String findStatement() {
      return "SELECT ID, title, artistID FROM albums WHERE
↳ID = ?";
   }

class AbstractMapper...

   abstract protected String findStatement();
   protected DomainObject abstractFind(Long id) {
      DomainObject result = (DomainObject) loadedMap.get(id);
      if (result != null) return result;
      PreparedStatement stmt = null;
      ResultSet rs = null;
      try {
         stmt = DB.prepare(findStatement());
         stmt.setLong(1, id.longValue());
         rs = stmt.executeQuery();
         rs.next();
         result = load(rs);
```

```
            return result;
        } catch (SQLException e) {
            throw new ApplicationException(e);
        } finally {cleanUp(stmt, rs);}
    }
    private Map loadedMap = new HashMap();
```

Метод поиска вызывает метод загрузки, чтобы выполнить фактическую загрузку данных в объект альбома.

```
class AbstractMapper...

    protected DomainObject load(ResultSet rs)
↳throws SQLException {
        Long id = new Long(rs.getLong(1));
        if (loadedMap.containsKey(id)) return (DomainObject)
↳loadedMap.get(id);
        DomainObject result = doLoad(id, rs);
        doRegister(id, result);
        return result;
    }
    protected void doRegister(Long id, DomainObject result) {
        Assert.isFalse(loadedMap.containsKey(id));
        loadedMap.put(id, result);
    }
    abstract protected DomainObject doLoad(Long id, ResultSet rs)
↳throws SQLException;

class AlbumMapper...

    protected DomainObject doLoad(Long id, ResultSet rs)
↳throws SQLException {
        String title = rs.getString(2);
        long artistID = rs.getLong(3);
        Artist artist = MapperRegistry.artist().find(artistID);
        Album result = new Album(id, title, artist);
        return result;
    }
```

Для обновления альбома преобразователь извлекает значение внешнего ключа его исполнителя.

```
class AbstractMapper...

    abstract public void update(DomainObject arg);

class AlbumMapper...

    public void update(DomainObject arg) {
        PreparedStatement statement = null;
        try {
            statement = DB.prepare(
                "UPDATE albums SET title = ?, artistID = ?
```

```
↳WHERE id = ?");
         statement.setLong(3, arg.getID().longValue());
         Album album = (Album) arg;
         statement.setString(1, album.getTitle());
         statement.setLong(2,
↳album.getArtist().getID().longValue());
         statement.execute();
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {
         cleanUp(statement);
      }
   }
```

Пример: многотабличный поиск (Java)

Обычно область действия SQL-запроса ограничивается одной таблицей. Однако на практике это не слишком удобно, поскольку выполнение SQL-запросов связано с удаленным вызовом функций, а удаленные вызовы крайне медлительны. Изменим предыдущий пример так, чтобы получение информации об альбоме и о его исполнителе осуществлялось в рамках одного SQL-запроса. Первое изменение коснется SQL-выражения, предназначенного для проведения поиска.

```
class AlbumMapper...

   public Album find(Long id) {
      return (Album) abstractFind(id);
   }
   protected String findStatement() {
      return "SELECT a.ID, a.title, a.artistID, r.name " +
            " from albums a, artists r " +
            " WHERE ID = ? and a.artistID = r.ID";
   }
```

Кроме того, воспользуемся другим методом загрузки, который будет загружать данные и об альбоме и об исполнителе.

```
class AlbumMapper...

   protected DomainObject doLoad(Long id, ResultSet rs)
↳throws SQLException {
      String title = rs.getString(2);
      long artistID = rs.getLong(3);
      ArtistMapper artistMapper = MapperRegistry.artist();
      Artist artist;
      if (artistMapper.isLoaded(artistID))
         artist = artistMapper.find(artistID);
      else
         artist = loadArtist(artistID, rs);
      Album result = new Album(id, title, artist);
      return result;
   }
   private Artist loadArtist(long id, ResultSet rs)
```

```
↳throws SQLException {
      String name = rs.getString(4);
      Artist result = new Artist(new Long(id), name);
      MapperRegistry.artist().register(result.getID(), result);
      return result;
   }
```

При выполнении этой задачи возникает вопрос: куда поместить метод, который отображает результат выполнения SQL-запроса на объект исполнителя? С одной стороны, его лучше поместить в преобразователь исполнителя, так как именно этот класс выполняет загрузку объекта исполнителя. С другой стороны, метод загрузки тесно связан с выполнением SQL-запроса и потому должен оставаться в том классе, в котором определен SQL-запрос. В данном примере я рекомендую второе.

Пример: коллекция ссылок (C#)

Этот случай возникает тогда, когда у объекта есть поле, содержащее коллекцию. В качестве примера рассмотрим ситуацию с командами и игроками. При этом будем исходить из предположения, что игроки не могут быть **отображениями зависимых объектов** (рис. 12.5).

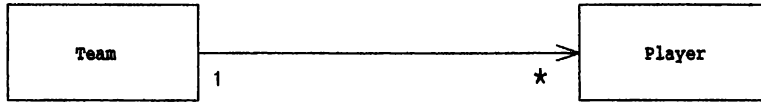


Рис. 12.5. Команда, состоящая из нескольких игроков

```
class Team...
      public String Name;
      public IList Players {
         get {return ArrayList.ReadOnly(playersData);}
         set {playersData = new ArrayList(value);}
      }
      public void AddPlayer(Player arg) {
         playersData.Add(arg);
      }
      private IList playersData = new ArrayList();
```

В базе данных этой связи будет соответствовать запись об игроке, содержащая в себе внешний ключ команды (рис. 12.6).



Рис. 12.6. Структура базы данных для команды, состоящей из нескольких игроков

```
class TeamMapper...

      public Team Find(long id) {
         return (Team) AbstractFind(id);
      }

class AbstractMapper...

      protected DomainObject AbstractFind(long id) {
         Assert.True (id != DomainObject.PLACEHOLDER_ID);
         DataRow row = FindRow(id);
         return (row == null) ? null : Load(row);
      }
      protected DataRow FindRow(long id) {
         String filter = String.Format("id = {0}", id);
         DataRow[] results = table.Select(filter);
         return (results.Length == 0) ? null : results[0];
      }
      protected DataTable table {
         get {return dsh.Data.Tables[TableName];}
      }
      public DataSetHolder dsh;
      abstract protected String TableName {get;}

class TeamMapper...

      protected override String TableName {
         get {return "Teams";}
      }
```

Класс `DataSetHolder` содержит в себе используемое множество данных (объект `DataSet`), а также объекты `DataAdapter`, необходимые для передачи обновлений объекта `DataSet` в базу данных.

```
class DataSetHolder...

      public DataSet Data = new DataSet();
      private Hashtable DataAdapters = new Hashtable();
```

В этом примере мы исходим из предположения, что объект `DataSetHolder` уже был заполнен результатами выполнения нескольких соответствующих запросов.

Метод поиска вызывает метод загрузки для выполнения фактической загрузки данных в новый объект.

```
class AbstractMapper...

      protected DomainObject Load (DataRow row) {
         long id = (int) row ["id"];
         if (identityMap[id] != null) return (DomainObject)
↳identityMap[id];
         else {
            DomainObject result = CreateDomainObject();
            result.Id = id;
```

```
            identityMap.Add(result.Id, result);
            doLoad(result,row);
            return result;
         }
      }
      abstract protected DomainObject CreateDomainObject();
      private IDictionary identityMap = new Hashtable();
      abstract protected void doLoad (DomainObject obj,
↳DataRow row);

class TeamMapper...

      protected override void doLoad (DomainObject obj,
↳DataRow row) {
         Team team = (Team) obj;
         team.Name = (String) row["name"];
         team.Players = MapperRegistry.Player.FindForTeam(
↳team.Id);
      }
```

Чтобы загрузить список игроков, я реализую в классе `PlayerMapper` специальный метод поиска.

```
class PlayerMapper...

      public IList FindForTeam(long id) {
         String filter = String.Format("teamID = {0}", id);
         DataRow[] rows = table.Select(filter);
         IList result = new ArrayList();
         foreach (DataRow row in rows) {
            result.Add(Load (row));
         }
         return result;
      }
```

Для обновления сведений о команде объект `TeamMapper` сохраняет ее собственные данные и передает управление объекту `PlayerMapper`, чтобы тот сохранил обновленные данные в таблице игроков.

```
class AbstractMapper...

      public virtual void Update (DomainObject arg) {
         Save (arg, FindRow(arg.Id));
      }
      abstract protected void Save (DomainObject arg,
↳DataRow row);

class TeamMapper...

      protected override void Save (DomainObject obj,
↳DataRow row){
         Team team = (Team) obj;
         row["name"] = team.Name;
```

```
            savePlayers(team);
        }
        private void savePlayers(Team team){
            foreach (Player p in team.Players) {
                MapperRegistry.Player.LinkTeam(p, team.Id);
            }
        }

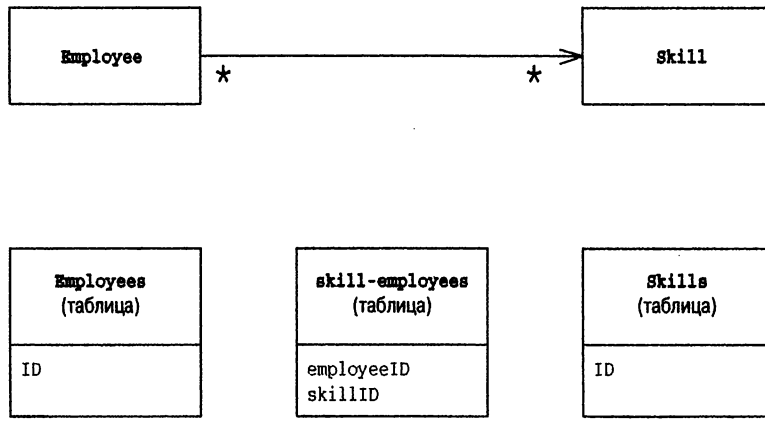
class PlayerMapper...

        public void LinkTeam (Player player, long teamID) {
            DataRow row = FindRow(player.Id);
            row["teamID"] = teamID;
        }
```

В этом примере код обновления довольно прост, потому что ссылка игрока на команду является обязательной. Даже если игрок перейдет из одной команды в другую, не придется выполнять утомительное сравнение для сортировки игроков на добавленных, удаленных и т.п. — вся информация об игроке будет обновлена при обновлении сведений о соответствующих командах. Более сложный случай с необязательными ссылками я выношу на самостоятельное рассмотрение читателей.

Отображение с помощью таблицы ассоциаций (Association Table Mapping)

Сохраняет множество ассоциаций в виде таблицы, содержащей внешние ключи таблиц, связанных ассоциациями



Объекты легко справляются с многозначными полями — для этого значение поля достаточно сделать коллекцией. Реляционные базы данных не обладают подобной возможностью, в результате чего их поля могут иметь только одно значение. Как уже отмечалось, для отображения связи типа "один ко многим" можно воспользоваться

отображением внешних ключей (Foreign Key Mapping, 258), указав в "многозначной стороне" ссылки внешний ключ "однозначной стороны". Однако данное типовое решение не может применяться для отображения связи типа "многие ко многим", поскольку однозначной стороны здесь нет. Как же тогда представить подобную связь?

Ответом на этот вопрос является классическое решение, применяемое в реляционных базах данных на протяжении уже нескольких десятилетий: создать дополнительную таблицу отношений, а затем воспользоваться типовым решением **отображение с помощью таблицы ассоциаций**, чтобы отобразить многозначное поле на таблицу отношений.

Принцип действия

Основной идеей, лежащей в основе **отображения с помощью таблицы ассоциаций**, является хранение ассоциаций в таблице отношений. Последняя содержит только значения внешних ключей двух таблиц, связанных отношением. Таким образом, каждой паре взаимосвязанных объектов соответствует одна строка таблицы отношений.

Таблице отношений не соответствует объект приложения, вследствие чего у нее нет идентификатора объекта. Первичным ключом данной таблицы является совокупность двух первичных ключей таблиц, которые связаны отношениями.

Чтобы загрузить данные из таблицы отношений, необходимо выполнить два запроса. Для большей наглядности представьте себе загрузку списка профессиональных качеств служащего. В этом случае выполнение запросов (по крайней мере концептуально) выполняется в два этапа. На первом этапе запрос находит все строки таблицы отношений `skillEmployees`, которые ссылаются на нужного служащего. На втором этапе запрос находит все профессиональные качества, соответствующие их идентификаторам в найденных строках таблицы `skillEmployees`.

Описанная схема работает хорошо, если вся необходимая информация уже загружена в память. Если же это не так, реализация подобного алгоритма может привести к огромным расходам, связанным с количеством запросов, потому что для определения каждого профессионального качества, идентификатор которого содержится в таблице отношений, придется выполнять отдельный запрос. Для сокращения расходов к таблицам ссылок и профессиональных качеств можно применить операцию соединения, что позволит извлекать все данные посредством одного запроса, хотя и увеличит сложность отображения.

Обновление данных об ассоциациях связано с массой сложностей, касающихся выполнения обновлений многозначных полей. К счастью, с таблицей отношений можно обращаться так же, как и с **отображением зависимых объектов (Dependent Mapping, 283)**, что значительно упрощает дело. На таблицу отношений не должна ссылаться никакая другая таблица, поэтому вы можете свободно добавлять и удалять отношения по мере необходимости.

Назначение

Каноническим примером использования **отображения с помощью таблицы ассоциаций** является связь типа "многие ко многим", поскольку альтернативы данному решению просто нет.

Отображение с помощью таблицы ассоциаций может быть использовано и для других типов связей. Разумеется, поскольку данное типовое решение является более сложным, чем **отображение внешних ключей**, а также требует дополнительной операции соединения,

его выбор не всегда может быть удачным. Впрочем, оно незаменимо в ситуациях, когда у разработчика нет полного контроля над схемой базы данных. Иногда вам может понадобиться связать две существующие таблицы, к которым нельзя добавить новые столбцы. В этом случае вы можете создать новую таблицу и воспользоваться **отображением с помощью таблицы ассоциаций**. Другой возможный вариант использования данного типового решения состоит в том, что существующая схема базы данных включает в себя таблицу отношений, даже если эта таблица на самом деле не нужна. В этом случае для представления отношений легче воспользоваться **отображением с помощью таблицы ассоциаций**, чем пытаться упростить схему базы данных.

Иногда таблицу отношений проектируют таким образом, чтобы она содержала в себе некоторые сведения об отношении. В качестве примера можно привести таблицу отношений "служащие–компании", которая помимо внешних ключей будет содержать информацию о должности, занимаемой служащим в данной компании. В этом случае таблица "служащие–компании" будет соответствовать полноценному объекту домена.

Пример: служащие и профессиональные качества (C#)

Рассмотрим простой пример для модели, о которой шла речь в начале раздела. Итак, у нас есть класс служащих, содержащий коллекцию профессиональных качеств, каждым из которых может обладать более чем один служащий.

```
class Employee...

      public IList Skills {
         get {return ArrayList.ReadOnly(skillsData);}
         set {skillsData = new ArrayList(value);}
      }
      public void AddSkill (Skill arg) {
         skillsData.Add(arg);
      }
      public void RemoveSkill (Skill arg) {
         skillsData.Remove(arg);
      }
      private IList skillsData = new ArrayList();
```

Чтобы загрузить объект `Employee`, нужно извлечь из базы данных список профессиональных качеств соответствующего служащего. Для этого воспользуемся преобразователем `EmployeeMapper`. У каждого объекта `EmployeeMapper` есть метод поиска, создающий объект `Employee`. Общее поведение преобразователей вынесено в наследуемый ими абстрактный класс `AbstractMapper`.

```
class EmployeeMapper...

      public Employee Find(long id) {
         return (Employee) AbstractFind(id);
      }

class AbstractMapper...

      protected DomainObject AbstractFind(long id) {
```

```
         Assert.True (id != DomainObject.PLACEHOLDER_ID);
         DataRow row = FindRow(id);
         return (row == null) ? null : Load(row);
      }
      protected DataRow FindRow(long id) {
         String filter = String.Format("id = {0}", id);
         DataRow[] results = table.Select(filter);
         return (results.Length == 0) ? null : results[0];
      }
      protected DataTable table {
         get {return dsh.Data.Tables[TableName];}
      }
      public DataSetHolder dsh;
      abstract protected String TableName {get;}

class EmployeeMapper...

      protected override String TableName {
         get {return "Employees";}
      }
```

Объект `DataSetHolder` содержит в себе объект ADO.NET `DataSet`, а также объекты `DataAdapter`, необходимые для записи содержимого объекта `DataSet` в базу данных.

```
class DataSetHolder...

      public DataSet Data = new DataSet();
      private Hashtable DataAdapters = new Hashtable();
```

Чтобы еще более упростить задачу, будем исходить из предположения, что объект `DataSet` уже был заполнен необходимым содержимым.

Метод поиска вызывает методы `Load` и `doLoad` для загрузки сведений о служащем.

```
class AbstractMapper...

      protected DomainObject Load (DataRow row) {
         long id = (int) row ["id"];
         if (identityMap[id] != null) return (DomainObject)
↳identityMap[id];
         else {
            DomainObject result = CreateDomainObject();
            result.Id = id;
            identityMap.Add(result.Id, result);
            doLoad(result,row);
            return result;
         }
      }
      abstract protected DomainObject CreateDomainObject();
      private IDictionary identityMap = new Hashtable();
      abstract protected void doLoad (DomainObject obj,
↳DataRow row);

class EmployeeMapper...
```

```
      protected override void doLoad (DomainObject obj,
↳DataRow row) {
         Employee emp = (Employee) obj;
         emp.Name = (String) row["name"];
         loadSkills(emp);
      }
```

Загрузка сведений о профессиональных качествах служащего достаточно сложна, поэтому она была вынесена в отдельный метод.

```
class EmployeeMapper...

      private IList loadSkills (Employee emp) {
         DataRow[] rows = skillLinkRows(emp);
         IList result = new ArrayList();
         foreach (DataRow row in rows) {
            long skillID = (int)row["skillID"];
            emp.AddSkill(MapperRegistry.Skill.Find(skillID));
         }
         return result;
      }
      private DataRow[] skillLinkRows(Employee emp) {
         String filter = String.Format("employeeID = {0}",
↳emp.Id);
         return skillLinkTable.Select(filter);
      }
      private DataTable skillLinkTable {
         get {return dsh.Data.Tables["skillEmployees"];}
      }
```

Для обработки изменений в сведениях о профессиональных качествах будет применяться метод обновления, реализованный в классе `AbstractMapper`.

```
class AbstractMapper...

      public virtual void Update (DomainObject arg) {
         Save (arg, FindRow(arg.Id));
      }
      abstract protected void Save (DomainObject arg,
↳DataRow row);
```

Метод обновления вызывает метод сохранения, реализованный в производном классе.

```
class EmployeeMapper...

      protected override void Save (DomainObject obj,
↳DataRow row) {
         Employee emp = (Employee) obj;
         row["name"] = emp.Name;
         saveSkills(emp);
      }
```

И здесь сохранение профессиональных качеств было вынесено в отдельный метод.

```
class EmployeeMapper...

      private void saveSkills(Employee emp) {
         deleteSkills(emp);
         foreach (Skill s in emp.Skills) {
            DataRow row = skillLinkTable.NewRow();
            row["employeeID"] = emp.Id;
            row["skillID"] = s.Id;
            skillLinkTable.Rows.Add(row);
         }
      }
      private void deleteSkills(Employee emp) {
         DataRow[] skillRows = skillLinkRows(emp);
         foreach (DataRow r in skillRows) r.Delete();
      }
```

Данная логика очень проста: она удаляет все существующие строки таблицы отношений и создает новые. Это избавляет от необходимости разбираться в том, какие профессиональные качества были добавлены, а какие — удалены.

Пример: использование SQL для непосредственного обращения к базе данных (Java)

Библиотека ADO.NET имеет одно замечательное преимущество: позволяет обсуждать основы объектно-реляционного отображения, не вдаваясь в многочисленные детали минимизации запросов. Все другие схемы реляционного отображения более тесно связаны с SQL и не могут рассматриваться без учета его особенностей.

Вообще говоря, при непосредственном доступе к базе данных очень важно минимизировать количество запросов. В первом варианте решения нашей задачи извлечение данных о служащем и его профессиональных качествах выполняется посредством двух запросов. Это очень простой, но далеко не оптимальный метод, поэтому приверженцам минимизации запросов придется немного потерпеть.

Вот как выглядит описание наших таблиц:

```
create table employees (ID int primary key, firstname varchar,
↳lastname varchar)
create table skills (ID int primary key, name varchar)
create table employeeSkills (employeeID int, skillID int,
↳primary key (employeeID, skillID))
```

Для загрузки одного объекта `Employee` я воспользуюсь тем же приемом, что и в предыдущем примере. Объект `EmployeeMapper` содержит простой метод-оболочку для абстрактного метода поиска, определенного в **супертипе слоя (Layer Supertype, 491)**.

```
class EmployeeMapper...

   public Employee find(long key) {
      return find (new Long (key));
   }
```

```
   public Employee find (Long key) {
      return (Employee) abstractFind(key);
   }
   protected String findStatement() {
      return
         "SELECT " + COLUMN_LIST +
         "    FROM employees" +
         "    WHERE ID = ?";
   }
   public static final String COLUMN_LIST = " ID, lastname,
↳firstname ";

class AbstractMapper...

   protected DomainObject abstractFind(Long id) {
      DomainObject result = (DomainObject) loadedMap.get(id);
      if (result != null) return result;
      PreparedStatement stmt = null;
      ResultSet rs = null;
      try {
         stmt = DB.prepare(findStatement());
         stmt.setLong(1, id.longValue());
         rs = stmt.executeQuery();
         rs.next();
         result = load(rs);
         return result;
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {DB.cleanUp(stmt, rs);
      }
   }
   abstract protected String findStatement();
   protected Map loadedMap = new HashMap();
```

После выполнения всех необходимых действий методы поиска вызывают методы загрузки. Абстрактный метод загрузки обрабатывает загрузку идентификатора служащего, а все основные данные загружаются посредством метода, определенного в классе `EmployeeMapper`.

```
class AbstractMapper...

   protected DomainObject load(ResultSet rs)
↳throws SQLException {
      Long id = new Long(rs.getLong(1));
      return load(id, rs);
   }
   public DomainObject load(Long id, ResultSet rs)
↳throws SQLException {
      if (hasLoaded(id)) return (DomainObject)
↳loadedMap.get(id);
      DomainObject result = doLoad(id, rs);
      loadedMap.put(id, result);
      return result;
```

```
   }
   abstract protected DomainObject doLoad(Long id, ResultSet rs)
↳throws SQLException;

class EmployeeMapper...

   protected DomainObject doLoad(Long id, ResultSet rs)
↳throws SQLException {
      Employee result = new Employee(id);
      result.setFirstName(rs.getString("firstname"));
      result.setLastName(rs.getString("lastname"));
      result.setSkills(loadSkills(id));
      return result;
   }
```

Чтобы загрузить профессиональные качества служащего, объекту `EmployeeMapper` необходимо выполнить отдельный запрос. Впрочем, все профессиональные качества могут быть легко загружены посредством одного запроса. Для этого объект `EmployeeMapper` вызывает объект `SkillMapper`, выполняющий загрузку сведений о конкретном профессиональном качестве.

```
class EmployeeMapper...

   protected List loadSkills(Long employeeID) {
         PreparedStatement stmt = null;
         ResultSet rs = null;
         try {
            List result = new ArrayList();
            stmt = DB.prepare(findSkillsStatement);
            stmt.setObject(1, employeeID);
            rs = stmt.executeQuery();
            while (rs.next()) {
               Long skillId = new Long (rs.getLong(1));
               result.add((Skill)
↳MapperRegistry.skill().loadRow(skillId, rs));
            }
            return result;
         } catch (SQLException e) {
            throw new ApplicationException(e);
         } finally {DB.cleanUp(stmt, rs);
         }
   }
   private static final String findSkillsStatement =
      "SELECT skill.ID, " + SkillMapper.COLUMN_LIST +
      "    FROM skills skill, employeeSkills es " +
      "    WHERE es.employeeID = ? AND skill.ID = es.skillID";

class SkillMapper...

   public static final String COLUMN_LIST = " skill.name
↳skillName ";

class AbstractMapper...
```

```
   protected DomainObject loadRow (Long id, ResultSet rs)
↳throws SQLException {
      return load (id, rs);
   }

class SkillMapper...

   protected DomainObject doLoad(Long id, ResultSet rs)
↳throws SQLException {
      Skill result = new Skill (id);
      result.setName(rs.getString("skillName"));
      return result;
   }
```

Класс `AbstractMapper` может пригодиться и для поиска сведений о служащих.

```
class EmployeeMapper...

   public List findAll() {
      return findAll(findAllStatement);
   }
   private static final String findAllStatement =
      "SELECT " + COLUMN_LIST +
      "    FROM employees employee" +
      "    ORDER BY employee.lastname";

class AbstractMapper...

   protected List findAll(String sql) {
      PreparedStatement stmt = null;
      ResultSet rs = null;
      try {
         List result = new ArrayList();
         stmt = DB.prepare(sql);
         rs = stmt.executeQuery();
         while (rs.next())
            result.add(load(rs));
         return result;
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {DB.cleanUp(stmt, rs);
      }
   }
```

Описанный способ извлечения данных довольно надежен и очень прост в реализации. Тем не менее он далеко не идеален, потому что для загрузки сведений о служащем необходимо выполнить два SQL-запроса. Хотя основные данные обо всех служащих можно загрузить с помощью одного и того же запроса, нам понадобится еще по одному запросу на каждого служащего для того, чтобы загрузить его профессиональные качества. Таким образом, загрузка сведений о 100 служащих потребует выполнения 101 запроса.

Пример: загрузка сведений о нескольких служащих посредством одного запроса (Java)

В случае необходимости сведения обо всех служащих, включая и их профессиональные качества, можно загрузить с помощью единственного запроса. Это хороший пример оптимизации многотабличных запросов, но, к сожалению, он довольно сложен. В связи с этим могу дать совет: применяйте этот способ только в случае необходимости и не рассматривайте его как панацею на все случаи жизни. Гораздо лучше уделить внимание ускорению действительно медленных запросов, чем пытаться ускорить все подряд.

Сначала рассмотрим более простой случай, когда все профессиональные качества служащего будут извлекаться в том же запросе, что и основные данные. Для этого я воспользуюсь довольно сложным SQL-выражением, которое будет выполнять соединение всех трех таблиц.

```
class EmployeeMapper...

   protected String findStatement() {
      return
         "SELECT " + COLUMN_LIST +
         "    FROM employees employee, skills skill,
↳employeeSkills es" +
         "    WHERE employee.ID = es.employeeID AND skill.ID =
↳es.skillID AND employee.ID = ?";
   }
   public static final String COLUMN_LIST =
         " employee.ID, employee.lastname,
↳employee.firstname, " +
         "    es.skillID, es.employeeID, skill.ID skillID, " +
         SkillMapper.COLUMN_LIST;
```

Методы суперкласса `abstractFind` и `load` аналогичны таковым в предыдущем примере, поэтому здесь они не приводятся. В свою очередь, загрузка данных объектом `EmployeeMapper` выполняется по-другому (а именно с учетом множественных строк данных).

```
class EmployeeMapper...

   protected DomainObject doLoad(Long id, ResultSet rs)
↳throws SQLException {
      Employee result = (Employee) loadRow(id, rs);
      loadSkillData(result, rs);
      while (rs.next()){
         Assert.isTrue(rowIsForSameEmployee(id, rs));
         loadSkillData(result, rs);
      }
      return result;
   }
   protected DomainObject loadRow(Long id, ResultSet rs)
↳throws SQLException {
      Employee result = new Employee(id);
      result.setFirstName(rs.getString("firstname"));
      result.setLastName(rs.getString("lastname"));
```

```
      return result;
   }
   private boolean rowIsForSameEmployee(Long id, ResultSet rs)
↳throws SQLException {
      return id.equals(new Long(rs.getLong(1)));
   }
   private void loadSkillData(Employee person, ResultSet rs)
↳throws SQLException {
      Long skillID = new Long(rs.getLong("skillID"));
      person.addSkill ((Skill)MapperRegistry.skill().loadRow(
↳skillID, rs));
   }
```

В этом примере метод загрузки объекта `EmployeeMapper` фактически просматривает всю оставшуюся часть результирующего множества данных, чтобы загрузить все данные.

Как видите, загрузить сведения об одном служащем довольно просто. Тем не менее настоящие преимущества многотабличных запросов проявляются только при загрузке данных о нескольких служащих. Реализовать считывание данных напрямую порой оказывается крайне сложно, особенно если не требуется, чтобы результирующее множество данных было сгруппировано по служащим. Поэтому воспользуемся вспомогательным классом, который будет просматривать саму таблицу отношений, загружая сведения о служащих и профессиональных качествах по мере их обнаружения.

Вначале идет описание SQL-выражения и вызов специального класса загрузчика.

```
class EmployeeMapper...

   public List findAll() {
      return findAll(findAllStatement);
   }
   private static final String findAllStatement =
         "SELECT " + COLUMN_LIST +
         "    FROM employees employee, skills skill,
↳employeeSkills es" +
         "    WHERE employee.ID = es.employeeID AND skill.ID =
↳es.skillID" +
         "    ORDER BY employee.lastname";
   protected List findAll(String sql) {
      AssociationTableLoader loader = new
↳AssociationTableLoader(this, new SkillAdder());
      return loader.run(findAllStatement);
   }

class AssociationTableLoader...

   private AbstractMapper sourceMapper;
   private Adder targetAdder;
   public AssociationTableLoader(AbstractMapper primaryMapper,
↳Adder targetAdder) {
      this.sourceMapper = primaryMapper;
      this.targetAdder = targetAdder;
   }
```

Не беспокойтесь насчет объекта `SkillAdder` — о нем речь идет немного позднее. Пока же обратите внимание на создание экземпляра загрузчика. Как видите, создаваемый экземпляр получает ссылку на преобразователь и затем выполняет загрузку данных в соответствии с необходимым запросом. Это типичная структура объекта метода. *Объект метода (method object)* [6] представляет собой способ превращения сложного метода в отдельный объект. Преимущество данного способа состоит в том, что он позволяет помещать необходимые значения в поля объекта, вместо того чтобы передавать их в качестве параметров. Обычно, чтобы воспользоваться объектом метода, его создают, запускают нужный метод и затем спокойно уничтожают.

Выполнение загрузки происходит в три этапа.

```
class AssociationTableLoader...

   protected List run(String sql) {
      loadData(sql);
      addAllNewObjectsToIdentityMap();
      return formResult();
   }
```

Метод `loadData` формирует вызов SQL-запроса, выполняет его и перебирает результирующее множество данных. Поскольку `AssociationTableMapper` — это объект метода, я поместил результирующее множество данных в поле, поэтому его не нужно передавать в качестве параметра.

```
class AssociationTableLoader...

   private ResultSet rs = null;
   private void loadData(String sql) {
      PreparedStatement stmt = null;
      try {
         stmt = DB.prepare(sql);
         rs = stmt.executeQuery();
         while (rs.next())
            loadRow();
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {DB.cleanUp(stmt, rs);
      }
   }
```

Метод `loadRow` загружает данные из одной строки результирующего множества данных. Он довольно сложен.

```
class AssociationTableLoader...

   private List resultIds = new ArrayList();
   private Map inProgress = new HashMap();
   private void loadRow() throws SQLException {
      Long ID = new Long(rs.getLong(1));
      if (!resultIds.contains(ID)) resultIds.add(ID);
      if (!sourceMapper.hasLoaded(ID)) {
```

```
            if (!inProgress.keySet().contains(ID))
               inProgress.put(ID, sourceMapper.loadRow(ID, rs));
            targetAdder.add((DomainObject) inProgress.get(ID), rs);
         }
      }

class AbstractMapper...

      boolean hasLoaded(Long id) {
         return loadedMap.containsKey(id);
      }
```

Загрузчик сохраняет порядок строк результирующего множества данных, поэтому выходной список служащих будет отсортирован так же, как и ранее. Таким образом, я буду вести список идентификаторов служащих в той последовательности, в которой они ко мне поступают. Переходя к очередному идентификатору, я проверяю, был ли соответствующий объект полностью загружен в преобразователь (как правило, в соответствии с результатами предыдущего запроса). Если это не так, я загружаю те данные, которые у меня есть, и сохраняю идентификатор в списке идентификаторов служащих, находящихся в состоянии загрузки. Подобный список нужен на тот случай, если сведения о профессиональных качествах служащего будут состоять из нескольких строк, не идущих последовательно одна за другой.

Самое сложное в написании всего этого кода — гарантировать, что я смогу добавить профессиональное качество, которое загружается в список профессиональных качеств служащих, и одновременно сохранить загрузчик универсальным, чтобы он не зависел от служащих или их профессиональных качеств. Для этого мне пришлось изрядно покопаться в своих "закромах", чтобы извлечь оттуда внутренний интерфейс по имени `Adder`.

```
class AssociationTableLoader...

      public static interface Adder {
         void add(DomainObject host, ResultSet rs)
↳throws SQLException;
      }
```

Объект, вызывающий метод поиска, должен реализовать интерфейс `Adder`, чтобы привязать его к конкретным потребностям объекта служащего или профессионального качества.

```
class EmployeeMapper...

      private static class SkillAdder implements
↳AssociationTableLoader.Adder {
         public void add(DomainObject host, ResultSet rs)
↳throws SQLException {
            Employee emp = (Employee) host;
            Long skillId = new Long (rs.getLong("skillId"));
            emp.addSkill((Skill) MapperRegistry.skill().loadRow(
↳skillId, rs));
         }
      }
```

Подобная реализация более естественна для языков, имеющих замыкания или указатели функций, но мы сумели заменить их классом и интерфейсом. (В данном случае они не обязательно должны быть внутренними, однако это помогает ограничить их область действия, которая на самом деле довольно узка.)

Возможно, вы заметили, что методы `load` и `loadRow` определены в суперклассе и что реализацией метода `loadRow` является вызов метода `load`. Я сделал это для того, чтобы исключить смещение результирующего множества данных. Другими словами, метод `load` выполняет все необходимое, чтобы загрузить объект, а метод `loadRow` следит за тем, чтобы загрузка данных из строки не приводила к смещению курсора базы данных. В большинстве случаев эти два метода делают одно и то же, однако в нашей реализации объекта `EmployeeMapper` они различны.

Теперь все нужные сведения находятся в результирующем множестве данных. У меня есть две коллекции: список идентификаторов всех служащих, которые были в результирующем множестве данных, в порядке их появления, и список новых объектов, которые еще не были добавлены в **коллекцию объектов (Identity Map, 216)** преобразователя `EmployeeMapper`.

Следующий этап — поместить все новые объекты в **коллекцию объектов**.

```
class AssociationTableLoader...

   private void addAllNewObjectsToIdentityMap() {
      for (Iterator it = inProgress.values().iterator();
↳it.hasNext();)
         sourceMapper.putAsLoaded((DomainObject)it.next());
   }

class AbstractMapper...

   void putAsLoaded (DomainObject obj) {
      loadedMap.put (obj.getID(), obj);
   }
```

И наконец последний шаг — собрать окончательный список служащих и профессиональных качеств путем просмотра идентификаторов, загруженных преобразователем.

```
class AssociationTableLoader...

   private List formResult() {
      List result = new ArrayList();
      for (Iterator it = resultIds.iterator(); it.hasNext();) {
         Long id = (Long)it.next();
         result.add(sourceMapper.lookUp(id));
      }
      return result;
   }

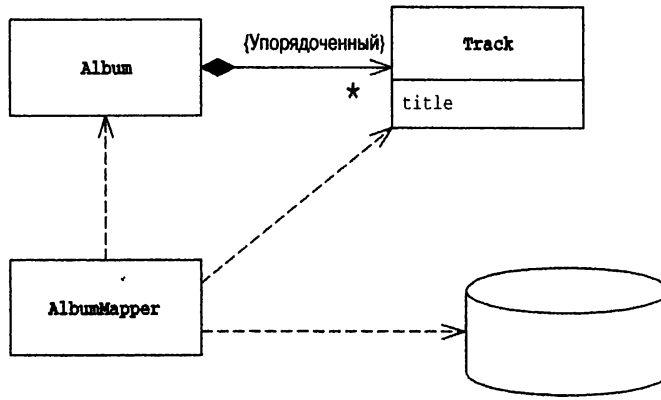
class AbstractMapper...

   protected DomainObject lookUp (Long id) {
      return (DomainObject) loadedMap.get(id);
   }
```

Данный код более сложен, чем обычный код загрузки, однако он позволяет максимально сократить количество запросов. Из-за своей сложности подобный прием должен использоваться избирательно — в основном тогда, когда взаимодействие с базой данных осуществляется не слишком быстро. Тем не менее он является хорошим примером того, как **преобразователь данных (Data Mapper, 187)** может обеспечить эффективное выполнение запросов, не уведомляя слой домена о сложности их реализации.

Отображение зависимых объектов (Dependent Mapping)

Передает некоторому классу полномочия по выполнению отображения для дочернего класса



Некоторые объекты в силу своей семантики применяются в контексте других объектов. Например, композиции альбома могут загружаться или сохраняться тогда же, когда и сам альбом. Если на композиции альбома не ссылается никакая другая таблица базы данных, процедуру отображения можно значительно упростить, передав полномочия по выполнению отображения для композиций объекту, выполняющему отображение для альбома. Подобная схема получила название *отображения зависимых объектов (dependent mapping)*.

Принцип действия

Главная идея, лежащая в основе типового решения **отображение зависимых объектов**, состоит в том, что один класс (*зависимый объект*) передает другому классу (*владельцу*) все свои полномочия по взаимодействию с базой данных. При этом у каждого зависимого объекта должен быть один и только один владелец.

Данный принцип проявляет себя в терминах классов, выполняющих отображение. В случае с **активной записью (Active Record, 182)** и **шлюзом записи данных (Row Data Gateway, 175)** зависимый класс не будет содержать никакого кода, касающегося выполнения отображения на базу данных; этот код будет реализован в классе-владельце. В случае с **преобразователем данных (Data Mapper, 187)** у зависимого класса не будет своего

преобразователя; все необходимое отображение будет выполняться преобразователем класса-владельца. И наконец, в случае **шлюза таблицы данных (Table Data Gateway, 167)** зависимого класса не будет вообще; всю обработку зависимых данных будет осуществлять класс-владелец.

В большинстве случаев вместе с объектом-владельцем загружаются и его зависимые объекты. Если загрузка зависимых объектов связана с большими расходами, а сами они используются нечасто, можно применить **загрузку по требованию (Lazy Load, 220)**, чтобы извлекать зависимые объекты только тогда, когда они понадобятся.

Важным свойством зависимого объекта является то, что он не имеет **поля идентификации (Identity Field, 237)** и, следовательно, не заносится в **коллекцию объектов (Identity Map, 216)**. Таким образом, зависимый объект нельзя загрузить посредством метода, выполняющего поиск по идентификатору. Впрочем, для зависимого объекта вообще не предусмотрено отдельных методов поиска, так как весь поиск выполняется объектом-владельцем.

Зависимый объект может быть владельцем другого зависимого объекта. В этом случае владелец первого зависимого объекта отвечает и за второй зависимый объект. Вообще говоря, у вас может быть целая иерархия зависимых объектов, контролируемых единственным первичным владельцем.

Первичные ключи зависимых объектов в базе данных удобно делать составными, чтобы они включали в себя первичный ключ владельца. Других внешних ключей в таблице, соответствующей зависимому объекту, быть не должно (за исключением разве что внешних ключей объектов, которые имеют того же владельца). Как следствие этого, ни один объект приложения, кроме владельца или его зависимых объектов, не может ссылаться на данный зависимый объект. С формальной точки зрения это правило можно немного ослабить, реализовав ссылку, которая не сохраняется в базе данных, однако наличие непостоянной ссылки само по себе способно вызвать массу недоразумений.

В языке UML для представления отношений между владельцем и зависимыми объектами применяют операцию композиции.

Поскольку записью и сохранением зависимых объектов в базе данных занимается их владелец и на эти объекты нет внешних ссылок, их обновление может быть реализовано путем удаления и вставки. Другими словами, чтобы обновить коллекцию зависимых объектов, вы можете без всяких опасений удалить все строки, которые ссылаются на объект-владелец, и затем заново вставить все строки, соответствующие зависимым объектам. Это избавит вас от необходимости анализировать, что было удалено, а что добавлено в коллекцию зависимых объектов владельца.

Во многих отношениях зависимые объекты напоминают **объекты-значения (Value Object, 500)**, хотя для реализации первых не нужны такие сложные механизмы, которые применяются для превращения сущности в **объект-значение** (например, переопределение метода проверки на равенство). Главное же различие между теми и другими состоит в том, что в контексте объектной модели зависимые объекты ничем не отличаются от обычных. "Зависимая" природа таких объектов проявляется только при отображении на базу данных.

Использование **отображения зависимых объектов** затрудняет отслеживание изменений объекта-владельца. При каждом изменении зависимого объекта владелец должен быть помечен как измененный, чтобы записать внесенные изменения в базу данных. Для упрощения этой процедуры зависимый объект можно сделать неизменяемым, т.е. изменение

будет осуществляться путем удаления старого объекта и добавления нового. Это несколько усложнит работу с объектной моделью, однако значительно упростит отображение на базу данных. Хотя при использовании **преобразователя данных** отображения на объекты и на базу данных должны выполняться независимо друг от друга, на практике время от времени случаются компромиссы.

Назначение

Отображение зависимых объектов используется тогда, когда в приложении есть объект, на который ссылается только какой-нибудь другой объект (например, когда у объекта есть коллекция зависимых от него объектов). Данное типовое решение прекрасно подходит для ситуаций, при которых у объекта-владельца есть коллекция ссылок на зависимые объекты, однако нет обратных указателей. Если в приложении есть множество объектов, которым не нужны собственные идентификаторы, использование **отображения зависимых объектов** значительно облегчает управление или в базе данных.

Отображение зависимых объектов может применяться только при соблюдении следующих условий:

- у каждого зависимого объекта должен быть строго один владелец;
- на зависимый объект может ссылаться только его владелец.

Существует направление объектно-ориентированного проектирования, которое использует понятие объектов-сущностей и зависимых объектов при разработке **модели предметной области (Domain Model, 140)**. Я же предпочитаю рассматривать **отображение зависимых объектов** как один из приемов, направленных на упрощение объектно-реляционного отображения, а не как фундаментальный принцип проектирования. В частности, я избегаю создавать большие графы зависимых объектов. Как известно, на зависимые объекты нельзя ссылаться извне, поэтому наличие у корневого объекта множества зависимых требует разработки весьма сложных схем поиска по графу объектов.

Не рекомендуется использовать **отображение зависимых объектов** вместе с **единицей работы (Unit of Work, 205)**. Стратегия удаления и повторной вставки окажется бесполезной, если выполнение обновлений будет отслеживаться **единицей работы**. Кроме того, **единица работы** не контролирует зависимые объекты. Майк Реттиг (Mike Rettig) рассказывал мне о приложении, в котором **единица работы** должна была отследить строки, вставляемые в целях тестирования, и затем удалить их все по окончании работы. Поскольку упомянутая **единица работы** не отслеживала вставку зависимых объектов, в базе данных появились "осиротевшие" строки, которые испортили всю картину тестирования.

Пример: альбомы и композиции (Java)

В модели предметной области, показанной на рис. 12.7, объект-альбом содержит коллекцию композиций. В этом простом (и, надо сказать, совершенно бесполезном) приложеньице на композиции не будет ссылаться ничто, кроме альбома, поэтому оно является очевидным кандидатом на применение **отображения зависимых объектов**. (На самом деле, глядя на эту модель, можно понять, что она была сконструирована именно под данное типовое решение.)

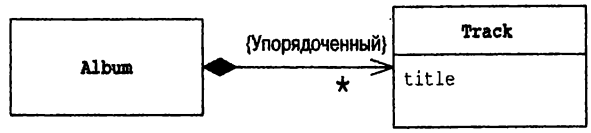


Рис. 12.7. К альбому, ссылающемуся на композиции, нельзя не применить **отображение зависимых объектов**

Композиция имеет только название. Я определил ее как неизменяемый класс.

```
class Track...

   private final String title;
   public Track(String title) {
      this.title = title;
   }
   public String getTitle() {
      return title;
   }
```

Список композиций хранится в классе альбома.

```
class Album...

   private List tracks = new ArrayList();
   public void addTrack(Track arg) {
      tracks.add(arg);
   }
   public void removeTrack(Track arg) {
      tracks.remove(arg);
   }
   public void removeTrack(int i) {
      tracks.remove(i);
   }
   public Track[] getTracks() {
      return (Track[]) tracks.toArray(new Track[tracks.size()]);
   }
```

Класс `AlbumMapper` выполняет все SQL-операции для работы с композициями и поэтому содержит определение SQL-выражений, предназначенных для доступа к таблице композиций.

```
class AlbumMapper...

   protected String findStatement() {
      return
         "SELECT ID, a.title, t.title as trackTitle" +
         "   FROM albums a, tracks t" +
         "    WHERE a.ID = ? AND t.albumID = a.ID" +
         "    ORDER BY t.seq";
   }
```

При каждой загрузке альбома в него загружается список композиций.

```
class AlbumMapper...

   protected DomainObject doLoad(Long id, ResultSet rs)
↳throws SQLException {
      String title = rs.getString(2);
      Album result = new Album(id, title);
      loadTracks(result, rs);
      return result;
   }
   public void loadTracks(Album arg, ResultSet rs)
↳throws SQLException {
      arg.addTrack(newTrack(rs));
      while (rs.next()) {
         arg.addTrack(newTrack(rs));
      }
   }
   private Track newTrack(ResultSet rs) throws SQLException {
      String title = rs.getString(3);
      Track newTrack = new Track (title);
      return newTrack;
   }
```

Для упрощения задачи я вынес загрузку композиций в отдельный запрос. Если же вы беспокоитесь о производительности, попробуйте загрузить список композиций вместе с альбомом, как было показано в примере на стр. 265.

При обновлении альбома список его композиций удаляется и затем снова вставляется.

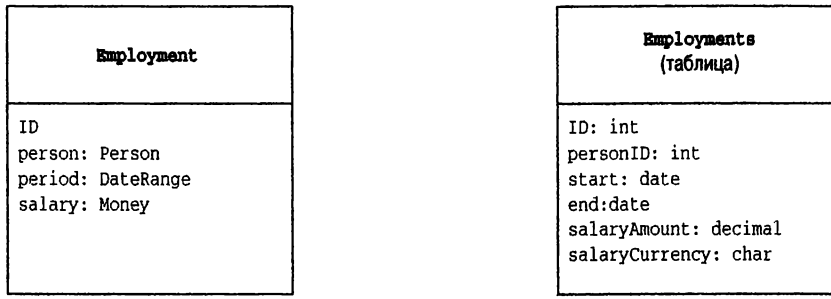
```
class AlbumMapper...

   public void update(DomainObject arg) {
      PreparedStatement updateStatement = null;
      try {
         updateStatement = DB.prepare("UPDATE albums
↳SET title = ? WHERE id = ?");
         updateStatement.setLong(2, arg.getID().longValue());
         Album album = (Album) arg;
         updateStatement.setString(1, album.getTitle());
         updateStatement.execute();
         updateTracks(album);
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {DB.cleanUp(updateStatement);
      }
   }
   public void updateTracks(Album arg) throws SQLException {
      PreparedStatement deleteTracksStatement = null;
      try {
         deleteTracksStatement = DB.prepare("DELETE from tracks
↳WHERE albumID = ?");
         deleteTracksStatement.setLong(1,
↳arg.getID().longValue());
         deleteTracksStatement.execute();
         for (int i = 0; i < arg.getTracks().length; i++) {
```

```
            Track track = arg.getTracks()[i];
            insertTrack(track, i + 1, arg);
         }
      } finally {DB.cleanUp(deleteTracksStatement);
      }
   }
   public void insertTrack(Track track, int seq, Album album)
↳throws SQLException {
      PreparedStatement insertTracksStatement = null;
      try {
         insertTracksStatement =
         DB.prepare("INSERT INTO tracks (seq, albumID, title)
↳VALUES (?, ?, ?)");
         insertTracksStatement.setInt(1, seq);
         insertTracksStatement.setLong(2,
↳album.getID().longValue());
         insertTracksStatement.setString(3, track.getTitle());
         insertTracksStatement.execute();
      } finally {DB.cleanUp(insertTracksStatement);
      }
   }
```

Внедренное значение (Embedded Value)

Отображает объект на несколько полей таблицы, соответствующей другому объекту



В приложениях часто встречаются небольшие объекты, которые имеют смысл в объектной модели, однако совершенно бесполезны в качестве таблиц базы данных. Хорошим примером таких объектов являются денежные значения в определенной валюте или диапазоны дат. Хотя общепринятая практика предусматривает сохранение объектов в виде таблиц, ни один здравомыслящий разработчик не станет создавать таблицу денежных значений.

Типовое решение **внедренное значение** отображает значения полей объекта на поля записи его владельца. В качестве примера можно привести объект-должность, который ссылается на такие объекты, как диапазон дат и денежное значение. В результирующей

таблице поля этих объектов (диапазон дат и денежное значение) будут отображаться на поля таблицы должностей, а не на записи собственных таблиц.

Принцип действия

На самом деле все очень просто. Когда объект-владелец (должность) загружается в базу данных или сохраняется в ней, вместе с ним загружаются или сохраняются и зависимые объекты (диапазон дат и денежное значение). Зависимые классы не имеют собственных методов загрузки и сохранения, поскольку все эти операции выполняются их владельцем. Для большей наглядности **внедренное значение** можно рассматривать как частный случай **отображения зависимых объектов (Dependent Mapping, 283)**, где значение поля таблицы является отдельным зависимым объектом.

Назначение

Понять принцип действия **внедренного значения** легко, а вот определить область его применения гораздо труднее.

Наиболее очевидными кандидатами для применения **внедренного значения** являются простые и понятные **объекты-значения (Value Object, 500)** наподобие денежных значений и диапазонов дат. Поскольку у **объектов-значений** нет идентификаторов, их можно свободно создавать и уничтожать, не беспокоясь о таких вещах, как синхронизация с помощью **коллекций объектов (Identity Map, 216)**. Вообще говоря, все **объекты-значения** должны храниться в виде **внедренных значений**, поскольку им никогда не понадобятся отдельные таблицы.

Главные сомнения возникают относительно того, стоит ли использовать **внедренное значение** для хранения полноценных объектов, связанных ссылками, например заказа и транспортной накладной. Здесь важно решить, имеет ли транспортная накладная какой-либо смысл вне контекста соответствующего заказа, в частности в плане загрузки и сохранения. Если сведения о транспортировке загружаются в память только вместе с заказом, стоит подумать о том, чтобы хранить их в одной таблице. Еще один спорный момент связан с тем, нужно ли осуществлять доступ к транспортным накладным отдельно от заказов. Это может быть крайне важно, если вы создаете отчеты посредством SQL-запросов и не используете для этого отдельной базы данных.

Выполняя отображение объектов на существующую схему базы данных, вы можете воспользоваться **внедренным значением** для отображения на таблицу, разбитую в оперативной памяти на несколько объектов. Это случается тогда, когда определенное поведение сущности выносят в отдельный объект, однако сохраняют ее неизменной в базе данных. В этом случае вам следует быть предельно внимательным, поскольку каждое изменение зависимого объекта автоматически помечает его владельца как "измененного" — ситуация, которая не является проблемой для **объектов-значений**, изменение которых равносильно их замене в коллекции владельца.

Как правило, **внедренное значение** для объектов, связанных ссылками, используют только тогда, когда связь между ними является однозначной "с обеих сторон" ссылки (связь типа "один к одному"). Иногда **внедренное значение** применяют и в тех случаях, когда зависимых объектов несколько, однако их число невелико и фиксированно. В этом случае каждому объекту соответствует несколько пронумерованных полей. Это значительно запутывает структуру таблицы и усложняет постановку SQL-запросов, однако

дает выигрыш в производительности. Впрочем, в плане производительности гораздо предпочтительнее использовать **крупный сериализованный объект (Serialized LOB, 292)**.

Большинство аргументов за или против использования **внедренного значения** справедливы и для **крупного сериализованного объекта**. Поэтому нельзя не уделить внимания вопросу выбора между этими двумя типовыми решениями. Огромным преимуществом **внедренного значения** является возможность постановки SQL-запросов к полям зависимого объекта. Хотя сериализация объектов с использованием XML, а также появление дополнительных компонентов SQL, основанных на XML, могут изменить ситуацию в будущем, пока что использовать зависимые значения в SQL-запросе можно, только применяя **внедренное значение**. Это может быть важно для отделения в базе данных механизмов создания отчетов.

Внедренное значение может применяться только для одного или нескольких сравнительно простых, не связанных между собой зависимых объектов. Более сложные структуры, включая подграфы объектов, потенциально являющиеся крупными объектами, требуют применения **крупного сериализованного объекта**.

Дополнительные источники информации

В разные годы типовое решение **внедренное значение** фигурировало под несколькими различными именами. В технологии TOPLink его называют *агрегированным отображением (aggregate mapping)*, а в Visual Age — *компоновщиком (composer)*.

Пример: простой объект-значение (Java)

Рассмотрим классический пример объекта-значения, отображение которого выполняется посредством **внедренного значения**. Для начала создадим простой класс `Product Offering` ("Предложение на продажу"), содержащий перечисленные ниже поля.

```
class ProductOffering...

   private Product product;
   private Money baseCost;
   private Integer ID;
```

Здесь поле `ID` является **полем идентификации (Identity Field, 237)**, а поле `product` — отображением обычной записи о товаре. Для отображения базовой стоимости товара (поле `baseCost`), представленного объектом `Money`, воспользуемся **внедренным значением**. Чтобы еще более упростить задачу, все отображение будет выполняться посредством **активной записи (Active Record, 182)**.

Поскольку используется **активная запись**, нам понадобится реализовать методы сохранения и загрузки. Эти методы будут определены в классе-владельце, а именно в классе `ProductOffering`. У класса `Money` подобных методов не будет. Покажем, как выглядит метод загрузки.

```
class ProductOffering...

   public static ProductOffering load(ResultSet rs) {
      try {
         Integer id = (Integer) rs.getObject("ID");
```

```
            BigDecimal baseCostAmount = rs.getBigDecimal(
↳"base_cost_amount");
            Currency baseCostCurrency = Registry.getCurrency(
↳rs.getString("base_cost_currency"));
            Money baseCost = new Money(baseCostAmount,
↳baseCostCurrency);
            Integer productID = (Integer) rs.getObject("product");
            Product product = Product.find((Integer) rs.getObject(
↳"product"));
            return new ProductOffering(id, product, baseCost);
         } catch (SQLException e) {
            throw new ApplicationException(e);
         }
      }
```

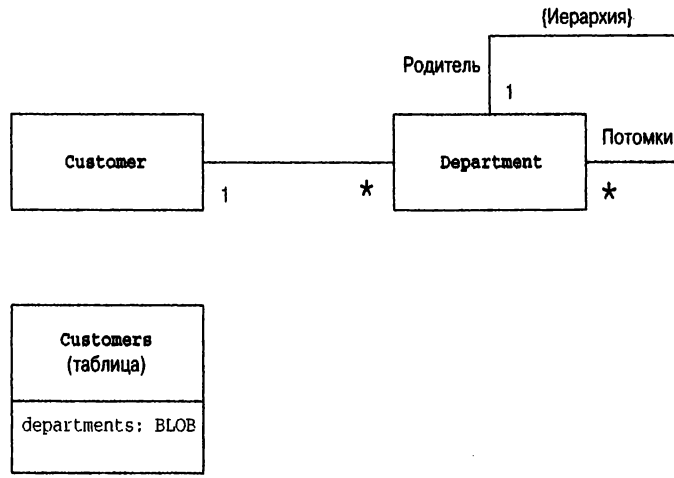
А теперь приведем метод обновления, который также представляет собой простую вариацию стандартных методов.

```
class ProductOffering...

   public void update() {
      PreparedStatement stmt = null;
      try {
         stmt = DB.prepare(updateStatementString);
         stmt.setBigDecimal(1, baseCost.amount());
         stmt.setString(2, baseCost.currency().code());
         stmt.setInt(3, ID.intValue());
         stmt.execute();
      } catch (Exception e) {
         throw new ApplicationException(e);
      } finally {DB.cleanUp(stmt);}
   }
   private String updateStatementString =
         "UPDATE product_offerings" +
         "    SET base_cost_amount = ?,
↳base_cost_currency = ? " +
         "    WHERE id = ?";
```

Сериализованный крупный объект (Serialized LOB)

Сохраняет граф объектов путем их сериализации в единый крупный объект (Large Object — LOB) и помещает его в поле базы данных



В объектных моделях часто встречаются сложные графы небольших объектов. Большая часть информации в подобных структурах заключается не столько в самих объектах, сколько в связях между ними. В качестве примера представьте себе иерархию отделов, обслуживающих покупателей некоторой организации. Объектная модель прекрасно подходит для представления подобных иерархий, и вы можете легко добавлять методы для извлечения объектов-предков, объектов-потомков, объектов, находящихся на том же уровне иерархии, что и данный, и т.п.

С реляционными базами данных дело обстоит намного сложнее. Основная схема как будто проста — таблица организации с внешними ключами "предков" соответствующих отделов. Тем не менее управление подобной таблицей требует выполнения множества соединений, которые довольно неприятны, да к тому же еще и медленны.

Между тем объекты не обязательно хранить в виде связанных между собой строк таблицы. Еще одной формой хранения объектов является сериализация, при которой весь граф объектов записывается в базу данных в виде так называемого *крупного объекта (Large Object — LOB)*. Соответствующее типовое решение получило название **крупного сериализованного объекта** и может рассматриваться как разновидность типового решения **memento (Memento)** [20].

Принцип действия

Существует два вида крупных объектов: *крупный двоичный объект (binary LOB — BLOB)* и *крупный символьный объект (character LOB — CLOB)*. Объекты BLOB проще создавать, поскольку многие платформы включают в себя возможность автоматической сериализации графов объектов. В этом случае сохранение графа представляет собой

простую сериализацию содержимого буфера и последующее сохранение этого буфера в соответствующем поле таблицы.

Преимуществами объектов BLOB являются простота кодирования (если они поддерживаются средой разработки) и небольшие размеры занимаемого пространства. Недостатки же заключаются в том, что для применения объектов BLOB используемая СУБД должна поддерживать двоичный формат данных и что структуру графа нельзя воссоздать без самого объекта, а потому содержимое этого поля совершенно непригодно для просмотра человеком. Впрочем, наиболее серьезная проблема использования объектов BLOB связана с версиями, например если вы измените класс отдела, то не сможете прочитать его предыдущие сериализации. Эта проблема не так уж незначительна, как может показаться на первый взгляд, поскольку содержимое базы данных способно храниться на протяжении долгого времени.

Альтернативой BLOB являются уже упоминавшиеся объекты CLOB. В этом случае граф объектов сериализуется в текстовую строку, содержащую всю необходимую информацию. Полученный текст вполне читабелен, что заметно облегчает работу при простом просмотре базы данных. Тем не менее сериализованный объект в текстовом формате занимает гораздо больший объем памяти и может потребовать написания специального анализатора. Кроме того, обработка символьных объектов обычно занимает больше времени, чем обработка двоичных.

Большую часть недостатков объектов CLOB можно преодолеть путем сериализации в формат XML. Анализаторы XML доступны всем, поэтому писать собственный анализатор вам не придется. Кроме того, в последние годы XML превратился в общепринятый стандарт и для работы с этим форматом данных постоянно появляется все больше и больше программных средств. К сожалению, использование XML не решает проблемы с занимаемым пространством. Вообще говоря, оно ее только усложняет, поскольку описания объектов на языке XML слишком "многословны". Данную проблему можно решить путем использования сжатого формата XML в качестве объекта BLOB; разумеется, это лишит его читабельности, однако значительно сократит используемое дисковое пространство.

При использовании **крупного сериализованного объекта** следует обращать особое внимание на проблемы идентификации. Предположим, вы хотите, чтобы в записи о заказе содержались сведения о покупателе. Не помещайте LOB с данными покупателя в таблицу заказов — в этом случае данные будут скопированы в каждый заказ, что значительно затруднит обновление одного из них. (Тем не менее этот подход может оказаться весьма удобным, если вы хотите сохранить данные о покупателе в том состоянии, в котором они находились в момент размещения заказа; это избавит от необходимости создавать временные отношения.) Если вы хотите, чтобы обновление данных о покупателе выполнялось при обновлении каждого заказа в классическом реляционном понимании этого процесса, LOB следует поместить в таблицу покупателей, чтобы на него могли ссылаться сразу несколько заказов. Не беспокойтесь, если полученная таблица будет состоять всего из двух полей — идентификатора и LOB; это вполне нормально.

Следует также отметить, что для данного типового решения характерно дублирование данных. Зачастую дублированию подвергается не весь объект, а только его часть, которая перекрывается с другой частью. Чтобы избежать подобных проблем, необходимо тщательно следить за тем, какие данные помещаются в **крупный сериализованный объект**, и гарантировать, чтобы эти данные были доступны только объекту, который выступает в роли владельца **крупного сериализованного объекта**.

Назначение

Крупный сериализованный объект применяется далеко не так часто, как можно было бы подумать. Привлекательность данного типового решения подкрепляется возможностью сериализации в формат XML, существенным образом упрощающей кодирование. Тем не менее данный подход имеет большой недостаток, поскольку к содержимому подобной структуры нельзя осуществлять запросы средствами SQL. В последние годы на рынке программного обеспечения появились расширения языка SQL, позволяющие извлекать XML-данные в пределах поля, однако это еще не то, что нужно (по крайней мере подобные решения не обеспечивают переносимости).

Данное типовое решение хорошо применять тогда, когда требуется сохранить сложный фрагмент объектной модели в виде единого целого (в частности, объекта LOB). Для большей наглядности LOB можно рассматривать как способ сохранения группы объектов, к которым не будут поступать SQL-запросы из-за пределов данного приложения. В этом случае сохраненный граф объектов может быть описан с помощью SQL-схемы.

Крупный сериализованный объект крайне редко применяется в тех случаях, когда на содержимое графа объектов ссылаются извне. Для обработки подобных ситуаций вам придется создавать некое подобие схемы связей, которая будет поддерживать ссылки на объекты, находящиеся внутри объекта LOB. Нельзя сказать, что это невозможно, однако очень неприятно, по крайней мере настолько, чтобы этого не делать. И вновь данную ситуацию может несколько облегчить использование XML (а точнее, языка запросов XPath).

Если для составления отчетов используется отдельная база данных и через нее проходят все остальные SQL-запросы, LOB можно преобразовать в соответствующую табличную структуру. База данных, применяемая для составления отчетов, обычно не подпадает под определение нормальных форм. Это означает, что все структуры, применимые для построения **крупного сериализованного объекта**, зачастую подходят и для построения таблиц базы данных отчетов.

Пример: сериализация иерархии отделов в формат XML (Java)

В этом примере воспользуемся моделью покупателей и отделов, рассмотренной в начале раздела, и сериализуем иерархию отделов в объект CLOB формата XML. На момент написания этой книги средства Java для работы с XML были весьма примитивны и постоянно менялись, поэтому к тому времени, когда вы прочитаете этот текст, многое может измениться. (Кроме того, я использовал раннюю версию технологии JDOM.)

Рассмотренная объектная модель описывается следующей структурой классов:

```
class Customer...

   private String name;
   private List departments = new ArrayList();

class Department...

   private String name;
   private List subsidiaries = new ArrayList();
```

Соответствующая база данных состоит всего из одной таблицы.

```
create table customers (ID int primary key, name varchar,
↳departments varchar)
```

В нашем примере объект `Customer` будет выполнять роль **активной записи (Active Record, 182)**. Рассмотрим, как выполняется вставка графа отделов в базу данных.

```
class Customer...

   public Long insert() {
      PreparedStatement insertStatement = null;
      try {
         insertStatement = DB.prepare(insertStatementString);
         setID(findNextDatabaseId());
         insertStatement.setInt(1, getID().intValue());
         insertStatement.setString(2, name);
         insertStatement.setString(3,
XmlStringer.write(departmentsToXmlElement()));
         insertStatement.execute();
         Registry.addCustomer(this);
         return getID();
      } catch (SQLException e) {
         throw new ApplicationException(e);
      } finally {DB.cleanUp(insertStatement);
      }
   }
   public Element departmentsToXmlElement() {
      Element root = new Element("departmentList");
      Iterator i = departments.iterator();
      while (i.hasNext()) {
         Department dep = (Department) i.next();
         root.addContent(dep.toXmlElement());
      }
      return root;
   }

class Department...

   Element toXmlElement() {
      Element root = new Element("department");
      root.setAttribute("name", name);
      Iterator i = subsidiaries.iterator();
      while (i.hasNext()) {
         Department dep = (Department) i.next();
         root.addContent(dep.toXmlElement());
      }
      return root;
   }
```

У объекта `Customer` есть метод для сериализации содержимого поля `departments` в единую модель XML DOM (Document Object Model — объектная модель документа). У каждого объекта `Department` также есть метод для сериализации соответствующего

отдела (с рекурсивной сериализацией "дочерних" отделов) в свою модель DOM. После выполнения сериализации метод вставки преобразует модель DOM общей иерархии отделов в строку (с помощью служебного класса `XmlStringer`) и помещает ее в базу данных. Нас не очень волнует структура полученной строки. Она вполне читабельна, однако просматривать ее слишком часто не придется.

```
<?xml version="1.0" encoding="UTF-8"?>
<departmentList>
   <department name="US">
      <department name="New England">
         <department name="Boston" />
         <department name="Vermont" />
      </department>
      <department name="California" />
      <department name="Mid-West" />
   </department>
   <department name="Europe" />
</departmentList>
```

Считывание иерархии отделов выполняется прямо противоположно вставке.

```
class Customer...

   public static Customer load(ResultSet rs)
↳throws SQLException {
      Long id = new Long(rs.getLong("id"));
      Customer result = (Customer) Registry.getCustomer(id);
      if (result != null) return result;
      String name = rs.getString("name");
      String departmentLob = rs.getString("departments");
      result = new Customer(name);
      result.readDepartments(XmlStringer.read(departmentLob));
      return result;
   }
   void readDepartments(Element source) {
      List result = new ArrayList();
      Iterator it = source.getChildren("department").iterator();
      while (it.hasNext())
         addDepartment(Department.readXml((Element) it.next()));
   }

class Department...

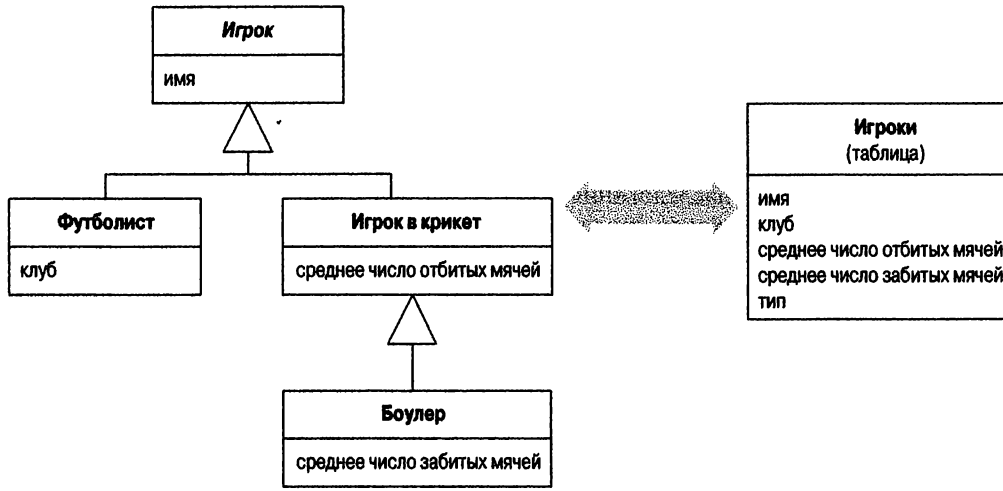
   static Department readXml(Element source) {
      String name = source.getAttributeValue("name");
      Department result = new Department(name);
      Iterator it = source.getChildren("department").iterator();
      while (it.hasNext())
         result.addSubsidiary(readXml((Element) it.next()));
      return result;
   }
```

Как видите, код загрузки является "зеркальным отражением" кода вставки. Объект `Department` содержит метод для восстановления отдела (и его дочерних отделов) из кода XML, а объект `Customer` оснащен методом, позволяющим извлечь из базы данных код XML и преобразовать полученную строку (с помощью служебного класса `XmlStringer`) в список отделов.

При сериализации данных в формат XML существует большая опасность, что кто-то попытается отредактировать содержимое полученного поля вручную, сделав его нечитабельным для методов загрузки. В качестве решения этой проблемы можно порекомендовать расширенные средства для работы с XML, которые поддерживают добавление к полю проверки на правильность путем применения шаблона DTD или схемы XML.

Наследование с одной таблицей (Single Table Inheritance)

Представляет иерархию наследования классов в виде одной таблицы, столбцы которой соответствуют всем полям классов, входящих в иерархию



Реляционные базы данных не поддерживают наследование. Выполняя отображение объектной модели на базу данных, необходимо найти способ, позволяющий отобразить структуру наследования. Разумеется, крайне важно минимизировать количество соединений, которое стремительно возрастает при попытке отображения структуры наследования на разные таблицы. На помощь приходит типовое решение **наследование с одной таблицей**, отображающее все поля всех классов структуры наследования на столбцы одной и той же таблицы.

Принцип действия

Итак, в данном случае структура наследования отображается на одну таблицу, которая содержит в себе все данные всех классов, входящих в иерархию наследования. Каждому классу (а точнее, его экземпляру) соответствует одна строка таблицы; при этом поля таблицы, которых нет в данном классе, остаются пустыми. Основное поведение объектов, выполняющих отображение, соответствует общей схеме **преобразователей наследования (Inheritance Mappers, 322)**.

Выполняя загрузку объекта в память, необходимо знать, в экземпляр какого класса следует поместить загружаемые данные. Для этого к таблице добавляется специальное поле, указывающее на то, экземпляр какого класса должен быть создан для загрузки данного объекта. Это может быть имя класса или какое-нибудь кодовое поле. Для отображения кодового поля на имя соответствующего класса необходим специальный код, который должен быть расширен при добавлении в иерархию нового класса. В свою очередь, указанное в таблице явное имя класса можно использовать непосредственно для создания экземпляра этого класса. Следует отметить, что явное имя класса занимает больше места и является более сложным для обработки при непосредственном использовании таблиц базы данных. Кроме того, оно теснее привязывает структуру классов к схеме базы данных.

Перед загрузкой данных необходимо считать код типа класса, чтобы узнать, экземпляр какого производного класса должен быть создан для загрузки объекта. При сохранении объекта в базе данных запись кода типа класса выполняется суперклассом.

Назначение

Типовое решение **наследование с одной таблицей** является одним из вариантов отображения иерархии наследования на реляционную базу данных. В число других возможных вариантов входят **наследование с таблицами для каждого класса (Class Table Inheritance, 305)** и **наследование с таблицами для каждого конкретного класса (Concrete Table Inheritance, 313)**.

Наследование с одной таблицей имеет ряд преимуществ.

- В структуру базы данных добавляется только одна таблица.
- Для извлечения данных не нужно выполнять соединение таблиц.
- Перемещение полей в производный класс или суперкласс не требует внесения изменений в структуру базы данных.

Несмотря на это, у данного типового решения есть и слабые стороны.

- Не все поля соответствуют содержимому каждого конкретного объекта, что может приводить в замешательство людей, работающих только с таблицами.
- Некоторые столбцы используются только одним-двумя производными классами, что приводит к бессмысленной трате свободного места. Критичность данной проблемы зависит от характеристик конкретных данных, а также от того, насколько хорошо сжимаются пустые поля. Например, в базах данных Oracle применяется высокая степень сжатия свободного пространства, особенно если "необязательные" столбцы выносятся в правую часть таблицы. Впрочем, у каждой базы данных есть свои приемы на этот счет.

- Полученная таблица может оказаться слишком большой, с множеством индексов и частыми блокировками, что будет оказывать негативное влияние на производительность базы данных. Во избежание этой проблемы можно создать отдельные таблицы индексов, которые будут содержать ключи строк, имеющих определенное свойство, или же копии подмножеств полей, имеющих отношение к индексам.
- Все имена столбцов таблицы принадлежат единому пространству имен, поэтому необходимо следить за тем, чтобы у полей разных классов не было одинаковых имен. Для облегчения работы рекомендую называть поля составными именами с указанием имени содержащего их класса в качестве префикса или суффикса.

Запомните: вы вовсе не обязаны использовать единственную форму отображения для *всей* иерархии наследования. Вполне естественно отобразить 5–10 классов с похожей структурой в общую таблицу, в то время как классы со множеством специфичных данных будут отображаться с использованием **наследования с таблицами для каждого конкретного класса**.

Пример: общая таблица игроков (C#)

Как и другие примеры обработки иерархий наследования, данный пример основан на использовании **преобразователей наследования** и объектной модели, изображенной на рис. 12.8. Каждый преобразователь должен ссылаться на таблицу `DataTable` объекта ADO.NET `DataSet`. Эта ссылка может быть реализована в общем виде в суперклассе `Mapper`. Значением свойства `Data` класса `Gateway` является объект `DataSet`, содержимое которого может быть загружено в результате выполнения запроса.

```
class Mapper...

      protected DataTable table {
         get {return Gateway.Data.Tables [TableName ];}
      }
      protected Gateway Gateway;
      abstract protected String TableName {get;}
```

Поскольку таблица всего одна, ее можно определить в абстрактном классе `AbstractPlayerMapper`.

```
class AbstractPlayerMapper...

      protected override String TableName {
         get {return "Players";}
      }
```

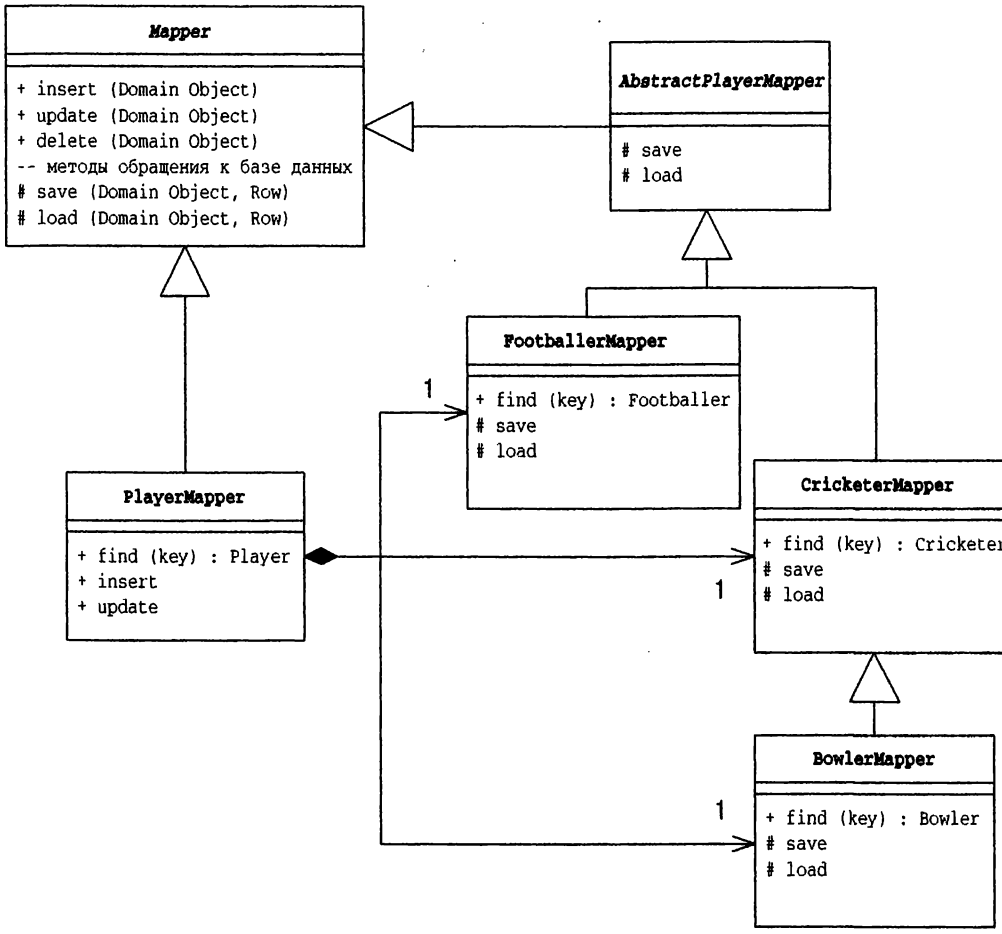


Рис. 12.8. Универсальная схема классов для **преобразователей наследования**

Каждому классу нужно поставить в соответствие код типа класса, чтобы преобразователь знал, с каким игроком он имеет дело. Код типа определяется в суперклассе и реализуется в производных классах.

```
class AbstractPlayerMapper...

      abstract public String TypeCode {get;}

class CricketerMapper...
      public const String TYPE_CODE = "C";
      public override String TypeCode {
         get {return TYPE_CODE;}
      }
```

Класс `PlayerMapper` содержит по одному полю на каждый из трех конкретных классов преобразователей (и соответственно на каждый из трех типов игроков).

```
class PlayerMapper...

      private BowlerMapper bmapper;
      private CricketerMapper cmapper;
      private FootballerMapper fmapper;
      public PlayerMapper (Gateway gateway) : base (gateway) {
         bmapper = new BowlerMapper(Gateway);
         cmapper = new CricketerMapper(Gateway);
         fmapper = new FootballerMapper(Gateway);
      }
```

Загрузка объекта из базы данных

Каждый конкретный класс преобразователя содержит метод поиска для извлечения объекта из базы данных.

```
class CricketerMapper...

      public Cricketer Find(long id) {
         return (Cricketer)AbstractFind(id);
      }
```

Для выполнения поиска данный метод вызывает универсальный метод суперкласса.

```
class Mapper...

      protected DomainObject AbstractFind(long id) {
         DataRow row = FindRow(id);
         return (row == null) ? null : Find(row);
      }
      protected DataRow FindRow(long id) {
         String filter = String.Format("id = {0}", id);
         DataRow[] results = table.Select(filter);
         return (results.Length == 0) ? null : results[0];
      }
      public DomainObject Find (DataRow row) {
         DomainObject result = CreateDomainObject();
         Load(result, row);
         return result;
      }
      abstract protected DomainObject CreateDomainObject();

class CricketerMapper...

      protected override DomainObject CreateDomainObject() {
         return new Cricketer();
      }
```

Для загрузки данных в новый объект я применяю группу методов загрузки — по одному в каждом классе иерархии.

```
class CricketerMapper...

      protected override void Load(DomainObject obj,
↳DataRow row) {
         base.Load(obj,row);
         Cricketer cricketer = (Cricketer) obj;
         cricketer.battingAverage = (double)row[
↳"battingAverage"];
      }

class AbstractPlayerMapper...

      protected override void Load(DomainObject obj,
↳DataRow row) {
         base.Load(obj, row);
         Player player = (Player) obj;
         player.name = (String)row["name"];
      }

class Mapper...

      protected virtual void Load(DomainObject obj,
↳DataRow row) {
         obj.Id = (int) row ["id"];
      }
```

Вместо этого я могу загрузить сведения об игроке с помощью преобразователя `PlayerMapper`. Он считывает данные и использует код типа класса, чтобы определить, какой конкретный преобразователь нужно использовать в данном случае.

```
class PlayerMapper...

      public Player Find (long key) {
         DataRow row = FindRow(key);
         if (row == null) return null;
         else {
            String typecode = (String) row["type"];
            switch (typecode){
               case BowlerMapper.TYPE_CODE:
                  return (Player) bmapper.Find(row);
               case CricketerMapper.TYPE_CODE:
                  return (Player) cmapper.Find(row);
               case FootballerMapper.TYPE_CODE:
                  return (Player) fmapper.Find(row);
               default:
                  throw new Exception("unknown type");
            }
         }
      }
```

Обновление объекта

Суть операции обновления одинакова для всех классов, поэтому ее можно определить в суперклассе `Mapper`.

```
class Mapper...

      public virtual void Update (DomainObject arg){
         Save (arg,FindRow(arg.Id));
      }
```

Метод сохранения аналогичен методу загрузки, т.е. определен в каждом производном классе для сохранения соответствующих данных.

```
class CricketerMapper...

      protected override void Save(DomainObject obj,
↳DataRow row) {
         base.Save(obj, row);
         Cricketer cricketer = (Cricketer) obj;
         row["battingAverage"] = cricketer.battingAverage;
      }

class AbstractPlayerMapper...

      protected override void Save(DomainObject obj,
↳DataRow row) {
         Player player = (Player) obj;
         row["name"] = player.name;
         row["type"] = TypeCode;
      }
```

Преобразователь `PlayerMapper` обращается к нужному конкретному преобразователю.

```
class PlayerMapper...

      public override void Update (DomainObject obj) {
         MapperFor(obj).Update(obj);
      }
      private Mapper MapperFor(DomainObject obj) {
         if (obj is Footballer)
            return fmapper;
         if (obj is Bowler)
            return bmapper;
         if (obj is Cricketer)
            return cmapper;
         throw new Exception("No mapper available");
      }
```

Вставка объекта

Выполнение вставки аналогично обновлению; единственная существенная разница состоит в том, что перед сохранением данных в таблице нужно создать новую строку.

```
class Mapper...

      public virtual long Insert (DomainObject arg) {
         DataRow row = table.NewRow();
         arg.Id = GetNextID();
         row["id"] = arg.Id;
         Save (arg, row);
         table.Rows.Add(row);
         return arg.Id;
      }

class PlayerMapper...

      public override long Insert (DomainObject obj) {
         return MapperFor(obj).Insert(obj);
      }
```

Удаление объекта

Удалить объект очень просто. Операции удаления определены на абстрактном уровне, а также в классе-оболочке `PlayerMapper`.

```
class Mapper...

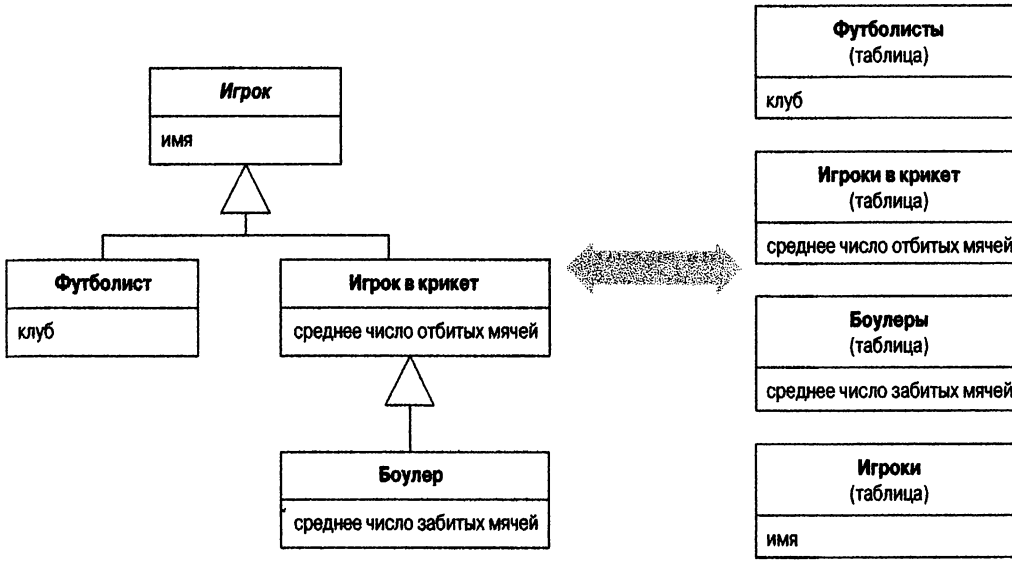
      public virtual void Delete(DomainObject obj) {
         DataRow row = FindRow(obj.Id);
         row.Delete();
      }

class PlayerMapper...

      public override void Delete (DomainObject obj) {
         MapperFor(obj).Delete(obj);
      }
```

Наследование с таблицами для каждого класса (Class Table Inheritance)

Представляет иерархию наследования классов, используя по одной таблице для каждого класса



Одним из наиболее очевидных несоответствий объектной и реляционной моделей является то, что реляционные базы данных не поддерживают наследование. Между тем требуется создать такую структуру базы данных, которая бы хорошо отображалась на объекты и сохраняла все возможные связи объектной модели. Для этого применяется типовое решение **наследование с таблицами для каждого класса**, которое использует по одной таблице на каждый класс структуры наследования.

Принцип действия

Идея **наследования с таблицами для каждого класса** проста и понятна: каждому классу модели предметной области соответствует своя таблица базы данных. Поля класса домена отображаются непосредственно на столбцы соответствующей таблицы. Как и в других схемах отображения иерархии наследования, в данном типовом решении применяется фундаментальный принцип **преобразователей наследования (Inheritance Mappers, 322)**.

При использовании **наследования с таблицами для каждого класса** возникает вопрос, как связать соответствующие строки таблиц базы данных. Возможным решением может стать использование общего значения первичного ключа: например, чтобы строка с ключом 101 в таблице футболистов и строка с ключом 101 в таблице игроков соответствовали одному и тому же объекту домена. Поскольку таблица суперкласса содержит по одной строке на каждую строку каждой таблицы производных классов, первичные ключи записей должны быть уникальными в пределах всех таблиц производных классов. Вместо

этого у каждой таблицы может быть собственный первичный ключ. В этом случае для привязки соответствующих строк в таблицу суперкласса добавляются внешние ключи таблиц производных классов.

Еще одна важная проблема реализации **наследования с таблицами для каждого класса** — эффективное извлечение данных из множества таблиц. Очевидно, выполнение отдельного запроса к каждой таблице не будет эффективным, поскольку потребует множества обращений к базе данных. Эту проблему можно решить путем соединения нескольких таблиц. Следует, однако, учесть, что в силу способа оптимизации, применяемого в большинстве баз данных, соединение трех или более таблиц будет обрабатываться крайне медленно.

И наконец, нельзя не обратить внимание на такой аспект: при выполнении запросов не всегда известно, к каким именно таблицам следует применить соединение. Если вы ищете сведения о футболисте, то, естественно, обратитесь к таблице футболистов; если же потребуются сведения о группе игроков, к каким таблицам обратиться? Разумеется, не все таблицы будут содержать необходимые данные. Для выполнения эффективных соединений с такими таблицами применяется внешнее соединение, однако оно нестандартно и достаточно медленно. В качестве альтернативы можно предложить следующее решение: вначале считать корневую таблицу, а затем использовать код типа класса, чтобы определить, какие таблицы следует считывать далее. К сожалению, этот способ требует выполнения нескольких запросов.

Назначение

Типовые решения **наследование с таблицами для каждого класса**, **наследование с одной таблицей (Single Table Inheritance, 305)** и **наследование с таблицами для каждого конкретного класса (Concrete Table Inheritance, 313)** применяются для отображения иерархии наследования на реляционную базу данных.

Наследование с таблицами для каждого класса имеет ряд преимуществ.

- Все поля таблицы соответствуют содержимому каждой ее строки (т.е. есть в каждом описываемом объекте), поэтому таблицы легки в понимании и не занимают лишнего места.
- Взаимосвязь между моделью домена и схемой базы данных проста и понятна.

Однако это типовое решение имеет и слабые стороны.

- Загрузка объекта охватывает сразу несколько таблиц, что требует их соединения либо множества обращений к базе данных с последующим "сшиванием" результатов в памяти.
- Перемещение полей в производный класс или суперкласс требует изменения структуры базы данных.
- Таблицы суперклассов могут стать "узким местом" в вопросах производительности, поскольку доступ к таким таблицам будет осуществляться слишком часто.
- Высокая степень нормализации может стать препятствием для выполнения запросов, не хранящихся в базе данных (ad hoc queries).

Напомню: вы вовсе не обязаны использовать единственную форму отображения для *всей* иерархии наследования. Например, вы можете применить **наследование с таблицами для каждого класса** для классов, находящихся на верхних уровнях иерархии, и несколько **наследований с таблицами для каждого конкретного класса** для классов на более низких уровнях.

Дополнительные источники информации

В литературе по программным продуктам и технологиям IBM данное типовое решение фигурирует под именем **отображение "корень–лист" (Root-Leaf Mapping)** [9].

Пример: семейство игроков (C#)

Рассмотрим реализацию модели, приведенной в начале раздела. И вновь я воспользуюсь уже знакомым (хотя и несколько странным) примером с игроками, а также стандартным принципом **преобразователей наследования** (рис. 12.9).

Каждый класс должен определить таблицу, которая будет содержать в себе его данные и тип кода.

```
class AbstractPlayerMapper...

      abstract public String TypeCode {get;}
      protected static String TABLENAME = "Players";

class FootballerMapper...

      public override String TypeCode {
         get {return "F";}
      }
      protected new static String TABLENAME = "Footballers";
```

Обратите внимание, что в данном примере (в отличие от других) не переопределяется имя таблицы. Это сделано потому, что у каждого класса должна быть своя таблица, даже если его экземпляр будет являться экземпляром производного класса.

Загрузка объекта

Если вы уже просматривали другие примеры отображений иерархии наследования, то наверняка помните, что выполнение поиска инициируется методом конкретного преобразователя.

```
class FootballerMapper...

      public Footballer Find(long id) {
         return (Footballer) AbstractFind (id, TABLENAME);
      }
```

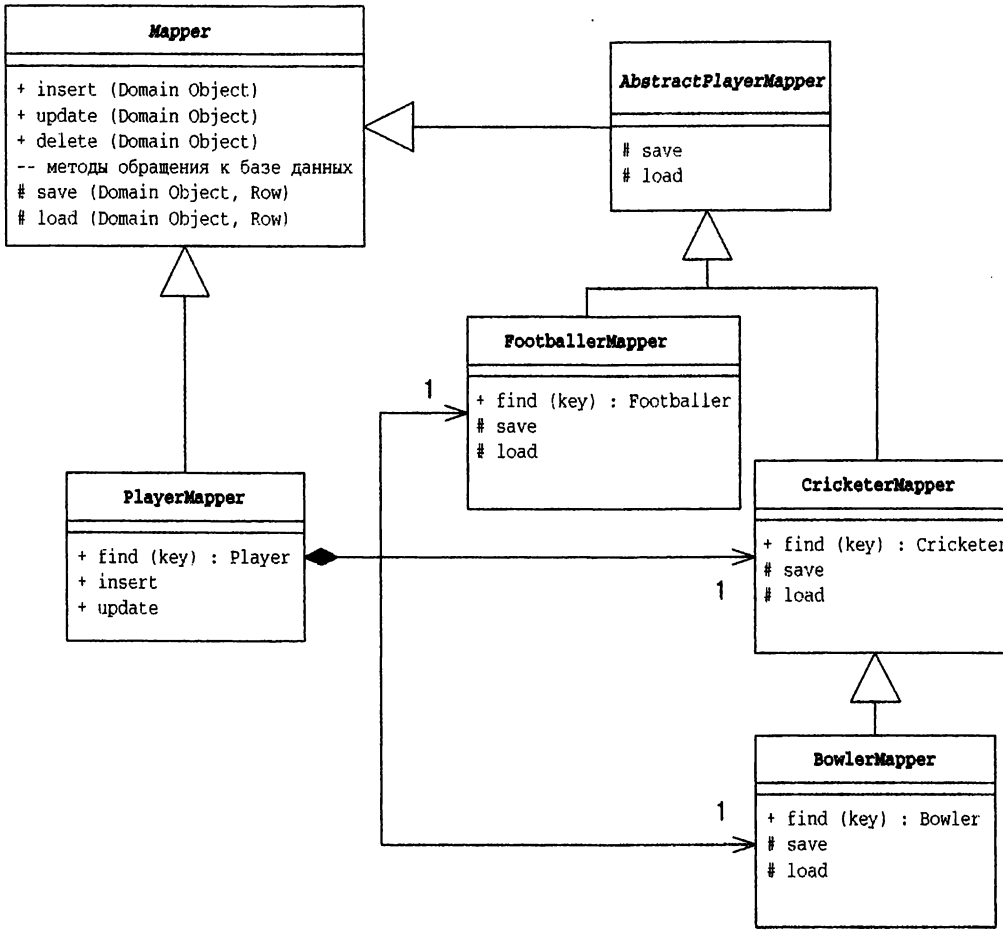


Рис. 12.9. Универсальная схема классов для **преобразователей наследования**

Метод `AbstractFind` ищет строку с заданным значением ключа. В случае успеха данный метод создает объект домена, после чего вызывает метод загрузки этого объекта.

```
class Mapper...

      public DomainObject AbstractFind(long id,
↳String tablename) {
         DataRow row = FindRow (id, tableFor(tablename));
         if (row == null) return null;
         else {
            DomainObject result = CreateDomainObject();
            result.Id = id;
            Load(result);
            return result;
         }
      }
```

```
    protected DataTable tableFor(String name) {
        return Gateway.Data.Tables[name];
    }
    protected DataRow FindRow(long id, DataTable table) {
        String filter = String.Format("id = {0}", id);
        DataRow[] results = table.Select(filter);
        return (results.Length == 0) ? null : results[0];
    }
    protected DataRow FindRow (long id, String tablename) {
        return FindRow(id, tableFor(tablename));
    }
    protected abstract DomainObject CreateDomainObject();

class FootballerMapper...

    protected override DomainObject CreateDomainObject(){
        return new Footballer();
    }
```

У каждого класса есть свой метод загрузки, который загружает свойства, определенные в этом классе.

```
class FootballerMapper...

    protected override void Load(DomainObject obj) {
        base.Load(obj);
        DataRow row = FindRow (obj.Id, tableFor(TABLENAME));
        Footballer footballer = (Footballer) obj;
        footballer.club = (String)row["club"];
    }

class AbstractPlayerMapper...

    protected override void Load(DomainObject obj) {
        DataRow row = FindRow (obj.Id, tableFor(TABLENAME));
        Player player = (Player) obj;
        player.name = (String)row["name"];
    }
```

Как и в других примерах (хотя здесь это, пожалуй, более очевидно), я исхожу из предположения, что данные были загружены в объект ADO.NET `DataSet` и кэшированы в оперативной памяти. Это позволяет выполнить несколько обращений к табличным данным без особых потерь в производительности. Если же вы обращаетесь непосредственно к базе данных, нагрузку придется уменьшить. В этом примере для уменьшения нагрузки можно создать соединение всех таблиц и в дальнейшем ставить свои запросы именно к нему.

Объект `PlayerMapper` определяет, какому типу принадлежит искомый игрок, и затем делегирует выполнение поиска нужному конкретному преобразователю.

```
class PlayerMapper...

    public Player Find (long key) {
        DataRow row = FindRow(key, tableFor(TABLENAME));
```

```
        if (row == null) return null;
        else {
           String typecode = (String) row["type"];
           if (typecode == bmapper.TypeCode)
              return bmapper.Find(key);
           if (typecode == cmapper.TypeCode)
              return cmapper.Find(key);
           if (typecode == fmapper.TypeCode)
              return fmapper.Find(key);
           throw new Exception("unknown type");
     }
  }
  protected static String TABLENAME = "Players";
```

Обновление объекта

Метод обновления определен в суперклассе `Mapper`.

```
class Mapper...

     public virtual void Update (DomainObject arg){
        Save (arg);
     }
```

Он реализован посредством группы методов сохранения — по одному на каждый класс иерархии.

```
class FootballerMapper...

     protected override void Save(DomainObject obj) {
        base.Save(obj);
        DataRow row = FindRow (obj.Id, tableFor(TABLENAME));
        Footballer footballer = (Footballer) obj;
        row["club"] = footballer.club;
     }

class AbstractPlayerMapper...

     protected override void Save(DomainObject obj) {
        DataRow row = FindRow (obj.Id, tableFor(TABLENAME));
        Player player = (Player) obj;
        row["name"] = player.name;
        row["type"] = TypeCode;
     }
```

Метод обновления класса `PlayerMapper` переопределяет универсальный метод, чтобы делегировать выполнение обновления нужному конкретному преобразователю.

```
class PlayerMapper...

     public override void Update (DomainObject obj) {
        MapperFor(obj).Update(obj);
     }
```

```
      private Mapper MapperFor(DomainObject obj) {
         if (obj is Footballer)
            return fmapper;
         if (obj is Bowler)
            return bmapper;
         if (obj is Cricketer)
            return cmapper;
         throw new Exception("No mapper available");
      }
```

Вставка объекта

Метод вставки объявлен в суперклассе `Mapper`. Выполнение вставки происходит в два этапа: вначале создаются новые строки базы данных, а затем с помощью методов обновления в созданные строки помещаются необходимые данные.

```
class Mapper...

      public virtual void Update (DomainObject arg) {
         Save (arg);
      }
```

Каждый класс, задействованный во вставке объекта, добавляет в свою таблицу новую строку.

```
class FootballerMapper...

   protected override void AddRow (DomainObject obj) {
      base.AddRow(obj);
      InsertRow (obj, tableFor(TABLENAME));
   }

class AbstractPlayerMapper...

   protected override void AddRow (DomainObject obj) {
      InsertRow (obj, tableFor(TABLENAME));
   }

class Mapper...

   abstract protected void AddRow (DomainObject obj);
   protected virtual void InsertRow (DomainObject arg,
↳DataTable table) {
      DataRow row = table.NewRow();
      row["id"] = arg.Id;
      table.Rows.Add(row);
   }
```

Объект `PlayerMapper` делегирует полномочия нужному конкретному преобразователю.

```
class PlayerMapper...

      public override long Insert (DomainObject obj) {
```

```
        return MapperFor(obj).Insert(obj);
    }
```

Удаление объекта

Чтобы удалить объект, каждый класс удаляет из своей таблицы соответствующую строку.

```
class FootballerMapper...

    public override void Delete(DomainObject obj) {
        base.Delete(obj);
        DataRow row = FindRow(obj.Id, TABLENAME);
        row.Delete();
    }

class AbstractPlayerMapper...

    public override void Delete(DomainObject obj) {
        DataRow row = FindRow(obj.Id, tableFor(TABLENAME));
        row.Delete();
    }

class Mapper...

    public abstract void Delete(DomainObject obj);
```

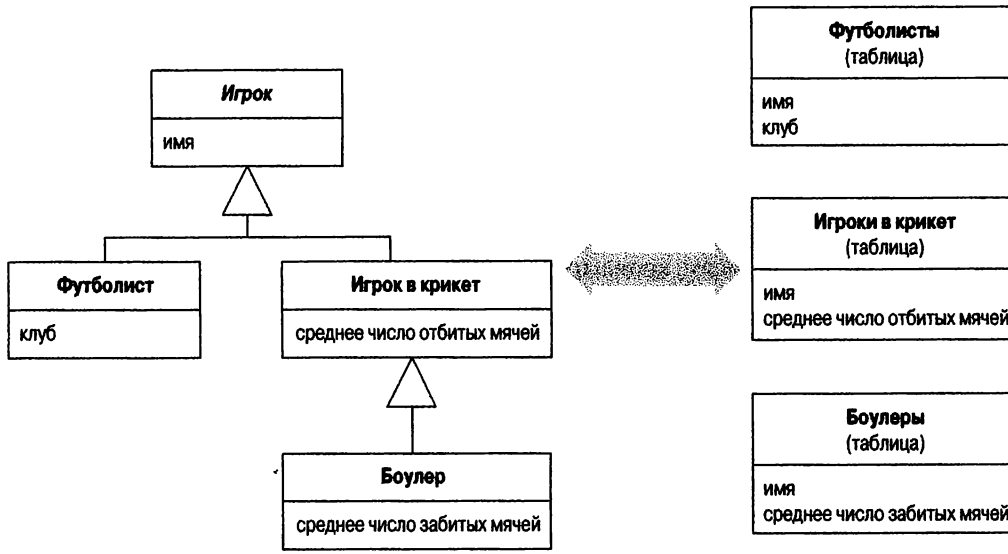
И снова объект `PlayerMapper` перекладывает всю работу на конкретный преобразователь.

```
class PlayerMapper...

    override public void Delete(DomainObject obj) {
        MapperFor(obj).Delete(obj);
    }
```

Наследование с таблицами для каждого конкретного класса (Concrete Table Inheritance)

Представляет иерархию наследования классов, используя по одной таблице для каждого конкретного класса этой иерархии



Спросите любого пуриста объектно-ориентированного подхода, и он с негодованием подтвердит, что реляционные базы данных не поддерживают наследование, — факт, который значительно затрудняет объектно-реляционное отображение. Если рассматривать таблицы с "точки зрения" экземпляров объектов, наиболее естественным будет отобразить каждый объект, находящийся в оперативной памяти, на отдельную строку базы данных. Этот подход реализован в типовом решении **наследование с таблицами для каждого конкретного класса**, которое подразумевает создание отдельной таблицы для каждого конкретного класса иерархии наследования.

Должен признаться, что при выборе названия для данного типового решения у меня возникли некоторые затруднения. Большинство разработчиков рассматривают его как основанное на использовании "листьев", поскольку в большинстве случаев таблицы базы данных соответствуют листьям иерархии объектов. Если следовать этой логике, данное типовое решение нужно было бы назвать, например, "наследование с таблицами для каждого листа" (leaf table inheritance). На самом деле во многих источниках и правда используют слово "лист" (leaf). Тем не менее иногда таблицы создаются и для конкретных классов, не являющихся листьями, поэтому с формальной точки зрения мое название более корректно (хотя и не совсем интуитивно).

Принцип действия

Данное типовое решение подразумевает создание отдельной таблицы для каждого конкретного класса иерархии наследования. При этом каждая таблица содержит столбцы, соответствующие полям конкретного класса и всех его "предков", а потому поля суперкласса дублируются во всех таблицах его производных классов. Как и остальные схемы отображения иерархии наследования, данное типовое решение основано на фундаментальном принципе **преобразователей наследования (Inheritance Mappers, 322)**.

Используя **наследование с таблицами для каждого конкретного класса**, необходимо обращать особое внимание на ключи. В данном случае они должны быть уникальны не только в пределах таблицы, но и в пределах всей иерархии. Классическим примером необходимости подобного решения является ситуация, когда для коллекции игроков используется **поле идентификации (Identity Field, 237)** с ключами, уникальными в пределах таблиц. Если значения ключей повторяются в разных таблицах конкретных классов, одному значению ключа может соответствовать целый набор строк. В этом случае вам понадобится система распределения ключей, которая будет следить за использованием их значений в разных таблицах; кроме того, вы не сможете полагаться на механизм уникальности первичных ключей базы данных.

Подобная ситуация может стать особенно неприятной, если вы работаете с базами данных, используемыми другими системами. В большинстве таких случаев вы не сможете гарантировать уникальность ключей в пределах таблиц. Чтобы избежать такого рода проблем, вам придется отказаться от использования полей суперклассов либо применить составной ключ, который будет включать в себя идентификатор таблицы.

Ситуацию можно упростить, если вообще не создавать полей, принадлежащих суперклассу, однако это потребует полного пересмотра объектной модели. Существует и альтернативный вариант: определить открытый интерфейс с методами доступа к супертипу, а в реализации использовать несколько закрытых полей для каждого конкретного типа. В этом случае интерфейс будет комбинировать значения закрытых полей. Если открытый интерфейс возвращает одномерное значение, он будет выбирать из значений закрытых полей то, которое не равно `NULL`. Если же открытый интерфейс возвращает значение-коллекцию, он будет использовать объединение значений закрытых полей.

Для представления составного ключа в качестве значения **поля идентификации** можно использовать специальный объект ключа. Чтобы обеспечить уникальность по всей иерархии наследования, такой ключ использует сочетание первичного ключа таблицы и ее имени.

Описанная проблема тесно связана с проблемой целостности на уровне ссылок. В качестве примера рассмотрим объектную модель, изображенную на рис. 12.10. Чтобы реализовать целостность на уровне ссылок, нужно создать таблицу отношений, которая будет содержать внешние ключи благотворительных акций и игроков. Но в базе данных нет таблицы игроков, поэтому для поля внешнего ключа нельзя создать ограничение целостности на уровне ссылок, которое бы применялось и для футболистов, и для игроков в крикет. В этом случае придется либо пренебречь целостностью на уровне ссылок, либо использовать несколько таблиц отношений (по одной на каждую реальную таблицу базы данных). Проблема становится еще более серьезной, если мы не сможем гарантировать уникальность ключа в пределах иерархии.

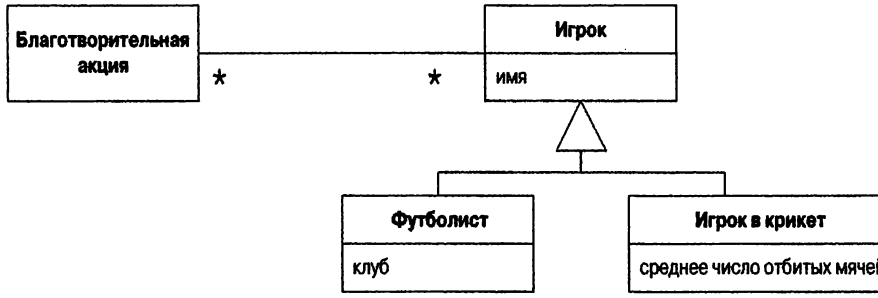


Рис. 12.10. Объектная модель, иллюстрирующая сложность реализации целостности на уровне ссылок при использовании типового решения **наследование с таблицами для каждого конкретного класса**

Выполняя поиск игроков с помощью оператора `SELECT`, вы должны просмотреть все таблицы, чтобы узнать, какая из них содержит нужное значение. Для этого придется совершить несколько запросов либо выполнить внешнее соединение. И то и другое решение крайне негативно отражается на производительности. Разумеется, вы не страдаете от падения производительности, когда точно знаете, какой класс вам нужен, однако для повышения производительности необходимо использовать только конкретные классы.

Данное типовое решение часто рассматривают как разновидность *наследования с таблицами для каждого листа (leaf table inheritance)*. Некоторые разработчики предпочитают создавать по одной таблице не для каждого конкретного класса, а для каждого листа иерархии наследования. Если в иерархии нет конкретных суперклассов, данные принципы отображения полностью совпадают. Впрочем, даже если подобные классы и есть, разница все равно невелика.

Назначение

Для отображения иерархии наследования на реляционную базу данных могут применяться **наследование с таблицами для каждого конкретного класса**, **наследование с таблицами для каждого класса (Class Table Inheritance, 305)** или **наследование с одной таблицей (Single Table Inheritance, 297)**.

Преимущества **наследования с таблицами для каждого конкретного класса** перечислены ниже.

- Каждая таблица является замкнутой и не содержит ненужных полей, вследствие чего ее удобно использовать в других приложениях, не работающих с объектами.
- При считывании данных посредством конкретных преобразователей не нужно выполнять соединений.
- Доступ к таблице осуществляется только в случае доступа к конкретному классу, что позволяет распределить нагрузку по всей базе данных.

Разумеется, данное типовое решение имеет и слабые стороны.

- Первичные ключи могут быть неудобны в обработке.
- Отсутствует возможность моделировать отношения между абстрактными классами.

- Если поля классов домена перемещаются в суперклассы или производные классы, придется вносить изменения в определения таблиц. Эти изменения будут не так часты, как в случае **наследования с таблицами для каждого класса**, однако их нельзя просто игнорировать, как было в случае с **наследованием с одной таблицей**.
- Если в суперклассе будет изменено какое-нибудь поле, понадобится изменить каждую таблицу, имеющую данное поле, поскольку поля суперкласса дублируются во всех таблицах его производных классов.
- Абстрактному методу поиска придется просматривать все таблицы производных классов, что потребует большого количества обращений к базе данных (либо выполнения весьма странных соединений).

Запомните: все три упомянутых типовых решения могут спокойно сосуществовать в пределах одной иерархии. Например, вы можете отобразить один или два производных класса посредством **наследования с таблицами для каждого конкретного класса**, а к остальным применить **наследование с одной таблицей**.

Пример: конкретные классы игроков (C#)

Рассмотрим реализацию модели, описанной в этом разделе. Как и в предыдущих примерах отображения иерархии наследования, я воспользуюсь фундаментальной концепцией **преобразователей наследования (Inheritance Mappers, 322)**, схема которой приведена на рис. 12.11.

Каждый преобразователь ссылается на таблицу, являющуюся источником данных для конкретного класса. Таблицы представлены объектами `DataTable` и хранятся в объекте ADO.NET `DataSet`.

```
class Mapper...

      public Gateway Gateway;
      private IDictionary identityMap = new Hashtable();
      public Mapper (Gateway gateway) {
         this.Gateway = gateway;
      }
      private DataTable table {
         get {return Gateway.Data.Tables[TableName];}
      }
      abstract public String TableName {get;}
```

Значением свойства `Data` класса `Gateway` является объект `DataSet`. Его содержимое может быть загружено путем выполнения соответствующих запросов.

```
class Gateway...

      public DataSet Data = new DataSet();
```

Каждый конкретный преобразователь должен ссылаться на таблицу, в которой хранятся данные соответствующего класса.

```
class CricketerMapper...

      public override String TableName {
         get {return "Cricketers";}
      }
```

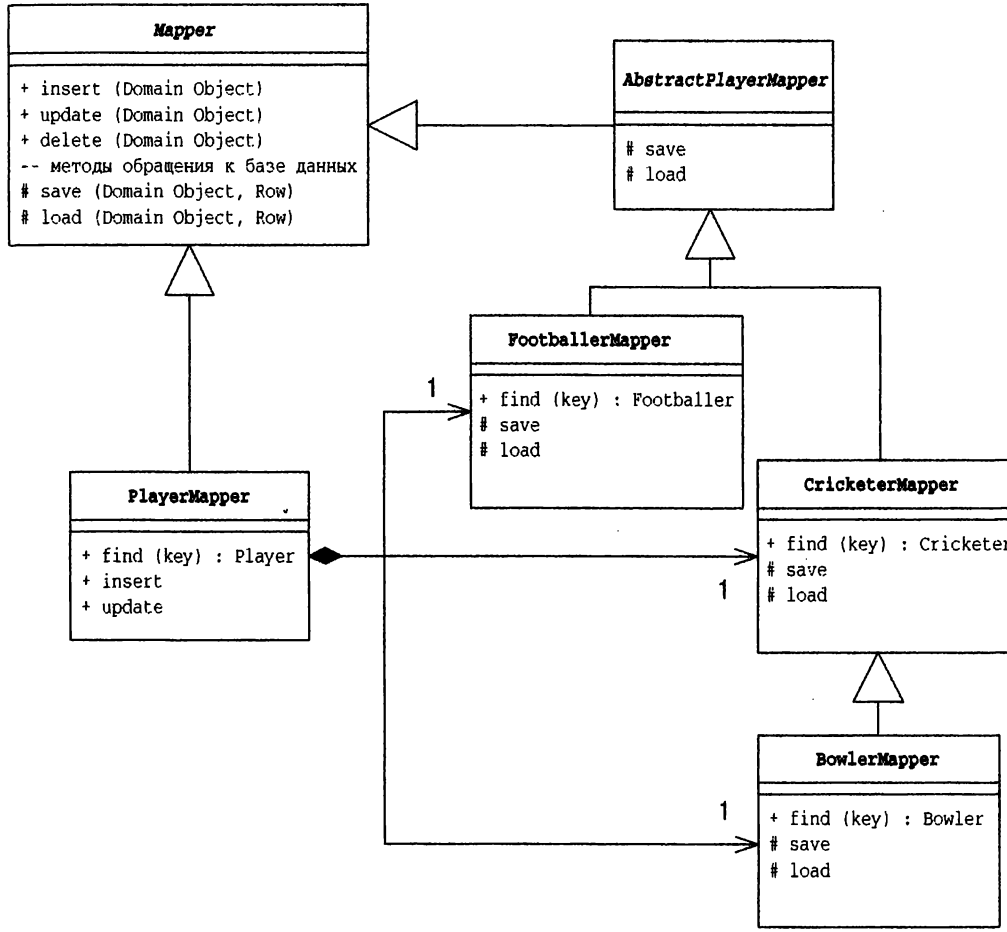


Рис. 12.11. Универсальная схема классов для **преобразователей наследования**

Класс `PlayerMapper` содержит по одному полю на каждый конкретный класс преобразователя.

```
class PlayerMapper...

      private BowlerMapper bmapper;
      private CricketerMapper cmapper;
      private FootballerMapper fmapper;
      public PlayerMapper (Gateway gateway) : base (gateway) {
```

```
        bmapper = new BowlerMapper(Gateway);
        cmapper = new CricketerMapper(Gateway);
        fmapper = new FootballerMapper(Gateway);
    }
```

Загрузка объекта из базы данных

У каждого конкретного класса преобразователя есть метод поиска, который возвращает объект по значению его первичного ключа.

```
class CricketerMapper...

    public Cricketer Find(long id) {
        return (Cricketer) AbstractFind(id);
    }
```

Абстрактный метод суперкласса находит строку с указанным идентификатором, создает новый объект домена нужного типа и заполняет его содержимым с помощью метода загрузки (последний будет описан немного позднее).

```
class Mapper...

    public DomainObject AbstractFind(long id) {
        DataRow row = FindRow(id);
        if (row == null) return null;
        else {
            DomainObject result = CreateDomainObject();
            Load(result, row);
            return result;
        }
    }
    private DataRow FindRow(long id) {
        String filter = String.Format("id = {0}", id);
        DataRow[] results = table.Select(filter);
        if (results.Length == 0) return null;
        else return results[0];
    }
    protected abstract DomainObject CreateDomainObject();

class CricketerMapper...

    protected override DomainObject CreateDomainObject() {
        return new Cricketer();
    }
```

Фактическая загрузка объекта из базы данных осуществляется посредством нескольких методов загрузки: по одному на каждый класс преобразователя и на все его суперклассы.

```
class CricketerMapper...

    protected override void Load(DomainObject obj,
↳DataRow row) {
```

```
        base.Load(obj,row);
        Cricketer cricketer = (Cricketer) obj;
        cricketer.battingAverage = (double)row[
↳"battingAverage"];
    }

class AbstractPlayerMapper...
    protected override void Load(DomainObject obj,
↳DataRow row) {
        base.Load(obj, row);
        Player player = (Player) obj;
        player.name = (String)row["name"];
    }

class Mapper...

    protected virtual void Load(DomainObject obj,
↳DataRow row) {
        obj.Id = (int) row ["id"];
    }
```

Данная логика описывает выполнение поиска с использованием преобразователя для конкретного класса. Вместо этого можно воспользоваться преобразователем для суперкласса, а именно объектом `PlayerMapper`, который позволяет найти объект, в какой бы из таблиц тот ни находился. Поскольку все данные уже загружены в оперативную память (объект `DataSet`), можно поступить так, как показано ниже.

```
class PlayerMapper...

    public Player Find (long key) {
        Player result;
        result = fmapper.Find(key);
        if (result != null) return result;
        result = bmapper.Find(key);
        if (result != null) return result;
        result = cmapper.Find(key);
        if (result != null) return result;
        return null;
    }
```

Запомните: это можно делать только потому, что данные уже хранятся в оперативной памяти. Если бы для доступа к данным нам пришлось совершить три обращения к базе данных (или даже больше, в зависимости от количества производных классов), приведенный метод оказался бы крайне медленным. В этом случае могу порекомендовать выполнить соединение всех таблиц конкретных классов, что позволит извлечь нужные сведения за одно обращение к базе данных. К сожалению, большие соединения медленны сами по себе, поэтому придется провести несколько "черновых" тестирований, чтобы понять, что хорошо, а что плохо для вашего приложения. Кроме того, это будет внешнее соединение, которое помимо низкой скорости работы обладает весьма непонятным и непереносимым синтаксисом.

Обновление объекта

Метод обновления может быть описан в суперклассе `Mapper`.

```
class Mapper...

      public virtual void Update (DomainObject arg) {
         Save (arg, FindRow(arg.Id));
      }
```

Для обновления, как и для загрузки, используется группа методов сохранения (по одному на каждый класс преобразователя).

```
class CricketerMapper...

      protected override void Save(DomainObject obj,
↳DataRow row) {
         base.Save(obj, row);
         Cricketer cricketer = (Cricketer) obj;
         row["battingAverage"] = cricketer.battingAverage;
      }

class AbstractPlayerMapper...

      protected override void Save(DomainObject obj,
↳DataRow row) {
         Player player = (Player) obj;
         row["name"] = player.name;
      }
```

Объект `PlayerMapper` выбирает нужный конкретный преобразователь и делегирует ему выполнение обновления.

```
class PlayerMapper...

      public override void Update (DomainObject obj) {
         MapperFor(obj).Update(obj);
      }
      private Mapper MapperFor(DomainObject obj) {
         if (obj is Footballer)
            return fmapper;
         if (obj is Bowler)
            return bmapper;
         if (obj is Cricketer)
            return cmapper;
         throw new Exception("No mapper available");
      }
```

Вставка объекта

Операцию вставки можно рассматривать как разновидность обновления. Дополнительное поведение заключается в создании новой строки и может быть реализовано в суперклассе.

```
class Mapper...

      public virtual long Insert (DomainObject arg) {
         DataRow row = table.NewRow();
         arg.Id = GetNextID();
         row["id"] = arg.Id;
         Save (arg, row);
         table.Rows.Add(row);
         return arg.Id;
      }
```

И здесь класс `PlayerMapper` делегирует выполнение вставки нужному преобразователю.

```
class PlayerMapper...

      public override long Insert (DomainObject obj) {
         return MapperFor(obj).Insert(obj);
      }
```

Удаление объекта

Схема удаления совсем проста. Как и раньше, у нас есть метод, определенный в суперклассе.

```
class Mapper...

      public virtual void Delete(DomainObject obj) {
         DataRow row = FindRow(obj.Id);
         row.Delete();
      }
```

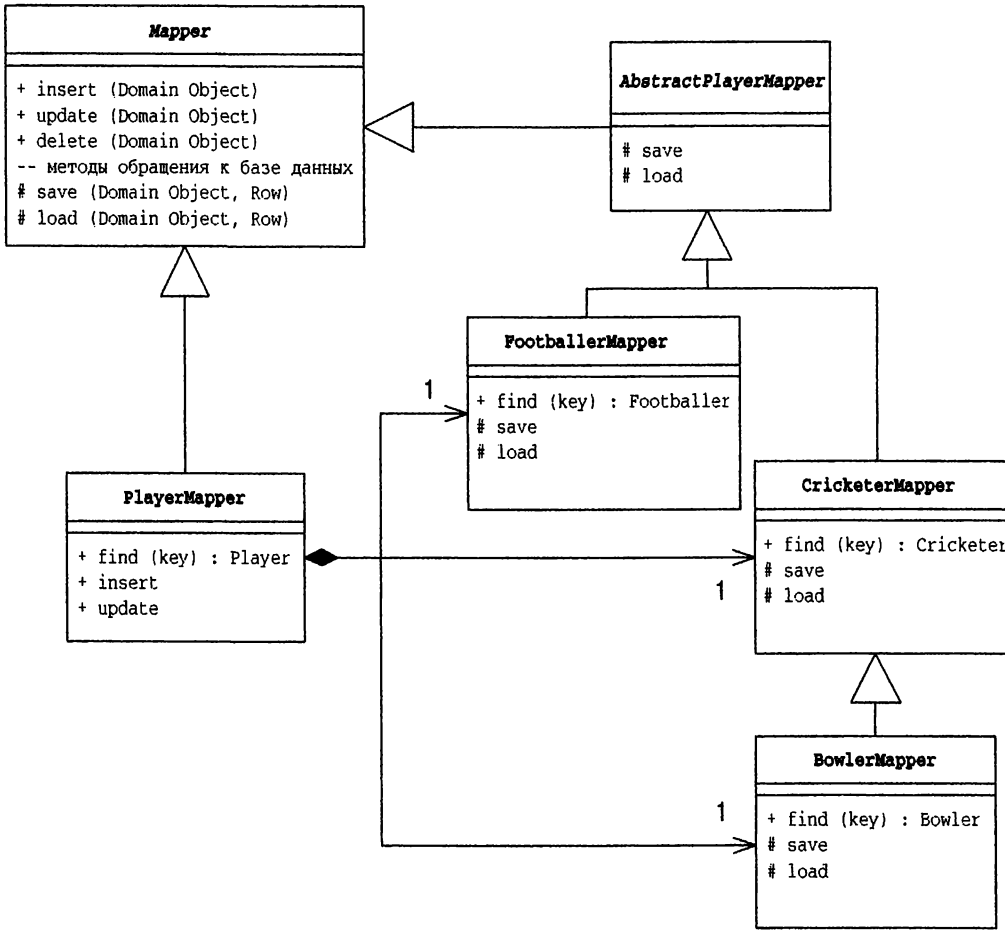
Есть также метод класса `PlayerMapper`, делегирующий выполнение удаления конкретному преобразователю.

```
class PlayerMapper...

      public override void Delete (DomainObject obj) {
         MapperFor(obj).Delete(obj);
      }
```

Преобразователи наследования (Inheritance Mappers)

Структура, предназначенная для организации преобразователей, которые работают с иерархиями наследования



При отображении объектно-ориентированной иерархии наследования на реляционную базу данных крайне важно минимизировать количество кода, необходимого для загрузки и сохранения содержимого базы данных. Кроме того, необходимо реализовать и абстрактное, и конкретное поведение, что позволит сохранять или загружать как экземпляр суперкласса, так и экземпляры производных классов.

Хотя детали поведения могут различаться в зависимости от выбранной схемы отображения (**наследование с одной таблицей (Single Table Inheritance, 297)**, **наследование с таблицами для каждого класса (Class Table Inheritance, 305)** и **наследование с таблицами для каждого конкретного класса (Concrete Table Inheritance, 313)**), общая структура остается одной и той же.

Принцип действия

Преобразователи можно организовать в иерархию, так, чтобы у каждого класса домена был свой преобразователь, который будет загружать и сохранять данные этого класса. В этом случае у нас есть одна точка, в которой можно изменить принцип отображения. Данный подход хорошо применять для конкретных преобразователей, которые "знают", как отображать конкретные объекты иерархии. Тем не менее иногда преобразователи нужны и для абстрактных классов. Это можно реализовать с помощью специальных преобразователей, которые находятся за пределами базовой иерархии, однако делегируют выполнение операций соответствующим конкретным преобразователям.

Чтобы попроще объяснить принцип действия **преобразователей наследования**, я начну с конкретных преобразователей. В нашей модели есть конкретные преобразователи для объектов `Footballer` (футболист), `Cricketer` (игрок в крикет) и `Bowler` (игрок в боулинг). Базовое поведение этих преобразователей включает в себя методы поиска, вставки, обновления и удаления.

Методы поиска объявлены в конкретных производных классах, поскольку они должны возвращать экземпляр конкретного класса. Другими словами, метод класса `BowlerMapper` должен возвращать не абстрактный объект, а объект `Bowler` (игрок в боулинг). Большинство объектно-ориентированных языков программирования не позволяют изменять объявленный тип метода, поэтому невозможно наследовать метод поиска от абстрактного класса и одновременно присвоить этому методу конкретный тип. Разумеется, можно возвращать абстрактный объект, однако это вынудит пользователя класса выполнять обратное приведение объекта к нужному типу, чего следует всячески избегать. (Данная проблема не возникает в языках с возможностью динамической типизации.)

Основное поведение метода поиска заключается в том, чтобы найти заданную строку базы данных, создать экземпляр объекта корректного типа (это определяется производным классом) и загрузить в новый объект найденные данные. Метод загрузки реализован в каждом преобразователе иерархии, выполняющем загрузку данных для соответствующего класса. Это значит, что метод загрузки класса `BowlerMapper` загружает данные, специфичные для класса `Bowler`, после чего вызывает метод суперкласса, чтобы тот загрузил данные, специфичные для класса `Cricketer`. Упомянутый суперкласс вызывает метод своего суперкласса и т.д.

Методы обновления и вставки выполняются по одной и той же схеме с использованием метода сохранения. В данном случае интерфейс этих методов можно определить в суперклассе, а точнее, в **супертипе слоя (Layer Supertype, 491)**. Метод вставки создает новую строку и затем сохраняет в ней данные объекта домена, используя для этого методы сохранения. Метод обновления просто сохраняет данные, используя для этого все те же методы сохранения. Последние выполняются аналогично методам загрузки: каждый класс сохраняет специфические для него данные и вызывает метод своего суперкласса.

Данная схема значительно упрощает написание преобразователей для сохранения информации об определенной части иерархии. Следующим шагом является реализация

загрузки и сохранения объекта абстрактного класса — в нашем случае это класс `Player` (игрок). Естественно, первое, что предполагается сделать, — разместить соответствующие методы прямо в суперклассе, однако это вовсе не так хорошо, как кажется. В отличие от конкретных преобразователей, которые могут просто использовать методы вставки и обновления абстрактного класса, классу `PlayerMapper` нужно переопределять эти методы, чтобы вызвать нужный конкретный преобразователь. Результатом подобных действий обычно является "лихое" сочетание обобщений и конкретизаций, которое способно свести с ума даже самых закаленных разработчиков.

Я предпочитаю разбивать преобразователи на два отдельных класса. В этом случае класс `AbstractPlayerMapper` будет отвечать за загрузку и сохранение данных о конкретном игроке. Это абстрактный класс, поведение которого может быть использовано только конкретными преобразователями. В свою очередь, класс `PlayerMapper` будет применяться в качестве интерфейса для операций над игроками вообще. Данный класс содержит собственный метод поиска и переопределяет методы обновления и вставки. Во всех этих случаях функции класса `PlayerMapper` заключаются в том, чтобы определить, какой конкретный преобразователь должен обрабатывать поставленное задание, и делегировать ему выполнение этого задания.

Хотя описанная схема является универсальной и может быть применена к любому типу отображения иерархии наследования, детали реализации будут слишком различны. Поэтому я не могу привести примеры кода для иллюстрации данного типового решения. Хорошие примеры для каждой схемы отображения иерархии наследования можно найти в разделах, посвященных типовым решениям **наследование с одной таблицей, наследование с таблицами для каждого класса** и **наследование с таблицами для каждого конкретного класса**.

Назначение

Данная схема может применяться для всех типов отображения иерархии наследования на реляционную базу данных. Возможные альтернативы включают в себя дублирование кода абстрактного преобразователя во всех конкретных преобразователях либо вынесение содержимого класса `PlayerMapper` в класс `AbstractPlayerMapper`. Первый из этих способов иначе как "гнусным преступлением против человечества" и не назовешь. Второй способ более реален, однако использовать общий (и довольно запутанный) класс `PlayerMapper` крайне неудобно. Вообще говоря, придумать действительно удачную альтернативу **преобразователям наследования** практически невозможно.

Глава 13

Типовые решения объектно-реляционного отображения с использованием метаданных

Отображение метаданных (Metadata Mapping)

Хранит описание деталей объектно-реляционного отображения в виде метаданных



Большая часть кода, касающегося объектно-реляционного отображения, задает соответствие между столбцами таблиц базы данных и полями объектов. Подобные фрагменты кода довольно однообразны и повторяются во многих местах. Типовое решение **отображение метаданных** позволяет разработчикам описать отображение в виде простой таблицы соответствий, которая затем может быть использована универсальным кодом для получения деталей относительно считывания, вставки и обновления данных.

Принцип действия

Прежде чем приступать к реализации **отображения метаданных**, необходимо решить, как метаданные будут использоваться кодом программы. Существует два основных варианта: генерация кода и метод отражения.

При использовании метода *генерации кода (code generation)* разработчик пишет программу, которая принимает на вход метаданные и возвращает исходный код классов, выполняющих отображение. Эти классы ничем не отличаются от написанных вручную, однако на деле являются полностью искусственным "продуктом", автоматически сгенерированным в процессе сборки (как правило, перед самым началом компиляции). Полученные классы преобразователей развертываются серверным кодом.

Если вы используете генерацию кода, убедитесь, что она полностью интегрирована в процесс сборки, какие бы сценарии в нем ни применялись. Сгенерированные классы никогда не должны дорабатываться вручную, а следовательно, не нуждаются в проверке программой контроля исходного кода.

При использовании метода *отражения (reflection)* рефлективная программа просматривает метаданные объекта, находит метод `setName` и выполняет его с соответствующим аргументом. Поскольку методы (и поля) воспринимаются программой как обычные данные, она может считывать имена полей и методов из файла с метаданными и затем использовать их для отображения. Обычно я не рекомендую использовать отражение — частично из-за его медлительности, однако в основном потому, что оно значительно затрудняет отладку кода. Несмотря на это, метод отражения прекрасно подходит для отображения на базу данных. Считывание имен полей и методов из файла позволяет в полной степени ощутить гибкость метода отражения.

Генерация кода представляет собой гораздо менее динамичный подход, поскольку любые изменения в отображении требуют перекомпиляции и нового развертывания, как минимум, этой части программного обеспечения. А применение метода отражения позволяет просто изменить файл с описанием отображения, и существующие классы будут использовать новые метаданные. Впрочем, как показывает практика, необходимость в изменении отображения возникает крайне редко, поскольку предполагает изменение кода или самой базы данных. Кроме того, современные среды разработки позволяют значительно упростить развертывание части приложения.

Слабым местом метода отражения является скорость. Значимость этой проблемы зависит от того, какую среду разработки вы используете — в некоторых средах медлительность рефлективных программ переходит всякие границы. Не следует, однако, забывать, что отражение осуществляется в контексте SQL-запроса, поэтому медленное выполнение отражения может быть совсем незаметным на фоне медленного выполнения удаленного вызова. Рекомендую попробовать отражение в своей среде разработки, чтобы узнать, насколько оно замедляет производительность.

Оба подхода довольно сложны в отладке. Выбор оптимального способа во многом зависит от того, насколько разработчик привык к сгенерированному или рефлективному коду. Сгенерированный код более явный, что облегчает отслеживание действий отладчика. Поэтому я предпочитаю именно генерацию кода и думаю, что она окажется более простой для недостаточно искушенных программистов (наверное, это заявление делает меня совсем неискушенным программистом).

В большинстве случаев метаданные хранятся в отдельном файле. В наши дни для описания метаданных используется XML, поскольку он предоставляет возможность

иерархического структурирования и вместе с тем освобождает от необходимости написания собственных анализаторов и других средств обработки. Специальный метод загрузки преобразует метаданные в структуру используемого языка программирования, после чего они могут применяться для генерации кода либо для выполнения отражения.

В некоторых случаях можно обойтись и без отдельного файла, поместив метаданные прямо в исходный код. Это избавит от необходимости применения анализатора, однако несколько усложнит редактирование метаданных.

Еще одной альтернативой является хранение метаданных в самой базе данных. Если схема базы данных будет изменена, информация об отображении будет всегда под рукой.

Выбирая способ хранения метаданных, можно с уверенностью пренебречь скоростью доступа и анализа. Если вы используете генерацию кода, доступ к метаданным и их анализ осуществляются только в процессе сборки и никак не влияют на выполнение программы. Если же вы используете метод отражения, доступ к метаданным и их анализ будут осуществляться и в процессе выполнения, но только однажды, а именно при запуске системы. После этого вы сможете воспользоваться представлением метаданных в оперативной памяти.

Одним из наиболее важных моментов является сложность метаданных. Общая схема объектно-реляционного отображения подразумевает хранение в метаданных огромного количества его характеристик. Между тем для большинства проектов могут применяться гораздо менее глобальные схемы отображений, что позволяет значительно упростить структуру метаданных. Вообще говоря, приложение хорошо усложнять по мере необходимости, а с применением метаданных добавлять новые возможности будет совсем нетрудно.

В 90% случаев простой схемы отображения метаданных оказывается более чем достаточно. Однако и из этого правила существуют исключения, способные изрядно потрепать нервы разработчиков. Для обработки этих довольно редких, но все же иногда встречающихся исключений схему отображения с использованием метаданных приходится существенно усложнять. В качестве неплохой альтернативы можно предложить переопределение универсального кода с помощью методов производных классов, написанных вручную. Последние могут быть произведены от сгенерированных классов или же классов, выполняющих отражение. К сожалению, поскольку данные классы предназначены для обработки весьма специфических случаев, каких-либо универсальных принципов переопределения не существует. Все, что я могу порекомендовать, — это действовать соответственно ситуации. Если вы почувствуете необходимость в описании частного случая, измените сгенерированный или рефлективный код, чтобы вынести поведение, которое должно быть переопределено, в отдельный метод и затем переопределите этот метод в производных классах, предназначенных для обработки частных случаев.

Назначение

Использование **отображения метаданных** позволяет значительно сократить объем работы, необходимый для реализации отображения на базу данных. Тем не менее подготовка **отображения метаданных** к работе требует некоторых усилий. Следует отметить и то, что, хотя это решение прекрасно подходит для большинства конкретных ситуаций, существуют исключения, которые значительно усложняют использование метаданных.

Как и следовало ожидать, данное типовое решение весьма охотно используют в коммерческих средствах объектно-реляционного отображения, поскольку удачная реализация **отображения метаданных** позволяет существенно снизить дальнейшие затраты на разработку.

Если вы проектируете собственную систему, попробуйте оценить преимущества и недостатки **отображения метаданных** сами. Сравните затраты на добавление новых отображений вручную с затратами на разработку этого типового решения. Если вы собираетесь использовать отражение, проверьте, как оно влияет на производительность; иногда отражение существенно замедляет скорость работы приложения, а иногда нет. Оценить значимость этой проблемы можно только с помощью собственных измерений.

Объем "ручной" работы можно значительно сократить, создав хороший **супертип слоя (Layer Supertype, 491)**, который будет содержать в себе все общее поведение преобразователей. В этом случае вам останется добавить всего несколько методов для каждого отображения, что можно еще более упростить посредством **отображения метаданных**.

Отображение метаданных может конфликтовать с рефакторингом, особенно если последний происходит с использованием автоматизированных средств. Если вы измените имя закрытого поля, это может неожиданно привести к отказу приложения. Даже автоматизированные средства рефакторинга будут не в состоянии обнаружить имя поля в дебрях XML-метаданных. В этом плане использование генерации кода немного проще, поскольку механизмы поиска могут отследить использование поля. Тем не менее при повторной генерации кода все автоматизированные обновления будут утеряны. Разумеется, программа может предупредить вас о возможных проблемах, однако вносить изменения в метаданные придется самому. Кроме того, при использовании метода отражения вы вообще не получите никаких предупреждений.

С другой стороны, **отображение метаданных** может упростить рефакторинг базы данных, поскольку метаданные представляют собой спецификацию ее интерфейса. Таким образом, для согласования программы с изменениями базы данных достаточно изменить метаданные.

Пример: использование метаданных и метода отражения (Java)

В большинстве примеров этой книги используется явный код, поскольку в нем проще разобраться. Тем не менее явное описание действий приводит к большому количеству однообразных и повторяющихся фрагментов кода, а наличие повторяющихся фрагментов — верный признак того, что с программой что-то не так. Чтобы избежать подобных проблем, можно воспользоваться метаданными.

Хранение метаданных

При реализации **отображения метаданных** прежде всего необходимо решить, в каком виде будут храниться метаданные. В нашем примере для хранения метаданных будут использоваться два класса. Класс `DataMap` описывает соответствие между классом приложения и таблицей базы данных. Это очень простое отображение, однако для наших целей оно подойдет как нельзя лучше.

```
class DataMap...

   private Class domainClass;
```

```
   private String tableName;
   private List columnMaps = new ArrayList();
```

Класс DataMap содержит коллекцию объектов ColumnMap, описывающих отображение столбцов таблицы на поля объекта.

```
class ColumnMap...

   private String columnName;
   private String fieldName;
   private Field field;
   private DataMap dataMap;
```

Как видите, данное отображение нельзя назвать слишком сложным. В нашем примере я воспользовался стандартными отображениями типов Java. Это значит, что при отображении между полями объекта и столбцами таблицы не происходит преобразования данных из одного типа в другой. Кроме того, при подобной организации отображения одна таблица базы данных должна соответствовать одному классу приложения и наоборот.

Итак, нами созданы структуры, которые будут применяться для хранения метаданных. Теперь необходимо подумать над тем, как эти структуры заполнить. В данном примере я собираюсь заполнять их кодом Java с помощью специализированных классов преобразователей. Последние могут показаться несколько странными, однако они дают возможность воспользоваться основным преимуществом метаданных — избежать повторяющегося кода.

```
class PersonMapper...

   protected void loadDataMap(){
      dataMap = new DataMap (Person.class, "people");
      dataMap.addColumn ("lastname", "varchar", "lastName");
      dataMap.addColumn ("firstname", "varchar", "firstName");
      dataMap.addColumn ("number_of_dependents", "int",
↳"numberOfDependents");
   }
```

Как вы, должно быть, заметили, я добавил к классу ColumnMap ссылку на поле объекта. Вообще говоря, это было сделано в целях оптимизации, чтобы не подсчитывать значение поля каждый раз, когда оно понадобится. Тем не менее наличие подобной ссылки значительно сокращает количество последующих обращений к базе данных, что немаловажно для моего маленького ноутбука.

```
class ColumnMap...

   public ColumnMap(String columnName, String fieldName,
↳DataMap dataMap) {
      this.columnName = columnName;
      this.fieldName = fieldName;
      this.dataMap = dataMap;
      initField();
   }
```

```
   private void initField() {
      try {
      field = dataMap.getDomainClass().getDeclaredField(
↳getFieldName());
      field.setAccessible(true);
      } catch (Exception e) {
         throw new ApplicationException (
↳"unable to set up field:" + fieldName, e);
      }
   }
```

Думаю, вам несложно понять, как написать методы для загрузки метаданных из базы данных или файла XML. Возможно, какие-то небольшие тонкости здесь все-таки есть, однако я выношу их на ваше самостоятельное рассмотрение.

Теперь, когда отображения определены, я могу ими воспользоваться. Преимуществом работы с метаданными является то, что все методы, предназначенные для манипулирования объектами, находятся в суперклассе, поэтому мне не нужно писать код отображения конкретных классов, как это делалось в предыдущих примерах.

Поиск по идентификатору

Для начала опишем выполнение поиска по идентификатору.

```
class Mapper...

   public Object findObject (Long key) {
      if (uow.isLoaded(key)) return uow.getObject(key);
      String sql = "SELECT" + dataMap.columnList() + " FROM
         " + dataMap.getTableName() + " WHERE ID =? ";
      PreparedStatement stmt = null;
      ResultSet rs = null;
      DomainObject result = null;
      try {
         stmt = DB.prepare(sql);
         stmt.setLong(1, key.longValue());
         rs = stmt.executeQuery();
         rs.next();
         result = load(rs);
      } catch (Exception e){throw new ApplicationException (e);
      } finally {DB.cleanUp(stmt, rs);
      }
     return result;
   }
   private UnitOfWork uow;
   protected DataMap dataMap;

class DataMap...

   public String columnList() {
      StringBuffer result = new StringBuffer(" ID");
      for (Iterator it = columnMaps.iterator(); it.hasNext();) {
         result.append(",");
         ColumnMap columnMap = (ColumnMap)it.next();
         result.append(columnMap.getColumnName());
```

```
        }
        return result.toString();
    }
    public String getTableName(){
        return tableName;
    }
```

В данном случае построение выражения SELECT выполняется более динамично, чем в других примерах, однако подготовить его для кэширования базой данных тоже не мешает. В случае необходимости список столбцов может быть определен в процессе построения выражения и затем кэширован, поскольку за время жизни объекта DataMap имена столбцов изменены не будут. В нашем примере для обработки сеанса соединения с базой данных используется **единица работы (Unit of Work, 205)**.

Как и в других примерах этой книги, я отделил выполнение загрузки от поиска, поэтому один и тот же метод загрузки может применяться различными методами поиска.

```
class Mapper...

    public DomainObject load(ResultSet rs)
            throwsInstantiationException, IllegalAccessException,
↪SQLException
    {
        Long key = new Long(rs.getLong("ID"));
        if (uow.isLoaded(key)) return uow.getObject(key);
        DomainObject result = (DomainObject)
↪dataMap.getDomainClass().newInstance();
        result.setID(key);
        uow.registerClean(result);
        loadFields(rs, result);
        return result;
    }
    private void loadFields(ResultSet rs, DomainObject result)
↪throws SQLException {
        for (Iterator it = dataMap.getColumns(); it.hasNext();) {
            ColumnMap columnMap = (ColumnMap)it.next();
            Object columnValue = rs.getObject(
↪columnMap.getColumnName());
            columnMap.setField(result, columnValue);
        }
    }

class ColumnMap...

    public void setField(Object result, Object columnValue) {
        try {
            field.set(result, columnValue);
        } catch (Exception e) { throw new ApplicationException (
↪"Error in setting " + fieldName, e);
        }
}
```

Перед вами классический пример отражения. Последовательно просматривая список объектов `ColumnMap`, можно использовать каждый из них для загрузки соответствующего поля объекта домена. Я выделил загрузку полей в отдельный метод `loadFields`, чтобы показать, как данный код может быть расширен для более сложных случаев. Если у вас есть класс и таблица, отображение которых не может быть описано в терминах простых метаданных, можно просто переопределить метод `loadFields` в производном классе преобразователя, сделав его код настолько сложным, насколько это потребуется. Создание отдельного метода, который может быть переопределен в нестандартных ситуациях, — весьма распространенный прием, который часто применяется в работе с метаданными. Это гораздо проще, чем создавать безумно сложные структуры метаданных только для того, чтобы изредка обрабатывать два-три частных случая.

Разумеется, если у вас уже есть производный класс, можете воспользоваться им, чтобы избежать необходимости обратного приведения объекта к нужному типу.

```
class PersonMapper...

   public Person find(Long key) {
      return (Person) findObject(key);
   }
```

Запись в базу данных

Все обновления выполняются с помощью одной и той же операции.

```
class Mapper...

   public void update (DomainObject obj) {
      String sql = "UPDATE " + dataMap.getTableName() +
↳dataMap.updateList() + " WHERE ID = ? ";
      PreparedStatement stmt = null;
      try {
         stmt = DB.prepare(sql);
         int argCount = 1;
         for (Iterator it = dataMap.getColumns();
↳it.hasNext();) {
            ColumnMap col = (ColumnMap) it.next();
            stmt.setObject(argCount++, col.getValue(obj));
         }
         stmt.setLong(argCount, obj.getID().longValue());
         stmt.executeUpdate();
         } catch (SQLException e) {throw new
↳ApplicationException (e);
         } finally {DB.cleanUp(stmt);
         }
   }

class DataMap...

   public String updateList() {
      StringBuffer result = new StringBuffer(" SET ");
      for (Iterator it = columnMaps.iterator(); it.hasNext();) {
```

```
            ColumnMap columnMap = (ColumnMap)it.next();
            result.append(columnMap.getColumnName());
            result.append("=?,");
         }
         result.setLength(result.length() - 1);
         return result.toString();
      }
      public Iterator getColumns() {
         return Collections.unmodifiableCollection(
↳columnMaps).iterator();
      }

class ColumnMap...

      public Object getValue (Object subject) {
         try {
            return field.get(subject);
         } catch (Exception e) {
            throw new ApplicationException (e);
         }
}
```

Вставка выполняется по той же схеме, что и обновление.

```
class Mapper...

      public Long insert (DomainObject obj) {
         String sql ="INSERT INTO "+ dataMap.getTableName() +
↳" VALUES (?" + dataMap.insertList() + ")";
         PreparedStatement stmt = null;
         try {
            stmt = DB.prepare(sql);
            stmt.setObject(1, obj.getID());
            int argCount = 2;
            for (Iterator it = dataMap.getColumns();
↳it.hasNext();) {
               ColumnMap col = (ColumnMap) it.next();
               stmt.setObject(argCount++, col.getValue(obj));
            }
            stmt.executeUpdate();
            } catch (SQLException e) {throw new
↳ApplicationException (e);
            } finally {DB.cleanUp(stmt);
            }
            return obj.getID();
      }

class DataMap...

      public String insertList() {
         StringBuffer result = new StringBuffer();
         for (int i = 0; i < columnMaps.size(); i++) {
            result.append(",");
```

```
            result.append("?");
        }
        return result.toString();
```

Извлечение множества объектов

Существует несколько способов извлечь множество объектов посредством одного запроса. Если вы хотите реализовать универсальный запрос в суперклассе преобразователя, можете создать метод поиска, который будет принимать в качестве аргумента SQL-оператор `WHERE`.

```
class Mapper...

    public Set findObjectsWhere (String whereClause) {
        String sql = "SELECT" + dataMap.columnList() +
↳" FROM " + dataMap.getTableName() + " WHERE " + whereClause;
        PreparedStatement stmt = null;
        ResultSet rs = null;
        Set result = new HashSet();
        try {
            stmt = DB.prepare(sql);
            rs = stmt.executeQuery();
            result = loadAll(rs);
        } catch (Exception e) {
            throw new ApplicationException (e);
        } finally {DB.cleanUp(stmt, rs);
        }
    return result;
    }
    public Set loadAll(ResultSet rs) throws SQLException,
↳InstantiationException, IllegalAccessException {
        Set result = new HashSet();
        while (rs.next()) {
            DomainObject newObj = (DomainObject)
                dataMap.getDomainClass().newInstance();
            newObj = load (rs);
            result.add(newObj);
        }
    return result;
}
```

Вместо этого можете реализовать специальные методы поиска в производных классах преобразователей.

```
class PersonMapper...

    public Set findLastNamesLike (String pattern) {
        String sql =
              "SELECT" + dataMap.columnList() +
              " FROM " + dataMap.getTableName() +
              " WHERE UPPER(lastName) like UPPER(?)";
        PreparedStatement stmt = null;
        ResultSet rs = null;
```

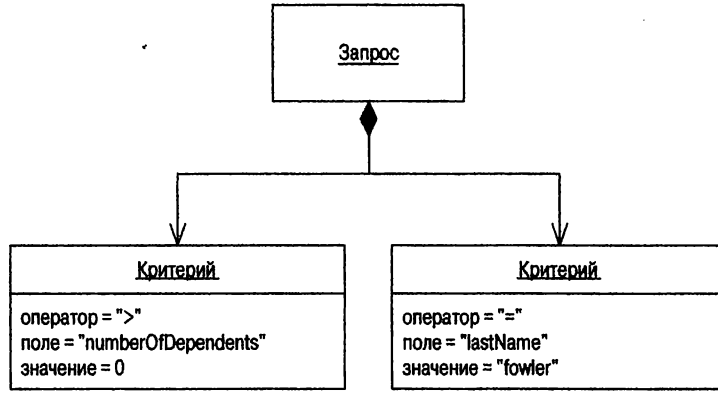
```
        try {
            stmt = DB.prepare(sql);
            stmt.setString(1, pattern);
            rs = stmt.executeQuery();
            return loadAll(rs);
        } catch (Exception e) {throw new ApplicationException (e);
        } finally {DB.cleanUp(stmt,rs);
        }
}
```

И наконец, для построения универсальных решений можно воспользоваться **объектом запроса (Query Object, 335)**.

Огромным преимуществом использования метаданных является простота добавления к отображению новых таблиц и классов. Все, что потребуется, — это предоставить метод загрузки метаданных, а также любые специализированные функции поиска, которые вы сочтете нужным.

Объект запроса (Query Object)

Объект, представляющий запрос к базе данных



Некоторые аспекты языка SQL довольно сложны, и не все разработчики хорошо их знают. Кроме того, для формирования запросов, необходимо иметь четкое представление о схеме базы данных. Чтобы избежать подобных проблем, можно создать специализированные методы поиска, которые будут "скрывать" SQL-выражения внутри параметризованных методов, однако это значительно усложнит формирование случайных, нерегламентированных запросов (ad hoc queries). Кроме того, при изменении схемы базы данных этот подход приведет к необходимости дублирования изменений во всех SQL-выражениях.

Объект запроса — это разновидность типового решения **интерпретатор (Interpreter)** [20], т.е. структура объектов, формирующих SQL-запрос. Чтобы создать такой запрос, достаточно сослаться на имена классов и полей, а не на имена таблиц и столбцов. В этом

случае формирование запросов может происходить независимо от схемы базы данных, а все изменения программы, связанные с изменениями схемы, будут локализованы в одном месте.

Принцип действия

Как уже отмечалось, **объект запроса** — это разновидность типового решения **интерпретатор**, предназначенная для представления SQL-запросов. Основная функция **объекта запроса** заключается в том, чтобы предоставить клиенту возможность формировать запросы различных типов и преобразовывать полученные структуры объектов в соответствующие SQL-выражения.

Чтобы представлять запросы различных типов, **объект запроса** должен обладать определенной гибкостью. Впрочем, зачастую приложения не нуждаются в полной мощи языка SQL, благодаря чему **объект запроса** можно сделать довольно простым. Разумеется, он не сможет применяться для представления всех и вся, однако его вполне хватит, чтобы удовлетворить конкретные нужды вашего приложения. Более того, усовершенствовать **объект запроса** в случае необходимости не сложнее, чем написать с "нуля". Поэтому рекомендую создавать **объект запроса** с минимальной функциональностью, достаточной для удовлетворения ваших текущих потребностей, и усложнять его по мере роста этих потребностей.

Особенностью **объекта запроса** является возможность представления запросов в терминах объектов приложения, а не базы данных. Другими словами, вместо имен таблиц и столбцов для составления запросов можно применять имена объектов и полей. Это не обязательно, если объектная модель и база данных имеют одну и ту же структуру, однако может весьма пригодиться, если они различны. Для обеспечения подобной возможности **объект запроса** должен "знать", как структура базы данных отображается на объектную модель, что, несомненно, требует применения **отображения метаданных (Metadata Mapping, 325)**.

Если **объект запроса** предназначается для использования с несколькими базами данных, его можно спроектировать таким образом, чтобы он генерировал различные SQL-запросы в зависимости от используемой базы данных. На самом простом уровне он может учитывать все те досадные различия в синтаксисе SQL, которые способны испортить жизнь при переходе к другой базе данных. Более серьезный уровень предполагает использование разных отображений для работы с одними и теми же классами, предназначенными для моделирования различных схем баз данных.

Менее очевидное применение **объекта запроса** — это уменьшение количества запросов к базе данных. Если вы заметили, что подобный запрос уже выполнялся в течение текущего сеанса соединения с базой данных, то можете использовать этот запрос, чтобы извлечь нужные сведения из **коллекции объектов (Identity Map, 216)** и таким образом избежать дополнительного обращения к базе данных. Более утонченный подход предполагает определение того, что запрос является частным случаем уже выполненного запроса, например что он такой же, как и предыдущий запрос, только имеет дополнительное выражение, присоединенное с помощью оператора `AND`.

Методы реализации подобных возможностей выходят за рамки данной книги, однако они относятся к той категории возможностей, которые обеспечиваются средствами объектно-реляционного отображения.

Как разновидность **объекта запроса** можно рассматривать запрос, определяемый экземпляром объекта домена. Например, вы можете создать экземпляр объекта `Person` ("сотрудник"), атрибут которого `lastName` ("фамилия") будет иметь значение `Fowler`, а все остальные атрибуты будут иметь значение `NULL`. Обрабатывая полученный объект как **объект запроса**, вы получаете запрос, который возвращает из базы данных сведения обо всех сотрудниках с фамилией Fowler. Данный метод весьма прост и удобен в применении, однако не подходит для выполнения сложных запросов.

Назначение

Область применения **объектов запроса** довольно сложна, поэтому они не используются в приложениях, у которых есть собственный слой источника данных. В действительности **объекты запроса** нужны только тогда, когда вы используете **модель предметной области (Domain Model, 140)** и **преобразователь данных (Data Mapper, 187)**; при этом, чтобы извлечь из **объектов запроса** реальную пользу, понадобится и **отображение метаданных**.

Объекты запроса не нужны и тогда, когда разработчики хорошо знают SQL. В этом случае многие детали схемы базы данных могут быть скрыты в специализированных методах поиска.

Настоящие преимущества **объектов запроса** раскрываются при наличии более сложных потребностей: инкапсуляции схем баз данных, поддержки работы с несколькими базами данных и оптимизации количества запросов. С помощью команды очень опытных и умелых разработчиков данное типовое решение можно реализовать и самостоятельно, однако в большинстве случаев те, кто использует **объекты запроса**, предпочитают обращаться к коммерческим средствам. Что касается меня, то я тоже отдаю предпочтение готовым продуктам.

Несмотря на сказанное выше, вы можете написать простой **объект запроса**, который будет прекрасно удовлетворять нужды вашего приложения, не требуя понимания всех сложностей реализации. Главное — максимально урезать функциональность **объекта запроса**, оставив только те возможности, которые вам действительно нужны.

Дополнительные источники информации

Пример **объекта запроса** можно найти в разделе книги [2], посвященном обсуждению разновидностей типового решения **интерпретатор**. Кроме того, **объект запроса** тесно связан с типовым решением **спецификация (Specification)**, описанным в [14, 15].

Пример: простой объект запроса (Java)

Рассмотрим пример простого **объекта запроса**, который не очень полезен в реальных ситуациях, однако вполне способен проиллюстрировать принцип работы данного типового решения. Наш **объект запроса** будет формировать запросы к одной таблице базы данных на основе набора критериев, соединенных оператором `AND` (в математике это называется конъюнкцией элементарных предикатов).

Формирование запросов будет осуществляться в терминах объектов домена, а не структуры базы данных. Таким образом, **объект запроса** "знает" класс, соответствующий таблице, к которой ставятся запросы, а также коллекцию критериев, соответствующих вариантам SQL-оператора `WHERE`.

```
class QueryObject...

   private Class klass;
   private List criteria = new ArrayList();
```

Простой критерий — это критерий, который сравнивает значение поля с заданным значением с помощью указанного SQL-оператора.

```
class Criteria...

   private String sqlOperator;
   protected String field;
   protected Object value;
```

Чтобы облегчить создание нужных критериев, я написал несколько соответствующих методов.

```
class Criteria...

   public static Criteria greaterThan(String fieldName,
↳int value) {
      return Criteria.greaterThan(fieldName,
↳new Integer(value));
   }
   public static Criteria greaterThan(String fieldName,
↳Object value) {
      return new Criteria(" > ", fieldName, value);
   }
   private Criteria(String sql, String field, Object value) {
      this.sqlOperator = sql;
      this.field = field;
      this.value = value;
   }
```

Теперь я могу извлечь из базы данных записи обо всех сотрудниках, имеющих подчиненных. Для этого достаточно создать критерий, приведенный ниже.

```
class Criteria...

      QueryObject query = new QueryObject(Person.class);
      query.addCriteria(Criteria.greaterThan(
↳"numberOfDependents", 0));
```

Предположим, у меня есть следующий объект `Person`:

```
class Person...

   private String lastName;
   private String firstName;
   private int numberOfDependents;
```

Тогда я могу извлечь записи обо всех сотрудниках, имеющих подчиненных, путем формирования запроса для объекта `Person` и добавления к нему только что созданного критерия.

```
      QueryObject query = new QueryObject(Person.class);
      query.addCriteria(Criteria.greaterThan(
↳"numberOfDependents", 0));
```

Этого достаточно, чтобы описать запрос. Теперь **объект запроса** должен преобразовать полученное выражение в SQL-оператор SELECT. В данном примере я буду исходить из предположения, что класс преобразователя Mapper поддерживает метод, выполняющий поиск объектов по заданному SQL-выражению WHERE.

```
class QueryObject...

   public Set execute(UnitOfWork uow) {
      this.uow = uow;
      return uow.getMapper(klass).findObjectsWhere(
↳generateWhereClause());
   }

class Mapper...

   public Set findObjectsWhere (String whereClause) {
      String sql = "SELECT" + dataMap.columnList() + " FROM " +
↳dataMap.getTableName() +" WHERE " + whereClause;
      PreparedStatement stmt = null;
      ResultSet rs = null;
      Set result = new HashSet();
      try {
         stmt = DB.prepare(sql);
         rs = stmt.executeQuery();
         result = loadAll(rs);
      } catch (Exception e) {
         throw new ApplicationException (e);
      } finally {DB.cleanUp(stmt, rs);
      }
      return result;
   }
```

В этом примере я использую **единицу работы (Unit of Work, 205)**, которая содержит преобразователи, индексированные по имени класса, а также преобразователь Mapper, использующий **отображение метаданных**. Код этого преобразователя такой же, как и в примере с **отображением метаданных**, поэтому я не стал повторять его здесь.

Для генерации SQL-выражения WHERE **объект запроса** последовательно просматривает список критериев и соединяет их в одну строку с помощью операторов AND.

```
class QueryObject...

   private String generateWhereClause() {
      StringBuffer result = new StringBuffer();
      for (Iterator it = criteria.iterator(); it.hasNext();) {
         Criteria c = (Criteria)it.next();
         if (result.length()!= 0)
            result.append(" AND ");
         result.append(c.generateSql(
↳uow.getMapper(klass).getDataMap()));
```

```
      }
      return result.toString();
   }

class Criteria...

   public String generateSql(DataMap dataMap) {
      return dataMap.getColumnForField(field) +
↳sqlOperator + value;
   }

class DataMap...

   public String getColumnForField (String fieldName) {
      for (Iterator it = getColumns(); it.hasNext();) {
         ColumnMap columnMap = (ColumnMap)it.next();
         if (columnMap.getFieldName().equals(fieldName))
            return columnMap.getColumnName();
      }
      throw new ApplicationException ("Unable to find
↳column for " + fieldName);
   }
```

Помимо критериев с простыми SQL-операторами, можно создавать и более сложные классы критериев. Давайте рассмотрим запрос на поиск слов по заданному шаблону без учета регистра, например запрос, который должен возвратить сведения обо всех сотрудниках, чьи фамилии начинаются на букву "F". Мы можем сформировать объект запроса для всех сотрудников с фамилиями на букву "F", имеющих подчиненных.

```
      QueryObject query = new QueryObject(Person.class);
      query.addCriteria(Criteria.greaterThan(
↳"numberOfDependents", 0));
      query.addCriteria(Criteria.matches("lastName", "f%"));
```

Для этого понадобится другой класс критериев, позволяющий формировать более сложные выражения `WHERE`.

```
class Criteria...

   public static Criteria matches(String fieldName,
↳String pattern) {
     return new MatchCriteria(fieldName, pattern);
   }

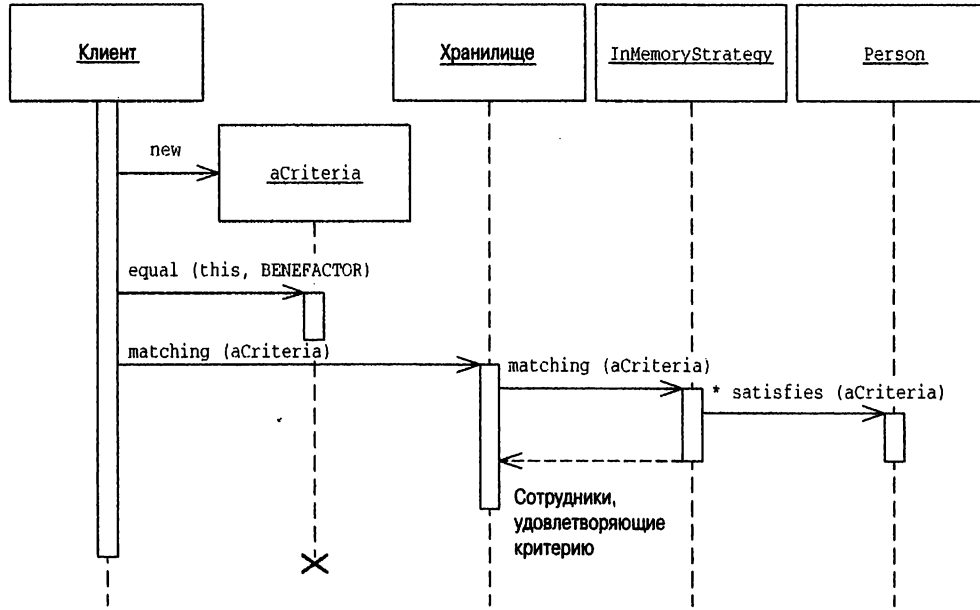
class MatchCriteria extends Criteria...

   public String generateSql(DataMap dataMap) {
      return "UPPER(" + dataMap.getColumnForField(field) + ")
↳LIKE UPPER('" + value + "')";
   }
```

Хранилище (Repository)

Эдвард Хайет и Роберт Ми

Выступает в роли посредника между слоем домена и слоем отображения данных, предоставляя интерфейс в виде коллекции для доступа к объектам домена



Системы с достаточно сложной моделью предметной области зачастую требуют применения слоя отображения, который бы отделил объекты домена от кода доступа к базе данных (как, например, слой **преобразователей данных (DataMapper, 187)**). В подобных системах имеет смысл наложить на слой отображения еще один слой абстракции, в котором будет находиться код, предназначенный для формирования запросов. Это становится особенно важным, когда приложение содержит слишком много классов домена или выполняет большое количество запросов. В этих случаях добавление дополнительного слоя позволяет избежать дублирования логики запросов.

Типовое решение **хранилище** располагается между слоем домена и слоем отображения данных, выполняя роль коллекции объектов домена в оперативной памяти. Клиентские объекты формируют спецификации запросов и отсылают их в **хранилище** для последующего удовлетворения. В **хранилище** можно добавлять и удалять объекты так же, как и в обычную коллекцию, а инкапсулированный в нем код отображения будет выполнять все необходимые операции. Идея **хранилища** состоит в том, чтобы инкапсулировать в себе множество объектов, содержащихся в источнике данных, и операции, которые над ними выполняются, тем самым обеспечивая объектно-ориентированный

аспект слоя хранения. Помимо этого, наличие **хранилища** помогает достичь полной изоляции и четкой односторонней зависимости между слоем домена и слоем отображения данных.

Принцип действия

Хранилище — достаточно сложное типовое решение, в реализации которого используется несколько других типовых решений, описанных в книге. В действительности оно напоминает небольшой фрагмент объектно-ориентированной базы данных и этим сходно с **объектом запроса (Query Object, 335)**, который чаще приобретают в составе коммерческих средств объектно-реляционного отображения, чем разрабатывают сами. Тем не менее, если команда разработчиков приложила усилия и сконструировала **объект запроса**, к нему не так уж сложно добавить функции **хранилища**. Последнее в сочетании с **объектом запроса** существенно увеличивает возможности слоя объектно-реляционного отображения.

Несмотря на сложные механизмы, реализованные внутри **хранилища**, его интерфейс очень прост. Клиент создает объект критериев, указывая характеристики объектов, которые должны быть возвращены в результате выполнения запроса. Например, чтобы найти сотрудников с указанным именем, необходимо задать объект критериев, описывая каждый критерий примерно следующим образом: `criteria.equals(Person.LAST_NAME, "Fowler")` и `criteria.like(Person.FIRST_NAME, "M")`. Затем нужно вызвать метод `repository.matching(criteria)`, чтобы возвратить список сотрудников с фамилией Fowler, имя которых начинается на букву "M". Для большего удобства в абстрактном классе **хранилища** можно определить и другие методы; например, для поиска единственного соответствия было бы неплохо воспользоваться методом наподобие `soleMatch(criteria)`, который бы возвращал объект, а не коллекцию объектов. В число других распространенных методов входит метод поиска по идентификатору `byObjectId(Id)`, который совсем несложно реализовать, используя метод `soleMatch`.

Клиентский код, работающий с **хранилищем**, воспринимает его как простую коллекцию объектов домена, расположенную в оперативной памяти. Тот факт, что объекты домена на самом деле не находятся в **хранилище**, полностью скрыт от клиентских программ. Разумеется, клиентский код должен быть осведомлен о том, что "видимая" коллекция объектов (например, товаров, распространяемых по каталогу) может отображаться на таблицу с сотнями тысяч записей, и применение метода `all()` к хранилищу `ProductRepository` — не самая удачная идея.

Методы **хранилища** заменяют собой специализированные методы поиска **преобразователя данных**, выполняя извлечение объектов на основе спецификаций [14]. Сравните это с непосредственным использованием **объекта запроса**, при котором клиент создает объект критериев (простая разновидность типового решения **спецификация (Specification)**), добавляет этот критерий непосредственно к **объекту запроса** и выполняет запрос. При работе с **хранилищем** клиентский код формирует критерии и передает их **хранилищу** с просьбой возвратить объекты, которые подойдут под заданное описание. Таким образом, с точки зрения клиента, *выполнение* запроса заменяется *удовлетворением*, т.е. извлечением объектов, удовлетворяющих спецификации запроса. Возможно, кому-то это различие покажется чисто формальным, однако оно хорошо иллюстрирует описательную

природу взаимодействия объектов посредством **хранилища**, которая составляет значительную часть мощи последнего.

Внутренняя структура **хранилища** представляет собой сочетание **отображения метаданных (Metadata Mapping, 325)** с **объектом запроса**, которое применяется для автоматической генерации SQL-кода на основе заданных критериев. То, какой объект будет отвечать за добавление критериев в объект **запроса** — объект критериев, **объект запроса** или само **отображение метаданных**, полностью зависит от предпочтений разработчика.

Источник объектов для **хранилища** вовсе не обязательно должен быть реляционной базой данных. Это вполне допустимо, потому что **хранилище** легко приспособить к смене слоя отображения посредством специальных объектов стратегий. Поэтому такое типовое решение особенно хорошо подходит для систем, работающих с несколькими схемами баз данных или имеющих несколько источников объектов домена. Пригодится **хранилище** и для тестирования, которое весьма желательно проводить только на объектах, расположенных в оперативной памяти, чтобы избежать проблем со скоростью доступа.

Хранилище может сослужить хорошую службу в улучшении читабельности и ясности кода, интенсивно использующего механизм запросов. Например, система, основанная на применении обозревателя и содержащая множество страниц запросов, нуждается в четком механизме преобразования объектов `HttpRequest` в результаты запроса. При наличии **хранилища** коду обработчика запросов будет гораздо легче преобразовать объект `HttpRequest` в объект критериев; передача критериев соответствующему **хранилищу** потребует только двух-трех дополнительных строк кода.

Назначение

Использование **хранилища** в большой системе с множеством типов объектов домена и массой разнообразных запросов позволяет сократить количество кода, необходимое для обработки всех этих запросов. Как уже упоминалось, **хранилище** использует типовое решение **спецификация** (в наших примерах оно представлено в виде объектов критериев), которое инкапсулирует запрос для выполнения последнего исключительно объектно-ориентированным способом. Это позволяет удалить из приложения весь код, с помощью которого выполняется настройка *объекта запроса (query object)* для обработки частных случаев. Клиентам никогда не придется писать запросы в SQL-выражениях; это можно осуществлять в терминах объектов.

Несмотря на это, настоящие возможности **хранилища** проявляются в системах с несколькими источниками данных. Предположим, вам время от времени нужно осуществлять доступ к простенькому источнику данных, расположенному в оперативной памяти. Как правило, это необходимо для того, чтобы проводить тестирование исключительно на объектах приложения. Большинство тестов выполняются гораздо быстрее, если в них не осуществляется доступ к базе данных. Создание средств для тестирования модулей приложения может оказаться еще проще, если от вас потребуется всего лишь построить несколько объектов домена и поместить их в коллекцию, вместо того чтобы сохранять в базе данных в начале работы и удалять оттуда по ее окончании.

Встречаются и такие ситуации, когда даже при обычной работе приложения некоторые типы объектов домена должны всегда находиться в оперативной памяти. Особенно это касается объектов домена, которые не могут быть изменены пользователем. После первой загрузки такие объекты должны находиться в памяти и больше никогда не запрашиваться из базы данных. Как вы узнаете в конце главы, небольшое расширение

хранилища позволяет применить ту или иную стратегию выполнения запроса в зависимости от конкретной ситуации.

И наконец, еще одной областью применения **хранилища** могут стать ситуации, когда в качестве источника объектов домена используются поточные данные, предоставляемые удаленным сервером, например поток данных XML, поступающий из Internet (возможно, посредством протокола SOAP). В этом случае можно реализовать стратегию `XmlFeedRepositoryStrategy`, которая будет считывать поток данных и преобразовывать код XML в объекты домена.

Дополнительные источники информации

Типовое решение **спецификация** еще не фигурирует в какой-либо серьезной справочной литературе. Таким образом, лучшим опубликованным источником по данному типовому решению остается [14]. Более удачное описание будет приведено в [15], однако на данный момент эта книга еще находится в стадии подготовки.

Пример: поиск подчиненных заданного сотрудника (Java)

С точки зрения клиента, использование **хранилища** не представляет собой ничего сложного. Чтобы извлечь из базы данных список подчиненных соответствующего сотрудника, объект `Person` создает объект `Criteria`, содержащий описание критерия поиска, и отсылает его нужному **хранилищу**.

```
public class Person {
   public List dependents() {
      Repository repository = Registry.personRepository();
      Criteria criteria = new Criteria();
      criteria.equal(Person.BENEFACTOR, this);
      return repository.matching(criteria);
   }
}
```

Для выполнения наиболее распространенных запросов можно реализовать специализированные производные классы хранилищ. В нашем примере можно было создать класс `PersonRepository`, производный от `Repository`, и переместить создание критерия поиска в новый класс.

```
public class PersonRepository extends Repository {
   public List dependentsOf(aPerson) {
      Criteria criteria = new Criteria();
      criteria.equal(Person.BENEFACTOR, aPerson);
      return matching(criteria);
   }
}
```

После этого объект `Person` сможет вызвать метод `dependentsOf()` прямо из своего хранилища `PersonRepository`.

```
public class Person {
   public List dependents() {
      return Registry.personRepository().dependentsOf(this);
   }
}
```

Пример: выбор стратегий хранилища (Java)

Поскольку интерфейс **хранилища** надежно скрывает источник данных от слоя домена, можно изменить реализацию логики запросов внутри **хранилища** без изменения вызовов, поступающих от клиентов. Действительно, код домена не должен "беспокоиться" об источнике или назначении объектов домена. Когда источник данных расположен в оперативной памяти, то для обработки этого случая необходимо изменить метод `matching()` так, чтобы он просматривал коллекцию объектов домена на предмет соответствия заданному критерию. При этом не предполагается изменять источник данных "навсегда"; скорее, нас интересует возможность быстро переключаться между источниками данных, когда это потребуется. Отсюда следует необходимость изменить метод `matching()` так, чтобы он делегировал выполнение запроса тому или иному объекту стратегии. Очевидное преимущество такого подхода — возможность определить несколько стратегий, настроив каждую из них для обработки конкретной ситуации. В нашем случае целесообразно создать два объекта стратегии: объект `RelationalStrategy`, который будет выполнять запрос к базе данных, и объект `InMemoryStrategy`, который будет просматривать коллекцию объектов домена в оперативной памяти. Каждый объект стратегии реализует интерфейс `RepositoryStrategy`, который предоставляет доступ к методу `matching()`, поэтому реализация класса `Repository` будет выглядеть, как показано ниже.

```
abstract class Repository {
   private RepositoryStrategy strategy;
   protected List matching(aCriteria) {
      return strategy.matching(aCriteria);
   }
}
```

Класс `RelationalStrategy` реализует метод `matching()` путем создания **объекта запроса** на основе заданного критерия и использования этого объекта для запроса к базе данных. Вы можете заполнить **объект запроса** полями и значениями, определенными заданным критерием, исходя из предположения, что **объект запроса** содержит собственный метод для добавления критериев.

```
public class RelationalStrategy implements RepositoryStrategy {
   protected List matching(Criteria criteria) {
      Query query = new Query(myDomainObjectClass())
      query.addCriteria(criteria);
      return query.execute(unitOfWork());
   }
}
```

Класс `InMemoryStrategy` реализует метод `matching()` путем итерации по коллекции объектов домена и проверки каждого объекта на соответствие заданному критерию. Объект критериев может реализовать выполнение этой проверки, используя метод отражения, чтобы запрашивать у объектов домена значения определенных полей. В этом случае извлечение объектов домена описывается следующим образом:

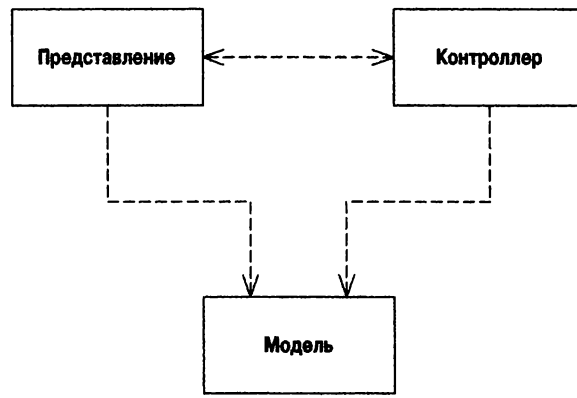
```
public class InMemoryStrategy implements RepositoryStrategy {
   private Set domainObjects;
   protected List matching(Criteria criteria) {
      List results = new ArrayList();
      Iterator it = domainObjects.iterator();
      while (it.hasNext()) {
         DomainObject each = (DomainObject) it.next();
         if (criteria.isSatisfiedBy(each))
            results.add(each);
      }
       return results;
   }
}
```

Глава 14

Типовые решения, предназначенные для представления данных в Web

Модель–представление–контроллер (Model View Controller)

Распределяет обработку взаимодействия с пользовательским интерфейсом между тремя участниками



Типовое решение **модель–представление–контроллер** — одно из наиболее часто цитируемых (и, к сожалению, неверно истолковываемых). Первоначально оно появилось в виде инфраструктуры, разработанной Тригве Реенскаугом (Trigve Reenskaug) для платформы Smalltalk в конце 70-х годов прошлого столетия. С тех пор оно сыграло значительную роль в разработке множества инфраструктур и легло в основу целого ряда концепций проектирования пользовательского интерфейса.

Принцип действия

Типовое решение **модель–представление–контроллер** подразумевает выделение трех отдельных ролей. Модель — это объект, предоставляющий некоторую информацию о домене. У модели нет визуального интерфейса, она содержит в себе все данные и поведение, не связанные с пользовательским интерфейсом. В объектно-ориентированном контексте наиболее "чистой" формой модели является объект **модели предметной области (Domain Model, 140)**. В качестве модели можно рассматривать и **сценарий транзакции (Transaction Script, 133)**, если он не содержит в себе никакой логики, связанной с пользовательским интерфейсом. Подобное определение не очень расширяет понятие модели, однако полностью соответствует распределению ролей в рассматриваемом типовом решении.

Представление отображает содержимое модели средствами графического интерфейса. Таким образом, если наша модель — это объект покупателя, соответствующее представление может быть фреймом с кучей элементов управления или HTML-страницей, заполненной информацией о покупателе. Функции представления заключаются только в отображении информации на экране. Все изменения информации обрабатываются третьим "участником" нашей системы — контроллером. Контроллер получает входные данные от пользователя, выполняет операции над моделью и указывает представлению на необходимость соответствующего обновления. В этом плане графический интерфейс можно рассматривать как совокупность представления и контроллера.

Говоря о типовом решении **модель–представление–контроллер**, нельзя не подчеркнуть два принципиальных типа разделения: отделение представления от модели и отделение контроллера от представления.

Отделение представления от модели — это один из фундаментальных принципов проектирования программного обеспечения. Наличие подобного разделения весьма важно по ряду причин.

- Представление и модель относятся к совершенно разным сферам программирования. Разрабатывая представление, вы думаете о механизмах пользовательского интерфейса и о том, как сделать интерфейс приложения максимально удобным для пользователя. В свою очередь, при работе с моделью ваше внимание сосредоточено на бизнес-политиках и, возможно, на взаимодействии с базой данных. Очевидно, при разработке модели и представления применяются разные (совершенно разные!) библиотеки. Кроме того, большинство разработчиков специализируются только в одной из этих областей.
- Пользователи хотят, чтобы, в зависимости от ситуации, одна и та же информация могла быть отображена разными способами. Отделение представления от модели позволяет разработать для одного и того же кода модели несколько представлений, а точнее, несколько абсолютно разных интерфейсов. Наиболее наглядно данный подход проявляется в том случае, когда одна и та же модель может быть представлена с помощью толстого клиента, Web-обозревателя, удаленного API или интерфейса командной строки. Даже в пределах одного и того же Web-интерфейса в разных местах приложения одному и тому же покупателю могут соответствовать разные страницы.

- Объекты, не имеющие визуального интерфейса, гораздо легче тестировать, чем объекты с интерфейсом. Отделение представления от модели позволяет легко протестировать всю логику домена, не прибегая к таким "ужасным" вещам, как средства написания сценариев для поддержки графического интерфейса пользователя.

Ключевым моментом в отделении представления от модели является направление зависимостей: представление зависит от модели, но модель не зависит от представления. Программисты, занимающиеся разработкой модели, вообще не должны быть осведомлены о том, какое представление будет использоваться. Это существенно облегчает разработку модели и одновременно упрощает последующее добавление новых представлений. Кроме того, это означает, что изменение представления не требует изменения модели.

Данный принцип тесно связан с распространенной проблемой. При использовании толстого клиента с множеством диалоговых окон на экране могут одновременно находиться несколько представлений одной и той же модели. Если пользователь внесет изменения в модель посредством одного представления, эти изменения должны быть отражены и во всех остальных представлениях. Чтобы это было возможным при отсутствии двунаправленной зависимости, необходимо реализовать типовое решение **наблюдатель (Observer)** [20] с использованием метода распространения событий (event propagation) или в виде типового решения **слушатель (Listener)**. В этом случае представление будет выполнять роль "наблюдателя" за моделью: как только модель будет изменена, представление генерирует соответствующее событие и все остальные представления обновляют свое содержимое.

Отделение контроллера от представления не играет такой важной роли, как предыдущий тип разделения. Действительно, по иронии судьбы практически во всех версиях Smalltalk разделение на контроллер и представление не проводилось. Классическим примером необходимости подобного разделения является поддержка редактируемого и нередактируемого поведения. Этого можно достичь при наличии одного представления и двух контроллеров (для двух вариантов использования), где контроллеры являются стратегиями [20], используемыми представлением. Между тем на практике в большинстве систем каждому представлению соответствует только один контроллер, поэтому разделение между ними не проводится. О данном решении вспомнили только при появлении Web-интерфейсов, где отделение контроллера от представления оказалось чрезвычайно полезным.

Тот факт, что в большинстве инфраструктур пользовательских интерфейсов не проводилось разделение на представление и контроллер, привел к множеству неверных толкований типового решения **модель–представление–контроллер**. Да, наличие модели и представления очевидно, но где же контроллер? Многие решили, что контроллер находится между моделью и представлением, как в **контроллере приложения (Application Controller, 397)**. Данное заблуждение еще более усугубил тот факт, что в обоих названиях фигурирует слово "контроллер". Между тем, несмотря на все положительные качества **контроллера приложения**, он ничем не похож на контроллер типового решения **модель–представление–контроллер**.

Описанные принципы — это все, что требуется для понимания данного набора типовых решений. Если же вам захочется еще немного покопаться в типовом решении **модель–представление–контроллер**, обратитесь к источнику [33] — там содержится лучшее на данный момент описание этого типового решения.

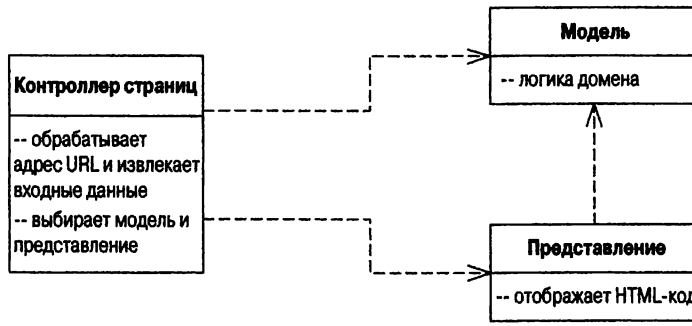
Назначение

Как уже отмечалось, ценность типового решения **модель–представление–контроллер** заключается в наличии двух типов разделения. Отделение представления от модели — один из основополагающих принципов, на котором держится все проектирование программного обеспечения, и пренебрегать им можно только тогда, когда речь идет о совсем простых системах, в которых модель вообще не имеет какого-либо реального поведения. Как только в приложении появляется невизуализированная логика, разделение становится крайне необходимым. К сожалению, во многих инфраструктурах пользовательских интерфейсов реализовать подобное разделение довольно сложно, а там, где это несложно, о нем все равно забывают.

Отделение представления от контроллера не так важно, поэтому рекомендую проводить его только в том случае, когда оно действительно нужно. Подобная необходимость практически не возникает в системах с толстыми клиентами, а вот в Web-интерфейсах отделение контроллера весьма полезно. Большинство рассматриваемых здесь типовых решений, предназначенных для проектирования Web-приложений, основаны именно на этом принципе.

Контроллер страниц (Page Controller)

Объект, который обрабатывает запрос к конкретной Web-странице или выполнение конкретного действия на Web-сайте



Большинство пользователей Internet когда-либо сталкивались со статическими HTML-страницами. Когда посетитель сайта запрашивает статическую страницу, он передает Web-серверу имя и путь к HTML-документу, который хранится на этом сервере. Ключевым моментом здесь является то, что каждая страница Web-сайта представлена на сервере отдельным документом. С динамическими страницами дело обстоит далеко не так просто из-за более сложных отношений между именами путей и соответствующими файлами. Несмотря на это, простая модель, при которой одному пути соответствует один документ, обрабатывающий запрос, может быть применена и здесь.

Типовое решение **контроллер страниц** предполагает наличие отдельного контроллера для каждой логической страницы Web-сайта. Этим контроллером может быть сама страница (как часто бывает в окружениях страниц сервера) или отдельный объект, соответствующий данной странице.

Принцип действия

В основе **контроллера страниц** лежит идея создания компонентов, которые будут выполнять роль контроллеров для каждой страницы Web-сайта. На практике количество контроллеров не всегда в точности соответствует количеству страниц, поскольку иногда при щелчке на ссылке открываются страницы с разным динамическим содержимым. Если говорить более точно, контроллер необходим для каждого *действия*, под которым подразумевается щелчок на кнопке или гиперссылке.

Контроллер страниц может быть реализован в виде сценария (сценария CGI, сервлета и т.п.) или страницы сервера (ASP, PHP, JSP и т.п.). Использование страницы сервера обычно предполагает сочетание в одном файле **контроллера страниц** и **представления по шаблону (Template View, 368)**. Это хорошо для **представления по шаблону**, но не очень подходит для **контроллера страниц**, поскольку значительно затрудняет правильное структурирование этого компонента. Данная проблема не столь важна, если страница применяется только для простого отображения информации. Тем не менее, если использование страницы предполагает наличие логики, связанной с извлечением пользовательских данных или выбором представления для отображения результатов, страница сервера может заполниться кошмарным кодом "скриптлета", т.е. внедренного сценария.

Чтобы избежать подобных проблем, можно воспользоваться вспомогательным объектом (helper object). При получении запроса страница сервера вызывает вспомогательный объект для обработки всей имеющейся логики. В зависимости от ситуации, вспомогательный объект может вернуть управление первоначальной странице сервера или же обратиться к другой странице сервера, чтобы она выступила в качестве представления. В этом случае обработчиком запросов является страница сервера, однако большая часть логики контроллера заключена во вспомогательном объекте.

Возможной альтернативой описанному подходу является реализация обработчика и контроллера в виде сценария. В этом случае при поступлении запроса Web-сервер передает управление сценарию; сценарий выполняет все действия, возложенные на контроллер, после чего отображает полученные результаты с помощью нужного представления.

Ниже перечислены основные обязанности **контроллера страниц**.

- Проанализировать адрес URL и извлечь данные, введенные пользователем в соответствующие формы, чтобы собрать все сведения, необходимые для выполнения действия.
- Создать объекты модели и вызвать их методы, необходимые для обработки данных. Все нужные данные из HTTP-запроса должны быть переданы модели, чтобы ее объекты были полностью независимы от этого запроса.
- Определить представление, которое должно быть использовано для отображения результатов, и передать ему необходимую информацию, полученную от модели.

Контроллер страниц не обязательно должен представлять собой единственный класс, зато все классы контроллеров могут использовать одни и те же вспомогательные объекты.

Это особенно удобно, если на Web-сервере должно быть несколько обработчиков, выполняющих аналогичные задания. В этом случае использование вспомогательного объекта позволяет избежать дублирования кода.

При необходимости одни адреса URL можно обрабатывать с помощью страниц сервера, а другие — с помощью сценариев. Те адреса URL, у которых нет или почти нет логики контроллера, хорошо обрабатывать с помощью страниц сервера, потому что последние представляют собой простой механизм, удобный для понимания и внесения изменений. Все остальные адреса, для обработки которых необходима более сложная логика, требуют применения сценариев. Я встречал приложения, в которых все страницы обрабатывались одним и тем же способом: либо с применением страниц сервера, либо с применением сценариев. Это обеспечивало хорошую согласованность, однако все ее преимущества обычно терялись на фоне страниц сервера, перегруженных внутренними сценариями, либо на фоне огромного множества простых сценариев.

Назначение

Выбирая способ обработки запросов к Web-сайту, необходимо решить, какой контроллер следует использовать — **контроллер страниц** или **контроллер запросов (Front Controller, 362)**. **Контроллер страниц** более прост в работе и представляет собой естественный механизм структуризации, при котором конкретные действия обрабатываются соответствующими страницами сервера или классами сценариев. **Контроллер запросов** гораздо сложнее, однако имеет массу разнообразных преимуществ, многие из которых крайне важны для Web-сайтов со сложной системой навигации.

Контроллер страниц хорошо применять для сайтов с достаточно простой логикой контроллера. В этом случае большинство адресов URL могут обрабатываться с помощью страниц сервера, а более сложные случаи — с применением вспомогательных объектов. Если логика контроллера довольно проста, использование **контроллера запросов** приведет к ненужным расходам.

Нередки ситуации, когда одни запросы к Web-сайту обрабатываются с помощью **контроллеров страниц**, а другие — с помощью **контроллеров запросов**, особенно когда разработчики выполняют рефакторинг элементов из одного контроллера в другой. В действительности эти типовые решения хорошо сочетаются между собой.

Пример: простое отображение с помощью контроллера-сервлета и представления JSP (Java)

Рассмотрим простой пример использования **контроллера страниц**, который будет что-нибудь отображать, например сведения об исполнителе (рис. 14.1). Адрес URL нашей страницы будет выглядеть как `http://www.thingy.com/recordingApp/artist?name=danielaMercury`.

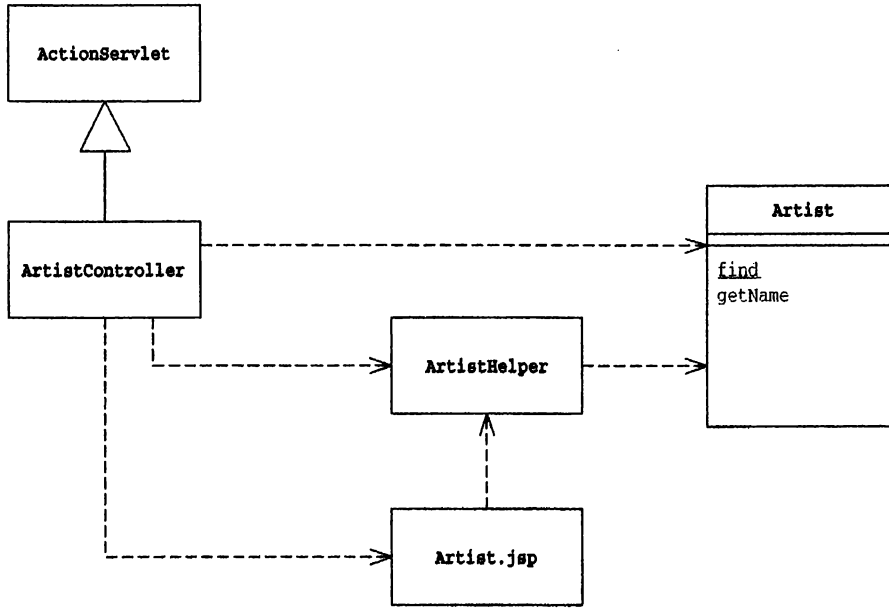


Рис. 14.1. Классы, принимающие участие в отображении данных вместе с **контроллером страниц**, *реализованным в виде сервлета, и представлением JSP*

Web-сервер должен быть настроен таким образом, чтобы распознавать элемент `/artist` как обращение к объекту `ArtistController`. В контейнере сервлетов Tomcat это делается путем добавления в файл `web.xml` приведенного далее кода.

```
<servlet>
   <servlet-name>artist</servlet-name>
   <servlet-class>actionController.ArtistController
↳</servlet-class>
</servlet>
<servlet-mapping>
   <servlet-name>artist</servlet-name>
   <url-pattern>/artist</url-pattern>
</servlet-mapping>
```

Класс `ArtistController` должен реализовать метод для обработки запроса.

```
class ArtistController...

   public void doGet(HttpServletRequest request,
↳HttpServletResponse response)
         throws IOException, ServletException {
      Artist artist = Artist.findNamed(
↳request.getParameter("name"));
      if (artist == null)
         forward("/MissingArtistError.jsp", request, response);
      else {
```

```
        request.setAttribute("helper",
↳new ArtistHelper(artist));
        forward("/artist.jsp", request, response);
    }
}
```

Хотя данный пример очень прост, он демонстрирует основные этапы обработки запроса. Вначале контроллер создает необходимые объекты моделей (здесь их функция заключается только в том, чтобы найти объект, который необходимо отобразить). Затем он помещает в HTTP-запрос нужную информацию (в нашем примере это вспомогательный объект), чтобы представление JSP смогло правильно выполнить отображение найденных сведений. И наконец, контроллер передает обработку отображения **представлению по шаблону**. Передача обработки отображения представляет собой поведение, общее для всех **контроллеров страниц**, а потому может быть реализована в их суперклассе.

```
class ActionServlet...

   protected void forward(String target,
                          HttpServletRequest request,
                          HttpServletResponse response)
         throws IOException, ServletException
   {
         RequestDispatcher dispatcher =
↳getServletContext().getRequestDispatcher(target);
         dispatcher.forward(request, response);
   }
```

Основная точка соприкосновения между **представлением по шаблону** и **контроллером страниц** — это имена параметров, указанных в запросе, значения которых должны быть переданы всем объектам, используемым представлением JSP.

Логика контроллера в данном примере действительно очень проста, однако даже при ее усложнении наш сервлет все еще может быть использован в качестве контроллера. Например, можно реализовать похожее поведение для отображения сведений об альбомах таким образом, что альбомам классической музыки будут соответствовать свой объект модели и свое представление JSP. Для реализации этого поведения используется еще один производный класс контроллера.

```
class AlbumController...

   public void doGet(HttpServletRequest request,
↳HttpServletResponse response)
          throws IOException, ServletException
   {
      Album album = Album.find(request.getParameter("id"));
      if (album == null) {
         forward("/missingAlbumError.jsp", request, response);
         return;
      }
      request.setAttribute("helper", album);
      if (album instanceof ClassicalAlbum)
         forward("/classicalAlbum.jsp", request, response);
```

```
      else
         forward("/album.jsp", request, response);
   }
```

Обратите внимание, что в этом примере я не создаю отдельный вспомогательный класс, а использую в качестве вспомогательных объекты модели. Это можно делать тогда, когда вспомогательный класс нужен только для того, чтобы передавать управление классу модели. Тем не менее, если вы решите поступить подобным образом, убедитесь, что класс модели не содержит никакого кода, зависимого от сервлета. В противном случае весь этот код должен быть вынесен в отдельный вспомогательный класс.

Пример: использование страницы JSP в качестве обработчика запросов (Java)

В качестве контроллера страниц можно использовать сервлет, однако обычно в этой роли выступает сама страница сервера. Недостаток данного подхода заключается в необходимости внедрения в страницу сервера кода скриптлета (а, как вы уже могли догадаться, я считаю, что код скриптлета имеет такое же отношение к качественному проектированию, как армрестлинг — к профессиональному спорту).

Несмотря на это, страницу сервера действительно можно применять в качестве обработчика запросов, перекладывая фактическое выполнение всех функций контроллера на вспомогательный объект. Я сделаю это для отображения сведений об альбоме, используя адрес URL вида `http://localhost:8080/isa/album.jsp?id=zero`. Большинство альбомов будут отображаться с помощью представления `album.jsp`, а альбомы с классической музыкой — с помощью представления `classicalAlbum.jsp`.

Поведение контроллера реализовано во вспомогательном классе, имя которого задается в коде самой страницы `album.jsp`.

```
album.jsp...

<jsp:useBean id="helper" class="actionController.AlbumConHelper
↳"/>
<%helper.init(request, response);%>
```

Вызов метода инициализации настраивает вспомогательный объект на выполнение функций контроллера.

```
class AlbumConHelper extends HelperController...

   public void init(HttpServletRequest request,
↳HttpServletResponse response) {
      super.init(request, response);
      if (getAlbum() == null) forward("missingAlbumError.jsp",
↳request, response);
      if (getAlbum() instanceof ClassicalAlbum) {
         request.setAttribute("helper", getAlbum());
         forward("/classicalAlbum.jsp", request, response);
      }
   }
```

Поведение, общее для всех контроллеров, можно вынести в суперкласс вспомогательного объекта.

```
class HelperController...

   public void init(HttpServletRequest request,
↳HttpServletResponse response) {
      this.request = request;
      this.response = response;
   }
   protected void forward(String target,
                  HttpServletRequest request,
                  HttpServletResponse response)
   {
      try {
         RequestDispatcher dispatcher =
↳request.getRequestDispatcher(target);
         if (dispatcher == null) response.sendError(
↳response.SC_NO_CONTENT);
         else dispatcher.forward(request, response);
      } catch (IOException e) {
         throw new ApplicationException(e);
      } catch (ServletException e) {
         throw new ApplicationException(e);
      }
   }
```

Обратите внимание на разницу в поведении контроллера-сервлета и контроллера — вспомогательного объекта. В последнем случае страница JSP, выступающая в роли обработчика запросов, одновременно является и стандартным представлением, а потому управление возвращается первоначальной странице сервера (кроме случая, когда оно передается другому представлению). Это очень удобно, когда большинство страниц JSP непосредственно выступают в роли представлений и передавать обработку отображения другим представлениям не нужно. Метод инициализации вспомогательного объекта запускает поведение модели и настраивает все параметры, необходимые для последующего отображения результатов. Данная схема вполне естественна, поскольку многие ассоциируют Web-страницу со страницей сервера, которая применяется для представления этой Web-страницы. Кроме того, зачастую подобная схема хорошо согласуется с конфигурацией Web-сервера.

Вызывать метод инициализации вспомогательного объекта довольно неудобно. В среде JSP это можно значительно упростить с помощью класса пользовательского дескриптора (custom tag). Такой дескриптор может автоматически создать нужный объект, поместить его в запрос и инициализировать. Все, что для этого нужно, — вставить в страницу JSP простой дескриптор примерно следующего вида:

```
<helper:init name = "actionController.AlbumConHelper"/>
```

Теперь вся работа по инициализации вспомогательного объекта ложится на реализацию класса пользовательского дескриптора.

```
class HelperInitTag extends HelperTag...

   private String helperClassName;
   public void setName(String helperClassName) {
      this.helperClassName = helperClassName;
   }
   public int doStartTag() throws JspException {
      HelperController helper = null;
      try {
         helper = (HelperController)
            Class.forName(helperClassName).newInstance();
      } catch (Exception e) {
         throw new ApplicationException("Unable to
↳instantiate " + helperClassName, e);
      }
      initHelper(helper);
      pageContext.setAttribute(HELPER, helper);
      return SKIP_BODY;
   }
   private void initHelper(HelperController helper) {
      HttpServletRequest request = (HttpServletRequest)
         pageContext.getRequest();
      HttpServletResponse response = (HttpServletResponse)
         pageContext.getResponse();
      helper.init(request, response);
   }

class HelperTag...

   public static final String HELPER = "helper";
```

Я могу создать еще один производный класс дескриптора, который будет применяться для доступа к свойству вспомогательного объекта (в нашем примере это будет название альбома).

```
class HelperGetTag extends HelperTag...

   private String propertyName;
   public void setProperty(String propertyName) {
      this.propertyName = propertyName;
   }
   public int doStartTag() throws JspException {
      try {
         pageContext.getOut().print(getProperty(propertyName));
      } catch (IOException e) {
         throw new JspException("unable to print to writer");
      }
      return SKIP_BODY;
   }

class HelperTag...

   protected Object getProperty(String property)
↳throws JspException {
```

```
      Object helper = getHelper();
      try {
         final Method getter =
            helper.getClass().getMethod(
↳gettingMethod(property), null);
         return getter.invoke(helper, null);
      } catch (Exception e) {
         throw new JspException ("Unable to invoke " +
↳gettingMethod(property) + " - " + e.getMessage());
      }
   }
   private Object getHelper() throws JspException {
      Object helper = pageContext.getAttribute(HELPER);
      if (helper == null) throw new JspException(
↳"Helper not found.");
      return helper;
   }
   private String gettingMethod(String property) {
      String methodName = "get" + property.substring(
↳0, 1).toUpperCase() + property.substring(1);
      return methodName;
   }
```

(Кое-кто из вас может возразить, что в данном примере было бы удобнее использовать механизм Java Beans, который бы позволил просто вызвать get-метод посредством отражения. Те, кто так думает, конечно же, правы... и, должно быть, достаточно умны для того, чтобы реализовать это самостоятельно.)

Определив объект `HelperGetTag`, я могу использовать его для извлечения информации из вспомогательного объекта. Для этого в страницу JSP придется вставить еще один дескриптор — между прочим, довольно короткий (что устранит вероятность неправильного написания имени вспомогательного объекта).

```
<B><helper:get property = "title"/></B>
```

Пример: обработка запросов страницей сервера с применением механизма разделения кода и представления (C#)

Компоненты .NET, предназначенные для создания Web-приложений, ориентированы на работу **с контроллером страниц** и **представлением по шаблону**, хотя, разумеется, вы можете обрабатывать Web-события и по-другому. В этом примере я воспользуюсь стандартным подходом .NET, накладывая слой представления на слой домена с использованием **модуля таблицы (Table Module, 148)**, а также применяя объекты `DataSet` в качестве "переносчиков" информации между слоями.

На этот раз будем иметь дело со страницей, которая отображает количество и интенсивность перебежек за одну подачу в крикетном матче. Конечно же, не все читатели могут быть знакомы с правилами этой поистине королевской игры. Впрочем, для нашего примера будет достаточно, что количество перебежек (runs scored) — это количество очков, набранное игроком, отбивающим мяч (бэтсменом), а интенсивность перебежек (run rate) — это количество перебежек, деленное на количество отбитых мячей. Сведения

о количестве перебежек и количестве отбитых мечей хранятся в базе данных. Интенсивность перебежек вычисляется приложением — небольшой, но крайне полезный в плане обучения кусочек логики домена.

В качестве обработчика запросов будем использовать Web-страницу ASP.NET, хранящуюся в файле с расширением `.aspx`. Как и другие подобные конструкции, данный файл позволяет внедрять логику домена прямо в код страницы сервера с помощью скриптлетов. Как вы уже знаете, я скорее выпью плохое пиво, чем стану писать скриптлеты. Поэтому воспользуемся *механизмом разделения кода и представления (code behind)*, который позволяет ассоциировать страницу ASP.NET с обычными классами, указав имя соответствующего файла в заголовке страницы.

```
<%@ Page language="c#" Codebehind="bat.aspx.cs"
↳AutoEventWireup="false" trace="False"
↳Inherits="batsmen.BattingPage" %>
```

Данная страница "наследует" класс `BattingPage`, реализующий логику домена, и потому может использовать все его защищенные свойства и методы. В этом случае объект страницы является обработчиком запросов, а фактическое выполнение логики домена будет осуществляться посредством метода `Page_Load`. Если большинство страниц будут обрабатывать запросы аналогичным способом, я могу определить **супертип слоя (Layer Supertype, 491)** с шаблонным методом `Page_Load` [20].

```
class CricketPage...

   protected void Page_Load(object sender, System.EventArgs e) {
      db = new OleDbConnection(DB.ConnectionString);
      if (hasMissingParameters())
         errorTransfer (missingParameterMessage);
      DataSet ds = getData();
      if (hasNoData (ds))
         errorTransfer ("No data matches your request");
      applyDomainLogic (ds);
      DataBind();
      prepareUI(ds);
   }
```

Шаблонный метод разбивает обработку запроса на несколько основных этапов. Это позволяет определить общий принцип обработки Web-запросов и в то же время реализовать в каждом **контроллере страниц** собственные варианты выполнения тех или иных этапов. В этом случае при написании нескольких **контроллеров страниц** вы будете знать, какие методы необходимо реализовать для подстановки в шаблонный метод. Если же какой-то странице понадобится выполнить нечто совершенно другое, она всегда сможет переопределить метод `Page_Load`.

Вначале нужно проверить правильность значений параметров, переданных пользователем странице ASP.NET. В реальных приложениях это могло бы потребовать общей проверки значений на отсутствие тривиальных ошибок, однако в данном случае просто проанализируем URL вида `http://localhost/batsmen/bat.aspx?team=England&innings=2&match=905` и проверим, присутствуют ли в нем все параметры, необходимые для выполнения запроса к базе данных. Как обычно, я слишком упрощаю обработку

ошибок, по крайней мере до тех пор, пока кто-нибудь не разработает хороших типовых решений для выполнения проверки на правильность. Итак, наша страница будет определять набор обязательных параметров, а **супертип слоя** выполнять проверку их наличия.

```
class CricketPage...

   abstract protected String[] mandatoryParameters();
   private Boolean hasMissingParameters() {
      foreach (String param in mandatoryParameters())
         if (Request.Params[param] == null) return true;
      return false;
   }
   private String missingParameterMessage {
      get {
         String result = "<P>This page is missing mandatory
↳parameters:</P>";
         result += "<UL>";
         foreach (String param in mandatoryParameters())
            if (Request.Params[param] == null)
               result += String.Format("<LI>{0}</LI>", param);
         result += "</UL>";
         return result;
      }
   }
   protected void errorTransfer (String message) {
      Context.Items.Add("errorMessage", message);
      Context.Server.Transfer("Error.aspx");
   }

class BattingPage...

   override protected String[] mandatoryParameters() {
      String[] result = {"team", "innings", "match"};
      return result;
   }
```

Следующий этап — извлечение данных из базы данных и помещение их в объект ADO.NET `DataSet` (напомню, что он применяется для доступа к данным в отсоединенном режиме). В нашем примере все данные извлекаются посредством одного запроса к таблице `batting`.

```
class CricketPage...

   abstract protected DataSet getData();
   protected Boolean hasNoData(DataSet ds) {
      foreach (DataTable table in ds.Tables)
         if (table.Rows.Count != 0) return false;
      return true;
   }

class BattingPage...

   override protected DataSet getData() {
```

```
      OleDbCommand command = new OleDbCommand(SQL, db);
      command.Parameters.Add(new OleDbParameter("team", team));
      command.Parameters.Add(new OleDbParameter("innings",
↳innings));
      command.Parameters.Add(new OleDbParameter("match",
↳match));
      OleDbDataAdapter da = new OleDbDataAdapter(command);
      DataSet result = new DataSet();
      da.Fill(result, Batting.TABLE_NAME);
      return result;
   }
   private const String SQL =
      @"SELECT * from batting
         WHERE team = ? AND innings = ? AND matchID = ?
         ORDER BY battingOrder";
```

Теперь пришла очередь выполнения логики домена, организованной в **модуль таблицы**. Контроллер передает **модулю таблицы** извлеченное множество данных для последующей обработки.

```
class CricketPage...

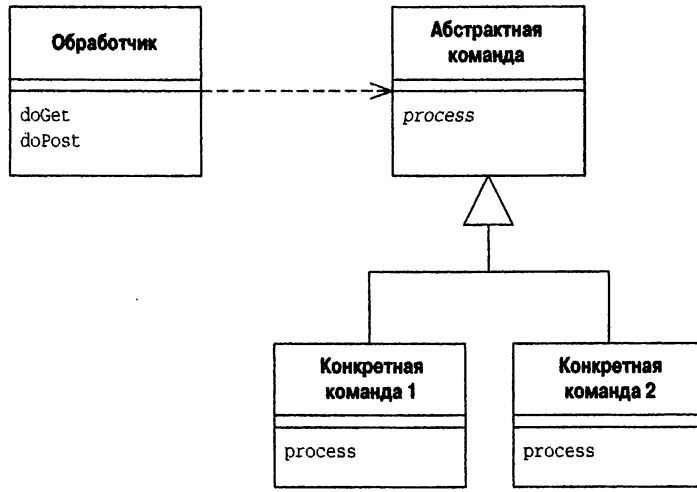
      protected virtual void applyDomainLogic (DataSet ds) {}

class BattingPage...
      override protected void applyDomainLogic (DataSet
↳dataSet) {
         batting = new Batting(dataSet);
         batting.CalculateRates();
      }
```

На этом этапе выполнение обязанностей контроллера закончено. Теперь, как следует из классической системы **модель–представление–контроллер (Model View Controller, 347)**, контроллер должен обратиться к представлению, чтобы передать ему обработку отображения. В нашем примере объект `BattingPage` выполняет роль и контроллера и представления, поэтому вызов метода `PrepareUI` в конце метода `Page_Load` фактически является частью поведения представления. Итак, пока что я могу попрощаться с данным примером. Однако смею надеяться, что вы не удовлетворены отсутствием эффектной концовки. Не огорчайтесь — мы еще вернемся к обсуждению этого вопроса.

Контроллер запросов (Front Controller)

Контроллер, который обрабатывает все запросы к Web-сайту



Обработка запросов к Web-сайтам со сложной структурой подразумевает выполнение большого количества аналогичных действий. Это проверка безопасности, обеспечение поддержки интернационализации и открытие специальных представлений для особых категорий пользователей. Если поведение входного контроллера будет разбросано по нескольким объектам, это приведет к дублированию большей части кода. Кроме того, это значительно затруднит изменение поведения контроллера во время выполнения.

Типовое решение **контроллер запросов** объединяет все действия по обработке запросов в одном месте, распределяя их выполнение посредством единственного объекта-обработчика. Как правило, этот объект реализует общее поведение, которое может быть изменено во время выполнения с помощью декораторов. Для выполнения конкретного запроса обработчик вызывает соответствующий объект команды.

Принцип действия

Контроллер запросов обрабатывает все запросы, поступающие к Web-сайту, и обычно состоит из двух частей: Web-обработчика и иерархии команд. Web-обработчик — это объект, который выполняет фактическое получение POST- или GET-запросов, поступивших на Web-сервер. Он извлекает необходимую информацию из адреса URL и входных данных запроса, после чего решает, какое действие необходимо инициировать, и делегирует его выполнение соответствующей команде (рис. 14.2).

Web-обработчик обычно реализуется в виде класса, а не страницы сервера, поскольку он не генерирует никаких откликов. Команды также являются классами, а не страницами сервера; более того, им не нужно знать о наличии Web-окружения, несмотря на то что

им часто передается информация из HTTP-запросов. В большинстве случаев Web-обработчик — это довольно простая программа, функции которой заключаются в выборе нужной команды.

Выбор команды может происходить статически или динамически. Статический выбор команды подразумевает проведение синтаксического анализа адреса URL и применение условной логики, а динамический — извлечение некоторого стандартного фрагмента адреса URL и динамическое создание экземпляра класса команды.

Преимущества статического выбора команды состоят в использовании явного кода, наличии проверки времени компиляции и высокой гибкости возможных вариантов написания адресов URL. В свою очередь, использование динамического подхода позволяет добавлять новые команды, не требуя изменения Web-обработчика.

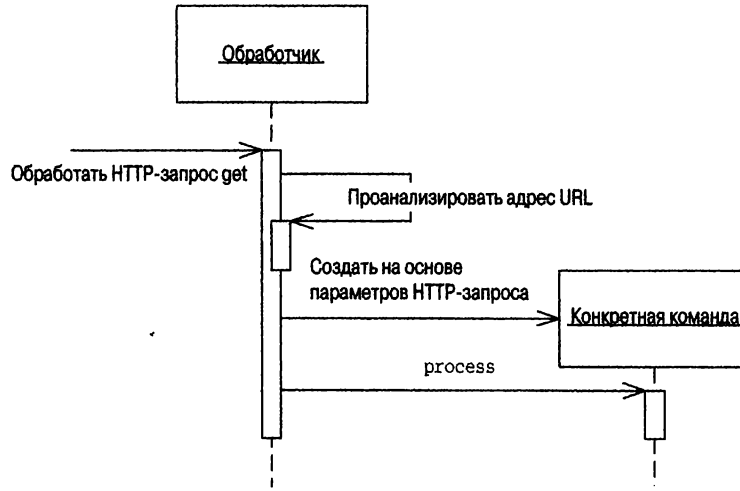


Рис. 14.2. Принцип работы **контроллера запросов**

При динамическом выборе команд имя класса команды можно поместить непосредственно в адрес URL либо воспользоваться файлом свойств, который будет привязывать адреса URL к именам классов команд. Разумеется, это потребует создания дополнительного файла свойств, однако позволит легко и непринужденно изменять имена классов, не просматривая все имеющиеся на сервере Web-страницы.

Контроллер запросов особенно хорошо сочетается с типовым решением **перехватывающий фильтр (Intercepting Filter)**, описанным в [3]. Последнее представляет собой декоратор, выполняющий роль оболочки для Web-обработчика контроллера запросов и позволяющий строить *цепочки фильтров (filter chains)*, т.е. последовательности фильтров, предназначенные для обработки таких процессов, как аутентификация, регистрация на сайте и выбор кодировки. Использование фильтров позволяет динамически настраивать их во время конфигурирования Web-сервера.

Роб Ми (Rob Mee) показал мне интересный вариант **контроллера запросов**, в котором Web-обработчик состоял из двух частей: "вырожденного" обработчика и диспетчера. Вырожденный Web-обработчик извлекает базовые данные из параметров HTTP-запроса и передает их диспетчеру таким образом, что тот оказывается полностью независимым от

инфраструктуры Web-сервера. Подобная схема значительно облегчает тестирование, поскольку позволяет запустить диспетчер, не запуская Web-сервер.

Не забывайте, что и обработчик и команды являются частью контроллера. Поэтому команды могут (и должны) выбирать, какое представление следует использовать для отображения результатов. Единственная обязанность обработчика — выбор той или иной команды. После этого обработчик перестает принимать участие в выполнении запроса.

Назначение

Контроллер запросов имеет более сложную структуру, чем его очевидный соперник — **контроллер страниц (Page Controller, 350)**. Чтобы быть удостоенным внимания разработчиков, **контроллер запросов** должен иметь хотя бы несколько преимуществ, и они у него, разумеется, есть.

Во-первых, Web-сервер должен быть настроен на использование только одного **контроллера запросов**; всю остальную работу по распределению запросов будет выполнять Web-обработчик. Это значительно облегчит конфигурирование Web-сервера (особенно если этот процесс достаточно трудоемок). Использование динамического выбора команд позволяет добавлять новые команды без каких-либо изменений. Кроме того, динамические команды облегчают переносимость, так как от вас требуется лишь "зарегистрировать" обработчик в настройках сервера.

Поскольку для выполнения каждого запроса создаются новые экземпляры классов команд, вам не придется беспокоиться о том, чтобы они функционировали в отдельных потоках. Это позволит избежать всех неприятностей, связанных с многопоточным программированием; тем не менее вам придется следить за отсутствием совместного использования других объектов, таких, как объекты модели.

Наиболее часто упоминаемым преимуществом **контроллера запросов** является возможность реализации общего кода, который так или иначе дублируется в многочисленных **контроллерах страниц**. Тем не менее многие забывают, что общее поведение последних с не меньшим успехом может быть вынесено в суперкласс **контроллеров страниц**.

Контроллер запросов только один, что позволяет легко изменять его поведение во время выполнения с помощью многочисленных декораторов [20]. Декораторы могут применяться для проведения аутентификации, выбора кодировки, обеспечения поддержки интернационализации и т.п. Более того, их можно добавлять не только с помощью файла настроек, но и прямо во время работы сервера. (Детальное описание этого подхода под названием **перехватывающий фильтр (Intercepting Filter)** приводится в [3].)

Дополнительные источники информации

В [3] содержится подробное описание реализации **контроллера запросов** на языке Java. Там же описано типовое решение **перехватывающий фильтр**, которое весьма хорошо сочетается с **контроллером запросов**.

Типовое решение **контроллер запросов** используется в целом ряде Web-инфраструктур Java. Хороший пример такого использования приведен в [37].

Пример: простое отображение (Java)

Рассмотрим простой пример использования **контроллера запросов** для выполнения исключительно оригинального и сверхважного задания — отображения сведений об исполнителе (рис. 14.3). Воспользуемся динамическим выбором команд, а также адресом URL вида `http://localhost:8080/isa/music?name=barelyWorks&command=Artist`. Параметр `command` будет указывать обработчику на то, какую команду следует использовать для выполнения того или иного запроса.

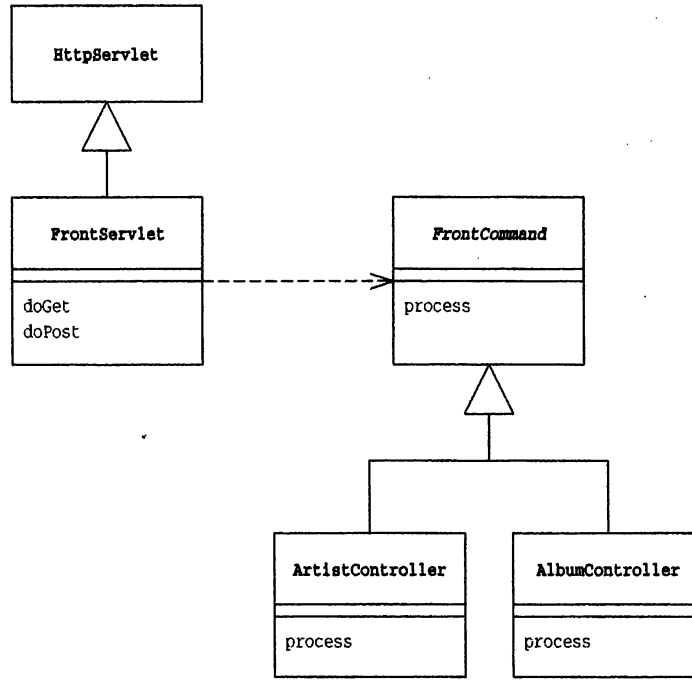


Рис. 14.3. Классы, принимающие участие в реализации **контроллера запросов**

Начнем с разработчика, который реализован в виде сервлета.

```
class FrontServlet...

   public void doGet(HttpServletRequest request,
↳HttpServletResponse response)
         throws IOException, ServletException {
      FrontCommand command = getCommand(request);
      command.init(getServletContext(), request, response);
      command.process();
   }
   private FrontCommand getCommand(HttpServletRequest request) {
      try {
         return (FrontCommand) getCommandClass(
↳request).newInstance();
```

```
      } catch (Exception e) {
         throw new ApplicationException(e);
      }
   }
   private Class getCommandClass(HttpServletRequest request) {
      Class result;
      final String commandClassName = "frontController." +
            (String) request.getParameter("command") +
↳"Command";
      try {
         result = Class.forName(commandClassName);
      } catch (ClassNotFoundException e) {
         result = UnknownCommand.class;
      }
      return result;
   }
```

Приведенная логика очень проста. Обработчик пытается создать экземпляр класса, имя которого образовано путем конкатенации имени команды и слова *Command*. Когда новый объект команды будет создан, обработчик инициализирует его с помощью данных, полученных от HTTP-сервера. В этом примере мне потребовалось передать команде совсем немного параметров. В реальных ситуациях могут понадобиться и другие параметры, например сеанс HTTP. На тот случай, если команда с указанным именем не будет найдена, я использовал типовое решение **частный случай (Special Case, 511)** и возвратил неизвестную команду (класс `UnknownCommand`). Как это обычно бывает, использование **частного случая** позволяет избежать дополнительной проверки на наличие ошибок.

Некоторая часть данных и поведения совместно используется различными объектами команд. Все эти объекты должны быть инициализированы с помощью информации, полученной от Web-сервера.

```
class FrontCommand...

   protected ServletContext context;
   protected HttpServletRequest request;
   protected HttpServletResponse response;
   public void init(ServletContext context,
                    HttpServletRequest request,
                    HttpServletResponse response)
   {
      this.context = context;
      this.request = request;
      this.response = response;
   }
```

Помимо этого, класс `FrontCommand` может реализовать общее поведение, например метод `forward`, а также определить абстрактный метод обработки, который будет переопределен конкретными классами команд.

```
class FrontCommand...

   abstract public void process() throws ServletException,
↳IOException ;
```

```
   protected void forward(String target)
↳throws ServletException, IOException
   {
      RequestDispatcher dispatcher =
↳context.getRequestDispatcher(target);
      dispatcher.forward(request, response);
   }
```

Объект команды очень прост, по крайней мере в нашем случае. Он содержит лишь реализацию метода обработки, который вызывает соответствующее поведение объектов модели, помещает в запрос информацию, необходимую для отображения результатов, и передает управление **представлению по шаблону (Template View, 368)**.

```
class ArtistCommand...

   public void process() throws ServletException, IOException {
      Artist artist = Artist.findNamed(request.getParameter(
↳"name"));
      request.setAttribute("helper", new ArtistHelper(artist));
      forward("/artist.jsp");
   }
```

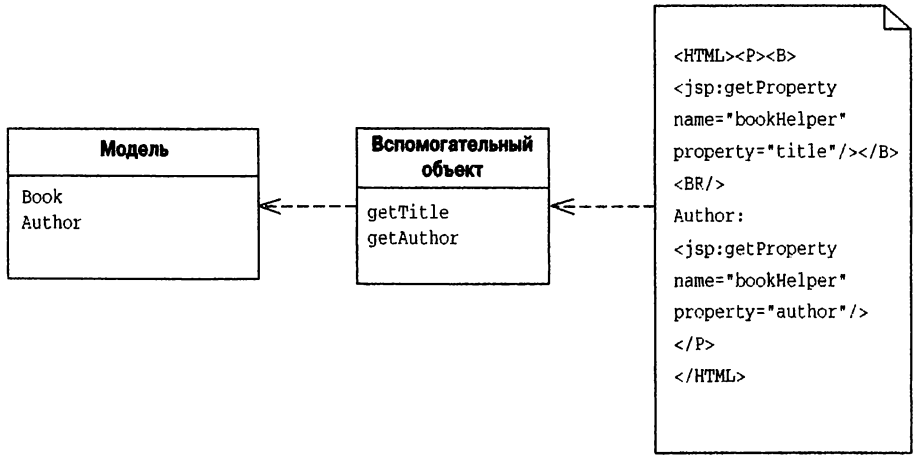
И наконец, класс `UnknownCommand` выводит на экран страницу с банальным сообщением об ошибке.

```
class UnknownCommand...

   public void process() throws ServletException, IOException {
      forward("/unknown.jsp");
   }
```

Представление по шаблону (Template View)

Преобразует результаты выполнения запроса в формат HTML путем внедрения маркеров в HTML-страницу



Написать программу, генерирующую код HTML, гораздо сложнее, чем может показаться на первый взгляд. Хотя современные языки программирования справляются с созданием текста гораздо лучше, чем в былые времена (некоторые еще помнят обработку символов в старом добром ФОРТРАНе или стандартной версии Pascal), создание и конкатенация строковых конструкций все еще связаны с массой неудобств. Данная проблема не так страшна, если текста не слишком много, однако создание целой HTML-страницы может потребовать множества "изуверских" манипуляций над текстом.

Для редактирования статических HTML-страниц (тех, содержимое которых не изменяется от запроса к запросу) можно использовать замечательные текстовые редакторы, работающие по принципу WYSIWYG (What You See Is What You Get — что видишь на экране, то и получишь при печати). Даже тем, кто предпочитает самые примитивные редакторы, набирать текст и дескрипторы намного приятнее, чем заниматься конкатенацией строк в коде программы.

Основные трудности связаны с созданием динамических Web-страниц — тех, которые принимают результаты выполнения какого-нибудь запроса (например, к базе данных) и внедряют их в код HTML. Содержимое такой страницы меняется с каждым запросом, а потому обыкновенные редакторы HTML здесь бессильны.

Наиболее удобный способ создания динамической Web-страницы — конструирование обычной статической страницы и последующая вставка в нее специальных маркеров, которые могут быть преобразованы в вызовы функций, предоставляющих динамическую информацию. Поскольку статическая часть страницы выступает в роли своеобразного шаблона, я называю это типовое решение **представлением по шаблону**.

Принцип действия

Основная идея, лежащая в основе типового решения **представление по шаблону**, — вставка маркеров в текст готовой статической HTML-страницы. При вызове страницы для обслуживания запроса эти маркеры будут заменены результатами некоторых вычислений (например, результатами выполнения запросов к базе данных). Подобная схема позволяет создавать статическую часть страницы с помощью обычных средств, например текстовых редакторов, работающих по принципу WYSIWYG, и не требует знания языков программирования. Для получения динамической информации маркеры обращаются к отдельным программам.

Представление по шаблону используется целым рядом программных средств. Таким образом, ваша задача состоит не столько в том, чтобы разработать данное решение самому, сколько в том, чтобы научиться его эффективно использовать и познакомиться с возможными альтернативами.

Вставка маркеров

Существует несколько способов внедрения маркеров в HTML-страницу. Один из них — это использование HTML-подобных дескрипторов. Данный способ хорошо подходит для редакторов, работающих по принципу WYSIWYG, поскольку они распознают элементы, заключенные в угловые скобки (<>), как специальное содержимое и поэтому игнорируют их либо обращаются с ними иначе, чем с обычным текстом. Если дескрипторы удовлетворяют правилам форматирования языка XML, для работы с полученным документом можно использовать средства XML (разумеется, при условии, что результирующий документ HTML является документом XHTML).

Еще один способ внедрения динамического содержимого — вставка специальных текстовых маркеров в тело страницы. В этом случае текстовые редакторы WYSIWYG будут воспринимать вставленные маркеры как обычный текст. Разумеется, содержимое маркеров от этого не изменится, однако может быть подвергнуто разнообразным назойливым операциям, например проверке орфографии. Тем не менее данный способ позволяет обойтись без запутанного синтаксиса HTML/XML.

Некоторые среды разработки содержат наборы готовых маркеров, однако все больше и больше платформ позволяют разработчику определять собственные дескрипторы и маркеры в соответствии с нуждами конкретных приложений.

Одной из наиболее популярных форм **представления по шаблону** является *страница сервера (server page)* — ASP, JSP или PHP. Вообще говоря, страницы сервера — это нечто большее, чем **представление по шаблону**, поскольку они позволяют внедрять в страницу элементы программной логики, называемые *скриптлетами (scriptlets)*. Я, однако, рассматриваю наличие скриптлетов как признак дурного тона, поэтому настоятельно рекомендую ограничивать поведение страниц сервера стандартной формой **представления по шаблону**.

Наиболее очевидный недостаток внедрения в страницу сервера множества скриптлетов состоит в том, что ее могут редактировать исключительно программисты. Данная проблема особенно критична, если проектированием страницы занимаются графические дизайнеры. Однако самые существенные недостатки скриптлетов связаны с тем, что страница — далеко не самый подходящий модуль для программы. Даже при использовании объектно-ориентированных языков программирования внедрение кода в текст страницы лишает вас возможности применять многие средства структурирования,

необходимые для построения модулей как в объектно-ориентированном, так и в процедурном стиле.

Однако гораздо неприятнее то, что внедрение в страницу большого числа скриптлетов приводит к "перемешиванию" слоев корпоративного приложения. Если логика домена запускается прямо на страницах сервера, она практически не поддается структурированию и вместе с тем может легко привести к дублированию логики на других страницах сервера. Что ни говори, а самым кошмарным кодом, который мне довелось увидеть за последние несколько лет, был код страницы сервера.

Вспомогательный объект

Чтобы избежать использования скриптлетов, каждой странице можно назначить собственный *вспомогательный объект (helper object)*. Этот объект будет содержать в себе всю фактическую логику домена, а сама страница — только вызовы вспомогательного объекта, что значительно упростит структуру страницы и максимально приблизит ее к "чистой" форме **представления по шаблону**. Более того, это обеспечит возможность "разделения труда", при котором непрограммисты смогут спокойно заняться редактированием страницы, а программисты — сосредоточиться на разработке вспомогательного объекта. В зависимости от используемого средства, все "шаблонное" содержимое страницы зачастую можно свести к набору HTML/XML-дескрипторов, что повысит согласованность страницы и сделает ее более пригодной для поддержки стандартными средствами.

Описанная схема кажется простой и вполне заслуживающей доверия. Тем не менее, как всегда, здесь есть несколько но. Самые простые маркеры — это те, которые получают информацию из остальной части системы и помещают эту информацию в нужное место страницы. Подобные маркеры могут быть легко преобразованы в вызовы методов вспомогательного объекта, результатом выполнения которых является текст (или то, что может быть легко преобразовано в текст). Этот текст и помещается в указанное место страницы.

Условное отображение

Гораздо сложнее реализовать условное поведение страницы. Самый простой пример условного поведения — это когда результат выполнения запроса отображается только в том случае, если значением условного выражения является истина. Подобное поведение может быть реализовано с помощью условных дескрипторов наподобие `<IF condition = "$падение_цены > 0.1"> ...отобразить_что_нибудь </IF>`. К сожалению, использование условных дескрипторов постепенно превращает шаблонные страницы в некое подобие языков программирования. Это чревато теми же проблемами, что и использование скриптлетов. Если вам нужен полноценный язык программирования, вы можете с таким же успехом воспользоваться скриптлетами, но вы же знаете, как я отношусь к этой идее!

Из всего сказанного выше можно понять, что я не приветствую использование условных дескрипторов и рекомендую всячески их избегать. Разумеется, это не всегда возможно, однако вы должны постараться придумать что-то более подходящее к потребностям конкретного приложения, чем универсальный дескриптор `<IF>`.

Если отображение текста должно выполняться при соблюдении определенных условий, их можно перенести во вспомогательный объект. В этом случае страница всегда будет отображать результаты выполнения методов вспомогательного объекта. Если условие

не соблюдено, вспомогательный объект возвратит пустую строку. Этот подход хорошо применять тогда, когда возвращаемый текст не нуждается в разметке либо допускает возможность возвращения пустой разметки, которая будет проигнорирована обозревателем.

Иногда данный прием не срабатывает, например если вы хотите акцентировать внимание посетителей сайта на наиболее продаваемых товарах, выделив их названия полужирным шрифтом. В этом случае, помимо отображения названий товаров, может понадобиться специальная разметка текста. Одно из возможных решений этой проблемы — генерация разметки вспомогательным объектом. Это позволит очистить страницу от программной логики, однако создаст дополнительную нагрузку для программиста, которому придется принять на себя обязанности дизайнера, реализуя механизм выделения текста.

Чтобы управление HTML-разметкой осталось в руках дизайнера страниц, необходимо воспользоваться условными дескрипторами. В такой ситуации очень важно не опуститься до применения простого дескриптора <IF>. Удачным решением является применение дескрипторов, направленных на выполнение определенных действий. Например, вместо дескриптора

```
<IF expression = "isHighSelling()"><B></IF>
<property name = "price"/>
<IF expression = "isHighSelling()"></B></IF>
```

можно применить дескриптор

```
<highlight condition = "isHighSelling" style = "bold">
   <property name = "price"/>
</highlight>
```

И в том и в другом случае очень важно, чтобы проверка условия выполнялась на основе единственного булева свойства вспомогательного объекта. Наличие на странице более сложных условных выражений будет означать перенесение логики на саму страницу.

Еще одним примером условного поведения является отображение той или иной информации в зависимости от используемого в системе регионального стандарта. Предположим, что некоторый текст должен отображаться на экране только для пользователей из США или Канады. В этом случае вместо универсального дескриптора

```
<IF expression = "locale = 'US' || 'CA'"> ...special text </IF>
```

рекомендуется использовать дескриптор вида

```
<locale includes = "US, CA"> ...special text </locale>
```

Итерация

Похожие проблемы связаны с итерацией по коллекциям объектов. Если вы хотите отобразить таблицу, где каждая строка будет соответствовать пункту заказа, вам понадобится реализовать конструкцию, которая позволит легко отображать сведения по каждой строке. В данном примере практически невозможно избежать итерации по дескриптору коллекций, однако обычно она выполняется довольно легко.

Разумеется, набор доступных дескрипторов ограничивается используемой средой разработки. Некоторые среды разработки предоставляют фиксированный набор дескрипторов, которых может оказаться недостаточно для реализации описанных выше приемов. Тем не менее в большинстве сред разработки набор доступных дескрипторов более обширен, а некоторые платформы даже позволяют создавать собственные библиотеки дескрипторов.

Обработка страницы

Как следует из названия, главной функцией типового решения **представление по шаблону** является выполнение роли представления в системе **модель–представление–контроллер (Model View Controller, 347)**. В большинстве корпоративных систем **представление по шаблону** должно выступать исключительно в качестве представления. В более простых системах его можно использовать как контроллер, а возможно, и как модель, хотя я настоятельно рекомендую максимально отделять модель от представления. Если **представление по шаблону** выполняет еще какие-либо функции, помимо функций представления, необходимо тщательно следить за тем, чтобы реализацию этих функций обеспечивал вспомогательный объект, а не страница сервера. Выполнение обязанностей контроллера и модели подразумевает наличие программной логики, которая, как и всякая другая программная логика, должна находиться во вспомогательном объекте.

Система, связанная с применением шаблонов, требует дополнительной обработки Web-сервером. Эта обработка может выполняться посредством компиляции страницы сразу после ее создания, компиляции страницы при поступлении первого запроса или же интерпретации страницы при поступлении каждого запроса. Последний вариант, очевидно, является неудачным, особенно если обработка страницы занимает определенное время.

Работая с **представлением по шаблону**, следует обращать внимание на исключения. Если исключение срабатывает "по пути" к Web-контейнеру, оно может прервать обработку страницы и привести к появлению в окне обозревателя весьма странного содержимого. Понаблюдайте, как Web-сервер обрабатывает исключения. Если он делает что-то не то, рекомендую обрабатывать все исключения самому, перехватывая их во вспомогательном классе (еще одна причина, по которой следует избегать скриптлетов).

Использование сценариев

Хотя наиболее популярной формой **представления по шаблону** были и остаются страницы сервера, в случае необходимости их можно заменить сценариями. Я уже встречал сценарии Perl, написанные по типу **представления по шаблону**. Главное при написании такого сценария — избежать конкатенации строк, используя вызовы функций, которые будут возвращать необходимые дескрипторы (это особенно хорошо демонстрируют сценарии CGI). Данный прием позволит создавать сценарий на привычном языке программирования, избегая досадных "вкраплений" в программную логику в виде функций вывода строк на экран.

Назначение

Основными альтернативами для реализации представления в системе **модель–представление–контроллер** являются **представление по шаблону** и **представление с преобразованием (Transform View, 379)**. Преимущество **представления по шаблону** состоит в том,

что оно позволяет оформлять представление в соответствии со структурой страницы. Данная возможность весьма привлекательна для тех, кто не умеет программировать. В частности, она позволяет воплотить в жизнь идею, когда графический дизайнер занимается проектированием страницы, а программист работает над вспомогательным объектом.

Представление по шаблону имеет два существенных недостатка. Во-первых, реализуя представление в виде страницы сервера, последнюю весьма легко переполнить логикой. Это значительно усложнит ее дальнейшую поддержку, особенно для людей, не являющихся программистами. Необходимо тщательно следить за тем, чтобы страница оставалась простой и ориентированной только на отображение, а вся программная логика была реализована во вспомогательном объекте. Во-вторых, **представление по шаблону** сложнее тестировать, чем **представление с преобразованием**. Большинство реализаций **представления по шаблону** ориентированы на использование в рамках Web-сервера, в результате чего их невозможно или практически невозможно протестировать в другом контексте. Реализации **представления с преобразованием** значительно легче поддаются тестированию и могут функционировать и при отсутствии запущенного Web-сервера.

Обсуждая варианты представления, нельзя не вспомнить о **двухэтапном представлении (Two Step View, 383)**. В зависимости от используемой схемы шаблона, это типовое решение можно реализовать и с применением специальных дескрипторов. Тем не менее реализация **двухэтапного представления** на основе **представления с преобразованием** может оказаться значительно легче. Не забывайте об этом!

Пример: использование страницы JSP в качестве представления с вынесением контроллера в отдельный объект (Java)

Если страница JSP применяется исключительно в качестве представления, она всегда будет запускаться контроллером, а не непосредственно контейнером сервлетов. Таким образом, странице JSP нужно передать всю информацию, необходимую для отображения результатов выполнения запроса. Для этого контроллер может создать вспомогательный объект и передать его странице JSP, используя HTTP-запрос. Этот прием уже рассматривался в примере простого отображения для **контроллера страниц (Page Controller, 350)**. Метод обработки запроса сервлетом выглядит как показано ниже.

```
class ArtistController...

   public void doGet(HttpServletRequest request,
↳HttpServletResponse response)
         throws IOException, ServletException {
      Artist artist = Artist.findNamed(request.getParameter(
↳"name"));
      if (artist == null)
         forward("/MissingArtistError.jsp", request, response);
      else {
         request.setAttribute("helper", new ArtistHelper(
↳artist));
         forward("/artist.jsp", request, response);
      }
   }
```

Итак, мы создали вспомогательный объект и поместили его в запрос. Теперь страница сервера сможет определить вспомогательный объект посредством дескриптора `useBean`.

```
<jsp:useBean id="helper" type="actionController.ArtistHelper"
↳scope="request"/>
```

Создав вспомогательный объект, можно использовать его для доступа к информации, которую нужно отобразить. Необходимая информация о модели была передана вспомогательному объекту в момент его создания.

```
class ArtistHelper...

   private Artist artist;
   public ArtistHelper(Artist artist) {
      this.artist = artist;
   }
```

Как уже отмечалось, вспомогательный объект применяется для извлечения из модели нужной информации. В самом простом случае можно реализовать метод, извлекающий некоторые простые данные, например имя исполнителя.

```
class ArtistHelper...

   public String getName() {
      return artist.getName();
   }
```

Затем осуществляется доступ к этой информации с помощью Java-выражения

```
<B> <%=helper.getName()%></B>
```

или обращения к свойству:

```
<B><jsp:getProperty name="helper" property="name"/></B>
```

Выбор между выражениями и свойствами зависит от предпочтений человека, который редактирует страницу JSP. Программистам больше нравится работать с выражениями, в основном из-за их читабельности и компактности, однако для Web-дизайнеров это будет довольно трудно. Непрограммисты, скорее всего, выберут обращение к свойству с помощью дескрипторов, потому что последние укладываются в общую структуру HTML и позволяют не наделать ошибок.

Использование вспомогательного объекта помогает избежать написания кошмарного кода скриптлетов. Если вы хотите отобразить список альбомов заданного исполнителя, вам понадобится реализовать цикл, что можно сделать посредством скриптлета, внедренного в страницу сервера.

```
<UL>
<%
   for (Iterator it = helper.getAlbums().iterator();
↳it.hasNext();) {
      Album album = (Album) it.next();%>
```

```
   <LI><%=album.getTitle()%></LI>
<%  }  %>
</UL>
```

По-моему, эта веселенькая смесь Java и HTML выглядит просто ужасно, не правда ли? Вместо написания скриптлета цикл можно переместить во вспомогательный объект.

```
class ArtistHelper...

   public String getAlbumList() {
      StringBuffer result = new StringBuffer();
      result.append("<UL>");
      for (Iterator it = getAlbums().iterator();
➥it.hasNext();) {
         Album album = (Album) it.next();
         result.append("<LI>");
         result.append(album.getTitle());
         result.append("</LI>");
      }
      result.append("</UL>");
      return result.toString();
   }
   public List getAlbums() {
      return artist.getAlbums();
}
```

На мой взгляд, данный подход гораздо легче, потому что в нем задействовано относительно небольшое количество кода HTML. Кроме того, он позволяет получать доступ к списку альбомов с помощью свойства, а не Java-выражения. Многие разработчики не любят помещать код HTML во вспомогательные объекты, и я с ними полностью согласен. Тем не менее при необходимости выбора между данным решением и скриптлетами я отдам безусловное предпочтение использованию кода HTML во вспомогательных объектах.

Вообще говоря, для итерации по списку альбомов лучше всего воспользоваться специальным дескриптором `forEach`.

```
<UL><tag:forEach host = "helper" collection = "albums"
➥id = "each">
   <LI><jsp:getProperty name="each" property="title"/></LI>
</tag:forEach></UL>
```

Это наилучшая альтернатива, поскольку она избавляет страницу JSP от необходимости внедрения скриптлетов, а вспомогательный объект — от необходимости использования кода HTML.

Пример: страница сервера ASP.NET (C#)

Продолжим пример, начатый нами при рассмотрении **контроллера страниц** (стр. 350). Напомню, что в нем шла речь об очках, заработанных бэтсменом за одну подачу в крикетном матче. Возможно, некоторые все еще думают, что крикет — это маленькое

назойливое насекомое[1]. Впрочем, я не стану проявлять благородное возмущение, а также распевать хвалебные оды, посвященные самому изысканному виду спорта, правила которого сохраняются неизменными на протяжении многих веков. Ограничусь лишь упоминанием того факта, что создаваемая нами страница ASP.NET отображает три элемента информации.

- Идентификатор матча.
- Команда, очки которой отображены на экране, и номер подачи, во время которой эти очки были набраны.
- Таблица, содержащая имя каждого бэтсмена, набранные им очки, а также интенсивность его перебежек (количество перебежек, деленное на количество отбитых мячей).

Пусть вас не беспокоит эта статистика. Крикет полон статистики; пожалуй, наибольший вклад этой игры в историю человечества — предоставление непонятной статистики для эксцентричных читателей газет.

Рассматривая пример **контроллера страниц**, мы обсуждали, как происходит обработка Web-запроса. Напомним, что в данном случае роль контроллера и представления исполняет страница ASP.NET с расширением `aspx`. Чтобы избежать внедрения скриптлетов в страницу-контроллер, мы применяем механизм разделения кода и представления (code behind), связывая страницу ASP.NET (представление) с отдельным классом (кодом).

```
<%@ Page language="c#" Codebehind="bat.aspx.cs"
↳AutoEventWireup="false" trace="False"
↳Inherits="batsmen.BattingPage" %>
```

Страница может напрямую использовать свойства и методы связанного с ней класса. Более того, в этом классе можно определить метод `Page_Load`, который будет выполнять обработку запроса. В нашем примере я реализовал метод `Page_Load` в виде шаблонного метода [20] и поместил его в **супертип слоя (Layer Supertype, 491)**.

```
class CricketPage...

      protected void Page_Load(object sender,
↳System.EventArgs e) {
         db = new OleDbConnection(DB.ConnectionString);
         if (hasMissingParameters())
            errorTransfer (missingParameterMessage);
         DataSet ds = getData();
         if (hasNoData (ds))
            errorTransfer ("No data matches your request");
         applyDomainLogic (ds);
         DataBind();
         prepareUI(ds);
   }
```

[1] В английском языке слово *cricket* означает и "крикет" и "сверчок". — *Прим. пер.*

Поскольку в данном примере речь идет **о представлении по шаблону**, в этом фрагменте кода нас интересуют только две последние строчки. Вызов метода `DataBind()` связывает переменные страниц с соответствующими источниками данных. Этого вполне достаточно для простых случаев. В более сложных случаях нам понадобится еще и последняя строка кода. Она вызывает метод класса, связанного с конкретной страницей, чтобы тот подготовил все объекты, необходимые для использования представления.

В нашем примере представление ASP.NET использует идентификатор матча, имя команды и номер подачи. Все эти значения передаются странице в виде параметров HTTP-запроса и могут быть определены как свойства класса, связанного со страницей.

```
class BattingPage...

      protected String team {
         get {return Request.Params["team"];}
      }
      protected String match {
         get {return Request.Params["match"];}
      }
      protected String innings {
         get {return Request.Params["innings"];}
      }
      protected String ordinalInnings{
         get {return (innings == "1") ? "1st" : "2nd";}
      }
```

Определив свойства, можно использовать их в тексте страницы.

```
<P>
   Match id:
   <asp:label id="matchLabel" Text="<%# match %>" runat="server"
↳font-bold="True">
   </asp:label> 
</P>
<P>
   <asp:label id=teamLabel Text="<%# team %>" runat="server"
↳font-bold="True">
   </asp:label> 
   <asp:Label id=inningsLabel Text="<%# ordinalInnings %>"
↳runat="server">
   </asp:Label> innings</P>
<P>
```

Отобразить таблицу немного сложнее. Впрочем, на практике этот процесс можно значительно упростить, используя средства графического проектирования Visual Studio. Эта среда разработки содержит элемент управления `DataGrid`, который может быть связан с таблицей из набора данных `DataSet`. Я могу выполнить связывание в методе `prepareUI`, который вызывается методом `Page_Load`.

```
class BattingPage...

      override protected void prepareUI(DataSet ds) {
```

```
        DataGrid1.DataSource = ds;
        DataGrid1.DataBind();
    }
```

Следует сказать несколько слов о классе `Batting`. Это **модуль таблицы (Table Module, 148)**, который содержит в себе логику домена для таблицы `batting` из базы данных крикетных матчей. Значением его свойств являются данные из таблицы `batting`, а также дополнительные данные, вычисленные посредством применения логики домена. В нашем случае дополнительными данными является интенсивность перебежек, которая вычисляется в **модуле таблицы**, а не извлекается из базы данных.

Используя дескриптор ASP.NET `DataGrid`, можно указать, какие столбцы таблицы должны быть отображены на Web-странице, а также описать форматирование этой таблицы. В данном примере понадобится отобразить столбцы с именем игрока, количеством перебежек (т.е. очков) и их интенсивностью.

```
<asp:DataGrid id="DataGrid1" runat="server" Width="480px"
      Height="171px" BorderColor="#336666"
      BorderStyle="Double" BorderWidth="3px"
      BackColor="White" CellPadding="4" GridLines="Horizontal"
      AutoGenerateColumns="False">
   <SelectedItemStyle Font-Bold="True" ForeColor="White"
↳BackColor="#339966"></
      SelectedItemStyle>
   <ItemStyle ForeColor="#333333" BackColor="White"></ItemStyle>
   <HeaderStyle Font-Bold="True" ForeColor="White"
      BackColor="#336666"></HeaderStyle>
   <FooterStyle ForeColor="#333333" BackColor="White">
   </FooterStyle>
   <Columns>
      <asp:BoundColumn DataField="name" HeaderText="Batsman">
         <HeaderStyle Width="70px"></HeaderStyle>
      </asp:BoundColumn>
      <asp:BoundColumn DataField="runs" HeaderText="Runs">
         <HeaderStyle Width="30px"></HeaderStyle>
      </asp:BoundColumn>
      <asp:BoundColumn DataField="rateString" HeaderText="Rate">
         <HeaderStyle Width="30px"></HeaderStyle>
      </asp:BoundColumn>
   </Columns>
   <PagerStyle HorizontalAlign="Center" ForeColor="White"
↳BackColor="#336666"
      Mode="NumericPages"></PagerStyle>
</asp:DataGrid></P>
```

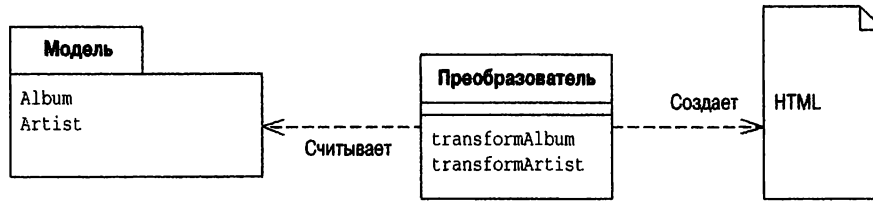
Конечно, приведенный выше код HTML выглядит довольно сложным. К счастью, если вы работаете в Visual Studio, вам не придется создавать его вручную. Вместо этого настройка элемента управления `DataGrid` (как, впрочем, и остальных элементов страницы) осуществляется посредством страниц свойств.

Возможность внедрять в Web-страницу элементы управления Web-форм, которые умеют работать с абстракциями ADO.NET `DataSet` и `DataTable`, имеет свои преимущества и ограничения. Основное преимущество такого подхода состоит в том, что средства

Visual Studio позволяют передавать данные элементам управления с помощью объектов `DataSet`. К сожалению, этот прием выглядит таким простым только при использовании типовых решений наподобие **модуля таблицы**. Если же логика домена очень сложна, она требует применения **модели предметной области (Domain Model, 140)**. В этом случае, чтобы воспользоваться средствами Visual Studio, для **модели предметной области** необходимо создать собственный объект `DataSet`.

Представление с преобразованием (Transform View)

Представление, которое поочередно обрабатывает элементы данных домена и преобразует их в код HTML



Выполняя запросы к слоям домена и источника данных, вы получаете от них нужные данные, однако без форматирования, необходимого для создания Web-страницы. Визуализацией полученных данных в элементы Web-страницы занимается объект, выполняющий роль представления в системе **модель–представление–контроллер (Model View Controller, 347)**. В типовом решении **представление с преобразованием** процесс визуализации данных рассматривается как преобразование, на вход которого подаются данные из модели, а на выходе принимается код HTML.

Принцип действия

В основе типового решения **представление с преобразованием** лежит идея написания программы, которая бы просматривала данные домена и преобразовывала их в код HTML. В процессе выполнения такая программа последовательно проходит по структуре данных домена и, обнаруживая новый фрагмент данных, создает их описание в терминах HTML. Для большей наглядности попробуйте представить себе метод `renderCustomer`, который преобразует объект `Customer` ("покупатель") в формат HTML. Если объект содержит множество заказов, метод `renderCustomer` последовательно просматривает список заказов, вызывая для каждого из них метод `renderOrder`.

Основное отличие **представления с преобразованием** от **представления по шаблону (Template View, 368)** заключается в способе организации представления. **Представление по шаблону** организовано с учетом размещения на экране выходных данных. **Представление с преобразованием** ориентировано на использование отдельных преобразований для каждого вида входных данных. Преобразованиями управляет нечто наподобие простого цикла, который поочередно просматривает каждый входной элемент, подбирает для него подходящее преобразование и применяет это преобразование. Таким образом, правила

представления с преобразованием могут быть организованы в любом порядке — на результат это не повлияет.

Представление с преобразованием может быть написано на любом языке. Тем не менее на данный момент наиболее популярным языком для написания **представлений с преобразованием** является XSLT. Интересно отметить, что XSLT является языком функционального программирования, как LISP, Haskell и другие языки, которые так и не смогли оказаться в первом эшелоне средств, применяемых для написания информационных систем. Подобно этим языкам, XSLT имеет довольно своеобразную структуру. Например, вместо того чтобы вызывать методы явно, XSLT выделяет XML-элементы в данных домена и запускает соответствующие преобразования, необходимые для визуализации этих элементов.

Для применения преобразований XSLT данные домена должны находиться в формате XML. Этого проще достичь, когда логика домена сразу же возвращает данные в формате XML или любом другом, который может быть легко преобразован в XML, например объекты .NET. В противном случае придется самостоятельно генерировать код XML — возможно, с использованием **объекта переноса данных (Data Transfer Object, 419)**, который способен сериализовать свое содержимое в формат XML. В этом случае сбор данных для помещения в **объект переноса данных** может осуществляться посредством подходящего интерфейса API. В более простых случаях можно воспользоваться **сценарием транзакции (Transaction Script, 133)**, который будет напрямую возвращать код XML.

Код XML, поступающий на вход преобразования, не обязательно должен быть строкой (если только этого не требует канал передачи данных). Как правило, гораздо быстрее и легче организовать входные данные в модель DOM.

Данные домена в формате XML передаются процессору XSLT. В последнее время на рынке программного обеспечения доступно все больше и больше коммерческих процессоров XSLT. Логика преобразования содержится в таблице стилей XSLT, которая также передается процессору. Последний применяет таблицу стилей к входным данным XML и преобразует их в код HTML, который сразу же может быть помещен в HTTP-запрос.

Назначение

Выбор между **представлением с преобразованием** и **представлением по шаблону** зависит от того, какую среду разработки предпочитает команда, занимающаяся проектированием представлений. Ключевым фактором здесь является наличие необходимых средств. Для написания **представлений по шаблону** можно использовать практически любой HTML-редактор, в то время как средства для работы с XSLT не так распространены, да и возможностей у них значительно меньше. Кроме того, язык XSLT представляет собой достаточно сложную смесь функционального программирования и синтаксиса XML, а потому овладеть им сможет далеко не каждый.

Одним из преимуществ XSLT является хорошая переносимость практически на все Web-платформы. Одну и ту же таблицу стилей XSLT можно применять для преобразования данных XML, созданных на основе объектов J2EE или .NET, что позволяет применять общие HTML-представления для данных, полученных из различных источников.

Применение XSLT значительно упрощает процесс отображения, если представление создается на основе документа XML. Другие среды разработки могут потребовать преобразования такого документа в объект или модель XML DOM, что не так просто сделать. В этом плане XSLT прекрасно вписывается в концепцию работы с XML.

Помимо всего прочего, **представление с преобразованием** лишено двух недостатков, присущих **представлению по шаблону**. Во-первых, преобразования изначально направлены на визуализацию данных в формат HTML, что позволяет избежать внедрения в представление слишком большого количества логики. Кроме того, **представление с преобразованием** легче тестировать, поскольку применение таблицы стилей не требует наличия функционирующего Web-сервера.

Представление с преобразованием преобразует данные домена в формате XML в код HTML. Чтобы кардинально изменить внешний вид Web-сайта, может понадобиться внести изменения в целый ряд программ преобразования. Этот процесс можно значительно облегчить, если использовать общие программы преобразования, в частности включения XSLT. Следует отметить, что применять общие преобразования гораздо проще с использованием **представления с преобразованием**, чем **представления по шаблону**. Если же вам необходим быстрый способ внесения глобальных изменений или поддержка разных способов отображения одних и тех же данных, подумайте об использовании **двухэтапного представления (Two Step View, 383)**.

Пример: простое преобразование (Java)

Для выполнения простого преобразования необходимо определить метод, который будет применять к результатам выполнения запроса нужную таблицу стилей. Кроме того, понадобится подготовить саму таблицу стилей. Большая часть логики по обработке запросов универсальна, поэтому в качестве контроллера можно использовать **контроллер запросов (Front Controller, 362)**. Здесь описан только необходимый объект команды, а применение объекта команды в выполнении запроса рассматривается в разделе, посвященном **контроллеру запросов**.

Все, что делает объект команды, — это вызывает методы модели для получения документа XML и передает его процессору XSLT.

```
class AlbumCommand...

   public void process() {
      try {
         Album album = Album.findNamed(request.getParameter(
↳"name"));
         Assert.notNull(album);
         PrintWriter out = response.getWriter();
         XsltProcessor processor = new SingleStepXsltProcessor(
↳"album.xsl");
         out.print(processor.getTransformation(
↳album.toXmlDocument()));
      } catch (Exception e) {
         throw new ApplicationException(e);
      }
   }
```

Полученный документ XML может выглядеть примерно так, как показано ниже.

```
<album>
   <title>Stormcock</title>
   <artist>Roy Harper</artist>
```

```
   <trackList>
      <track><title>Hors d'Oeuvres</title><time>8:37</time>
      </track>
      <track><title>The Same Old Rock</title><time>12:24</time>
      </track>
      <track><title>One Man Rock and Roll Band</title><time>7:23
      </time></track>
      <track><title>Me and My Woman</title><time>13:01</time>
      </track>
   </trackList>
</album>
```

Преобразование документа XML выполняется с помощью шаблона XSLT. Каждая инструкция `template` соответствует определенному элементу XML и преобразует его в соответствующий элемент HTML-страницы. В данном примере я применил очень простое форматирование, только чтобы продемонстрировать, как это делается. Перечисленные ниже инструкции соответствуют основным элементам файла XML.

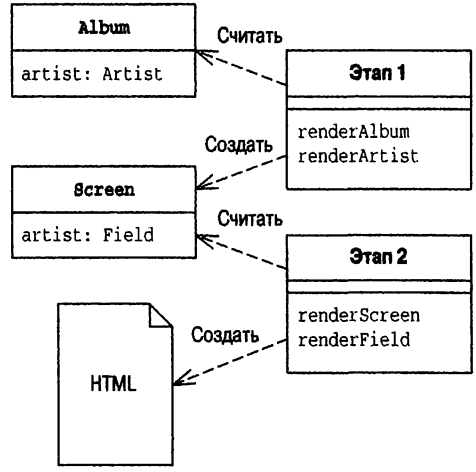
```
<xsl:template match="album">
   <HTML><BODY bgcolor="white">
   <xsl:apply-templates/>
   </BODY></HTML>
</xsl:template>
<xsl:template match="album/title">
   <h1><xsl:apply-templates/></h1>
</xsl:template>
<xsl:template match="artist">
       <P><B>Artist: </B><xsl:apply-templates/></P>
</xsl:template>
```

Остальные инструкции применяются для обработки таблицы, чередующиеся строки которой могут быть выделены разными цветами. Это хороший пример действия, которое невозможно выполнить путем наложения таблиц стилей, однако можно осуществить с использованием XML.

```
<xsl:template match="trackList">
   <table><xsl:apply-templates/></table>
</xsl:template>
<xsl:template match="track">
   <xsl:variable name="bgcolor">
      <xsl:choose>
        <xsl:when test="(position() mod 2) = 1">linen</xsl:when>
        <xsl:otherwise>white</xsl:otherwise>
      </xsl:choose>
   </xsl:variable>
   <tr bgcolor="{$bgcolor}"><xsl:apply-templates/></tr>
</xsl:template>
<xsl:template match="track/title">
   <td><xsl:apply-templates/></td>
</xsl:template>
<xsl:template match="track/time">
   <td><xsl:apply-templates/></td>
</xsl:template>
```

Двухэтапное представление (Two Step View)

Выполняет визуализацию данных домена в два этапа: вначале формирует некое подобие логической страницы, после чего преобразует логическую страницу в формат HTML



Если разрабатываемое Web-приложение содержит много страниц, вам, скорее всего, захочется оформить и организовать их в одном стиле. Действительно, сайт, каждая страница которого имеет совершенно другой вид, способен окончательно запутать посетителей. Кроме того, вам может понадобиться быстрый способ глобального изменения внешнего вида приложения. К сожалению, использование **представления по шаблону (Template View, 368)** или **представления с преобразованием (Transform View, 379)** значительно затруднит этот процесс. Данные типовые решения не исключают дублирования фрагментов представлений в нескольких страницах или компонентах преобразований, в результате чего глобальное изменение внешнего вида может потребовать внесения изменений в несколько различных файлов.

Избежать подобных проблем помогает **двухэтапное представление**, выполняющее преобразование данных в два этапа. Вначале данные модели преобразуются в логическое представление, не содержащее какого-либо специального форматирования. Затем полученное логическое представление путем применения необходимого форматирования трансформируется в HTML. В этом случае глобальные изменения внешнего вида Web-сайта приходится проводить только в одном месте, а именно на втором этапе преобразования. Кроме того, подобная схема позволяет отображать одни и те же данные несколькими способами, что также определяется на втором этапе.

Принцип действия

Главной особенностью этого типового решения является выполнение преобразования данных в формат HTML в два этапа. На первом этапе информация, полученная от

модели, организуется в некую логическую структуру, которая описывает визуальные элементы будущего отображения, однако еще не содержит кода HTML. На втором этапе полученная структура преобразуется в код HTML.

Упомянутая промежуточная структура представляет собой некоторое подобие логического "экрана". Ее элементами могут быть поля ввода, верхние и нижние колонтитулы, таблицы, переключатели и т.п. В связи с этим данную структуру можно по праву назвать моделью представления. Очевидно также, что она вынуждает будущие страницы сайта следовать одному стилю. Для большей наглядности модель представления можно представить себе как систему, которая определяет элементы управления страницы и содержащиеся в них данные, однако не описывает их внешний вид в терминах HTML.

Для построения каждого логического окна применяется собственный код. Методы первого этапа обращаются к модели данных домена, будь то база данных, **модель предметной области (Domain Model, 140)** или **объект переноса данных (Data Transfer Object, 419)**, извлекают из нее информацию, необходимую для построения экрана, и помещают эту информацию в логическое представление.

На втором этапе полученная логическая структура преобразуется в код HTML. Методы второго этапа "знают", какие элементы есть в логической структуре и как визуализировать каждый из этих элементов. Таким образом, система с множеством экранов может быть трансформирована в код HTML путем единственного прохождения второго этапа, благодаря чему решение о варианте преобразования в HTML принимается в одном месте. Разумеется, воспользоваться таким преимуществом удастся только в том случае, когда логические экраны компонуются по одному принципу.

Существует несколько способов построения **двухэтапного представления**. Наиболее простой из них — двухэтапное применение XSLT. Одноэтапное применение технологии XSLT следует подходу, реализованному в **представлении с преобразованием**, при котором каждой странице соответствует своя таблица стилей XSLT, преобразующая данные домена в формате XML в код HTML. В двухэтапном варианте этого подхода применяются две таблицы стилей XSLT: первая преобразует данные домена в формате XML в логическое представление в формате XML, а вторая преобразует логическое представление в формате XML в код HTML.

Возможной альтернативой этому подходу является использование классов. В этом случае логическое представление определяется как набор классов: класс таблицы, класс строки и т.п. (рис. 14.4). Методы первого этапа извлекают данные домена и создают экземпляры описанных классов, помещая данные в структуру, моделирующую логический экран. На втором этапе для полученных экземпляров классов генерируется код HTML. Это делают либо сами классы, либо специальный класс, предназначенный для выполнения визуализации.

И первый и второй подход основан на применении **представления с преобразованием**. Вместо этого **двухэтапное представление** может быть построено на основе **представления по шаблону**. В этом случае шаблон страницы составляется с учетом структуры логического экрана, например:

```
<field label = "Name" value = "getName" />
```

После этого система шаблонов преобразует "логические" дескрипторы, подобные показанному выше, в формат HTML. При такой схеме определение страницы не содержит кода HTML, а только дескрипторы логического экрана, вследствие чего полученная

страница будет представлять собой документ XML и не сможет быть отредактирована с помощью HTML-редакторов типа WYSIWYG.

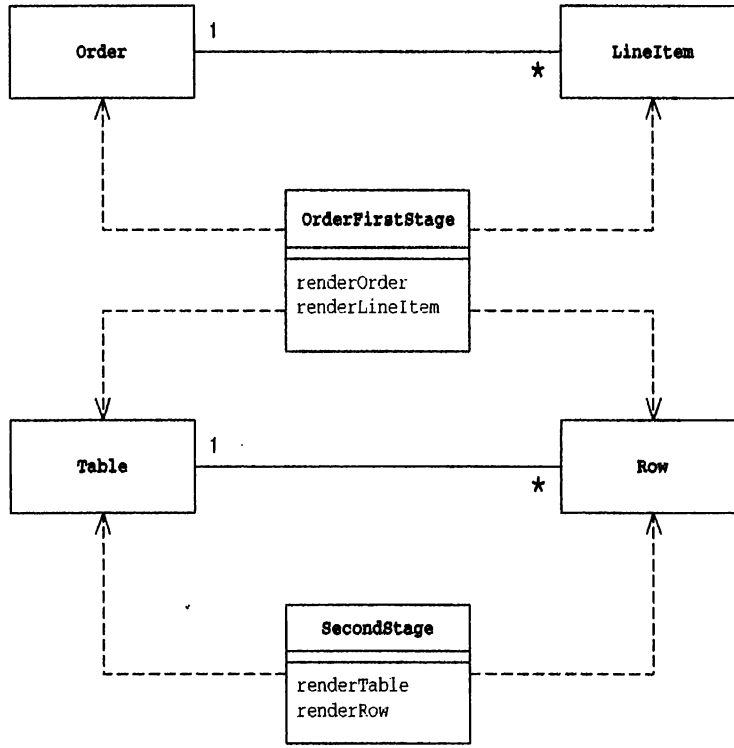


Рис. 14.4. Пример двухэтапного преобразования с использованием классов

Назначение

Главным преимуществом **двухэтапного представления** является возможность разбить преобразование данных на два этапа, что облегчает проведение глобальных изменений (рис. 14.5). Подобная схема преобразования может оказаться чрезвычайно полезной в двух ситуациях: Web-приложения с несколькими вариантами внешнего вида и Web-приложения с одним вариантом внешнего вида. Что касается Web-приложений с несколькими вариантами внешнего вида, они еще не так распространены, однако все больше и больше входят в употребление. В таких приложениях одна и та же базовая функциональность используется разными компаниями и сайт каждой компании имеет собственное оформление. Хорошим примером подобных приложений являются системы бронирования билетов различных авиакомпаний. Каждая из этих систем имеет свой внешний вид, однако, приглядевшись к структуре их страниц, легко заметить, что в их основе лежит одно и то же базовое решение. На мой взгляд, причина проста: всем авиакомпаниям нужна одинаковая функциональность, но (разумеется!) совершенно разное оформление.

Намного чаще встречаются приложения с одним вариантом внешнего вида. Как правило, таким приложением управляет одна компания, которая хочет, чтобы все страницы ее сайта были оформлены в одинаковом стиле. Это самое простое решение, что, собственно, и объясняет высокую степень распространенности подобных приложений.

Одноэтапный вариант представления, будь то **представление по шаблону** или **представление с преобразованием**, предусматривает по одному компоненту представления для каждой Web-страницы приложения (рис. 14.6). В **двухэтапном представлении** визуализация данных выполняется в два этапа, что требует по одному представлению первого этапа для каждой страницы приложения и единственное представление второго этапа для всего приложения (рис. 14.7). Последняя схема значительно облегчает изменение внешнего вида сайта на втором этапе, поскольку каждое такое изменение распространяется сразу на весь сайт.

Данное преимущество становится еще более очевидным для приложений с несколькими вариантами внешнего вида. Реализация таких приложений предусматривает по одному компоненту одноэтапного представления для каждой комбинации интерфейсного экрана и варианта внешнего вида (рис. 14.8). Таким образом, приложению, состоящему из 10 страниц и имеющему три варианта внешнего вида, потребуется 30 компонентов одноэтапных представлений! В свою очередь, при использовании **двухэтапного представления** (рис. 14.9) вы сможете обойтись десятью представлениями первого этапа и тремя представлениями второго этапа. Чем больше интерфейсных экранов и вариантов внешнего вида есть у приложения, тем большим будет выигрыш.

Несмотря на все преимущества **двухэтапного представления**, возможность его использования полностью зависит от того, сможете ли вы создать логическое представление, которое действительно соответствует вашим нуждам. Так, **двухэтапное представление** не подойдет для "навороченного" Web-сайта, каждая страница которого должна выглядеть по-другому. Экраны такого приложения имеют слишком мало общего, чтобы результирующее логическое представление получилось достаточно простым. Вообще говоря, использование единого логического представления существенно ограничивает возможности оформления Web-сайтов, и для многих из них такое ограничение является непосильно строгим.

Еще один недостаток **двухэтапного представления** заключается в необходимости использовать определенные программные средства. Создавать HTML-страницы в соответствии с **представлением по шаблону** могут и Web-дизайнеры, не имеющие навыков программирования, однако применение **двухэтапного представления** требует написания программного кода контроллера и классов, выполняющих преобразование. Каждое изменение такого сайта требует участия программистов.

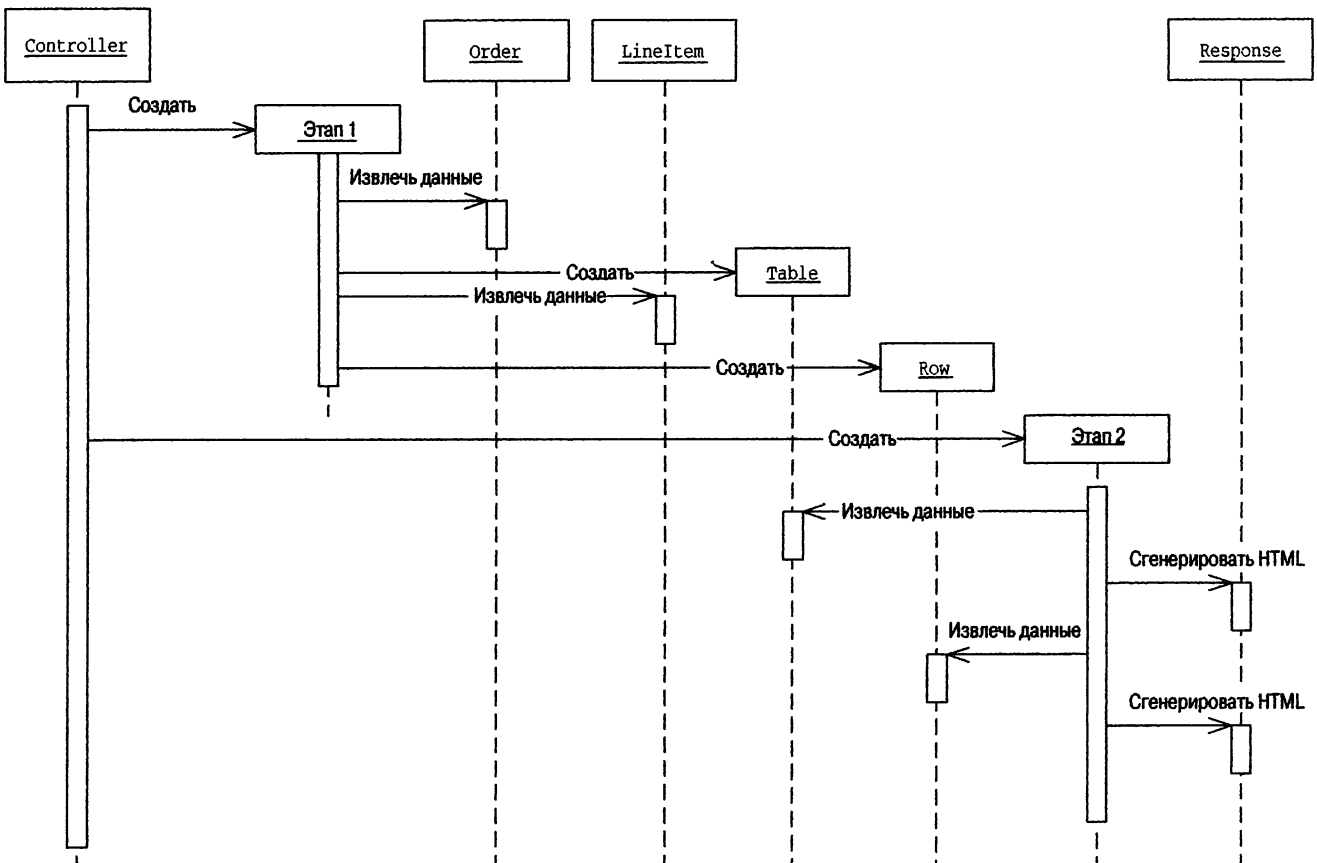


Рис. 14.5. Последовательность двухэтапного преобразования данных

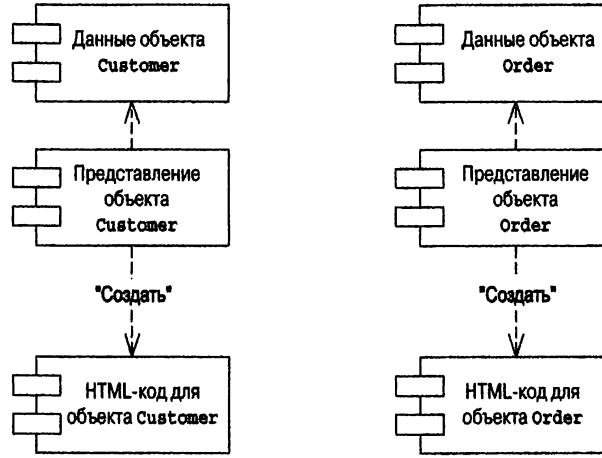


Рис. 14.6. Одноэтапное представление с одним вариантом внешнего вида

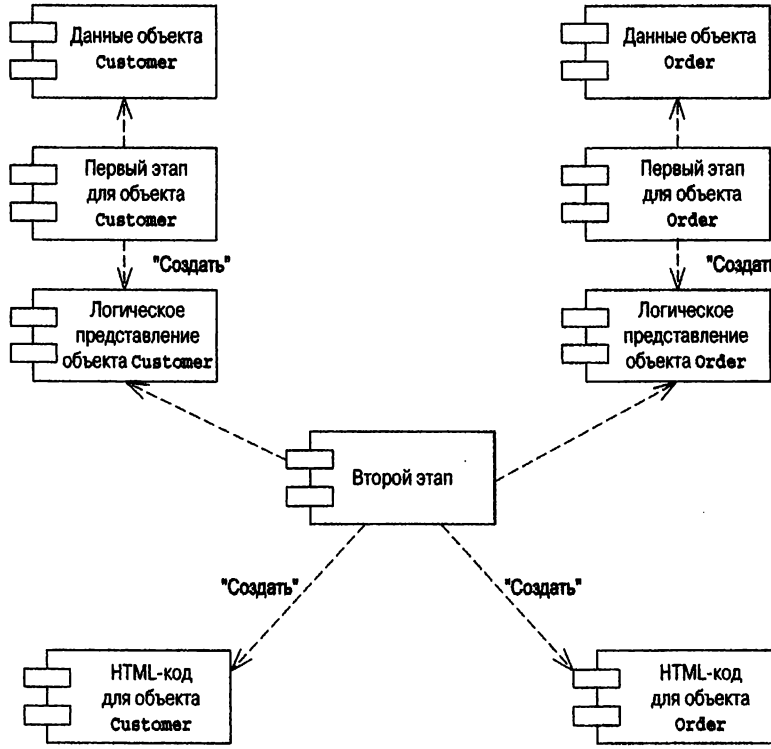


Рис. 14.7. Двухэтапное представление с одним вариантом внешнего вида

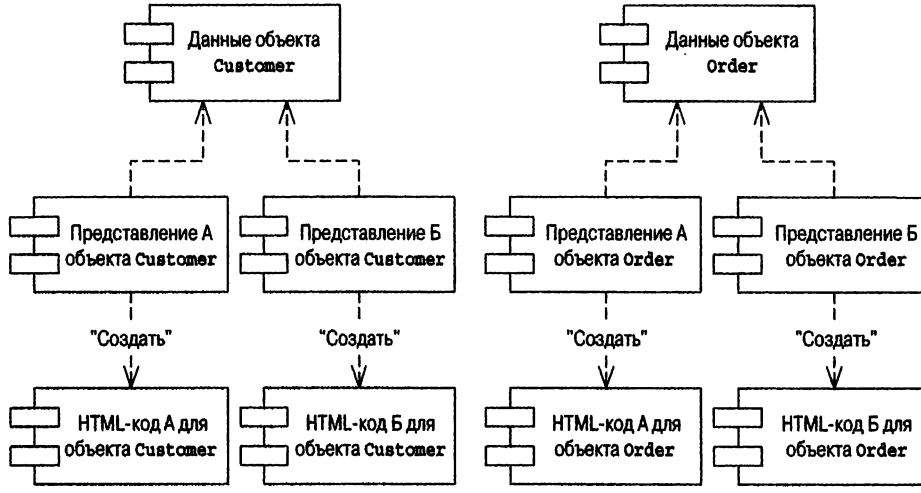


Рис. 14.8. Одноэтапное представление с двумя вариантами внешнего вида

Нельзя отрицать и того, что использование **двухэтапного представления** с его несколькими слоями значительно усложняет программную модель. Впрочем, при определенном навыке работы с данной моделью она перестает казаться сложной, а заодно и помогает избежать написания повторяющихся фрагментов кода.

Возможность использования нескольких вариантов внешнего вида допускает разные реализации второго этапа представления для различных устройств вывода, например обычного Web-обозревателя и карманного компьютера. И вновь обе реализации должны соответствовать одному и тому же логическому экрану, что может оказаться непосильным для слишком различающихся устройств.

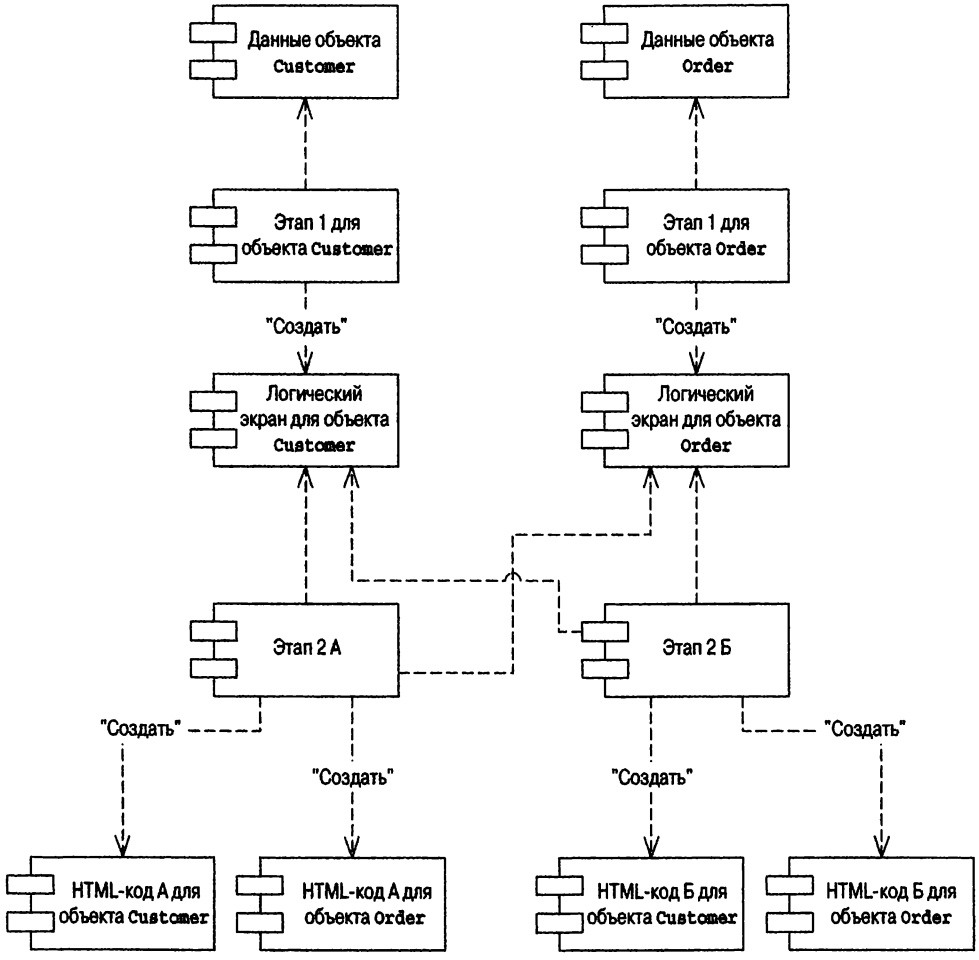


Рис. 14.9. Двухэтапное представление с двумя вариантами внешнего вида

Пример: двухэтапное применение XSLT (XSLT)

Рассмотрим вариант **двухэтапного преобразования** с применением таблиц стилей XSLT. На первом этапе данные домена, находящиеся в формате XML, преобразуются в логический экран в формате XML, а на втором этапе логический экран в формате XML преобразуется в код HTML.

Ниже приведены первоначальные данные домена в формате XML.

```
<album>
   <title>Zero Hour</title>
   <artist>Astor Piazzola</artist>
   <trackList>
     <track><title>Tanguedia III</title><time>4:39</time>
     </track>
```

```
      <track><title>Milonga del Angel</title><time>6:30</time>
      </track>
      <track><title>Concierto Para Quinteto</title><time>9:00
      </time></track>
      <track><title>Milonga Loca</title><time>3:05</time></track>
      <track><title>Michelangelo '70</title><time>2:50</time>
      </track>
      <track><title>Contrabajisimo</title><time>10:18</time>
      </track>
      <track><title>Mumuki</title><time>9:32</time></track>
   </trackList>
</album>
```

После выполнения первого этапа этот документ преобразуется в логическое представление также в формате XML.

```
<screen>
   <title>Zero Hour</title>
   <field label="Artist">Astor Piazzola</field>
   <table>
      <row><cell>Tanguedia III</cell><cell>4:39</cell></row>
      <row><cell>Milonga del Angel</cell><cell>6:30</cell></row>
      <row><cell>Concierto Para Quinteto</cell><cell>9:00</cell>
      </row>
      <row><cell>Milonga Loca</cell><cell>3:05</cell></row>
      <row><cell>Michelangelo '70</cell><cell>2:50</cell></row>
      <row><cell>Contrabajisimo</cell><cell>10:18</cell></row>
      <row><cell>Mumuki</cell><cell>9:32</cell></row>
   </table>
</screen>
```

Данное преобразование стало возможным благодаря шаблону XSLT.

```
<xsl:template match="album">
   <screen><xsl:apply-templates/></screen>
</xsl:template>
<xsl:template match="album/title">
   <title><xsl:apply-templates/></title>
</xsl:template>
<xsl:template match="artist">
   <field label="Artist"><xsl:apply-templates/></field>
</xsl:template>
<xsl:template match="trackList">
   <table><xsl:apply-templates/></table>
</xsl:template>
<xsl:template match="track">
   <row><xsl:apply-templates/></row>
</xsl:template>
<xsl:template match="track/title">
   <cell><xsl:apply-templates/></cell>
</xsl:template>
<xsl:template match="track/time">
   <cell><xsl:apply-templates/></cell>
</xsl:template>
```

Наше логическое представление получилось очень простым. Чтобы преобразовать его в код HTML, воспользуемся еще одним шаблоном XSLT.

```
<xsl:template match="screen">
   <HTML><BODY bgcolor="white">
   <xsl:apply-templates/>
   </BODY></HTML>
</xsl:template>
<xsl:template match="title">
   <h1><xsl:apply-templates/></h1>
</xsl:template><xsl:template match="field">
   <P><B><xsl:value-of select = "@label"/>: </B>
   <xsl:apply-templates/></P>
</xsl:template>
<xsl:template match="table">
   <table><xsl:apply-templates/></table>
</xsl:template>
<xsl:template match="table/row">
   <xsl:variable name="bgcolor">
      <xsl:choose>
          <xsl:when test="(position() mod 2) = 1">linen
          </xsl:when>
          <xsl:otherwise>white</xsl:otherwise>
      </xsl:choose>
   </xsl:variable>
   <tr bgcolor="{$bgcolor}"><xsl:apply-templates/></tr>
</xsl:template>
<xsl:template match="table/row/cell">
   <td><xsl:apply-templates/></td>
</xsl:template>
```

Для реализации двухэтапного представления и вынесения программного кода в отдельный класс я использовал **контроллер запросов (Front Controller, 362)**.

```
class AlbumCommand...

   public void process() {
      try {
         Album album = Album.findNamed(request.getParameter(
↳"name"));
         album = Album.findNamed("1234");
         Assert.notNull(album);
         PrintWriter out = response.getWriter();
         XsltProcessor processor = new TwoStepXsltProcessor(
↳"album2.xsl", "second.xsl");
         out.print(processor.getTransformation(
↳album.toXmlDocument()));
      } catch (Exception e) {
         throw new ApplicationException(e);
      }
   }
```

Сравните этот пример с одноэтапным применением XSLT, которое описывалось в разделе, посвященном **представлению с преобразованием**. Для изменения цвета чередующихся строк таблицы **в представлении с преобразованием** необходимо отредактировать все шаблоны XSLT, а в **двухэтапном представлении** — только один шаблон второго этапа. Подобную схему можно реализовать с использованием вызываемых шаблонов, однако это потребовало бы построения более изощренных конструкций языка XSLT. Тем не менее применение **двухэтапного представления** имеет и обратную сторону: полученный код HTML слишком ограничен структурой логического представления XML.

Пример: страницы JSP и пользовательские дескрипторы (Java)

Первое, что приходит в голову при выборе реализации для **двухэтапного представления**, — воспользоваться конструкциями XSLT. Между тем существуют и другие пути. В этом примере **двухэтапное представление** реализуется с помощью страниц JSP и пользовательских дескрипторов. Хотя эти средства более сложны в применении и обладают меньшими возможностями, чем XSLT, они способны продемонстрировать наличие принципиально иных способов реализации **двухэтапного представления**. Признаюсь, я немного блефую, потому что никогда не видел применения данной схемы в реальной жизни. Тем не менее я думаю, что даже такой теоретический пример поможет получить представление о возможных альтернативах шаблонам XSLT.

Ключевой момент реализации **двухэтапного представления** состоит в том, что выбор элементов для отображения на странице не зависит от выбора кода HTML, необходимого для этого отображения. В этом примере выполнением первого этапа занимаются страница JSP и ее вспомогательный объект, а выполнением второго этапа — пользовательские дескрипторы. Покажем, как выглядит упомянутая страница JSP.

```
<%@ taglib uri="2step.tld" prefix = "2step" %>
<%@ page session="false"%>
<jsp:useBean id="helper" class=
↳"actionController.AlbumConHelper"/>
<%helper.init(request, response);%>
<2step:screen>
<2step:title><jsp:getProperty name = "helper" property =
↳"title"/></2step:title>
<2step:field label = "Artist"><jsp:getProperty name = "helper"
↳property = "artist"/></2step:field>
<2step:table host = "helper" collection = "trackList" columns =
↳"title, time"/>
</2step:screen>
```

Данная страница JSP вместе со своим вспомогательным объектом выполняет роль **контроллера страниц**. Более подробно об этой схеме можно прочитать в разделе, посвященном **контроллеру страниц**, а пока рекомендую взглянуть на дескрипторы, принадлежащие пространству имен `2step`. Эти дескрипторы предназначены для запуска второго этапа преобразования. Кроме того, обратите внимание, что данная страница JSP не содержит кода HTML. Все, что в ней есть, — это дескрипторы, вызывающие запуск второго этапа, либо дескрипторы, предназначенные для извлечения данных из вспомогательного объекта.

Каждый дескриптор второго этапа имеет реализацию, которая генерирует код HTML для соответствующего элемента логического экрана. Самый простой из пользовательских дескрипторов — <title> ("название").

```
class TitleTag...

   public int doStartTag() throws JspException {
      try {
         pageContext.getOut().print("<H1>");
      } catch (IOException e) {
         throw new JspException("unable to print start");
      }
      return EVAL_BODY_INCLUDE;
   }
   public int doEndTag() throws JspException {
      try {
         pageContext.getOut().print("</H1>");
      } catch (IOException e) {
         throw new JspException("unable to print end");
      }
      return EVAL_PAGE;
   }
```

Для тех, кто еще не знает, поясню: класс пользовательского дескриптора реализует методы, которые вызываются в начале и в конце текста, заключенного в соответствующий дескриптор. В данном примере класс TitleTag просто заключает содержимое дескриптора <title> в HTML-дескриптор <H1>. Более сложный дескриптор, такой, как <field>, может содержать атрибут. Этот атрибут передается классу дескриптора с помощью set-метода.

```
class FieldTag...

   private String label;
   public void setLabel(String label) {
      this.label = label;
   }
```

После установки значения атрибута его можно использовать в качестве параметра преобразования второго этапа.

```
class FieldTag...

   public int doStartTag() throws JspException {
      try {
         pageContext.getOut().print("<P>" + label + ": <B>");
      } catch (IOException e) {
         throw new JspException("unable to print start");
      }
      return EVAL_BODY_INCLUDE;
   }
   public int doEndTag() throws JspException {
      try {
```

```
            pageContext.getOut().print("</B></P>");
         } catch (IOException e) {
            throw new JspException("how are checked exceptions
↳helping me here?");
         }
         return EVAL_PAGE;
      }
```

Самым сложным из дескрипторов второго этапа является `<table>`. Помимо выбора нужных столбцов, он описывает выделение разными цветами чередующихся строк таблицы. Реализация этого дескриптора применяется на втором этапе преобразования, поэтому выделение цветом будет относиться ко всем таблицам сайта (при наличии таковых).

Класс `TableTag` имеет закрытые поля `collectionName` (имя коллекции свойств), `hostName` (имя объекта, хранящего в себе коллекцию свойств) и `columns` (список разделенных запятыми имен столбцов).

```
class TableTag...

   private String collectionName;
   private String hostName;
   private String columns;
   public void setCollection(String collectionName) {
      this.collectionName = collectionName;
   }
   public void setHost(String hostName) {
      this.hostName = hostName;
   }
   public void setColumns(String columns) {
      this.columns = columns;
   }
```

Я создал вспомогательный метод для извлечения из объекта значения нужного свойства. Вообще говоря, в подобных случаях рекомендуется применять различные классы, поддерживающие компоненты Java Beans, а не просто вызывать методы типа "get*Что_то*", однако для нашего примера подойдет и последнее.

```
class TableTag...

   private Object getProperty(Object obj, String property)
↳throws JspException {
      try {
         String methodName = "get" + property.substring(
↳0, 1).toUpperCase() + property.substring(1);
         Object result = obj.getClass().getMethod(methodName,
↳null).invoke(obj, null);
         return result;
      } catch (Exception e) {
         throw new JspException("Unable to get property " +
↳property + " from " + obj);
      }
   }
```

Дескриптор <table> не имеет тела. При вызове метода doStartTag последний извлекает коллекцию с заданным именем из указанного свойства объекта запроса и последовательно перебирает объекты этой коллекции для генерации строк таблицы.

```
class TableTag...

   public int doStartTag() throws JspException {
      try {
         JspWriter out = pageContext.getOut();
         out.print("<table>");
         Collection coll = (Collection)
↳getPropertyFromAttribute(hostName, collectionName);
         Iterator rows = coll.iterator();
         int rowNumber = 0;
         while (rows.hasNext()) {
            out.print("<tr");
            if ((rowNumber++ % 2) == 0) out.print(
↳" bgcolor = " + HIGHLIGHT_COLOR);
            out.print(">");
            printCells(rows.next());
            out.print("</tr>");
         }
         out.print("</table>");
      } catch (IOException e) {
         throw new JspException("unable to print out");
      }
      return SKIP_BODY;
   }
   private Object getPropertyFromAttribute(String attribute,
↳String property)
      throws JspException
   {
      Object hostObject = pageContext.findAttribute(attribute);
      if (hostObject == null)
         throw new JspException("Attribute " + attribute + "
↳not found.");
      return getProperty(hostObject, property);
   }
   public static final String HIGHLIGHT_COLOR = "'linen'";
```

В процессе итерации каждая четная строка таблицы выделяется бежевым цветом.

Чтобы напечатать содержимое ячеек каждой строки, я использую имена столбцов для извлечения значений соответствующих свойств объектов коллекции.

```
class TableTag...

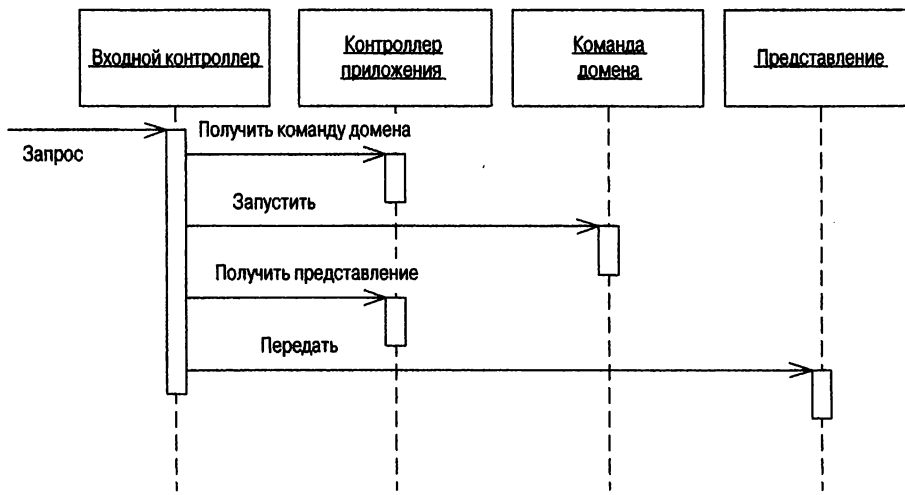
   private void printCells(Object obj) throws IOException,
↳JspException {
      JspWriter out = pageContext.getOut();
      for (int i = 0; i < getColumnList().length; i++) {
         out.print("<td>");
         out.print(getProperty(obj, getColumnList()[i]));
         out.print("</td>");
```

```
    }
}
private String[] getColumnList() {
    StringTokenizer tk = new StringTokenizer(columns, ", ");
    String[] result = new String[tk.countTokens()];
    for (int i = 0; tk.hasMoreTokens(); i++)
        result[i] = tk.nextToken();
    return result;
}
```

По сравнению с реализацией на языке XSLT данное решение накладывает меньше ограничений на структуру страниц. Например, автору отдельной страницы будет легче разместить на ней какой-нибудь "свой" элемент, которого нет на других страницах сайта. Эта возможность имеет и недостатки: допускает появление ненужных элементов оформления или случайных ошибок, сделанных людьми, которые плохо знакомы с данной схемой преобразования. Иногда ограничения действительно помогают избежать неприятностей. Команда разработчиков должна сама принять решение о том, какие преимущества для нее важнее.

Контроллер приложения (Application Controller)

Точка централизованного управления порядком отображения интерфейсных экранов и потоком функций приложения



Некоторые приложения содержат огромное количество логики, касающейся порядка отображения интерфейсных экранов: например, какой экран следует отобразить на той или иной стадии приложения. Подобное взаимодействие приложения с пользователем аналогично работе программ-мастеров, которые последовательно проводят пользователя

по набору диалоговых окон, расположенных в строго заданном порядке. В качестве других примеров можно привести диалоговые окна, которые отображаются только при определенных условиях, или же выбор нужного окна в зависимости от данных, введенных пользователем на предыдущей стадии.

В некоторой степени выполнение подобных функций обеспечивают входные контроллеры из системы **модель–представление–контроллер (Model View Controller, 347)**. Тем не менее по мере повышения сложности приложения логика выбора нужных экранов начинает дублироваться в нескольких контроллерах, что весьма нежелательно.

Во избежание подобного дублирования всю логику, касающуюся управления потоком функций приложения, можно поместить в **контроллер приложения**. В этом случае входные контроллеры будут обращаться к **контроллеру приложения** за командами, которые следует применить к модели, и за представлениями, которые необходимо использовать в определенном контексте приложения.

Принцип действия

Контроллер приложения выполняет две основные функции: выбор логики домена, которую нужно применить в конкретной ситуации, и выбор представления, которое следует отобразить в ответ на запрос. Для осуществления этих функций **контроллер приложения** поддерживает две коллекции ссылок на классы — одну для команд, выполняющихся в слое домена, и одну для представлений (рис. 14.10).

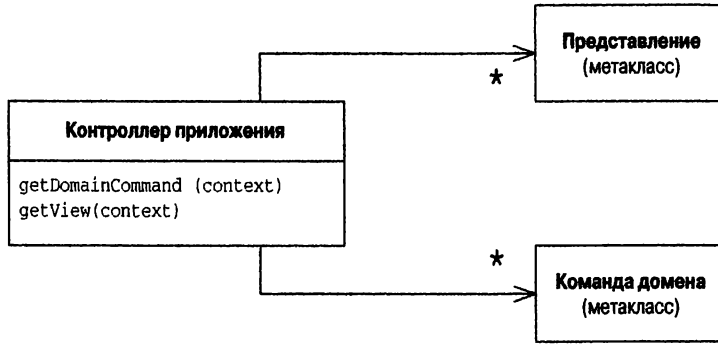


Рис. 14.10. **Контроллер приложения** *содержит две коллекции ссылок на классы: для логики домена и для представлений*

И в том и в другом случае **контроллеру приложения** необходим способ обращения к командам или представлениям. Удачным выбором является типовое решение **команда (Command)** [20], поскольку оно позволяет легко управлять выполнением фрагментов кода. При использовании языков, поддерживающих операции над функциями, **контроллер приложения** может содержать ссылки на объекты команд. Еще одной возможностью является хранение строки, которая может быть использована для вызова нужного метода посредством отражения.

В качестве команд домена могут выступать объекты команд, являющиеся частью слоя **контроллера приложения**, либо ссылки на **сценарий транзакции (Transaction Script, 133)** или методы объектов домена.

Если в качестве представлений используются страницы сервера, для вызова представления можно применить имя соответствующей страницы. Если же представление является классом, стоит подумать о хранении команды или строки для применения метода отражения. И наконец, представление может быть таблицей стилей XSLT, ссылка на которую в виде строки будет храниться в **контроллере приложения**.

Разрабатывая **контроллер приложения**, необходимо подумать о том, насколько он должен быть отделен от остальной части слоя представления. При этом необходимо учесть, зависит ли от **контроллера приложения** аппарат пользовательского интерфейса. Возможно, контроллер напрямую осуществляет доступ к данным из HTTP-сеанса, передает управление странице сервера или вызывает методы класса толстого клиента.

Что касается меня, то я отдаю предпочтение **контроллерам приложения**, не связанным с аппаратом пользовательского интерфейса. Прежде всего это позволяет тестировать **контроллер приложения** независимо от пользовательского интерфейса, что уже само по себе очень удобно. Кроме того, это дает возможность использовать один и тот же **контроллер приложения** с несколькими интерфейсами. В связи с этим многие разработчики рассматривают **контроллер приложения** как промежуточный слой между представлением и предметной областью.

Приложение может иметь несколько **контроллеров приложения**, управляющих его составными частями. Благодаря этому сложная логика выбора интерфейсных экранов может быть распределена по нескольким классам. В подобных случаях я рекомендую создавать отдельный **контроллер приложения** на каждую область пользовательского интерфейса. Более простому приложению достаточно и одного контроллера.

Если у приложения есть несколько представлений, например Web-интерфейс, толстый клиент и интерфейс для отображения на карманном компьютере, можете применить к каждому из них один и тот же **контроллер приложения**, но не перестарайтесь. Зачастую для получения действительно удобных интерфейсов каждому из них нужен свой порядок отображения экранов. Впрочем, повторное использование одного и того же **контроллера приложения** способно настолько сократить объем работы, что может вполне оправдать не слишком удобный пользовательский интерфейс.

Очень часто механизм пользовательского интерфейса рассматривают как модель состояний, в котором конкретные события могут инициировать различные отклики, в зависимости от состояния определенных объектов приложения. В этом случае для представления управляющей логики модели состояний удобно использовать метаданные. Эти метаданные могут задаваться средствами языка программирования (самый простой способ) или же храниться в отдельном файле настроек.

Иногда в **контроллер приложения** помещают логику домена, относящуюся к обработке конкретного запроса. Вообще говоря, я не приветствую подобные решения. Тем не менее следует признать, что граница между логикой приложения и логикой домена не всегда поддается точному определению. Предположим, я обрабатываю заявления о медицинском страховании и должен отобразить дополнительное окно с набором вопросов только в том случае, если заявитель — курильщик. Что это — логика приложения или логика домена? Если таких случаев не слишком много, я могу вынести эту логику в **контроллер приложения**, однако если подобные ситуации возникают по ходу всего приложения, придется реализовать их обработку в **модели предметной области (Domain Model, 140)**.

Назначение

Если логика Web-приложения, описывающая порядок отображения интерфейсных экранов, довольно проста (другими словами, если пользователь может открывать экраны приложения практически в любом порядке), применять **контроллер приложения** не имеет смысла. Основное преимущество данного типового решения состоит именно в определении порядка отображения страниц и выборе тех или иных представлений в зависимости от состояний объектов.

Необходимость использования **контроллера приложения** очевидна, если различные изменения хода приложения требуют применения схожей логики, касающейся выбора команд или представлений, особенно если такие изменения возникают во многих местах приложения.

Дополнительные источники информации

Большинство идей, лежащих в основе данного типового решения, были взяты из статьи [25]. Хотя подобные идеи выдвигались и раньше, объяснения, предложенные авторами этой статьи, показались мне наиболее четкими и логичными.

Пример: модель состояний контроллера приложения (Java)

Механизм пользовательского интерфейса нередко рассматривают как модель состояний. Подобная формализация особенно верна, когда приложение должно по-разному реагировать на события в зависимости от состояний некоторых объектов. В этом примере я использую простую модель состояний для описания нескольких операций, выполняемых над имуществом (рис. 14.11). Эксперты по лизингу компании ThoughtWork были бы повергнуты в шок таким вопиющим упрощением дел, однако в качестве примера **контроллера приложения**, основанного на использовании состояний, моя модель не так уж и плоха.

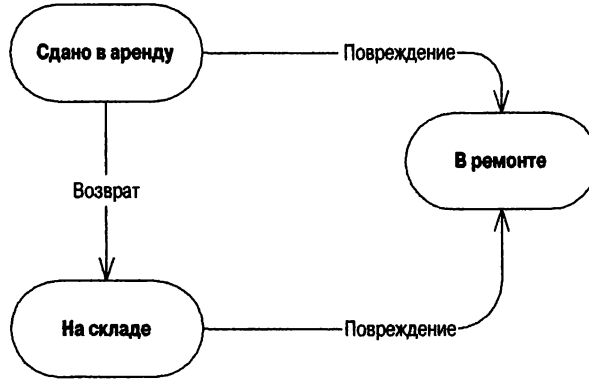


Рис. 14.11. Простая модель состояний имущества

Выполнение переходов в рассматриваемой модели состояний описывается тремя правилами.

- Если при получении команды `return` ("возврат") имущество находится в состоянии `ON_LEASE` ("сдано в аренду"), приложение отображает страницу для ввода данных о возврате имущества.
- Вызов команды `return` в состоянии `IN_INVENTORY` ("на складе") является недопустимым действием. В этом случае приложение отображает страницу с сообщением о попытке выполнения недопустимого действия.
- При получении команды `damage` ("повреждение") приложение отображает разные страницы, в зависимости от того, в каком состоянии находится имущество — `ON_LEASE` или `IN_INVENTORY`.

В качестве входного контроллера будет применяться **контроллер запросов (Front Controller, 362)**, который выполняет обслуживание запросов.

```
class FrontServlet...

   public void service(HttpServletRequest request,
↳HttpServletResponse response)
         throws IOException, ServletException
   {
      ApplicationController appController =
↳getApplicationController(request);
      String commandString = (String) request.getParameter(
↳"command");
      DomainCommand comm = appController.getDomainCommand(
↳commandString, getParameterMap(request));
      comm.run(getParameterMap(request));
      String viewPage = "/" + appController.getView(
↳commandString, getParameterMap(request)) + ".jsp";
      forward(viewPage, request, response);
   }
```

Логика метода `service` крайне проста: определяем контроллер приложения, соответствующий заданному запросу; спрашиваем у контроллера, какую команду домена следует применить; выполняем указанную команду; спрашиваем у контроллера, какое представление следует отобразить; передаем управление указанному представлению.

Данная схема предполагает наличие нескольких **контроллеров приложения**, реализующих общий интерфейс.

```
interface ApplicationController...

   DomainCommand getDomainCommand (String commandString,
↳Map params);
   String getView (String commandString, Map params);
```

В нашем примере **контроллер приложения** будет реализован в виде класса `AssetApplicationController`. Для хранения ссылок на команды домена и представления

контроллер приложения будет использовать класс `Response`. При этом обращение к командам домена будет осуществляться посредством ссылки на класс команды, а обращение к представлениям — с помощью строки, которая будет преобразована контроллером запросов в адрес URL, соответствующий странице JSP.

```
class Response...

   private Class domainCommand;
   private String viewUrl;
   public Response(Class domainCommand, String viewUrl) {
      this.domainCommand = domainCommand;
      this.viewUrl = viewUrl;
   }
   public DomainCommand getDomainCommand() {
      try {
         return (DomainCommand) domainCommand.newInstance();
      } catch (Exception e) {throw new ApplicationException (e);
      }
   }
   public String getViewUrl() {
      return viewUrl;
   }
```

Контроллер приложения управляет выбором откликов, используя коллекцию коллекций, индексированных по имени команды и состоянию имущества (рис. 14.12).

```
class AssetApplicationController...

   private Response getResponse(String commandString,
↳AssetStatus state) {
      return (Response) getResponseMap(commandString).get(
↳state);
   }
   private Map getResponseMap (String key) {
      return (Map) events.get(key);
   }
   private Map events = new HashMap();
```

Когда контроллер приложения получает запрос на выбор необходимой команды, он просматривает запрос, чтобы определить идентификатор предмета имущества, обращается к домену, чтобы выяснить состояние этого предмета имущества, находит соответствующий класс команды домена, создает экземпляр этого класса и возвращает созданный объект.

```
class AssetApplicationController...

   public DomainCommand getDomainCommand (String commandString,
↳Map params) {
      Response reponse = getResponse(commandString,
↳getAssetStatus(params));
      return reponse.getDomainCommand();
   }
```

```
private AssetStatus getAssetStatus(Map params) {
   String id = getParam("assetID", params);
   Asset asset = Asset.find(id);
   return asset.getStatus();
}
private String getParam(String key, Map params) {
   return ((String[]) params.get(key))[0];
}
```

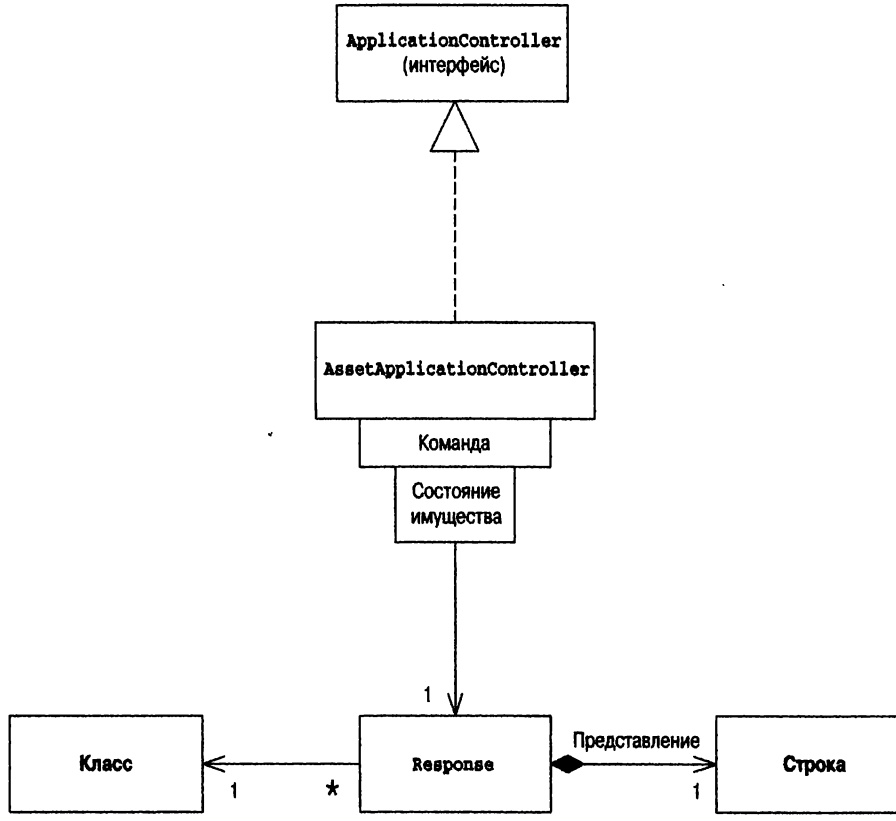


Рис. 14.12. Схема хранения контроллером приложения ссылок на команды домена и представления

Все классы команд домена реализуют простой интерфейс, который позволяет использовать их в модели контроллера запросов.

```
interface DomainCommand...

   abstract public void run(Map params);
```

После выполнения указанной команды контроллер запросов снова обращается к контроллеру приложения, на сей раз с просьбой определить необходимое представление.

```
class AssetApplicationController...

   public String getView (String commandString, Map params) {
      return getResponse(commandString,
↳getAssetStatus(params)).getViewUrl();
      }
```

В нашем случае **контроллер приложения** не возвращает полный адрес URL к представлению JSP. Вместо этого он возвращает строку, которую контроллер запросов преобразует в адрес URL. Я поступил так, чтобы избежать дублирования адресов URL в откликах контроллера. Кроме того, данный прием позволяет добавить еще один уровень опосредования ссылок на представления, если это вдруг понадобится позднее.

Для загрузки **контроллера приложения** можно применить код, приведенный ниже.

```
class AssetApplicationController...

   public void addResponse(String event, Object state,
↳Class domainCommand, String view) {
      Response newResponse = new Response (domainCommand, view);
      if ( ! events.containsKey(event))
         events.put(event, new HashMap());
      getResponseMap(event).put(state, newResponse);
   }
   private static void loadApplicationController(
↳AssetApplicationController appController) {
      appController = AssetApplicationController.getDefault();
      appController.addResponse("return", AssetStatus.ON_LEASE,
↳GatherReturnDetailsCommand.class, "return");
      appController.addResponse("return", AssetStatus.
↳IN_INVENTORY, NullAssetCommand.class, "illegalAction");
      appController.addResponse("damage", AssetStatus.
↳ON_LEASE, InventoryDamageCommand.class, "leaseDamage");
      appController.addResponse("damage", AssetStatus.
↳IN_INVENTORY, LeaseDamageCommand.class, "inventoryDamage");
   }
```

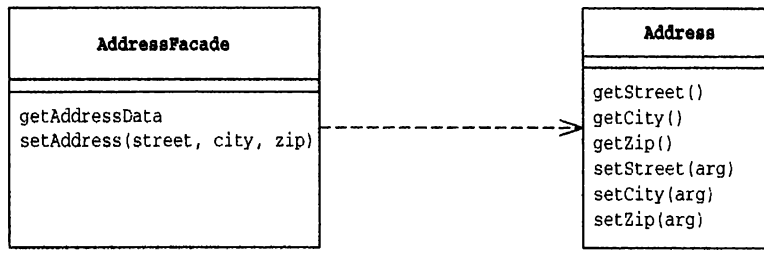
Загрузка **контроллера приложения** из файла настроек тоже не представляет никакой сложности. Попробуйте сделать это самостоятельно.

Глава 15

Типовые решения распределенной обработки данных

Интерфейс удаленного доступа (Remote Facade)

Предоставляет интерфейс с низкой степенью детализации для доступа к объектам, имеющим интерфейс с высокой степенью детализации, в целях повышения эффективности работы в сети



В объектно-ориентированной модели удобно работать с небольшими объектами, имеющими множество небольших методов. Это предоставляет возможность гибкого изменения поведения объектов, а также позволяет называть методы и атрибуты объектов конкретными именами, что значительно упрощает понимание кода приложения. Одним из последствий высокой степени детализации интерфейса является наличие большого количества взаимодействий между объектами и, как результат, большого количества вызовов методов.

Высокая детализация интерфейсов прекрасно подходит для работы в едином адресном пространстве, однако становится крайне проблематичной, когда вызовы осуществляются между процессами. Выполнение удаленных вызовов связано с довольно большими расходами на маршалинг данных, проверку безопасности, передачу пакетов между маршрутизаторами и т.п. Если процессы запущены на машинах, находящихся на противоположных концах земного шара, в дело вступает и скорость передачи данных. Как это ни горько сознавать, выполнение межпроцессного вызова обходится на

несколько порядков дороже, чем выполнение внутрипроцессного вызова, даже если оба процесса запущены на одном компьютере. Подобное влияние на производительность системы не могут не принимать во внимание даже самые стойкие приверженцы загрузки по требованию.

Таким образом, можно сделать вполне логичный вывод: каждый объект, потенциально предназначенный для удаленного доступа, должен иметь интерфейс с низкой степенью детализации, что позволит максимально уменьшить количество вызовов, необходимых для выполнения определенной процедуры. Это касается не только вызовов методов, но и самих объектов. Вместо того чтобы запрашивать пункты заказа отдельно от самого заказа, клиент должен извлекать и обновлять заказ и пункты заказа в одном вызове. К сожалению, подобная схема не может не повлиять на структуру объектов. Последняя становится менее детализированной, а разработчик теряет "тонкий" контроль над поведением объектов, который удавалось получить при помощи множества небольших объектов и методов. Как результат, программирование усложняется, а его производительность падает.

Интерфейс удаленного доступа — это интерфейс с низкой степенью детализации [20], предоставляющий доступ к сети объектов с высокой степенью детализации. Такие объекты не имеют удаленных интерфейсов, а **интерфейс удаленного доступа** не содержит логики домена и лишь преобразует вызовы методов с низкой степенью детализации в последовательности методов с высокой степенью детализации.

Принцип действия

Интерфейс удаленного доступа решает проблему распределенной обработки данных путем применения стандартного подхода объектно-ориентированного программирования, предполагающего разделение разных обязанностей между различными объектами. Поэтому **интерфейс удаленного доступа** можно рассматривать как стандартное решение данной проблемы. Как известно, для реализации сложной логики хорошо подходят объекты с высокой степенью детализации, поэтому вся сложная логика помещается в "детализированные" объекты, предназначенные для взаимодействия в рамках единого процесса. Чтобы обеспечить эффективный доступ к этим объектам со стороны удаленных систем, создается отдельный объект, выполняющий роль удаленного интерфейса. По своей сути **интерфейс удаленного доступа** — это всего лишь оболочка, которая осуществляет переключение между менее детализированным и более детализированным интерфейсами.

В простых случаях, например для объекта, представляющего почтовый адрес, **интерфейс удаленного доступа** заменяет обычные get- и set-методы объекта одним get-методом и одним set-методом, известными как *функции массового доступа (bulk accessors)*. Когда удаленный клиент вызывает такой set-метод, **интерфейс удаленного доступа** считывает параметры, переданные этому методу, после чего последовательно вызывает отдельные, "настоящие" set-методы объекта `Address` (рис. 15.1). Этим, собственно, и ограничиваются обязанности **интерфейса удаленного доступа**. Как видите, вся логика проверки на правильность и вычислений остается в объекте адреса и может быть использована другими объектами с высокой степенью детализации.

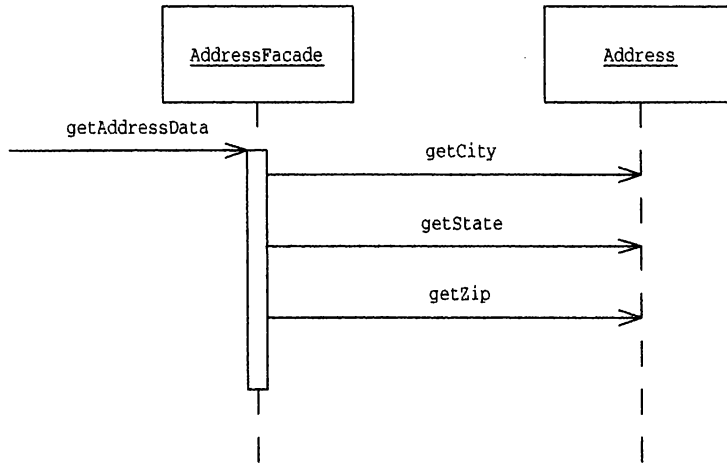


Рис. 15.1. Одно обращение удаленного клиента к интерфейсу инициирует несколько обращений интерфейса к объекту домена

В более сложных случаях один **интерфейс удаленного доступа** может выполнять роль шлюза для нескольких объектов с высокой степенью детализации. Например, **интерфейс удаленного доступа** для объекта заказа может применяться для извлечения и обновления сведений о заказе, обо всех пунктах этого заказа, а также некоторых сведений о покупателе, разместившем заказ.

Чтобы осуществлять массовую передачу данных по сети, последние должны находиться в определенном формате. Если соответствующие классы с высокой степенью детализации присутствуют по обе стороны соединения, а их содержимое поддается сериализации, для передачи по сети можно использовать копию объекта. В этом случае метод `getAddressData` создает копию исходного объекта адреса. Метод `setAddressData` получает копию объекта адреса и использует его для обновления содержимого исходного объекта адреса. (Утверждая это, я исхожу из предположения, что исходный объект адреса должен сохранить свой идентификатор и поэтому не может быть просто заменен новым объектом адреса.)

К сожалению, в большинстве ситуаций подобная тактика неприменима. Классы домена нечасто дублируются в нескольких процессах, а структура отношений между объектами может оказаться слишком сложной для сериализации фрагмента модели домена. Кроме того, клиенту может понадобиться не вся модель, а только упрощенное подмножество ее данных. В подобных случаях передачу данных рекомендуется осуществлять с использованием **объекта переноса данных (Data Transfer Object, 419)**.

Ранее упоминался пример, когда **интерфейс удаленного доступа** соответствовал одному объекту домена. Это достаточно простая и понятная, но далеко не единственная схема. Нередки ситуации, когда **интерфейс удаленного доступа** имеет целый ряд методов, каждый из которых предназначен для получения информации о нескольких объектах. Например, методы `getAddressData` и `setAddressData` могут быть определены в классе `CustomerService` ("данные о покупателе"), который помимо перечисленных включает в себя и такие методы, как `getPurchasingHistory` ("получить список покупок") или `updateCreditData` ("обновить сведения о кредите").

Одним из наиболее спорных моментов, связанных с реализацией **интерфейса удаленного доступа**, является степень детализации систем. Некоторые предпочитают создавать довольно простые **интерфейсы удаленного доступа**, например по одному на каждый вариант использования приложения. Что же касается меня, то я отдаю предпочтение менее детализированным системам с гораздо меньшим количеством **интерфейсов удаленного доступа**. На мой взгляд, приложениям среднего размера вполне достаточно одного **интерфейса удаленного доступа**, а крупным и даже очень крупным приложениям — не более пяти-шести. В этом случае каждый **интерфейс удаленного доступа** будет иметь довольно много методов, однако все они очень просты и вряд ли способны создать проблему для разработчика.

При проектировании **интерфейса удаленного доступа** следует опираться на потребности конкретного клиента. В большинстве случаев они сводятся к необходимости просматривать и обновлять информацию посредством пользовательского интерфейса. В этом случае вы можете создать один **интерфейс удаленного доступа** для группы диалоговых окон, каждому из которых будет соответствовать собственный метод массового доступа, загружающий и сохраняющий данные. Щелчок на элементе управления диалогового окна, например на кнопке изменения состояния заказа, будет запускать соответствующие команды интерфейса. Очень часто разные методы **интерфейса удаленного доступа** выполняют схожие операции над объектами домена. Это вполне нормально: назначение интерфейса — упрощать доступ к объектам для внешних пользователей, а не для внутренних методов. Если клиентский процесс воспринимает определенные действия как разные команды, это и будут разные команды, даже когда в действительности они соответствуют одному и тому же внутреннему методу.

Интерфейс удаленного доступа может быть с состояниями или без состояний. Применение **интерфейса удаленного доступа** без состояний позволит организовать пул соответствующих объектов, что оптимизирует использование ресурсов и повысит эффективность работы приложения, особенно если речь идет о модели "поставщик–потребитель". Тем не менее, если взаимодействие с клиентами потребует сохранения состояния объектов между сеансами, вам понадобится выбрать способ сохранения состояния сеанса. Обычно для этого применяют типовые решения **сохранение состояния сеанса на стороне клиента (Client Session State, 473)**, **сохранение состояния сеанса в базе данных (Database Session State, 479)** или реализацию **сохранения состояния сеанса на стороне сервера (Server Session State, 475)**. Поскольку **интерфейс удаленного доступа (Remote Facade)** может иметь собственные состояния, он позволяет легко реализовать **сохранение состояния сеанса на стороне сервера**, однако при наличии тысяч одновременно работающих пользователей данное решение может привести к значительному падению производительности.

Помимо предоставления интерфейса с низкой степенью детализации, **интерфейс удаленного доступа** может выполнять и другие функции, например обеспечение безопасности. Использование списков управления доступом позволяет задать, каким пользователям разрешено вызывать те или иные методы. Кроме того, данное типовое решение прекрасно подходит для управления транзакциями. Метод **интерфейса удаленного доступа** может начать транзакцию, выполнить всю необходимую внутреннюю работу и затем завершить транзакцию. Вызов метода очень удобно рассматривать как выполнение отдельной транзакции. При возвращении результатов клиенту транзакции должны быть закрыты, поскольку они не рассчитаны на такое длительное использование.

Одна из грубейших ошибок, которые мне доводилось видеть при реализации **интерфейса удаленного доступа**, — это помещение в него логики домена. Запомните: интерфейс удаленного доступа не содержит логики домена. Как и любой другой интерфейс, он должен представлять собой всего лишь простую оболочку с минимальным количеством функций. Если вам понадобится реализовать какую-либо логику домена, связанную с координацией откликов или рабочим процессом, оставьте ее в объектах, стоящих за интерфейсом, или вынесите в отдельный **сценарий транзакции (Transaction Script, 133)**, не предназначенный для удаленного доступа. У вас должна быть возможность локального запуска всего приложения без использования **интерфейсов удаленного доступа** или дублирования кода.

Интерфейс удаленного доступа и типовое решение интерфейс сеанса (Session Facade)

На протяжении последних нескольких лет сообщество J2EE активно популяризирует типовое решение **интерфейс сеанса**, описанное в [3]. В своих ранних черновиках я отождествлял **интерфейс удаленного доступа** с типовым решением **интерфейс сеанса** и использовал именно это название. Между тем в действительности данные типовые решения существенно различаются. **Интерфейс удаленного доступа** — это не более чем оболочка, призванная упростить доступ удаленных объектов (вот откуда моя неприязнь к помещению туда логики домена). Напротив, практически все описания типового решения **интерфейс сеанса** предполагают размещение в нем некоторых фрагментов логики домена, связанной, как правило, с рабочим процессом. Большую роль в этом сыграл распространенный подход, касающийся использования компонентов сеанса J2EE для доступа к компонентам сущностей. Любая координация действий компонентов сущностей должна выполняться другим объектом, поскольку компоненты сущностей не поддерживают возможность повторного вхождения.

Таким образом, я рассматриваю типовое решение **интерфейс сеанса** как помещение в удаленный интерфейс нескольких **сценариев транзакции**. Это весьма и весьма разумный подход, однако он не соответствует определению **интерфейса удаленного доступа**. Вообще говоря, поскольку типовое решение **интерфейс сеанса** содержит логику домена, я очень сомневаюсь, что оно имеет право называться "facade", т.е. "интерфейс"!

Слой служб

Концепция типовых решений **интерфейс удаленного доступа** и **интерфейс сеанса** имеет немало общего со **слоем служб (Service Layer, 156)**. Основное различие между ними заключается в том, что **слой служб** не применяется для удаленного доступа и потому не обязательно должен иметь только методы с низкой степенью детализации. Разумеется, стремление упростить **модель предметной области (Domain Model, 140)** зачастую приводит к появлению менее детализированных методов, однако в данном случае это делается для большей ясности, а не для повышения эффективности работы в сети. Кроме того, **слой служб** не нуждается в использовании **объекта переноса данных (Data Transfer Object, 419)** — в большинстве случаев клиенту возвращаются реальные объекты домена.

Если к **модели предметной области** будут обращаться и локальные и удаленные объекты, вы можете создать **слой служб** и наложить на него отдельный слой **интерфейса удаленного доступа**. Если же фрагмент модели будет использоваться только удаленными объектами, рекомендую упаковать **слой служб** в **интерфейс удаленного доступа** — разумеется,

при условии, что **слой служб** не содержит логики приложения. В противном случае **интерфейс удаленного доступа** лучше реализовать в виде отдельного объекта.

Назначение

Интерфейс удаленного доступа целесообразно применять во всех случаях, касающихся предоставления удаленного доступа к объектам с высокой степенью детализации. Это позволит оптимизировать скорость работы в сети, что присуще объектам с низкой степенью детализации, и в то же время сохранит все преимущества объектов с высокой степенью детализации. Таким образом, вы получаете все самое лучшее, что есть и у тех и у других.

Наиболее часто **интерфейс удаленного доступа** располагают между слоем представления и **моделью предметной области**, которые могут функционировать в двух различных процессах. В качестве примера можно привести взаимодействие интерфейса Swing и модели предметной области на стороне сервера или, скажем, сервлета и объектной модели, расположенной на сервере, если приложение и Web-серверы запущены в разных процессах.

В большинстве случаев описанные ситуации возникают при работе с процессами, запущенными на разных машинах. Между тем расходы на межпроцессный вызов в пределах одного компьютера также весьма велики. Поэтому интерфейс с низкой степенью детализации необходимо использовать для всех межпроцессных взаимодействий, независимо от того, где запущены эти процессы.

Если все операции по осуществлению доступа к объектам выполняются в рамках одного процесса, необходимость в применении **интерфейса удаленного доступа** отпадает. Таким образом, я бы не рекомендовал использовать данное типовое решение для взаимодействия между клиентской **моделью предметной области** и его же слоем представления или, например, между сценарием CGI и **моделью предметной области**, функционирующими на одном Web-сервере. Как правило, **интерфейс удаленного доступа** не применяется и со **сценарием транзакции**, потому что последний и так предполагает низкую степень детализации.

В концепции **интерфейса удаленного доступа** применен *синхронный* подход к распределенной обработке данных (с использованием удаленных вызовов процедур). Зачастую быстроту реагирования приложения можно значительно увеличить за счет *асинхронного* взаимодействия, основанного на сообщениях. Вообще говоря, асинхронный подход имеет массу неопровержимых преимуществ. К сожалению, рассмотрение типовых решений, предназначенных для обеспечения асинхронного взаимодействия, выходит за рамки данной книги.

Пример: использование компонента сеанса Java в качестве интерфейса удаленного доступа (Java)

Если вы работаете с платформой Enterprise Java, в качестве распределенного интерфейса удобно использовать компонент сеанса, поскольку он является удаленным объектом и может быть с состояниями или без состояний. В этом примере я запущу группу объектов POJO (plain old Java objects — простые объекты Java) внутри EJB-контейнера и буду осуществлять к ним удаленный доступ посредством компонента сеанса, спроектированного в виде **интерфейса удаленного доступа**.

Здесь я должен сделать небольшое лирическое отступление. Во-первых, меня страшно удивляет огромное количество людей, которые полагают, будто в EJB-контейнере нельзя запустить простые объекты Java. Меня часто спрашивают, являются ли объекты домена компонентами сущностей. Разумеется, они могут быть компонентами сущностей, но это вовсе *не обязательно*. В нашем примере вполне достаточно и простых объектов Java. Во-вторых, хочу обратить ваше внимание, что существуют и другие способы применения компонентов сеанса, например в качестве оболочки для **сценариев транзакции**.

В этом примере я буду обращаться к удаленным интерфейсам для получения сведений о музыкальных альбомах. Используемая **модель предметной области** состоит из объектов с высокой степенью детализации, представляющих исполнителя, альбом и композиции. Помимо них, у меня есть несколько других пакетов, применяемых приложением в качестве источников данных (рис. 15.2).

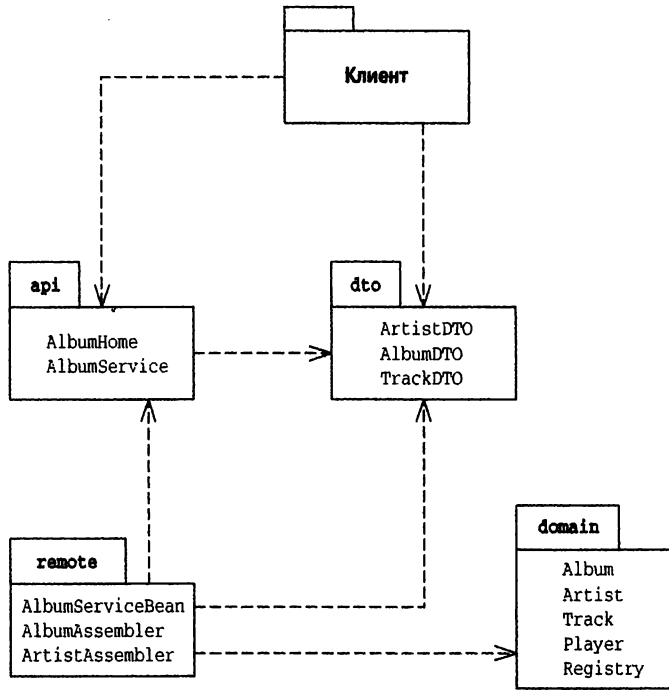


Рис. 15.2. Пакеты, принимающие участие в работе интерфейса удаленного доступа

Пакет `dto`, показанный на рис. 15.2, содержит **объекты переноса данных**, которые применяются для передачи данных по сети удаленному клиенту. Эти объекты имеют простые методы наподобие get- и set-функций, а также умеют выполнять сериализацию собственного содержимого в двоичный формат или текстовый формат XML. В пакете `remote` содержатся объекты-сборщики, которые помещают данные домена в **объекты переноса данных**. Если вас интересует выполнение этого процесса, обратитесь к разделу, посвященному упомянутому типовому решению.

Для упрощения задачи я предположу, что у меня уже реализованы способы помещения данных в **объект переноса данных** и извлечения из него. Таким образом, я могу сконцентрировать все внимание на интерфейсах удаленного доступа. Один логический компонент сеанса Java в действительности состоит из трех классов. Два из них образуют удаленный API (и, по сути, являются интерфейсами Java), а третий — это класс, реализующий данный API. Упомянутые интерфейсы носят имена `AlbumService` и `AlbumHome`. Объект `AlbumHome` используется службой `AlbumService` для получения доступа к распределенному интерфейсу, однако это детали реализации EJB, которые я, пожалуй, опущу. Нас интересует сам **интерфейс удаленного доступа**, а именно объект `AlbumService`. Объявление его интерфейса, предназначенное для использования клиентом, содержится в пакете `api` и представляет собой всего лишь список методов.

```
class AlbumService...

   String play(String id) throws RemoteException;
   String getAlbumXml(String id) throws RemoteException;
   AlbumDTO getAlbum(String id) throws RemoteException;
   void createAlbum(String id, String xml)
↳throws RemoteException;
   void createAlbum(String id, AlbumDTO dto)
↳throws RemoteException;
   void updateAlbum(String id, String xml)
↳throws RemoteException;
   void updateAlbum(String id, AlbumDTO dto)
↳throws RemoteException;
   void addArtistNamed(String id, String name)
↳throws RemoteException;
   void addArtist(String id, String xml)
↳throws RemoteException;
   void addArtist(String id, ArtistDTO dto)
↳throws RemoteException;
   ArtistDTO getArtist(String id) throws RemoteException;
```

Обратите внимание: даже в этом небольшом примере я применяю методы для доступа к содержимому двух различных классов **модели предметной области**, а именно `Artist` и `Album`. Кроме того, в моем интерфейсе можно заметить незначительно отличающиеся разновидности одного и того же метода. Единственные различия этих методов состоят в способе передачи данных удаленной службе: посредством **объекта переноса данных** или же в виде XML-строки. Это позволяет клиенту выбрать нужную форму передачи данных в зависимости от характера самого клиента, а также от параметров соединения. Как видите, даже маленькое приложение может потребовать наличия в **интерфейсе удаленного доступа** большого количества методов.

К счастью, все эти методы очень просты. Приведем реализацию методов, предназначенных для выполнения операций над альбомами.

```
class AlbumServiceBean...

   public AlbumDTO getAlbum(String id) throws RemoteException {
```

```
        return new AlbumAssembler().writeDTO(
↳Registry.findAlbum(id));
   }
   public String getAlbumXml(String id) throws RemoteException {
        AlbumDTO dto = new AlbumAssembler().writeDTO(
↳Registry.findAlbum(id));
        return dto.toXmlString();
   }
   public void createAlbum(String id, AlbumDTO dto)
↳throws RemoteException {
        new AlbumAssembler().createAlbum(id, dto);
   }
   public void createAlbum(String id, String xml)
↳throws RemoteException {
        AlbumDTO dto = AlbumDTO.readXmlString(xml);
        new AlbumAssembler().createAlbum(id, dto);
   }
   public void updateAlbum(String id, AlbumDTO dto)
↳throws RemoteException {
        new AlbumAssembler().updateAlbum(id, dto);
   }
   public void updateAlbum(String id, String xml)
↳throws RemoteException {
        AlbumDTO dto = AlbumDTO.readXmlString(xml);
        new AlbumAssembler().updateAlbum(id, dto);
   }
```

Как видите, единственная функция каждого из этих методов заключается в делегировании выполнения действия другому объекту, поэтому тело метода состоит всего из одной-двух строк кода. Данный фрагмент замечательно демонстрирует, как должен выглядеть распределенный интерфейс: длинный список коротких методов, содержащих очень малое количество логики. В этом случае интерфейс представляет собой всего лишь механизм "упаковки", чем, собственно, он и должен быть.

Прежде чем закончить рассмотрение этого примера, скажу несколько слов о тестировании. Для достижения максимальной эффективности как можно большую часть тестирования рекомендуется проводить в одном процессе. В данном примере я могу написать объекты для непосредственного тестирования реализации компонента сеанса: они могут быть запущены без развертывания компонентов сеанса в EJB-контейнер.

```
class XmlTester...

   private AlbumDTO kob;
   private AlbumDTO newkob;
   private AlbumServiceBean facade = new AlbumServiceBean();
   protected void setUp() throws Exception {
      facade.initializeForTesting();
      kob = facade.getAlbum("kob");
      Writer buffer = new StringWriter();
      kob.toXmlString(buffer);
      newkob = AlbumDTO.readXmlString(new StringReader(
↳buffer.toString()));
   }
```

```
public void testArtist() {
   assertEquals(kob.getArtist(), newkob.getArtist());
}
```

Это один из тестов JUnit, предназначенный для запуска в оперативной памяти. Он показывает, как создать и протестировать экземпляр компонента сеанса за пределами контейнера, чтобы достичь более высокой скорости тестирования.

Пример: Web-служба (C#)

Однажды я разговаривал с Майком Хендриксоном (Mike Hendrickson), редактором издательства Addison-Wesley, работавшим над этой книгой. Майк, которого страшно привлекают все новомодные словечки, спросил меня, нет ли в моей книге чего-нибудь о Web-службах. Вообще говоря, мне претит гнаться за модой — процесс издания книги столь долог, что к тому времени, когда вы сможете прочитать эту книгу, изложенный в ней "модный" материал окажется неактуальным. Тем не менее я все-таки решил коснуться вопросов, связанных с Web-службами, чтобы показать, как ключевые решения могут сохранить свою значимость даже при появлении новых технологий.

По своей сути Web-служба — это не более чем интерфейс удаленного доступа (снабженный для солидности достаточно медленным этапом синтаксического анализа). Реализация Web-служб соответствует основному принципу построения **интерфейса удаленного доступа**: функциональность размещается в объектах с высокой степенью детализации, после чего на эту модель накладывают слой **интерфейса удаленного доступа**.

В этом примере я воспользуюсь той же моделью домена, что и в предыдущем, однако на сей раз сосредоточу свои усилия только на получении сведений о конкретном альбоме. Взаимодействие классов, принимающих участие в работе Web-службы, показано на рис. 15.3. Это уже знакомые нам группы классов: класс `AlbumService`, выполняющий роль **интерфейса удаленного доступа**, два **объекта переноса данных**, три объекта **модели предметной области** и один объект-сборщик, извлекающий данные из **модели предметной области** в **объект переноса данных**.

Рассматриваемая **модель предметной области** до смешного проста. Вообще говоря, в подобном примере было бы проще воспользоваться **шлюзом таблицы данных (Table Data Gateway, 167)**, чтобы напрямую создавать **объекты переноса данных**. Однако это бы испортило пример **интерфейса удаленного доступа**, накладываемого на модель предметной области.

```
class Album...

      public String Title;
      public Artist Artist;
      public IList Tracks {
         get {return ArrayList.ReadOnly(tracksData);}
      }
      public void AddTrack (Track arg) {
         tracksData.Add(arg);
      }
      public void RemoveTrack (Track arg) {
         tracksData.Remove(arg);
      }
      private IList tracksData = new ArrayList();
```

```
class Artist...

      public String Name;

class Track...

      public String Title;
      public IList Performers {
         get {return ArrayList.ReadOnly(performersData);}
      }
      public void AddPerformer (Artist arg) {
         performersData.Add(arg);
      }
      public void RemovePerformer (Artist arg) {
         performersData.Remove(arg);
      }
      private IList performersData = new ArrayList();
```

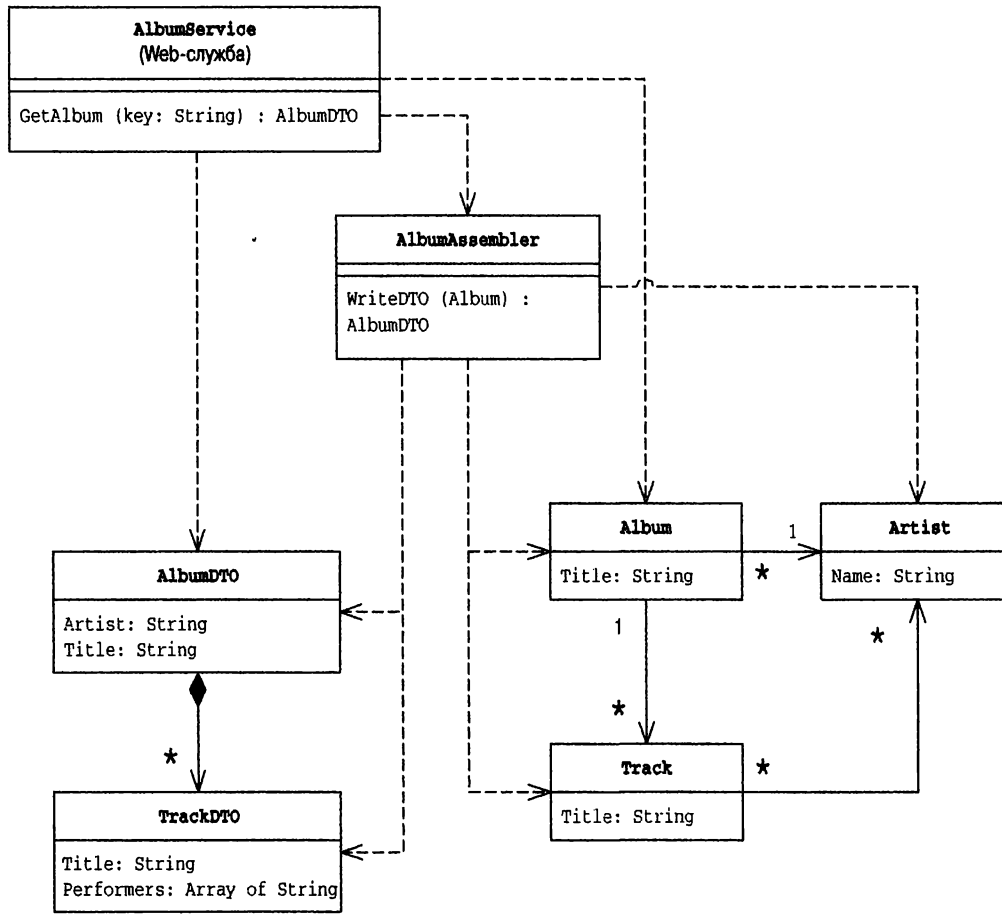


Рис. 15.3. Классы, принимающие участие в работе Web-службы AlbumService

Для передачи данных по сети я воспользовался **объектами переноса данных**. Они представляют собой диспетчеры, которые упрощают структуру передаваемых данных для удобства использования их Web-службой.

```
class AlbumDTO...

      public String Title;
      public String Artist;
      public TrackDTO[] Tracks;

class TrackDTO...

      public String Title;
      public String[] Performers;
```

Поскольку я работаю в .NET, мне не нужно писать отдельного кода для сериализации данных в XML. В состав инфраструктуры .NET входит специальный класс, который самостоятельно выполняет сериализацию и восстановление данных.

Для разработки Web-службы структуру **объекта переноса данных** необходимо описать в коде WSDL. Что приятно, средства Visual Studio позволяют автоматически сгенерировать код WSDL (а я, как известно, очень ленив). Ниже приведено определение схемы XML, соответствующее **объекту переноса данных**.

```
<s:complexType name="AlbumDTO">
   <s:sequence>
      <s:element minOccurs="1" maxOccurs="1" name="Title"
↳nillable="true" type="s:string" />
      <s:element minOccurs="1" maxOccurs="1" name="Artist"
↳nillable="true" type="s:string" />
      <s:element minOccurs="1" maxOccurs="1" name="Tracks"
↳nillable="true" type="s0:ArrayOfTrackDTO" />
   </s:sequence>
</s:complexType>
<s:complexType name="ArrayOfTrackDTO">
   <s:sequence>
      <s:element minOccurs="0" maxOccurs="unbounded"
↳name="TrackDTO" nillable="true" type="s0:TrackDTO" />
   </s:sequence>
</s:complexType>
<s:complexType name="TrackDTO">
   <s:sequence>
      <s:element minOccurs="1" maxOccurs="1" name="Title"
↳nillable="true" type="s:string" />
      <s:element minOccurs="1" maxOccurs="1" name="Performers"
↳nillable="true" type="s0:ArrayOfString" />
   </s:sequence>
</s:complexType>
<s:complexType name="ArrayOfString">
   <s:sequence>
      <s:element minOccurs="0" maxOccurs="unbounded"
```

```
↳name="string" nillable="true" type="s:string" />
   </s:sequence>
</s:complexType>
```

Как и все остальное, что написано на XML, данное определение весьма многословно, однако оно делает именно то, что нам нужно.

Для извлечения данных из **модели предметной области** и помещения их в **объект переноса данных** требуется объект-сборщик.

```
class AlbumAssembler...

      public AlbumDTO WriteDTO (Album subject) {
         AlbumDTO result = new AlbumDTO();
         result.Artist = subject.Artist.Name;
         result.Title = subject.Title;
         ArrayList trackList = new ArrayList();
         foreach (Track t in subject.Tracks)
            trackList.Add (WriteTrack(t));
         result.Tracks = (TrackDTO[]) trackList.ToArray(
↳typeof(TrackDTO));
         return result;
      }
      public TrackDTO WriteTrack (Track subject) {
         TrackDTO result = new TrackDTO();
         result.Title = subject.Title;
         result.Performers = new String[
↳subject.Performers.Count];
         ArrayList performerList = new ArrayList();
         foreach (Artist a in subject.Performers)
            performerList.Add (a.Name);
         result.Performers = (String[]) performerList.ToArray(
↳typeof (String));
         return result;
      }
```

И наконец, необходимо определение самой службы. Вначале это делается в классе C#.

```
class AlbumService...

      [ WebMethod ]
      public AlbumDTO GetAlbum(String key) {
         Album result = new AlbumFinder()[key];
         if (result == null)
            throw new SoapException ("unable to find album with
↳key: " + key, SoapException.ClientFaultCode);
         else return new AlbumAssembler().WriteDTO(result);
      }
```

Разумеется, это не настоящее определение Web-службы — последнее содержится в файле WSDL. Далее приведены наиболее важные фрагменты этого определения.

```
<portType name="AlbumServiceSoap">
   <operation name="GetAlbum">
      <input message="s0:GetAlbumSoapIn" />
      <output message="s0:GetAlbumSoapOut" />
   </operation>
</portType>
<message name="GetAlbumSoapIn">
   <part name="parameters" element="s0:GetAlbum" />
</message>
<message name="GetAlbumSoapOut">
   <part name="parameters" element="s0:GetAlbumResponse" />
</message>
<s:element name="GetAlbum">
   <s:complexType>
      <s:sequence>
         <s:element minOccurs="1" maxOccurs="1" name="key"
↳nillable="true" type="s:string" />
      </s:sequence>
   </s:complexType>
</s:element>
<s:element name="GetAlbumResponse">
   <s:complexType>
      <s:sequence>
         <s:element minOccurs="1" maxOccurs="1"
↳name="GetAlbumResult" nillable="true" type="s0:AlbumDTO" />
      </s:sequence>
   </s:complexType>
</s:element>
```

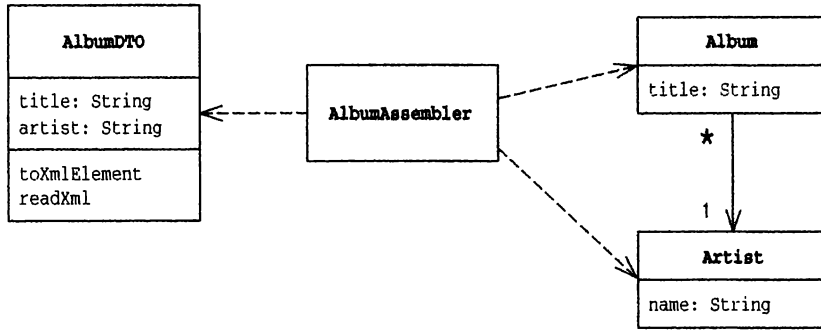
Как видите, описание на WSDL оказалось более многословным, чем любой среднестатистический политик. Впрочем, в отличие от политиков, оно добросовестно выполняет свою работу. Теперь я могу запустить Web-службу, отослав ей SOAP-сообщение.

```
?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/
↳XMLSchema-instance" xmlns:xsd="http://www.w3.org/
↳2001/XMLSchema" xmlns:soap="http://schemas.xmlsoap.org/
↳soap/envelope/">
   <soap:Body>
      <GetAlbum xmlns="http://martinfowler.com">
         <key>aKeyString</key>
      </GetAlbum>
   </soap:Body>
</soap:Envelope>
```

Приводя этот пример, я хотел продемонстрировать не различные "примочки" SOAP и .NET, а фундаментальный подход наложения слоев. Спроектируйте приложение без учета распределенного использования объектов, а затем наложите на него слой распределенного доступа посредством **интерфейсов удаленного доступа** и объектов **переноса данных**.

Объект переноса данных (Data Transfer Object)

Применяется для переноса данных между процессами в целях уменьшения количества вызовов



Каждое обращение к **интерфейсу удаленного доступа (Remote Facade, 405)** связано с большими затратами. Поэтому клиенту необходимо минимизировать число удаленных вызовов, а значит, каждый вызов должен возвращать как можно больше информации. На первый взгляд в этой ситуации удобнее всего использовать методы с множеством параметров. Однако реализовать подобное решение крайне сложно, а иногда и просто невозможно, например в таких языках, как Java, методы которых могут возвращать только одно значение.

В качестве альтернативного решения этой проблемы можно воспользоваться **объектом переноса данных**, который будет содержать в себе все данные, возвращаемые клиенту за один вызов. Разумеется, чтобы такой объект мог быть передан по сети, он должен поддерживать возможность сериализации. Как правило, для перемещения данных между **объектом переноса данных** и объектами домена применяется объект-сборщик, расположенный на стороне сервера.

Многие разработчики, имеющие дело с платформой Sun, используют для данного типового решения термин *объект-значение (value object)*. Я же вкладываю в это понятие совершенно другой смысл (см. описание типового решения **объект-значение (Value Object, 500)**).

Принцип действия

Если бы я был заботливой мамой, то обязательно сказал бы своему ребенку: "Никогда не пиши объекты переноса данных!" В большинстве случаев **объекты переноса данных** представляют собой не более чем раздутый набор полей, а также соответствующих get- и set-методов. Ценность этого омерзительного монстра состоит исключительно в возможности передавать по сети несколько элементов информации за один вызов — прием, который имеет большое значение для распределенных систем.

Когда удаленному объекту нужны какие-либо данные, он обращается к необходимому **объекту переноса данных**. Последний, как правило, содержит намного больше данных, чем было запрошено удаленным объектом. Это сделано не случайно: **объект переноса**

данных должен содержать в себе все данные, которые могут понадобиться удаленному объекту через какое-то время. Поскольку удаленные вызовы связаны с большими расходами, лучше послать слишком много данных, чем выполнять большее количество вызовов.

Зачастую **объект переноса данных** содержит гораздо больше информации, чем обычный серверный объект. Он собирает данные из всех серверных объектов, которые могут понадобиться удаленному объекту. Таким образом, если удаленный объект запросит данные об объекте заказа, **объект переноса данных** возвратит данные о заказе, покупателе, пунктах заказа, условиях доставки — словом все, что имело отношение к запрашиваемому заказу.

Объект переноса данных не может передавать объекты из **модели предметной области (Domain Model, 140)**. Как правило, эти объекты связаны между собой сложной системой отношений, которую весьма трудно, а то и вовсе невозможно сериализовать. Кроме того, объекты домена вообще не следует помещать на сторону клиента — это было бы эквивалентно копированию туда всей **модели предметной области**. Поэтому для переноса данных обычно используют несколько упрощенную форму объектов домена.

Поля **объектов переноса данных** довольно просты. Обычно они содержат значения стандартных типов наподобие строк или дат, а также другие **объекты переноса данных**. Любые связи между такими объектами должны укладываться в рамки простого графа, как правило иерархии, в противоположность сложным структурам, которые можно наблюдать в **модели предметной области**. Помимо поддержки сериализации, атрибуты **объектов переноса данных** должны "распознаваться" и передающей и принимающей стороной. Поэтому классы **объектов переноса данных**, а также классы, на которые они ссылаются, должны присутствовать на обоих концах соединения.

Объект переноса данных рекомендуется проектировать с учетом потребностей конкретного клиента. Вот почему **объекты переноса данных** часто соответствуют Web-страницам или экранам сценариев CGI. Более того, в зависимости от конкретного экрана, одному и тому же объекту могут соответствовать разные **объекты переноса данных**. Разумеется, если различные представления используют схожие данные, для обработки всех их запросов можно применять общий **объект переноса данных**.

Все эти размышления приводят к логичному вопросу: нужно ли использовать общий **объект переноса данных** для обработки всех имеющихся взаимодействий или следует создавать отдельные объекты для каждого запроса? Использование различных **объектов переноса данных** позволяет проще отследить, какие данные передаются в каждом вызове, однако требует создания множества объектов. В свою очередь, использование одного **объекта переноса данных** предполагает меньший объем работы по написанию кода, однако значительно затрудняет отслеживание передачи данных. Я рекомендую использовать один **объект переноса данных**, особенно если передаваемые данные содержат много общего, однако я не возражаю против использования нескольких объектов, если этого требует конкретный запрос. Вообще говоря, однозначного решения этой проблемы не существует, поэтому для обработки большинства взаимодействий вы можете использовать общий **объект переноса данных**, а для обработки двух-трех запросов и ответов — несколько других объектов.

Похожий вопрос связан с тем, нужно ли использовать общий **объект переноса данных** для запросов и ответов или же следует воспользоваться отдельными объектами для каждого из них. Общего правила здесь тоже нет. Если данные, передаваемые в запросе

и в ответе, схожи, используйте один **объект переноса данных**. Если же данные слишком различны, используйте два объекта.

Некоторые разработчики предпочитают делать **объекты переноса данных** неизменяемыми. В этом случае приложение получает от клиента один экземпляр **объекта переноса данных**, а возвращает другой, даже если они принадлежат одному и тому же классу. Другие же допускают изменение **объекта переноса данных**, отправленного клиентом в ходе запроса. Что касается меня, то я еще не выработал окончательного мнения, однако в целом отдаю предпочтение изменяемому **объекту переноса данных**. Это позволяет постепенно наполнять его данными, даже если в качестве ответа на запрос будет создан новый объект. Некоторые аргументы в пользу неизменяемого **объекта переноса данных** связаны с пониманием этого типового решения как объекта-значения, что и привело к совпадению названий с моим **объектом-значением (Value Object, 500)**.

Наиболее распространенной формой реализации **объекта переноса данных** является **множество записей (Record Set, 523)** — набор табличных записей, который возвращается в результате выполнения SQL-запроса. На самом деле **множество записей** можно рассматривать как **объект переноса данных** для базы данных SQL. Подобная схема часто используется при проектировании архитектурных решений. Модель домена может сгенерировать **множество записей** и отослать их клиенту. Клиент же будет воспринимать полученное **множество записей** так, как если бы оно было возвращено непосредственно в результате выполнения SQL-запроса. Это очень удобно, если у клиента есть средства для связывания **множеств записей** с элементами управления. **Множество записей** может быть полностью сгенерировано логикой домена, однако в большинстве случаев оно возвращается как результат выполнения SQL-запроса и обрабатывается логикой домена, после чего передается слою представления. Данный принцип работы был позаимствован для типового решения **модуль таблицы (Table Module, 148)**.

Еще одной формой реализации **объекта переноса данных** может быть универсальная структура данных по типу коллекции. Иногда в этом качестве применяют массивы данных, однако мне такой подход не по душе, поскольку индексы массивов только запутывают код. На мой взгляд, более удачной разновидностью коллекции является словарь, потому что в качестве ключей можно использовать значащие строки. Впрочем, у словаря есть и свои недостатки, связанные с потерей явного интерфейса и строгой типизации. Таким образом, словарь стоит использовать в нерегламентированных случаях, когда под рукой нет генератора объектов, поскольку манипулировать словарем легче, чем писать объект с явным интерфейсом вручную. Тем не менее при наличии генератора рекомендую придерживаться явного интерфейса, особенно если его предполагается использовать в качестве "протокола" общения между различными компонентами.

Сериализация объекта переноса данных

Помимо предоставления простых get- и set-методов, **объект переноса данных** может осуществлять сериализацию собственного содержимого в формат, пригодный для передачи по сети. Выбор формата зависит от того, что находится на противоположном конце соединения, что допускается передавать по самому соединению и насколько просто выполнить сериализацию передаваемой структуры данных. Многие платформы содержат встроенные средства для сериализации простых объектов. Например, Java обладает встроенными средствами сериализации в двоичный формат, а .NET — в двоичный формат и формат XML. Как правило, наличие встроенных средств значительно упрощает

процесс сериализации, поскольку **объекты переноса данных** являются простыми структурами, лишенными сложностей **модели предметной области**. Поэтому я всегда стараюсь использовать автоматизированные средства, если они есть.

Если у вас нет встроенных средств сериализации, их можно создать самому. Мне попадались генераторы кода, которые принимали на вход простые описания и генерировали соответствующие классы, способные содержать в себе данные, предоставлять методы доступа к этим данным, а также проводить сериализацию и восстановление. При написании генераторов очень важно не усложнить их структуру и не перегрузить их ненужными средствами, которые вряд ли когда-нибудь вам понадобятся. Рекомендую написать первые несколько классов вручную и затем ориентироваться на них при построении генератора.

Для выполнения сериализации можно применить и метод отражения. В этом случае придется единожды описать механизмы сериализации и восстановления данных и затем поместить их в суперкласс. Разумеется, применение отражения может негативно сказаться на производительности, однако реальную значимость этой проблемы можно определить только в конкретных условиях.

Выбирая механизм сериализации, помните, что он должен быть применим на обоих концах соединения. Если вы контролируете оба конца, выберите самый простой механизм; в противном случае для противоположного конца соединения можно обеспечить "переходник". Тогда вы сможете использовать простой **объект переноса данных** на обоих концах соединения и затем применить переходник, который адаптирует полученный объект к компоненту клиента.

Одной из наиболее распространенных проблем, связанных с сериализацией **объекта переноса данных**, является выбор формата. Объекты, сериализованные в текстовый формат, вполне читабельны и подходят для эпизодического просмотра пользователем, что позволяет отслеживать, какие данные передаются на другой конец соединения. В последнее время огромную популярность завоевал формат XML, поскольку на рынке программного обеспечения существует множество средств, предназначенных для создания и синтаксического анализа XML-документов. К сожалению, передача текстовых данных требует гораздо большей пропускной способности соединения (что особенно верно по отношению к XML), а также может значительно снизить производительность приложения.

Важным фактором, который следует учитывать при выполнении сериализации, является синхронизация **объектов переноса данных** на обоих концах соединения. Теоретически при изменении определения **объекта переноса данных** на стороне сервера подобное обновление происходит и на стороне клиента, однако на практике так бывает далеко не всегда. Попытка доступа к серверу с использованием устаревшего клиента приводит к проблемам, однако механизм сериализации может еще более усугубить эти проблемы. Если сериализация **объекта переноса данных** выполняется в двоичный формат, взаимодействие между клиентом и сервером будет полностью утеряно, поскольку любые изменения структуры такого объекта делают невозможным восстановление данных. Даже безобидное изменение, например добавление нового поля, полностью изменяет двоичное представление объекта. Таким образом, применение двоичного формата сериализации делает взаимодействие с удаленным клиентом слишком чувствительным к изменениям.

Избежать подобных проблем помогают другие схемы сериализации. Одна из них — это сериализация в формат XML. В большинстве случаев ее можно описать таким образом, чтобы результирующая структура данных стала более устойчивой к изменениям. Другой возможной альтернативой является сериализация в двоичный формат с использованием словаря. Хотя обычно я не рекомендую реализовывать **объект переноса данных** в виде словаря, последний можно рассматривать как более удачный вариант сериализации в двоичный формат, поскольку он позволяет справиться с некоторыми отклонениями в синхронизации.

Сборка объекта переноса данных из объектов домена

Объект переноса данных не "знает" о том, как взаимодействовать с объектами домена. Как известно, **объект переноса данных** должен быть развернут на обоих концах соединения, поэтому крайне нежелательно, чтобы он был зависим от объектов домена. Точно так же объекты домена не должны зависеть от **объекта переноса данных**, потому что последний будет меняться при каждом изменении интерфейса. Итак, в общем случае модель домена должна оставаться независимой от внешних интерфейсов.

Добиться независимости модели домена от **объекта переноса данных** можно путем реализации отдельного объекта-сборщика, который будет создавать **объект переноса данных** на основе данных домена и наоборот — обновлять модель домена данными из **объекта переноса данных** (рис. 15.4). Подобный сборщик можно рассматривать как разновидность **преобразователя (Mapper, 489)**, поскольку он выполняет отображение **объектов переноса данных** на объекты домена.

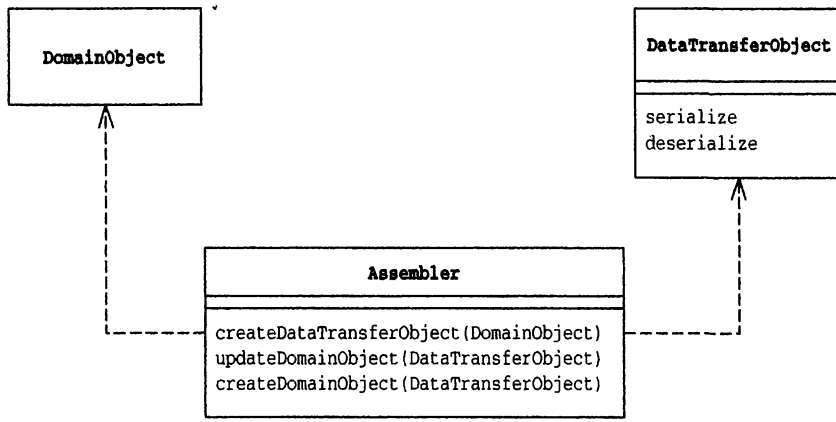


Рис. 15.4. Использование сборщика помогает сохранить независимость модели домена от **объекта переноса данных**

Один и тот же **объект переноса данных** может использоваться несколькими сборщиками. Главной причиной подобного поведения является необходимость применения одних и тех же данных в разных сценариях обновления. Еще одним аргументом в пользу выделения объекта-сборщика является тот факт, что **объект переноса данных** может быть автоматически сгенерирован на основе простых описаний данных. Сгенерировать же сборщик гораздо труднее, а подчас вовсе невозможно.

Назначение

Объект переноса данных применяется во всех случаях перемещения данных между процессами, когда за один вызов необходимо передать большое количество данных.

Существует несколько альтернатив использованию **объекта переноса данных**, хотя мне они не слишком нравятся. Один из возможных вариантов — вообще не создавать вспомогательного объекта, а воспользоваться set-методом со множеством аргументов или же get-методом с механизмом передачи параметров по ссылке. Данный подход имеет существенный недостаток: во многих языках, в частности Java, метод может возвращать только один объект. Таким образом, хотя подобная схема вполне подходит для обновлений, она не может быть использована для извлечения данных без хитроумных манипуляций с обратными вызовами.

Еще одна альтернатива предполагает непосредственную передачу данных в виде строки, без участия специального объекта в качестве интерфейса. Недостатком этого подхода является слишком тесная зависимость механизма передачи от строкового представления данных. Вообще говоря, конкретное представление рекомендуется скрывать за явным интерфейсом. В этом случае, если понадобится изменить строковое представление данных или заменить его двоичной структурой, вам не придется изменять все остальное.

Объект переноса данных особенно удобен тогда, когда взаимодействие между клиентским и серверным компонентами осуществляется с применением XML. Модель XML DOM довольно неудобна в обращении, поэтому ее гораздо проще инкапсулировать в **объекте переноса данных**, особенно с учетом простоты автоматической генерации последнего.

Довольно часто **объект переноса данных** применяется в качестве общего источника данных для компонентов различных слоев приложения. Каждый компонент вносит в **объект переноса данных** некоторые изменения и передает его следующему слою. Хорошим примером использования подобной схемы является применение **множества записей (Record Set, 523)** в COM и .NET. Здесь каждый слой приложения "умеет" обращаться с данными, представленными в виде множества записей, независимо от того, пришло ли это множество прямо из базы данных SQL или же было изменено другими слоями. Платформа .NET еще более расширяет данную схему, предоставляя встроенный механизм сериализации множеств записей в формат XML.

Хотя эта книга посвящена синхронным системам, **объект переноса данных** может применяться и для асинхронного перемещения данных. Подобная необходимость возникает тогда, когда интерфейс доступа к данным желательно использовать и синхронно и асинхронно. При синхронном извлечении данных **объект переноса данных** будет возвращаться обычным способом, а при асинхронном — с помощью **загрузки по требованию (Lazy Load, 220)**. Последнюю необходимо использовать везде, где могут фигурировать результаты выполнения асинхронных вызовов. Пользователь **объекта переноса данных** будет блокировать доступ остальных пользователей только тогда, когда попытается извлечь результаты выполнения вызова.

Дополнительные источники информации

В [3] типовое решение **объект переноса данных** рассматривается под именем **объект-значение (Value Object)**. Как уже отмечалось, последнее не имеет ничего общего с типовым решением **объект-значение (Value Object, 500)**. Это не более чем досадное совпадение

имен; многие используют термин *value object* в том же значении, что и я. Насколько мне известно, **объект переноса данных** называется "объектом-значением" только в J2EE, поэтому я решил последовать более распространенному пониманию этого слова.

Объект-сборщик рассмотрен в [3] под именем **сборщик объектов-значений (Value Object Assembler)**. Я не стал выносить сборщик в отдельное типовое решение, хотя и применил термин "сборщик", а не **преобразователь**.

Книга [28] содержит описание **объекта переноса данных** и нескольких вариантов его реализации. И наконец, в статье [34] рассмотрены гибкие механизмы сериализации, в частности переключение между двоичным и текстовым форматами.

Пример: передача информации об альбомах (Java)

В этом примере я воспользуюсь моделью домена, показанной на рис. 15.5. Мы будем передавать сведения о взаимосвязанных объектах `Album`, `Track` и `Artist`, используя **объекты переноса данных**, изображенные на рис. 15.6.

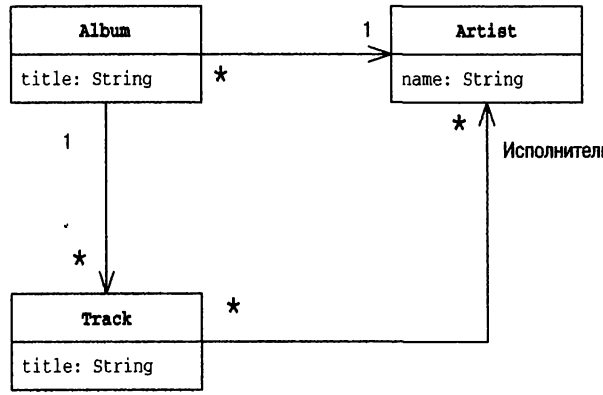


Рис. 15.5. Схема классов исполнителей и альбомов

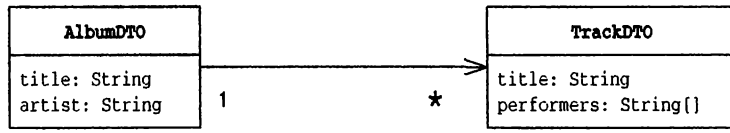


Рис. 15.6. Схема классов для **объектов переноса данных**

Наличие **объектов переноса данных** несколько упрощает структуру передаваемой информации. Необходимые данные из объекта `Artist` помещаются в объект `AlbumDTO`, а перечень исполнителей композиции сохраняется в объекте `TrackDTO` в виде массива строк. Это типичный пример свертывания структуры данных для последующей передачи с помощью **объекта переноса данных**. В нашем примере используются два **объекта переноса данных** — один для альбомов, а другой для композиций. **Объект переноса данных** для исполнителей мне не нужен, потому что все их данные уже присутствуют в двух других **объектах переноса данных**. Что касается **объекта переноса данных** для композиций, то он

понадобился мне, поскольку в альбоме есть несколько композиций и каждая из них может содержать в себе более одного элемента данных.

Ниже показано, как заполнить **объект переноса данных** содержимым модели домена. Соответствующий сборщик вызывается любым объектом, обрабатывающим удаленный интерфейс, например **интерфейс удаленного доступа**.

```
class AlbumAssembler...

   public AlbumDTO writeDTO(Album subject) {
      AlbumDTO result = new AlbumDTO();
      result.setTitle(subject.getTitle());
      result.setArtist(subject.getArtist().getName());
      writeTracks(result, subject);
      return result;
   }
   private void writeTracks(AlbumDTO result, Album subject) {
      List newTracks = new ArrayList();
      Iterator it = subject.getTracks().iterator();
      while (it.hasNext()) {
         TrackDTO newDTO = new TrackDTO();
         Track thisTrack = (Track) it.next();
         newDTO.setTitle(thisTrack.getTitle());
         writePerformers(newDTO, thisTrack);
         newTracks.add(newDTO);
      }
      result.setTracks((TrackDTO[]) newTracks.toArray(
↳new TrackDTO[0]));
   }
   private void writePerformers(TrackDTO dto, Track subject) {
      List result = new ArrayList();
      Iterator it = subject.getPerformers().iterator();
      while (it.hasNext()) {
         Artist each = (Artist) it.next();
         result.add(each.getName());
      }
      dto.setPerformers((String[]) result.toArray(
↳new String[0]));
   }
```

Обновление модели содержимым **объекта переноса данных** немного сложнее. В нашем примере создание нового альбома происходит не так, как обновление существующего. Ниже приведен код создания альбома.

```
class AlbumAssembler...

   public void createAlbum(String id, AlbumDTO source) {
      Artist artist = Registry.findArtistNamed(
↳source.getArtist());
      if (artist == null)
         throw new RuntimeException("No artist named " +
↳source.getArtist());
      Album album = new Album(source.getTitle(), artist);
      createTracks(source.getTracks(), album);
```

```
        Registry.addAlbum(id, album);
    }
    private void createTracks(TrackDTO[] tracks, Album album) {
        for (int i = 0; i < tracks.length; i++) {
            Track newTrack = new Track(tracks[i].getTitle());
            album.addTrack(newTrack);
            createPerformers(newTrack, tracks[i].getPerformers());
        }
    }
    private void createPerformers(Track newTrack, String[]
↳performerArray) {
        for (int i = 0; i < performerArray.length; i++) {
            Artist performer = Registry.findArtistNamed(
↳performerArray[i]);
            if (performer == null)
                throw new RuntimeException("No artist named " +
↳performerArray[i]);
            newTrack.addPerformer(performer);
        }
    }
```

Чтобы реализовать считывание **объекта переноса данных**, необходимо принять несколько решений относительно общей организации создания объектов, в частности объектов исполнителей. Я исхожу из предположения, что при создании объекта альбома объекты исполнителей уже находятся в **реестре (Registry, 495)**, поэтому, если я не могу найти исполнителя, это рассматривается как ошибка. Вместо этого я бы мог создавать объекты исполнителей тогда, когда их имена извлекаются из **объекта переноса данных**.

Как уже упоминалось, процесс обновления альбома несколько отличается от процесса его создания.

```
class AlbumAssembler...

    public void updateAlbum(String id, AlbumDTO source) {
        Album current = Registry.findAlbum(id);
        if (current == null)
            throw new RuntimeException("Album does not exist: " +
↳source.getTitle());
        if (source.getTitle() != current.getTitle())
↳current.setTitle(source.getTitle());
        if (source.getArtist() != current.getArtist().getName()) {
            Artist artist = Registry.findArtistNamed(
↳source.getArtist());
            if (artist == null)
                throw new RuntimeException("No artist named " +
↳source.getArtist());
            current.setArtist(artist);
        }
        updateTracks(source, current);
    }
    private void updateTracks(AlbumDTO source, Album current) {
        for (int i = 0; i < source.getTracks().length; i++) {
            current.getTrack(i).setTitle(
```

```
↳source.getTrackDTO(i).getTitle());
            current.getTrack(i).clearPerformers();
            createPerformers(current.getTrack(i),
               source.getTrackDTO(i).getPerformers());
        }
    }
```

Чтобы произвести обновление, вы можете обновить существующий объект домена или же уничтожить его и заменить новым объектом. При этом необходимо учитывать, ссылаются ли на обновляемый объект другие объекты домена. В этом примере я вынужден обновлять существующий объект альбома, потому что на него и его композиции ссылаются другие объекты. Тем не менее, обновляя название альбома и перечень исполнителей композиции, я могу просто заменить существующие объекты новыми.

Еще одна проблема связана с изменением исполнителя. Следует ли изменить имя существующего исполнителя или же перенаправить ссылку альбома на другой объект исполнителя? Эти вопросы должны решаться отдельно для каждого варианта использования приложения. Здесь я перенаправляю ссылку на новый объект исполнителя.

В нашем примере я использовал обыкновенную сериализацию в двоичный формат, а значит, должен тщательно следить за синхронизацией классов **объектов переноса данных** на обоих концах соединения. Если я изменю структуру **объекта переноса данных** на стороне сервера и не внесу соответствующие изменения на стороне клиента, попытка передачи данных приведет к возникновению ошибки. Чтобы сделать процесс передачи данных менее чувствительным к изменениям структуры, можно сериализовать данные в коллекцию.

```
class TrackDTO...

    public Map writeMap() {
        Map result = new HashMap();
        result.put("title", title);
        result.put("performers", performers);
        return result;
    }
    public static TrackDTO readMap(Map arg) {
        TrackDTO result = new TrackDTO();
        result.title = (String) arg.get("title");
        result.performers = (String[]) arg.get("performers");
        return result;
    }
```

Теперь, если к классу **объекта переноса данных** на стороне сервера будет добавлено новое поле, а на стороне клиента — нет, клиент не сможет загрузить содержимое нового поля, зато успешно загрузит все остальные данные.

Конечно же, заниматься написанием однообразных процедур сериализации и восстановления данных довольно утомительно. Чтобы избежать подобного однообразия, можно воспользоваться методом отражения, например реализовав его в **супертипе слоя (Layer Supertype, 491)**.

```
class DataTransferObject...

    public Map writeMapReflect() {
```

```
        Map result = null;
        try {
           Field[] fields = this.getClass().getDeclaredFields();
           result = new HashMap();
           for (int i = 0; i < fields.length; i++)
              result.put(fields[i].getName(),
↳fields[i].get(this));
        } catch (Exception e) {throw new ApplicationException (e);
        }
        return result;
     }
     public static TrackDTO readMapReflect(Map arg) {
        TrackDTO result = new TrackDTO();
        try {
           Field[] fields = result.getClass().getDeclaredFields();
           for (int i = 0; i < fields.length; i++)
              fields[i].set(result, arg.get(fields[i].getName()));
        } catch (Exception e) {throw new ApplicationException (e);
        }
        return result;
     }
```

Подобные методы хорошо справляются с сериализацией и восстановлением большинства объектов (хотя вам придется добавить еще несколько строк кода для обработки стандартных типов данных).

Пример: сериализация с использованием XML (Java)

В наши дни у платформы Java появляется все больше и больше средств для обработки XML, а интерфейсы API хотя и остаются довольно непостоянными, в целом становятся все лучше и лучше. Вполне возможно, что к тому времени, когда вы прочитаете этот раздел, изложенный материал может оказаться устаревшим или вообще противоречащим современным нормам, однако основная концепция преобразования данных в формат XML измениться не должна.

Вначале я определяю структуру **объекта переноса данных**, а затем принимаю решение о том, как ее сериализовать. В Java для выполнения обыкновенной двоичной сериализации нужно всего-навсего воспользоваться опознавательным интерфейсом (marker interface). Для **объекта переноса данных** выполнение сериализации в двоичный формат полностью автоматизировано, поэтому выбор двоичного формата наиболее очевиден. Тем не менее иногда данные необходимо сериализовать в текстовый формат, например в формат XML.

В этом примере я воспользуюсь интерфейсом JDOM, поскольку его намного удобнее применять для работы с XML, чем стандартные интерфейсы W3C. Ниже приведены методы для создания XML-элементов на основе соответствующих **объектов переноса данных** и для обратного преобразования XML-элементов в **объекты переноса данных**.

```
class AlbumDTO...

   Element toXmlElement() {
      Element root = new Element("album");
      root.setAttribute("title", title);
```

```
      root.setAttribute("artist", artist);
      for (int i = 0; i < tracks.length; i++)
      root.addContent(tracks[i].toXmlElement());
      return root;
   }
   static AlbumDTO readXml(Element source) {
      AlbumDTO result = new AlbumDTO();
      result.setTitle(source.getAttributeValue("title"));
      result.setArtist(source.getAttributeValue("artist"));
      List trackList = new ArrayList();
      Iterator it = source.getChildren("track").iterator();
      while (it.hasNext())
         trackList.add(TrackDTO.readXml((Element) it.next()));
      result.setTracks((TrackDTO[]) trackList.toArray(
↳new TrackDTO[0]));
      return result;
   }

class TrackDTO...

   Element toXmlElement() {
      Element result = new Element("track");
      result.setAttribute("title", title);
      for (int i = 0; i < performers.length; i++) {
         Element performerElement = new Element("performer");
         performerElement.setAttribute("name", performers[i]);
         result.addContent(performerElement);
      }
      return result;
   }
   static TrackDTO readXml(Element arg) {
      TrackDTO result = new TrackDTO();
      result.setTitle(arg.getAttributeValue("title"));
      Iterator it = arg.getChildren("performer").iterator();
      List buffer = new ArrayList();
      while (it.hasNext()) {
         Element eachElement = (Element) it.next();
         buffer.add(eachElement.getAttributeValue("name"));
      }
      result.setPerformers((String[]) buffer.toArray(
↳new String[0]));
      return result;
   }
```

Разумеется, приведенные методы могут создавать только элементы модели XML DOM. Чтобы выполнять сериализацию и восстановление данных, мне нужно реализовать методы для преобразования элементов модели DOM в XML-строку и наоборот. Поскольку сведения о композициях передаются только в контексте альбома, мне понадобится только приведенный ниже код.

```
class AlbumDTO...

   public void toXmlString(Writer output) {
      Element root = toXmlElement();
```

```
        Document doc = new Document(root);
        XMLOutputter writer = new XMLOutputter();
        try {
            writer.output(doc, output);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    public static AlbumDTO readXmlString(Reader input) {
        try {
            SAXBuilder builder = new SAXBuilder();
            Document doc = builder.build(input);
            Element root = doc.getRootElement();
            AlbumDTO result = readXml(root);
            return result;
        } catch (Exception e) {
            e.printStackTrace();
            throw new RuntimeException();
        }
    }
```

Написать этот код было не слишком сложно. Впрочем, я надеюсь, что средства JAXB в скором времени вообще сделают его ненужным.

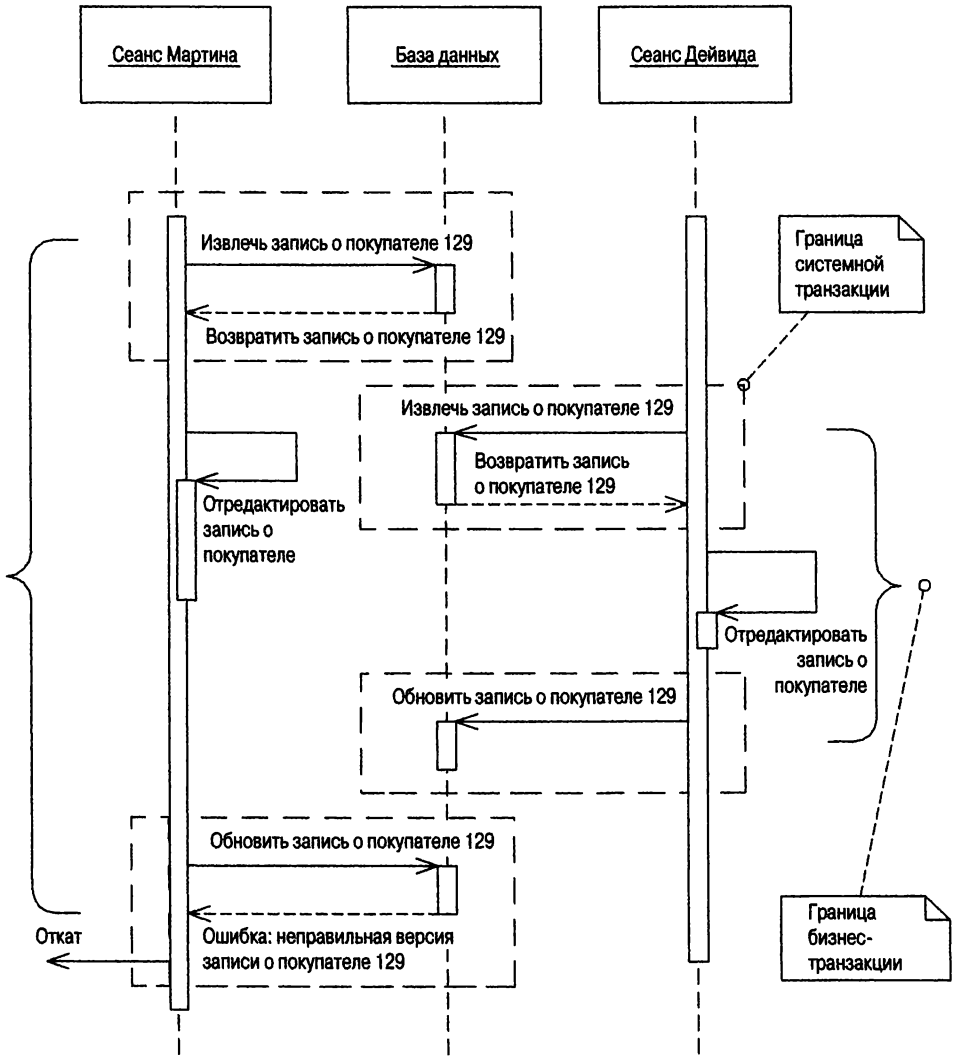
Глава 16

Типовые решения для обработки задач автономного параллелизма

Оптимистическая автономная блокировка (Optimistic Offline Lock)

Дейвид Райс

Предотвращает возникновение конфликтов между параллельными бизнес-транзакциями путем обнаружения конфликта и отката транзакции



Довольно часто выполнение бизнес-транзакции охватывает несколько системных транзакций. В подобных случаях диспетчер СУБД не сможет гарантировать, что записи базы данных останутся в согласованном состоянии. Любая попытка доступа нескольких сеансов к одним и тем же записям грозит нарушением целостности данных и значительно повышает риск утраты выполненных изменений. Кроме того, когда один сеанс обновляет данные, которые считывает другой сеанс, последний может столкнуться с несогласованностью чтения.

Оптимистическая автономная блокировка решает эту проблему путем проверки, не вступят ли изменения, которые должны быть зафиксированы одним сеансом, в конфликт с изменениями, выполненными другим сеансом. Успешная проверка означает наложение блокировки на изменяемые записи и разрешает зафиксировать результаты транзакции. Поскольку проверка и фиксация изменений совершаются в рамках одной системной транзакции, выполнение бизнес-транзакции не вызовет несогласованности данных.

В отличие от **пессимистической автономной блокировки (Pessimistic Offline Lock, 445)**, которая предполагает, что вероятность возникновения конфликта высока, и поэтому ограничивает возможность параллельной работы в системе, при **оптимистической автономной блокировке** вероятность возникновения конфликта крайне мала. Благодаря такому подходу с одними и теми же данными одновременно могут работать несколько пользователей.

Принцип действия

При оптимистическом блокировании сеансу разрешается зафиксировать изменение записи в базе данных, если со времени, прошедшего после загрузки этой записи текущим сеансом, она не была изменена никаким другим сеансом. **Оптимистическая автономная блокировка** может быть применена в любое время, однако срок ее действия ограничивается системной транзакцией, в процессе которой она была установлена. Таким образом, чтобы выполнение бизнес-транзакции не привело к утрате изменений или несогласованности данных, следует применять **оптимистическую автономную блокировку** к каждой записи, изменяемой во время системной транзакции.

Наиболее распространенный прием отслеживания изменений — сохранение вместе с каждой записью номера ее версии. После загрузки такой записи номер ее версии будет сохраняться вместе с остальными элементами состояния сеанса. Наложение **оптимистической автономной блокировки** состоит в сравнении номера версии, хранящегося в состоянии сеанса, с текущим номером версии этой же записи в базе данных. Если проверка пройдет успешно, текущий пользователь сможет зафиксировать в базе данных все изменения, включая и увеличенный номер версии. Последнее позволит избежать несогласованности данных, поскольку любой другой сеанс с более старой версией записи уже не сможет получить блокировку и зафиксировать свои изменения.

В реляционных СУБД реализация описанного приема требует проверки номера версии в любом SQL-операторе, применяемом для обновления или удаления записи. Один SQL-оператор может установить блокировку и обновить содержимое записи. Последний шаг бизнес-транзакции — проверка количества строк, измененных в результате выполнения SQL-оператора. Если количество измененных строк равно 1, это означает, что обновление прошло успешно. Если же количество измененных строк равно 0, следовательно, со времени загрузки строки текущим сеансом она была изменена или удалена кем-то

другим. В этом случае бизнес-транзакция должна осуществить откат системной транзакции, чтобы аннулировать все изменения, выполненные в ходе последней, а затем аварийно завершить выполнение или же попытаться разрешить конфликт и произвести обновление еще раз (рис. 16.1).

Для разрешения конфликтов, возникающих при параллельной обработке данных, удобно сохранять не только номер версии, но и сведения о том, кто провел последнее обновление записи. При выдаче уведомления об ошибке, связанной с наличием параллельных сеансов, хорошее приложение всегда сообщит пользователю, кем и когда было выполнено последнее обновление. Не стоит, однако, заменять номер версии временной меткой — системные часы слишком ненадежны, особенно если речь идет о координации работы нескольких пользователей.

Вместо сравнения номера версии в выражении `WHERE` оператора `UPDATE` можно выполнить сравнение по всем полям текущей строки, что позволяет обойтись без специального поля версии. Это особенно удобно тогда, когда изменить таблицы базы данных для добавления в них нового поля невозможно. К сожалению, данный прием требует манипулирования потенциально большими выражениями `WHERE`, что может негативно сказаться на производительности всего приложения (в зависимости от того, насколько умело СУБД использует индекс по первичному ключу).

Зачастую реализация **оптимистической автономной блокировки** ограничивается проверкой номера версии в каждом операторе `UPDATE` или `DELETE`. К сожалению, это не решает проблемы несогласованности чтения. Представьте себе расчетную систему, которая определяет стоимость заказа и вычисляет сумму налога с продажи. Сеанс подсчитывает суммарную стоимость заказа и затем проверяет место проживания покупателя, чтобы подсчитать величину налога на сумму заказа[1]. Что произойдет, если в это же время другой сеанс, относящийся к службе работы с клиентами, изменит адрес покупателя? Так как величина налога зависит от места проживания, значение, подсчитанное первым сеансом, может оказаться неправильным, однако, поскольку первый сеанс не пытался вносить изменения в адрес покупателя, а только его считывал, конфликт не будет обнаружен.

Несмотря на изложенное, **оптимистическая автономная блокировка** вполне применима и для обнаружения несогласованного чтения. В рассмотренном примере сеанс подсчета суммы заказа должен определить, что правильность его действий зависит от значения адреса соответствующего покупателя. Таким образом, проверка номера версии должна применяться и к записи, содержащей адрес покупателя, возможно, путем искусственного добавления адреса в набор изменений или поддержки специального списка элементов, для которых нужно проверить номер версии. Последнее немного сложнее реализовать, однако назначение полученного при этом кода будет более понятным. Если вместо искусственного обновления вы собираетесь просто перепроверить номер версии, обратите особое внимание на используемый в системе уровень изоляции транзакций. Повторное считывание номера версии можно проводить только при уровне изоляции "повторяемое чтение" или выше. В противном случае номер версии придется искусственно увеличить.

---

[1] В США величина налога с продажи зависит от штата, а иногда и от города, в котором живет покупатель. — *Прим. пер.*

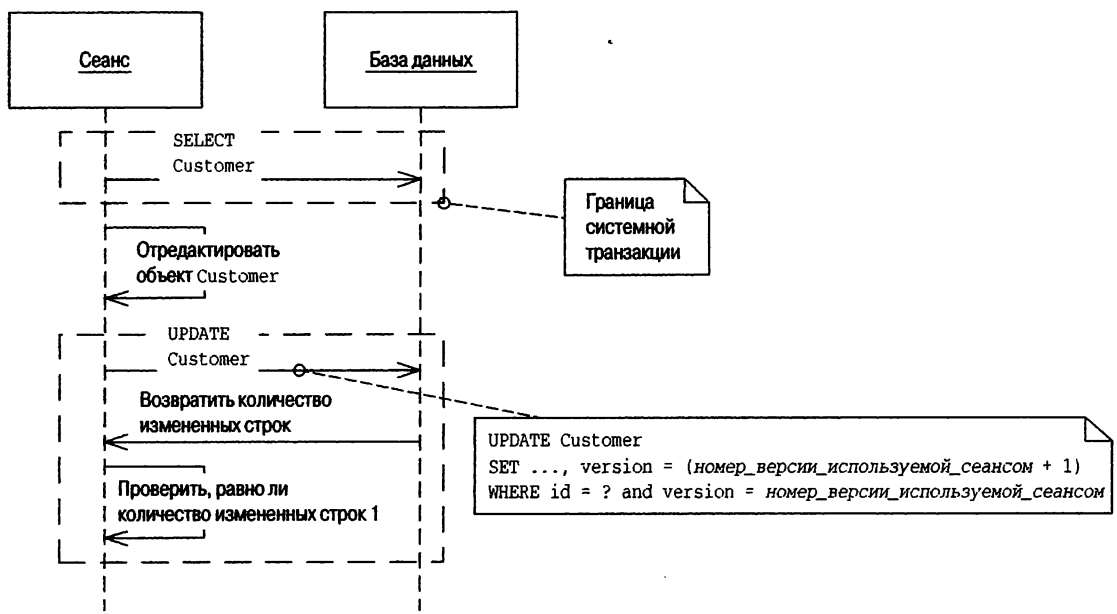


Рис. 16.1. Проверка номера версии при оптимистическом блокировании с помощью оператора UPDATE

Для некоторых проблем несогласованного чтения проверка номера версии может оказаться чересчур строгой. Зачастую выполнение транзакции зависит от самого факта присутствия в базе данных определенной записи или, скажем, от значения только одного из ее полей. В этом случае для повышения степени параллелизма системы вместо проверки номера версии рекомендуется применять более мягкие критерии, в результате чего меньшее количество параллельных обновлений будут заканчиваться откатами бизнес-транзакций. Чем лучше вы понимаете проблемы параллельной обработки данных конкретной системы, тем более эффективно сможете управлять ими в коде приложения.

Для решения некоторых проблем несогласованного чтения удобно воспользоваться **блокировкой с низкой степенью детализации (Coarse-Grained Lock, 457)**, которая рассматривает группу объектов как единый блокируемый элемент. Еще одной возможной альтернативой является простое выполнение всех шагов "проблемной" бизнес-транзакции в рамках одной длинной транзакции. Простота реализации последнего решения может перевесить чрезмерное расходование ресурсов, связанное с наличием нескольких длинных транзакций в разных местах приложения.

Обнаружить факты несогласованного чтения становится сложнее, если успешность транзакции зависит не от считывания конкретных строк, а от результатов выполнения динамического запроса. В этом случае для применения **оптимистической автономной блокировки** можно сохранить первоначальные результаты выполнения запроса и затем сравнить их с результатами выполнения этого же запроса, полученными непосредственно перед фиксацией изменений.

Как и другие схемы блокирования, **оптимистическая автономная блокировка** сама по себе не в состоянии предоставить адекватных решений некоторых наиболее "скользких" моментов параллельного выполнения заданий в корпоративных приложениях. Запомните главное: при управлении параллельными заданиями в бизнес-приложениях следует учитывать не только технические вопросы, но и специфику предметной области. Является ли в действительности описанный конфликт с адресом покупателя таким важным, как это представлялось? Возможно, я был прав, когда подсчитал величину налогов с учетом старого адреса покупателя, но какую из сумм я должен использовать в данный момент? Этот вопрос относится не к сфере программирования, а к бизнес-правилам предприятия. Теперь представьте себе обновление коллекции: как следует поступить, если два сеанса одновременно добавят в коллекцию новый элемент? Обычная схема **оптимистической автономной блокировки** сочтет выполнение подобной операции вполне допустимым, даже если это будет противоречить бизнес-правилам приложения.

Пожалуй, наиболее удачным примером использования **оптимистической автономной блокировки** является хорошо знакомая многим из вас система управления исходным кодом (source code management — SCM). Когда такая система обнаруживает конфликт между изменениями, вносимыми различными программистами, она пытается определить, как осуществить слияние этих изменений, и выполняет повторную попытку зафиксировать результаты. Хорошая стратегия слияния превращает **оптимистическую автономную блокировку** в действительно мощную схему блокирования, причем благодаря не только высокой степени параллелизма подобных приложений, но и тому, что пользователям крайне редко приходится переделывать свою работу. Разумеется, система управления исходным кодом существенно отличается от корпоративных приложений — последним обычно требуется не одна, а сотни стратегий слияния. Некоторые из этих стратегий могут быть так сложны, что просто не стоят усилий, потраченных на их кодирование.

Другие же могут оказаться настолько ценными для ведения бизнеса, что требуют реализации любыми возможными и невозможными способами. Хотя на практике слияние бизнес-объектов выполняется крайне редко, оно все же возможно. Более того, как мне кажется, стратегии слияния бизнес-данных требуют разработки собственных типовых решений. Я больше не буду углубляться в эту тему, однако еще раз обращаю ваше внимание на то, что правильная стратегия слияния на несколько порядков усиливает мощность **оптимистической автономной блокировки**.

В оптимистической схеме блокирования успешность или неуспешность завершения бизнес-транзакции становится известной только во время последней системной транзакции. Между тем иногда о возникновении конфликта хотелось бы узнавать пораньше. Для этого можно реализовать метод `checkCurrent`, который будет определять, были ли обновляемые данные изменены кем-то другим. Конечно, этот метод не гарантирует отсутствия конфликтов, но сможет вовремя предупредить вас о том, что начатый процесс неизбежно закончится неудачей и выполнение изменений следует прекратить. Используйте метод `checkCurrent` всегда, когда о наличии конфликта и необходимости прекращения изменений необходимо узнавать как можно раньше, однако помните, что он не гарантирует отсутствия конфликта во время фиксации результатов.

Назначение

Оптимистическое блокирование применяется тогда, когда вероятность возникновения конфликта между двумя параллельными бизнес-транзакциями мала. В противном случае данная схема окажется весьма недружелюбной по отношению к пользователю, поскольку о том, что фиксация изменений невозможна, ему сообщат только тогда, когда он закончит выполнять все свои действия. После нескольких подобных неудач пользователь устанет, разочаруется и просто прекратит работать с системой. Таким образом, если вероятность конфликта довольно высока или откат проделанных изменений неприемлем, имеет смысл прибегнуть к **пессимистической автономной блокировке**.

Оптимистическая автономная блокировка намного легче в реализации и не подвержена такому количеству недостатков и ошибок времени выполнения, как **пессимистическая автономная блокировка**. Поэтому оптимистическое блокирование должно рассматриваться как стандартный подход к управлению параллельными бизнес-транзакциями в любой корпоративной системе. В действительности пессимистическое блокирование хорошо сочетается с оптимистическим, поэтому их можно использовать и совместно. Вопрос должен заключаться не в том, следует ли *вообще* применять оптимистическое блокирование, а в том, когда одного лишь оптимистического блокирования окажется недостаточно. Правильный подход к управлению параллельными заданиями максимизирует степень параллелизма системы и в то же время минимизирует число конфликтов.

Пример: слой домена с преобразователями данных (Java)

Вообще говоря, чтобы продемонстрировать применение **оптимистической автономной блокировки**, мне бы вполне хватило одной таблицы базы данных со столбцом, содержащим номер версии, и операторов `UPDATE` и `DELETE`, использующих номер версии как часть критерия для выполнения обновлений. Тем не менее я думаю, что вы разрабатываете более сложные приложения, поэтому приведу пример реализации **оптимистической автономной блокировки** с использованием **модели предметной области (Domain Model, 140)**

и **преобразователей данных (Data Mapper, 187)**. Эта схема позволит затронуть больше типичных моментов, возникающих при внедрении в жизнь **оптимистической автономной блокировки**.

Вначале следует убедиться, что **супертип слоя (Layer Supertype, 491)** предметной области может содержать в себе все данные, необходимые для реализации **оптимистической автономной блокировки**, а именно: сведения о времени изменения, лице, выполнившем изменение, и номере версии.

```
class DomainObject...

   private Timestamp modified;
   private String modifiedBy;
   private int version;
```

Все данные хранятся в реляционной базе данных, поэтому каждая таблица также должна содержать в себе эти сведения. Ниже приведена схема таблицы `customer`, а также реализация стандартного набора SQL-операций CRUD (Create Read Update Delete — создание, чтение, обновление, удаление), необходимых для поддержки **оптимистической автономной блокировки**.

```
table customer...

   create table customer(id bigint primary key, name varchar,
↳createdby varchar, created datetime, modifiedby varchar,
↳modified datetime, version int)

SQL customer CRUD...

   INSERT INTO customer VALUES (?, ?, ?, ?, ?, ?, ?)
   SELECT * FROM customer WHERE id = ?
   UPDATE customer SET name = ?, modifiedBy = ?, modified = ?,
↳version = ? WHERE id = ? and version = ?
   DELETE FROM customer WHERE id = ? and version = ?
```

При наличии хотя бы нескольких таблиц и объектов домена, несомненно, понадобится создать **супертип слоя преобразователей данных** и вынести в него все рутинные, повторяющиеся фрагменты объектно-реляционного отображения. Это не только сократит объем работы при написании **преобразователей данных**, но и позволит применить **неявную блокировку (Implicit Lock, 468)** для нейтрализации наиболее рассеянных разработчиков, которые могут забыть описать несколько случаев блокировки и тем самым разрушить весь механизм блокирования.

Первое, что необходимо вынести в абстрактный класс преобразователя, — это построение SQL-выражений. Для реализации подобной схемы преобразователям понадобится предоставить определенные метаданные об имеющихся таблицах. Вместо того чтобы конструировать SQL-выражения во время выполнения, их можно создавать путем автоматической генерации кода. Впрочем, я оставлю построение SQL-выражений в качестве домашнего задания для читателей. Как видно из приведенного ниже кода, я сделал несколько предположений относительно имен и расположения столбцов, содержащих данные о последних изменениях строк. Такой прием не совсем подходит для

уже существующих таблиц баз данных. В этом случае каждый конкретный преобразователь должен предоставить абстрактному преобразователю определенную порцию метаданных столбцов.

Когда у абстрактного преобразователя появятся SQL-выражения, он сможет управлять выполнением операций CRUD. Покажем, как выглядит метод поиска.

```
class AbstractMapper...

   public AbstractMapper(String table, String[] columns) {
      this.table = table;
      this.columns = columns;
      buildStatements();
   }
   public DomainObject find(Long id) {
      DomainObject obj =
         AppSessionManager.getSession().getIdentityMap().get(
↳id);
      if (obj == null) {
         Connection conn = null;
         PreparedStatement stmt = null;
         ResultSet rs = null;
         try {
            conn = ConnectionManager.INSTANCE.getConnection();
            stmt = conn.prepareStatement(loadSQL);
            stmt.setLong(1, id.longValue());
            rs = stmt.executeQuery();
            if (rs.next()) {
               obj = load(id, rs);
               String modifiedBy = rs.getString(
↳columns.length + 2);
               Timestamp modified = rs.getTimestamp(
↳columns.length + 3);
               int version = rs.getInt(columns.length + 4);
               obj.setSystemFields(modified, modifiedBy,
↳version);
               AppSessionManager.getSession().getIdentityMap().
↳put(obj);
            } else {
               throw new SystemException(table + " " + id +
↳" does not exist");
            }
         } catch (SQLException sqlEx) {
            throw new SystemException("unexpected error
↳finding " + table + " " + id);
         } finally {
            cleanupDBResources(rs, conn, stmt);
         }
      }
      return obj;
   }
   protected abstract DomainObject load(Long id, ResultSet rs)
↳throws SQLException;
```

Приведу несколько пояснений. Вначале преобразователь проверяет **коллекцию объектов (Identity Map, 216)**, чтобы убедиться, что искомый объект еще не был загружен. Невыполнение этой операции может привести к тому, что в разные моменты бизнес-транзакции пользователь загрузит несколько версий одного и того же объекта. В этом случае приложение поведет себя совершенно непредсказуемым образом, а выполнение всех проверок номеров версий будет окончательно спутано. После получения результирующего множества данных преобразователь передает управление абстрактному методу загрузки, реализованному в каждом конкретном преобразователе, для извлечения значений соответствующих полей и возвращения активизированного объекта. По завершении этой операции преобразователь вызывает метод `setSystemFields()`, чтобы установить значения номера версии и параметров последних изменений в полях абстрактного объекта домена. Конечно, для заполнения объекта можно было использовать и конструктор, однако это потребовало бы перенести часть логики по сохранению номеров версий в конкретные преобразователи и объекты домена, тем самым ослабив **неявную блокировку**.

Приведем реализацию метода `load()` в конкретном классе преобразователя.

```
class CustomerMapper extends AbstractMapper...

   protected DomainObject load(Long id, ResultSet rs)
↳throws SQLException {
      String name = rs.getString(2);
      return Customer.activate(id, name, addresses);
   }
```

Операции обновления и удаления имеют схожую структуру. В каждом из этих случаев для успешного завершения транзакции преобразователю необходимо убедиться, что в результате применения операции к базе данных возвращается количество измененных строк, равное 1. Если строка не была обновлена, пользователь не сможет получить право на применение оптимистической блокировки и преобразователь будет вынужден сгенерировать исключение `ConcurrencyException`. Как выглядит операция удаления, показано ниже.

```
class class AbstractMapper...

   public void delete(DomainObject object) {
      AppSessionManager.getSession().getIdentityMap().remove(
↳object.getId());
      Connection conn = null;
      PreparedStatement stmt = null;
      try {
         conn = ConnectionManager.INSTANCE.getConnection();
         stmt = conn.prepareStatement(deleteSQL);
         stmt.setLong(1, object.getId().longValue());
         int rowCount = stmt.executeUpdate();
         if (rowCount == 0) {
            throwConcurrencyException(object);
         }
      } catch (SQLException e) {
         throw new SystemException("unexpected error deleting");
      } finally {
```

```
            cleanupDBResources(conn, stmt);
        }
    }
    protected void throwConcurrencyException(DomainObject object)
↳throws SQLException {
        Connection conn = null;
        PreparedStatement stmt = null;
        ResultSet rs = null;
        try {
            conn = ConnectionManager.INSTANCE.getConnection();
            stmt = conn.prepareStatement(checkVersionSQL);
            stmt.setInt(1, (int) object.getId().longValue());
            rs = stmt.executeQuery();
            if (rs.next()) {
                int version = rs.getInt(1);
                String modifiedBy = rs.getString(2);
                Timestamp modified = rs.getTimestamp(3);
                if (version > object.getVersion()) {
                    String when = DateFormat.getDateTimeInstance().
↳format(modified);
                    throw new ConcurrencyException(table + " " +
↳object.getId() + " modified by " + modifiedBy + " at " + when);
                } else {
                    throw new SystemException("unexpected error
↳checking timestamp");
                }
            } else {
                throw new ConcurrencyException(table + " " +
↳object.getId() + " has been deleted");
            }
        } finally {
            cleanupDBResources(rs, conn, stmt);
        }
    }
```

SQL-оператор, применяемый для проверки номера версии в методе `throwConcurrencyException`, должен быть известен и абстрактному преобразователю. Последний будет конструировать его вместе с выражениями CRUD. Данное выражение выглядит примерно так, как показано ниже.

```
checkVersionSQL...

    SELECT version, modifiedBy, modified FROM customer
↳WHERE id = ?
```

Приведенный код не вполне отражает тот факт, что выполнение бизнес-транзакции охватывает несколько системных транзакций. Важно помнить, что для поддержания согласованности данных применение **оптимистической автономной блокировки** должно осуществляться в рамках той же системной транзакции, что и сама фиксация изменений. В нашем примере проверка номера версии вставлена в операторы `UPDATE` или `DELETE`, поэтому данное условие выполняется автоматически.

Взгляните на пример совместно используемого объекта версии в разделе, посвященном **блокировке с низкой степенью детализации (Coarse-Grained Lock, 457)**. Подобная блокировка достаточно хорошо справляется с некоторыми проблемами несогласованного чтения. Впрочем, обнаружить проблемы несогласованного чтения можно и с помощью простого объекта версии, поскольку он очень удобен для реализации логики оптимистических проверок, например методов `increment()` или `checkVersionIsLatest()`. Ниже приведен пример **единицы работы (Unit of Work, 205)**, содержащей проверку на согласованность чтения. Нам пришлось применить довольно строгие меры в виде увеличения номера версии, поскольку не известен уровень изоляции транзакций.

```
class UnitOfWork...

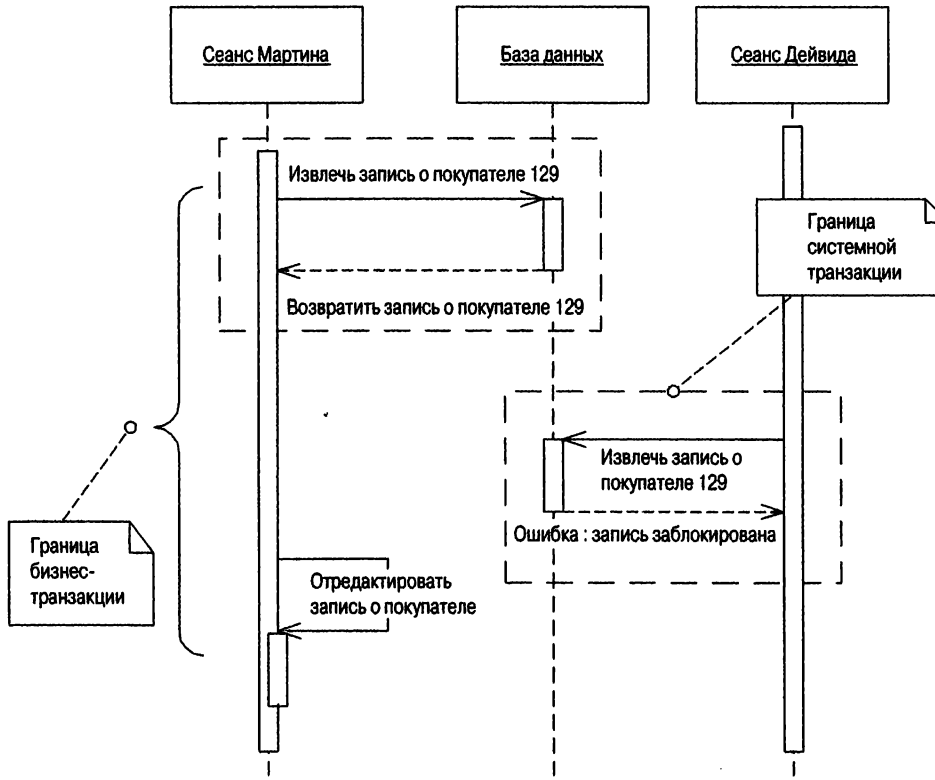
   private List reads = new ArrayList();
   public void registerRead(DomainObject object) {
      reads.add(object);
   }
   public void commit() {
      try {
         checkConsistentReads();
         insertNew();
         deleteRemoved();
         updateDirty();
      } catch (ConcurrencyException e) {
         rollbackSystemTransaction();
         throw e;
      }
   }
   public void checkConsistentReads() {
      for (Iterator iterator = reads.iterator();
↳iterator.hasNext();) {
         DomainObject dependent = (DomainObject)
↳iterator.next();
         dependent.getVersion().increment();
      }
   }
```

Обратите внимание, что **единица работы** выполняет откат системной транзакции, когда обнаруживает нарушение параллелизма (`ConcurrencyException`). Скорее всего, подобное действие понадобится предусмотреть и для всех остальных исключений, которые могут возникнуть в процессе фиксации изменений. Не забывайте об этом шаге! Вместо использования специальных объектов версий проверку номеров версий можно добавить в интерфейс преобразователя.

Пессимистическая автономная блокировка (Pessimistic Offline Lock)

Дейвид Райс

Предотвращает возникновение конфликтов между параллельными бизнес-транзакциями, предоставляя доступ к данным в конкретный момент времени только одной бизнес-транзакции



Управление параллельными заданиями в автономном режиме предполагает, что одна бизнес-транзакция может совершать несколько обращений к базе данных. Наиболее простое и очевидное решение — оставить системную транзакцию открытой на все время выполнения бизнес-транзакции. К сожалению, зачастую этот подход неприменим, так как системы транзакций не приспособлены к работе с длинными транзакциями. Поэтому выполнение бизнес-транзакции приходится разбивать на несколько системных транзакций, что требует реализации собственных средств для управления параллельным доступом к данным.

Разумеется, можно попробовать применить **оптимистическую автономную блокировку (Optimistic Offline Lock, 434)**. Однако данное типовое решение имеет ряд недостатков.

Если к одним и тем же данным одновременно осуществляют доступ несколько пользователей, один из них сможет легко зафиксировать выполнение транзакции, а вот другим в сохранении результатов будет отказано. Поскольку наличие конфликта обнаруживается только в конце выполнения бизнес-транзакции, "жертвам" подобной схемы блокирования придется нелегко. Представьте себе, каково тщательно выполнять свою работу только затем, чтобы по ее окончании тебе сообщили, что все пропало. Если подобные неприятности случаются сплошь и рядом, с системой просто откажутся работать.

Пессимистическая автономная блокировка помогает предотвратить возникновение конфликта самым радикальным способом — просто не допускает его. В пессимистической схеме блокирования бизнес-транзакция накладывает блокировку на данные прежде, чем начинает с ними работать, поэтому пользователь может быть уверен, что завершит транзакцию без негативных последствий со стороны параллельных процессов.

Принцип действия

Пессимистическая автономная блокировка реализуется в три этапа: определение необходимого типа блокировок, построение диспетчера блокировки и составление правил применения блокировок. Кроме того, если **пессимистическая автономная блокировка** используется в качестве дополнения к **оптимистической автономной блокировке**, понадобится определить типы записей, которые необходимо блокировать.

Первый возможный тип блокировки — это *монопольная блокировка записи (exclusive write lock)*, которая требует наложения блокировки только для редактирования данных. Это дает возможность избежать конфликта, не позволяя двум бизнес-транзакциям одновременно вносить изменения в одну и ту же строку. Данная схема блокирования полностью игнорирует операции чтения, поэтому вполне подходит в том случае, если параллельный сеанс допускает считывание несколько устаревших данных.

Если бизнес-транзакции обязательно нужны самые свежие данные вне зависимости от того, собирается она их редактировать или нет, используйте *монопольную блокировку чтения (exclusive read lock)*. В этом случае бизнес-транзакция накладывает блокировку на данные уже при загрузке последних. Разумеется, применение подобной стратегии жестко ограничивает возможность параллельного доступа к данным. В большинстве корпоративных систем монопольная блокировка записи обеспечивает более высокую степень параллелизма, чем монопольная блокировка чтения.

Третья стратегия сочетает в себе первые два типа блокировки, что ведет к жесткому ограничению доступа, свойственному монопольной блокировке чтения, и одновременному повышению степени параллелизма системы, присущему монопольной блокировке записи. Она называется *блокировкой чтения/записи (read/write lock)* и имеет немного более сложную схему, чем первые два типа. Данная схема подразумевает определенные отношения между блокировками чтения и записи.

- Блокировки чтения и записи являются взаимоисключающими. Строка базы данных не может быть заблокирована для записи, если другая бизнес-транзакция уже заблокировала ее для чтения. Точно так же строка не может быть заблокирована для чтения, если другая бизнес-транзакция уже установила блокировку на запись.
- Допускаются параллельные блокировки чтения. Наличие хотя бы одной блокировки чтения делает невозможным редактирование строки всеми другими бизнес-транзакциями, поэтому выполнение других параллельных процессов, просматривающих эту строку, вреда не принесет.

Как можно догадаться, последний фактор повышает степень параллелизма системы. Недостатками приведенной стратегии являются сложность ее реализации и большое количество "подводных камней", которые могут заставить экспертов изрядно поломать голову при моделировании системы.

Выбирая необходимый тип блокировки, ориентируйтесь на максимизацию степени параллелизма системы, удовлетворение бизнес-требований и минимизацию сложности кода. Не забывайте также, что выбранная стратегия блокирования должна быть понятна системным аналитикам и проектировщикам предметной области. Блокировка — это не только техническая проблема. Неправильный тип блокировки или блокирование неправильных типов записей могут свести на нет эффективность всей **пессимистической автономной блокировки**. Неэффективная стратегия блокировки не сможет сдержать натиск бизнес-транзакций или, наоборот, опустит возможность параллельного доступа до уровня однопользовательской системы. Подобную стратегию не спасет даже гениальная техническая реализация. В действительности было бы отнюдь не лишним включить **пессимистическую автономную блокировку** в модель предметной области.

Определившись с типом блокировок, необходимо создать диспетчер блокировки, работа которого заключается в удовлетворении или отклонении запроса бизнес-транзакции на получение или снятие блокировки. Для этого диспетчеру необходимо знать, что блокируется, а также иметь представление о предполагаемом владельце блокировки, а именно о бизнес-транзакции. Здесь и начинаются первые сложности. Бизнес-транзакция — это не *вещь*, которая может быть однозначно идентифицирована, поэтому передать бизнес-транзакцию диспетчеру блокировки довольно сложно. Между тем в вашем распоряжении почти наверняка окажется объект *сеанса*, поэтому концепцию бизнес-транзакции можно подменить концепцией сеанса. Вообще говоря, термины "сеанс" и "бизнес-транзакция" в какой-то степени взаимозаменяемы. Поскольку бизнес-транзакция представляет собой последовательность системных транзакций, выполняемых в пределах одного сеанса, последний может с успехом выполнять роль владельца **пессимистической автономной блокировки**. Более подробно этот вопрос рассматривается несколько ниже.

В основе диспетчера блокировки лежит обыкновенная таблица, которая отображает блокировки на их владельцев. Простейший вариант диспетчера блокировки может представлять собой оболочку для хэш-таблицы, расположенной в оперативной памяти, или же быть таблицей базы данных. В любом случае в системе должна быть одна и только одна таблица блокировки, поэтому, если она расположена в оперативной памяти, обязательно воспользуйтесь типовым решением **единственный элемент (Singleton)** [20], которое гарантирует, что в системе будет существовать только один экземпляр соответствующего класса. Если сервер приложений разбит на несколько кластеров, таблица блокировки, расположенная в оперативной памяти, будет работать только в том случае, когда она прикреплена к одному экземпляру сервера. Поэтому в кластеризованных окружениях лучше использовать таблицы блокировки, расположенные в базе данных.

Вне зависимости от реализации (в виде объекта или SQL-оператора, применяемого к таблице базы данных), блокировка должна быть доступна только диспетчеру блокировки. Бизнес-транзакции должны взаимодействовать только с диспетчером, а не с самим объектом блокировки.

Пришло время определить протокол, согласно которому бизнес-транзакции будут взаимодействовать с диспетчером блокировки. В этом протоколе должно быть указано, что и когда блокировать, когда снимать блокировку и как действовать в том случае, если в предоставлении блокировки будет отказано.

Вопрос "что блокировать?" зависит от вопроса "когда блокировать?", поэтому вначале займемся последним. Как правило, бизнес-транзакции накладывают блокировку перед загрузкой данных. В противном случае применение блокировки не сможет гарантировать, что бизнес-транзакция использует последнюю версию необходимых данных. Впрочем, поскольку применение блокировки происходит в рамках системной транзакции, существуют обстоятельства, при которых последовательность загрузки и наложения блокировки не играет роли. В качестве подобных обстоятельств можно рассматривать использование уровня изоляции с поддержкой упорядочиваемых транзакций или уровня изоляции "повторяемое чтение" при наличии определенных типов блокировки. Возможной альтернативой является применение **пессимистической автономной блокировки** и последующая проверка номера версии объекта по оптимистической схеме. Тем не менее необходимо быть полностью уверенным в наличии последней версии данных, что обычно предполагает наложение блокировки перед их загрузкой.

Так что же все-таки нужно блокировать? Нам кажется, что мы блокируем записи или объекты (или нечто наподобие этого), однако на самом деле блокировка накладывается на идентификатор или первичный ключ, применяемый для обнаружения этих записей или объектов. Данная схема позволяет применять блокировку еще до загрузки самих объектов. Блокирование отдельных объектов хорошо всегда, когда не требует нарушать правило о наличии последней версии объекта после применения блокировки.

В большинстве случаев снятие блокировки осуществляется по окончании бизнес-транзакции. Иногда снимать блокировку допустимо и до окончания транзакции, в зависимости от типа блокировки и вашего намерения повторно использовать тот же объект в рамках текущей транзакции. Тем не менее, если у вас нет особых причин снимать блокировку как можно раньше (например, для обслуживания особенно большого количества параллельных сеансов), придерживайтесь принципа снятия блокировки по окончании бизнес-транзакции.

А что же делать, если в предоставлении блокировки будет отказано? Наиболее простой выход — аварийное завершение бизнес-транзакции. Подобная схема должна удовлетворить пользователя, поскольку об отказе в предоставлении блокировки сообщается довольно рано, когда основная работа еще не сделана. Проектировщик и разработчик должны тщательно следить за тем, чтобы отказ в получении особенно "дефицитных" блокировок осуществлялся как можно раньше. Вообще говоря, по возможности все блокировки следует запрашивать еще до того, как пользователь начнет работу.

Доступ к каждому блокируемому элементу должен быть упорядочен. Если таблица блокировки расположена в оперативной памяти, самый простой способ решения этой проблемы — упорядочить доступ ко всему диспетчеру блокировки с помощью конструкций используемого языка программирования. Все остальные схемы слишком сложны и обсуждаться не будут.

Если таблица блокировки расположена в базе данных, доступ к ней (первое правило!) должен осуществляться в рамках одной системной транзакции. В этом случае вы можете воспользоваться всеми средствами упорядочения доступа, имеющимися в базе данных. При наличии монопольных блокировок чтения или записи упорядочение доступа сводится

к простому наложению ограничения уникальности на столбец, содержащий идентификатор блокируемого элемента. Хранение в базе данных блокировок чтения/записи более сложно. Такая стратегия блокирования подразумевает предоставление доступа к таблице блокировки не только для вставки, но и для чтения, а потому требует специальной обработки несогласованного чтения. Полную защиту от подобных ситуаций может гарантировать уровень изоляции с поддержкой упорядочиваемых транзакций. Разумеется, использование таких транзакций по всей системе значительно снизит ее производительность. Впрочем, вы можете применять упорядочиваемые транзакции только для наложения блокировок, а в остальных ситуациях использовать менее строгие уровни изоляции. Кроме того, для управления блокировками можно воспользоваться хранимыми процедурами. Обработка параллельных заданий — занятие далеко не простое, поэтому в спорных моментах не бойтесь полагаться на средства базы данных.

Необходимость упорядочения доступа к таблице блокировки не может не отразиться на скорости работы приложения. В связи с этим стоит подумать о снижении детализации блокирования, поскольку меньшее количество блокировок позволит снизить нехватку производительности. В частности, для устранения соперничества за право доступа к таблице блокировки можно воспользоваться **блокировкой с низкой степенью детализации (Coarse-Grained Lock, 457)**.

Если необходимые данные окажутся заблокированными кем-то другим, пессимистические схемы блокирования системных транзакций наподобие оператора `SELECT FOR UPDATE` или компонентов сущностей EJB предполагают ожидание до тех пор, пока блокировка не будет снята, а следовательно, допускают возникновение взаимоблокировок. Для большей наглядности механизм возникновения взаимоблокировки можно представить следующим образом. Пусть двум пользователям одновременно понадобились ресурсы А и Б. Если первый пользователь заблокирует ресурс А, а второй заблокирует ресурс Б, обе транзакции будут бесконечно ожидать высвобождения второго ресурса. Поскольку бизнес-транзакция охватывает несколько системных транзакций, подобное ожидание теряет всякий смысл, особенно если учесть, что выполнение бизнес-транзакции может занять около 20 минут, а то и больше. Никто не захочет так долго ждать предоставления блокировки. Это и хорошо, поскольку в противном случае разработчику пришлось бы возиться с кодированием механизма времени ожидания, что отнюдь не просто. Рекомендую придерживаться стандартной схемы, при которой в случае отказа предоставления блокировки диспетчер выдает исключение. Это позволит полностью избавиться от неприятностей, связанных с взаимоблокировками.

И наконец, при управлении блокировкой необходимо позаботиться о механизме времени ожидания, необходимом для обработки утраченных транзакций. Если в процессе выполнения транзакции машина клиента выйдет из строя, транзакция никогда не будет завершена, а следовательно, блокировка не будет снята. Данная проблема особенно критична для Web-приложений, которые слишком часто забывают закрывать должным образом. В идеале обработкой времени ожидания должно заниматься не само приложение, а сервер приложений. Для этого серверы Web-приложений применяют HTTP-сеансы. Механизм времени ожидания может быть реализован путем регистрации объекта, который автоматически высвобождает все блокировки, когда HTTP-сеанс станет недействительным. Вместо этого каждой блокировке можно присвоить временную метку и считать недействительными все блокировки, время жизни которых больше определенного значения.

Назначение

Пессимистическая автономная блокировка применяется тогда, когда вероятность возникновения конфликта между параллельными сеансами достаточно высока. Пользователь никогда не должен переделывать свою работу. К пессимистической схеме блокирования следует обращаться и в том случае, когда стоимость ликвидации последствий конфликта слишком велика независимо от природы последнего. Разумеется, блокирование каждой сущности в системе почти наверняка спровоцирует огромную конкуренцию доступа к данным, поэтому **пессимистическая автономная блокировка** должна рассматриваться как дополнение к **оптимистической автономной блокировке** и применяться только тогда, когда без нее действительно не обойтись.

Прежде чем окончательно принимать решение в пользу **пессимистической автономной блокировки**, подумайте о применении длинных транзакций. Хотя они никогда не считались удачным выбором, в некоторых ситуациях они способны нанести приложению не больше вреда, чем **пессимистическая автономная блокировка**, а программировать их гораздо легче. Рекомендую провести несколько тестов, чтобы посмотреть, какой из этих приемов наносит меньше урона параллельному доступу к данным.

Не используйте упомянутые приемы, если бизнес-транзакция умещается в рамки одной системной транзакции. Большинство серверов приложений и баз данных поставляются с собственными схемами пессимистического блокирования системных транзакций, в числе которых можно назвать SQL-оператор `SELECT FOR UPDATE` (для баз данных) и компоненты сущностей EJB (для серверов приложений). Зачем же утруждать себя подбором времени ожидания, областью видимости блокировки и другими подобными вещами, если все это уже есть? Понимание перечисленных типов блокировки способно сыграть существенную роль в реализации **пессимистической автономной блокировки**. Однако обратите внимание на то, что противоположное утверждение в корне неверно! Материал, изложенный в этой главе, никак не научит вас писать диспетчеры баз данных или системы управления транзакциями. Все схемы автономного блокирования, рассматриваемые в этой книге, предназначены исключительно для систем, имеющих собственные средства управления транзакциями.

Пример: простой диспетчер блокировки (Java)

В этом примере создадим диспетчер блокировки для наложения монопольных блокировок чтения (напомню, что эти блокировки применяются для считывания или записи объекта). Затем продемонстрируем, как использовать диспетчер блокировки для управления бизнес-транзакцией, которая охватывает несколько системных транзакций.

Вначале определим интерфейс диспетчера блокировки.

```
interface ExclusiveReadLockManager...

   public static final ExclusiveReadLockManager INSTANCE =
↳(ExclusiveReadLockManager) Plugins.getPlugin(
↳ExclusiveReadLockManager.class);
   public void acquireLock(Long lockable, String owner) throws
↳ConcurrencyException;
   public void releaseLock(Long lockable, String owner);
   public void relaseAllLocks(String owner);
```

Обратите внимание, что параметр `lockable` ("блокируемый элемент") имеет тип `Long`, а параметр `owner` ("владелец") — тип `String`. Это сделано по двум причинам. Во-первых, каждая таблица нашей базы данных использует первичный ключ типа `Long`. Этот ключ уникален в пределах базы данных и поэтому прекрасно подходит на роль блокируемого идентификатора (который должен быть уникален для всех объектов, обрабатываемых таблицей блокировки). Во-вторых, мы рассматриваем Web-приложение, а потому в качестве владельца блокировки удобно использовать идентификатор HTTP-сеанса, являющийся строкой.

В нашем примере диспетчер будет взаимодействовать не с объектом блокировки, а непосредственно с таблицей блокировки, расположенной в базе данных. Обратите внимание, что таблица по имени `lock`, как и все другие таблицы приложения, создана нами и не является частью внутреннего механизма блокирования базы данных. Получение блокировки эквивалентно вставке в таблицу блокировки новой строки, а высвобождение блокировки — удалению соответствующей строки. Ниже приведена схема таблицы `lock` и часть реализации диспетчера блокировки.

```
table lock...

   create table lock(lockableid bigint primary key,
↳ownerid bigint)

class ExclusiveReadLockManagerDBImpl implements
↳ExclusiveLockManager...

   private static final String INSERT_SQL =
↳"insert into lock values(?, ?)";
   private static final String DELETE_SINGLE_SQL =
↳"delete from lock where lockableid = ? and ownerid = ?";
   private static final String DELETE_ALL_SQL =
↳"delete from lock where ownerid = ?";
   private static final String CHECK_SQL = "select lockableid
↳from lock where lockableid = ? and ownerid = ?";
   public void acquireLock(Long lockable, String owner) throws
↳ConcurrencyException {
      if (!hasLock(lockable, owner)) {
         Connection conn = null;
         PreparedStatement pstmt = null;
         try {
            conn = ConnectionManager.INSTANCE.getConnection();
            pstmt = conn.prepareStatement(INSERT_SQL);
            pstmt.setLong(1, lockable.longValue());
            pstmt.setString(2, owner);
            pstmt.executeUpdate();
         } catch (SQLException sqlEx) {
            throw new ConcurrencyException("unable to lock " +
↳lockable);
         } finally {
            closeDBResources(conn, pstmt);
         }
      }
   }
```

```
   public void releaseLock(Long lockable, String owner) {
      Connection conn = null;
      PreparedStatement pstmt = null;
      try {
         conn = ConnectionManager.INSTANCE.getConnection();
         pstmt = conn.prepareStatement(DELETE_SINGLE_SQL);
         pstmt.setLong(1, lockable.longValue());
         pstmt.setString(2, owner);
         pstmt.executeUpdate();
      } catch (SQLException sqlEx) {
         throw new SystemException("unexpected error releasing
↳lock on " + lockable);
      } finally {
         closeDBResources(conn, pstmt);
      }
   }
```

В приведенном фрагменте кода не показана реализация открытого метода `releaseAllLocks()` и закрытого метода `hasLock()`. Суть метода `releaseAllLocks()` точно отражена в его имени: он снимает все существующие блокировки указанного владельца. Метод `hasLock()` обращается к базе данных с запросом о том, не существует ли у владельца блокировки на указанный элемент. При выполнении транзакций нередки ситуации, когда сеанс запрашивает блокировку, которая у него уже есть. Поэтому перед вставкой в таблицу блокировки новой строки метод `acquireLock()` должен обратиться к таблице и посмотреть, нет ли в ней этой же блокировки. Поскольку таблица блокировки обычно является точкой соперничества за право обладания ресурсами, подобные обращения могут негативно сказаться на производительности приложения. Таким образом, для выполнения проверки на наличие блокировок может понадобиться кэшировать их на уровне сеанса. Будьте очень внимательны, кэшируя блокировки!

Перейдем к созданию простенького Web-приложения, которое будет предоставлять записи о покупателях. Вначале создадим некоторое подобие инфраструктуры для обработки бизнес-транзакций. Слоям, находящимся под уровнем Web-компонентов, необходим объект пользовательского сеанса, поэтому одного лишь HTTP-сеанса будет недостаточно. Чтобы отличить новый объект сеанса от HTTP-сеанса, назовем его сеансом приложения. Класс сеанса приложения будет содержать идентификатор сеанса, имя пользователя и **коллекцию объектов (Identity Map, 216)** для кэширования объектов, загруженных или созданных во время бизнес-транзакции. Объекты сеанса будут ассоциированы с текущим потоком выполнения в порядке их обнаружения этим потоком.

```
class AppSession...

   private String user;
   private String id;
   private IdentityMap imap;
   public AppSession(String user, String id, IdentityMap imap) {
      this.user = user;
      this.imap = imap;
      this.id = id;
   }

class AppSessionManager...
```

```
   private static ThreadLocal current = new ThreadLocal();
   public static AppSession getSession() {
      return (AppSession) current.get();
   }
   public static void setSession(AppSession session) {
      current.set(session);
   }
```

Для обработки запросов к Web-приложению будет применяться **контроллер запросов (Front Controller, 362)**, поэтому нам понадобится определить объект команды. Первое, что должен выполнять объект команды, — указывать на необходимость начала новой бизнес-транзакции или продолжения текущей. Для этого следует соответственно установить новый сеанс приложения или возвратить текущий. Ниже приведено описание абстрактного класса команды, содержащего общие методы по установке контекста бизнес-транзакций.

```
interface Command...

   public void init(HttpServletRequest req, HttpServletResponse
↳rsp);
   public void process() throws Exception;

abstract class BusinessTransactionCommand implements Command...

   public void init(HttpServletRequest req, HttpServletResponse
↳rsp) {
      this.req = req;
      this.rsp = rsp;
   }
   protected void startNewBusinessTransaction() {
      HttpSession httpSession = getReq().getSession(true);
      AppSession appSession = (AppSession)
↳httpSession.getAttribute(APP_SESSION);
      if (appSession != null) {
         ExclusiveReadLockManager.INSTANCE.relaseAllLocks(
↳appSession.getId());
      }
      appSession = new AppSession(getReq().getRemoteUser(),
↳httpSession.getId(), new IdentityMap());
      AppSessionManager.setSession(appSession);
      httpSession.setAttribute(APP_SESSION, appSession);
      httpSession.setAttribute(LOCK_REMOVER,
      new LockRemover(appSession.getId()));
   }
   protected void continueBusinessTransaction() {
      HttpSession httpSession = getReq().getSession();
      AppSession appSession = (AppSession)
↳httpSession.getAttribute(APP_SESSION);
      AppSessionManager.setSession(appSession);
   }
   protected HttpServletRequest getReq() {
      return req;
```

```
    }
    protected HttpServletResponse getRsp() {
        return rsp;
    }
```

Обратите внимание, что при установке нового сеанса приложения мы снимаем все блокировки существующего сеанса. Мы также добавляем класс `LockRemover`, реализующий интерфейс `HttpSessionBindingListener`, который будет удалять все блокировки, принадлежащие сеансу приложения, по истечении срока жизни соответствующего HTTP-сеанса.

```
class LockRemover implements HttpSessionBindingListener...

    private String sessionId;
    public LockRemover(String sessionId) {
        this.sessionId = sessionId;
    }
    public void valueUnbound(HttpSessionBindingEvent event) {
        try {
            beginSystemTransaction();
            ExclusiveReadLockManager.INSTANCE.relaseAllLocks(
↳this.sessionId);
            commitSystemTransaction();
        } catch (Exception e) {
            handleSeriousError(e);
        }
    }
```

В нашем примере объекты команд содержат как бизнес-логику, так и логику управления блокировками. Кроме того, каждая команда должна выполняться в рамках одной системной транзакции. Для этого можно модифицировать поведение команды с помощью декоратора [20], представленного объектом `TransactionalCommand`. Реализуя объекты команд, убедитесь, что все операции по управлению блокировками и вся стандартная бизнес-логика по обработке одного запроса находятся в рамках единой системной транзакции. Методы, определяющие рамки системной транзакции, зависят от контекста развертывания приложения. Откат системной транзакции происходит при генерации исключения `ConcurrencyException` (а в нашем примере — и всех других исключений). Это позволит предотвратить внесение каких-либо постоянных изменений при возникновении конфликта.

```
class TransactionalCommand implements Command...

    public TransactionalCommand(Command impl) {
        this.impl = impl;
    }
    public void process() throws Exception {
        beginSystemTransaction();
        try {
            impl.process();
            commitSystemTransaction();
        } catch (Exception e) {
```

```
            rollbackSystemTransaction();
            throw e;
        }
    }
```

Теперь нужно написать сервлет, выполняющий роль контроллера, и конкретные классы команд. Сервлет контроллера добавляет к объектам команд методы управления транзакциями; конкретные классы команд необходимы для установки контекста бизнес-транзакции, выполнения логики домена, а также наложения и снятия блокировок там, где это потребуется.

```
class ControllerServlet extends HttpServlet...

    protected void doGet(HttpServletRequest req,
↳HttpServletResponse rsp) throws ServletException, IOException {
        try {
            String cmdName = req.getParameter("command");
            Command cmd = getCommand(cmdName);
            cmd.init(req, rsp);
            cmd.process();
        } catch (Exception e) {
            writeException(e, rsp.getWriter());
        }
    }
    private Command getCommand(String name) {
        try {
            String className = (String) commands.get(name);
            Command cmd = (Command) Class.forName(
↳className).newInstance();
            return new TransactionalCommand(cmd);
        } catch (Exception e) {
            e.printStackTrace();
            throw new SystemException("unable to create command
↳object for " + name);
        }
    }

class EditCustomerCommand implements Command...

    public void process() throws Exception {
        startNewBusinessTransaction();
        Long customerId = new Long(getReq().getParameter(
↳"customer_id"));
        ExclusiveReadLockManager.INSTANCE.acquireLock(
↳customerId, AppSessionManager.getSession().getId());
        Mapper customerMapper =
↳MapperRegistry.INSTANCE.getMapper(Customer.class);
        Customer customer = (Customer) customerMapper.find(
↳customerId);
        getReq().getSession().setAttribute("customer", customer);
        forward("/editCustomer.jsp");
    }
```

```
class SaveCustomerCommand implements Command...

   public void process() throws Exception {
      continueBusinessTransaction();
      Customer customer = (Customer)
            getReq().getSession().getAttribute("customer");
      String name = getReq().getParameter("customerName");
      customer.setName(name);
      Mapper customerMapper =
            MapperRegistry.INSTANCE.getMapper(Customer.class);
      customerMapper.update(customer);
      ExclusiveReadLockManager.INSTANCE.releaseLock(
↳customer.getId(), AppSessionManager.getSession().getId());
      forward("/customerSaved.jsp");
   }
```

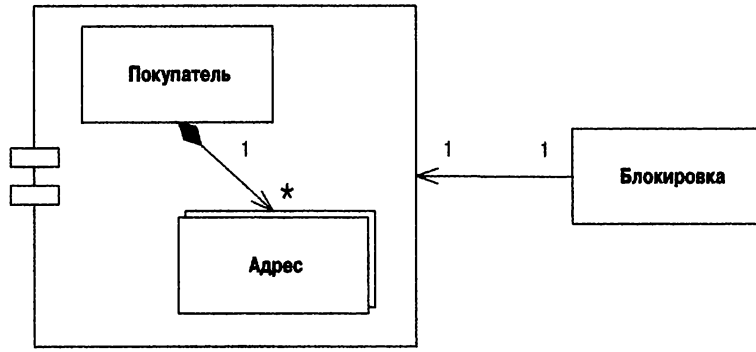
Описанные команды предотвращают одновременный доступ двух сеансов к одному и тому же объекту покупателя. Любая другая команда приложения, работающая с объектом покупателя, должна либо применить блокировку, либо работать только с тем объектом, который был заблокирован предыдущей командой в рамках текущей бизнес-транзакции. Поскольку диспетчер блокировки выполняет проверку на наличие существующей блокировки с помощью метода `hasLock()`, мы могли бы не усложнять свою задачу и применять блокировку для каждой команды. Это бы немного снизило производительность, однако гарантировало наличие блокировки. Более подробно механизмы защиты от неправильного использования блокировок рассматриваются далее в главе.

Как видите, для описания инфраструктуры понадобилось гораздо большее количество кода, чем для реализации самой логики домена. Действительно, внедрение в жизнь **пессимистической автономной блокировки** требует, как минимум, виртуозного пританцовывания вокруг сеанса приложения, бизнес-транзакции, диспетчера блокировки и системной транзакции, что, разумеется, достаточно сложно. Кроме того, в данном примере есть множество слабых мест, поэтому его следует рассматривать скорее как идею, чем как полноценное архитектурное решение.

Блокировка с низкой степенью детализации (Coarse-Grained Lock)

Дейвид Райс и Мэттью Фоммел

Блокирует группу взаимосвязанных объектов как единый элемент



Довольно часто объекты приходится редактировать в составе группы. Предположим, у вас есть объект покупателя и множество его адресов. В этом случае при необходимости блокировки одного из элементов имеет смысл заблокировать и все остальные. Между тем применение к каждому объекту отдельной блокировки связано с определенными проблемами. Во-первых, разработчику придется писать код, который бы обнаруживал все объекты группы, чтобы их заблокировать. Это не слишком сложно для покупателя и его адресов, однако при наличии большего количества групп блокировки реализация подобного механизма может стать затруднительной. А что, если группы будут иметь более сложную структуру? Если стратегия блокирования подразумевает, что для наложения блокировки объект должен быть загружен (принцип **оптимистической автономной блокировки (Optimistic Offline Lock, 434)**), блокирование большой группы объектов значительно снизит производительность. В свою очередь, при использовании **пессимистической автономной блокировки (Pessimistic Offline Lock, 445)** наличие большой группы объектов крайне запутывает управление блокировками и повышает конкуренцию за получение доступа к таблице блокировки.

Типовое решение **блокировка с низкой степенью детализации** накладывает блокировку сразу на группу объектов. Это не только упрощает применение блокировки, но и освобождает от необходимости загружать все члены группы, чтобы заблокировать каждый из них.

Принцип действия

Чтобы реализовать **блокировку с низкой степенью детализации**, необходимо создать единую точку соперничества за право доступа к группе блокируемых элементов. В этом случае для блокирования всей группы объектов потребуется только одна блокировка. После этого вам понадобится предоставить кратчайший путь для обнаружения точки

блокировки, чтобы для ее наложения пришлось идентифицировать (и при необходимости загрузить) как можно меньше объектов группы.

При использовании **оптимистической автономной блокировки** для создания единой точки доступа к группе объектов необходимо, чтобы они совместно использовали *один и тот же* (а не *такой же*) объект версии (рис. 16.2). Увеличение номера версии приведет к наложению на группу объектов *общей блокировки (shared lock)*. В этом случае для минимизации пути к точке блокировки достаточно, чтобы каждый объект группы указывал на общий объект версии.

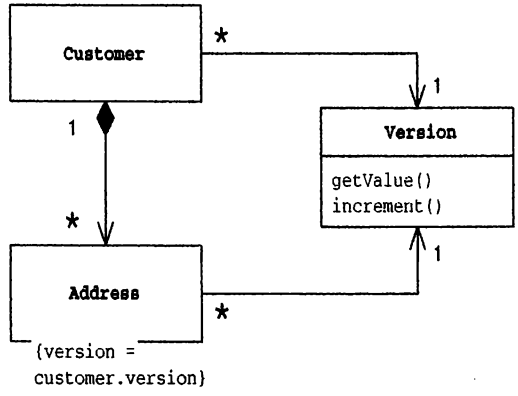


Рис. 16.2. Совместное использование объекта версии

Применение общей **пессимистической автономной блокировки** требует, чтобы все члены группы совместно использовали некий объект наподобие маркера, на который будет накладываться блокировка. Поскольку **пессимистическая автономная блокировка** часто выступает в качестве дополнения к **оптимистической автономной блокировке**, на роль блокируемого маркера прекрасно подойдет общий объект версии (рис. 16.3).

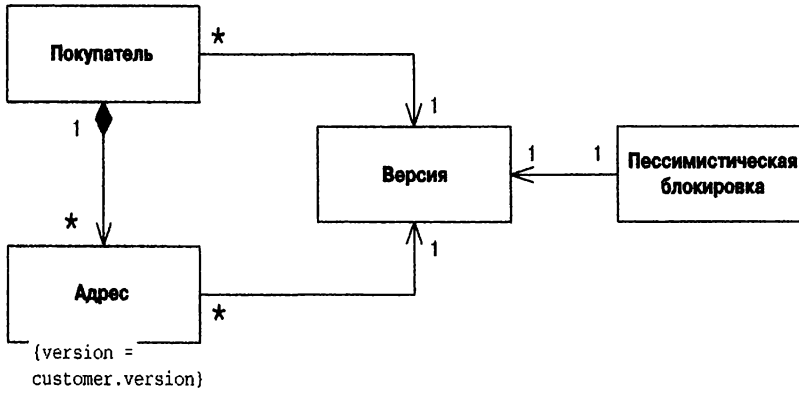


Рис. 16.3. Блокирование совместно используемого объекта версии

Эрик Эванс (Eric Evans) и Дейвид Сигель (David Siegel) [15] определяют *агрегат (aggregate)* как совокупность взаимосвязанных объектов, рассматриваемых с точки внесения изменений как единое целое. У каждого агрегата есть *корневой элемент (root)*, являющийся единственной точкой доступа к объектам этого агрегата, а также *граница (boundary)*, определяющая, какие объекты входят в агрегат. Перечисленные свойства агрегата требуют применения **блокировки с низкой степенью детализации**, потому что для работы с одним из элементов агрегата необходимо заблокировать и все остальные элементы. Блокирование агрегата представляет собой еще одну форму **блокировки с низкой степенью детализации**, отличную от общей блокировки. Назовем ее *блокировкой корневого элемента (root lock)* (рис. 16.4). По определению блокирование корневого элемента распространяется на все элементы агрегата. Таким образом, корневой элемент является единственной точкой соперничества за доступ к группе объектов, входящих в агрегат.

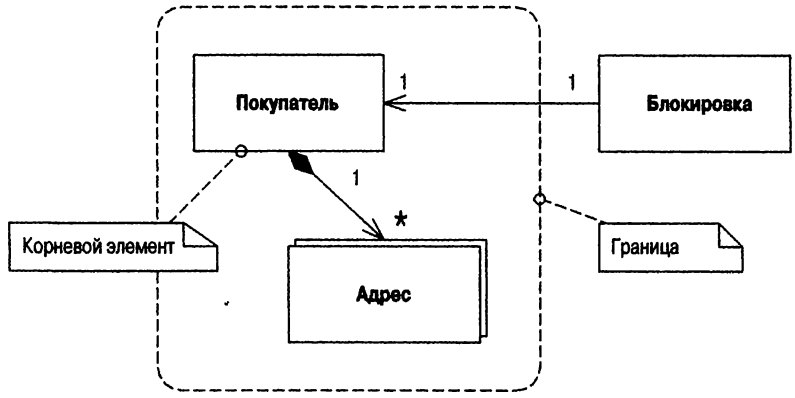


Рис. 16.4. Блокирование корневого элемента

Использование блокирования корневого элемента в качестве **блокировки с низкой степенью детализации** требует наличия механизма перехода к корневому элементу графа объектов. В этом случае при запросе на получение блокировки для одного из объектов графа механизм блокирования сможет перейти к корневому элементу и заблокировать только его. Для реализации механизма перехода можно воспользоваться прямым соединением каждого объекта агрегата с корневым элементом или же последовательностью промежуточных соединений. В качестве примера реализации этих способов можно рассмотреть иерархию наследования. Очевидно, корневым элементом этого агрегата является родитель верхнего уровня, поэтому каждый элемент иерархии может непосредственно ссылаться на вершину графа. Вместо этого каждый узел графа может ссылаться на своего прямого родителя, а корневой элемент будет достигаться путем последовательного перемещения вверх по таким ссылкам. Последняя стратегия требует загрузки каждого следующего родителя, чтобы определить, есть ли у него свой родитель, что может привести к значительному падению производительности в больших графах объектов. Поэтому для перемещения к корневому элементу необходимо применять **загрузку по требованию (Lazy Load, 220)**. Это позволит не только избежать загрузки ненужных объектов, но и справиться с циклическими ссылками. Не забывайте, однако, что применение **загрузки по требованию** к одному агрегату может растянуться на несколько системных транзакций, поэтому

различные части агрегата могут оказаться несогласованными. Разумеется, это очень и очень плохо.

Обратите внимание, что общая блокировка может применяться и для блокирования агрегата, поскольку блокирование любого объекта агрегата автоматически заблокирует корневой элемент.

Обе реализации **блокировки с низкой степенью детализации** — в виде общей блокировки или же блокировки корневого элемента — имеют как преимущества, так и недостатки. Применение общей блокировки к записям реляционной базы данных требует выполнения соединений с таблицей версий в каждом операторе `SELECT`. В свою очередь, последовательная загрузка объектов при перемещении к корневому элементу также может привести к падению производительности. Вообще говоря, сочетание блокировки корневого элемента и **пессимистической автономной блокировки** удачным не назовешь. К тому времени как вы загрузите все необходимые объекты и доберетесь до корневого элемента, вам может понадобиться перезагрузить несколько объектов, чтобы гарантировать наличие последних версий. И наконец, большие ограничения на выбор реализации накладывает построение системы для работы с существующим источником данных. Сколько бы ни было вариантов реализации, тонкостей их применения еще больше. Убедитесь, что выбранная реализация в точности соответствует потребностям вашего приложения.

Назначение

В большинстве случаев **блокировка с низкой степенью детализации** применяется для удовлетворения бизнес-требований. Это особенно справедливо при блокировании агрегатов. Представьте себе объект, содержащий коллекцию предметов имущества, сдаваемых в аренду. Скорее всего, вам не захочется допускать ситуации, когда один пользователь редактирует объект аренды, а другой — один из предметов имущества, входящих в коллекцию. Поэтому блокирование объекта аренды или одного из предметов имущества должно приводить к блокированию объекта аренды и всех предметов имущества, входящих в его коллекцию.

Огромным преимуществом использования **блокировки с низкой степенью детализации** является относительная дешевизна наложения и снятия блокировки, что, несомненно, служит серьезным аргументом в пользу данного типового решения. Общая блокировка может быть применена и к концепции агрегата [15]. Следует, однако, быть осторожным, используя данную схему блокирования для удовлетворения нефункциональных требований, например повышения производительности. Кроме того, реализация **блокировки с низкой степенью детализации** может привести к неестественным отношениям между объектами.

Пример: общая оптимистическая автономная блокировка (Java)

В данном примере воспользуемся моделью домена с **супертипом слоя (Layer Supertype, 491)** и **преобразователями данных (Data Mapper, 187)**. В качестве постоянного хранилища данных будет выступать реляционная база данных.

Вначале нужно создать класс и таблицу для работы с номерами версий. Для простоты создадим достаточно обширный класс `Version`, который будет содержать в себе не только значение номера версии, но и статический метод поиска. Обратите внимание, что для кэширования объектов версий, применяемых в текущем сеансе, будет использоваться

коллекция объектов (Identity Map, 216). Если блокируемые объекты совместно используют объект версии, они обязательно должны указывать на один и тот же экземпляр последнего. Поскольку класс `Version` является частью модели домена, он совершенно не подходит для размещения в нем кода работы с базой данных. Этот код лучше вынести в слой преобразователей, что вы вполне сможете проделать самостоятельно.

```
table version...

   create table version(id bigint primary key, value bigint,
↳modifiedBy varchar, modified datetime)

class Version...

   private Long id;
   private long value;
   private String modifiedBy;
   private Timestamp modified;
   private boolean locked;
   private boolean isNew;
   private static final String UPDATE_SQL =
↳"UPDATE version SET VALUE = ?, modifiedBy = ?,
↳modified = ? " + "WHERE id = ? and value = ?";
   private static final String DELETE_SQL =
↳"DELETE FROM version WHERE id = ? and value = ?";
   private static final String INSERT_SQL =
↳"INSERT INTO version VALUES (?, ?, ?, ?)";
   private static final String LOAD_SQL =
↳"SELECT id, value, modifiedBy, modified FROM
↳version WHERE id = ?";
   public static Version find(Long id) {
      Version version =
         AppSessionManager.getSession().getIdentityMap().
↳getVersion(id);
      if (version == null) {
         version = load(id);
      }
      return version;
   }
   private static Version load(Long id) {
      ResultSet rs = null;
      Connection conn = null;
      PreparedStatement pstmt = null;
      Version version = null;
      try {
         conn = ConnectionManager.INSTANCE.getConnection();
         pstmt = conn.prepareStatement(LOAD_SQL);
         pstmt.setLong(1, id.longValue());
         rs = pstmt.executeQuery();
         if (rs.next()) {
            long value = rs.getLong(2);
            String modifiedBy = rs.getString(3);
            Timestamp modified = rs.getTimestamp(4);
            version = new Version(id, value, modifiedBy,
```

```
↳modified);
            AppSessionManager.getSession().getIdentityMap().
↳putVersion(version);
         } else {
            throw new ConcurrencyException("version " + id +
↳" not found.");
         }
      } catch (SQLException sqlEx) {
         throw new SystemException("unexpected sql error loading
↳version", sqlEx);
      } finally {
         cleanupDBResources(rs, conn, pstmt);
      }
      return version;
   }
```

Класс `Version` содержит метод для создания нового экземпляра объекта версии. Вставка соответствующей строки в базу данных отделена от создания объекта версии, что позволяет отложить выполнение вставки до тех пор, пока в базу данных не будет добавлен хотя бы один владелец указанного объекта версии. Каждый **преобразователь данных** нашей предметной области может без опасения вызывать метод вставки объекта версии при вставке соответствующего объекта домена. Перед проведением вставки объект версии проверяет, является ли данная версия новой, чтобы убедиться, что она не будет вставлена дважды.

```
class Version...

   public static Version create() {
      Version version = new Version(
↳IdGenerator.INSTANCE.nextId(), 0,
↳AppSessionManager.getSession().getUser(), now());
      version.isNew = true;
      return version;
   }
   public void insert() {
      if (isNew()) {
         Connection conn = null;
         PreparedStatement pstmt = null;
         try {
            conn = ConnectionManager.INSTANCE.getConnection();
            pstmt = conn.prepareStatement(INSERT_SQL);
            pstmt.setLong(1, this.getId().longValue());
            pstmt.setLong(2, this.getValue());
            pstmt.setString(3, this.getModifiedBy());
            pstmt.setTimestamp(4, this.getModified());
            pstmt.executeUpdate();
            AppSessionManager.getSession().getIdentityMap().
↳putVersion(this);
            isNew = false;
         } catch (SQLException sqlEx) {
            throw new SystemException("unexpected sql error
↳inserting version", sqlEx);
```

```
            } finally {
                cleanupDBResources(conn, pstmt);
            }
        }
    }
```

Теперь нужно реализовать метод `increment()`, который будет увеличивать номер версии в соответствующей строке базы данных. Довольно часто объект версии совместно используется несколькими объектами из текущего набора изменений, поэтому, прежде чем увеличить свой номер, объект версии должен убедиться, что он еще не был заблокирован. После обращения к базе данных метод `increment()` проверяет, действительно ли строка с номером версии была обновлена. Если количество измененных строк равно нулю, метод `increment()` распознает нарушение параллелизма и выдает соответствующее исключение.

```
class Version...

    public void increment() throws ConcurrencyException {
        if (!isLocked()) {
            Connection conn = null;
            PreparedStatement pstmt = null;
            try {
                conn = ConnectionManager.INSTANCE.getConnection();
                pstmt = conn.prepareStatement(UPDATE_SQL);
                pstmt.setLong(1, value + 1);
                pstmt.setString(2, getModifiedBy());
                pstmt.setTimestamp(3, getModified());
                pstmt.setLong(4, id.longValue());
                pstmt.setLong(5, value);
                int rowCount = pstmt.executeUpdate();
                if (rowCount == 0) {
                    throwConcurrencyException();
                }
                value++;
                locked = true;
            } catch (SQLException sqlEx) {
                throw new SystemException("unexpected sql error
↳incrementing version", sqlEx);
            } finally {
                cleanupDBResources(conn, pstmt);
            }
        }
    }
    private void throwConcurrencyException() {
        Version currentVersion = load(this.getId());
        throw new ConcurrencyException(
↳"version modified by " + currentVersion.modifiedBy
↳+ " at " + DateFormat.getDateTimeInstance().
↳format(currentVersion.getModified()));
    }
```

Реализованный приведенным способом метод `increment()` должен вызываться только в той системной транзакции, в которой происходит фиксация результатов бизнес-транзакции. Флаг `isLocked` срабатывает таким образом, что увеличение номера версии в более ранних транзакциях приводит к ложному наложению блокировки задолго до выполнения фиксации. Это неправильно, поскольку основная идея оптимистического блокирования заключается именно в том, чтобы накладывать блокировку только во время фиксации результатов.

При использовании оптимистической схемы блокирования может понадобиться проверить базу данных на наличие последней версии объекта в более ранних системных транзакциях. Для этого к классу `Version` можно добавить метод `checkCurrent()`, который будет просто проверять нужный объект на возможность получения **оптимистической автономной блокировки** без выполнения каких-либо обновлений.

В предыдущих фрагментах кода не показан метод `delete`, который вызывает SQL-оператор для удаления объекта версии из базы данных. Если в результате выполнения этого оператора возвращается количество измененных строк, равное нулю, метод выдает исключение `ConcurrencyException`. Это означает, что при удалении последнего из объектов, использующих данный объект версии, **оптимистическая автономная блокировка** могла быть не получена, чего никогда не следует допускать. Самое сложное в реализации удаления — определить момент, когда общий объект версии может быть уничтожен. Если объект версии совместно используется элементами агрегата, этот объект нужно удалить после удаления корневого элемента агрегата. В других случаях определить корректный момент удаления объекта версии гораздо сложнее. В качестве возможного решения можно предложить хранение объектом версии количества своих владельцев и удаление объекта, когда это количество достигнет нуля. К сожалению, подобная схема требует разработки довольно сложного объекта версии — настолько сложного, что его может понадобиться реализовать в виде полноценного объекта домена. Это, конечно, не так уж плохо, однако учтите, что полученный объект домена будет отличаться от всех остальных отсутствием номера собственной версии.

Теперь рассмотрим совместное использование объекта версии. **Супертип слоя** домена содержит поле `version`, значением которого является не номер версии, а целый объект. **Преобразователь данных** может установить значение поля `version` при загрузке соответствующего объекта домена.

```
class DomainObject...

   private Long id;;
   private Timestamp modified;
   private String modifiedBy;
   private Version version;
   public void setSystemFields(Version version, Timestamp
↳modified, String modifiedBy) {
      this.version = version;
      this.modified = modified;
      this.modifiedBy = modifiedBy;
   }
```

Вначале рассмотрим создание объектов. В качестве примера воспользуемся агрегатом, состоящим из объекта покупателя (корневой элемент) и его адресов. Метод `create` объекта `Customer` будет создавать общий объект версии. Кроме того, у объекта `Customer` есть метод `getAddress()`, который создает объект адреса, передавая ему соответствующий объект версии. Преобразователь `AbstractMapper` будет вставлять в базу данных запись о новом объекте версии до вставки записей о соответствующих объектах домена. Напомню: методы объекта версии гарантируют, что он будет вставлен в базу данных только один раз.

```
class Customer extends DomainObject...

   public static Customer create(String name) {
      return new Customer(IdGenerator.INSTANCE.nextId(),
↳Version.create(), name, new ArrayList());
   }

class Customer extends DomainObject...

   public Address addAddress(String line1, String city,
↳String state) {
      Address address = Address.create(this, getVersion(),
↳line1, city, state);
      addresses.add(address);
      return address;
   }

class Address extends DomainObject...

   public static Address create(Customer customer, Version
↳version, String line1, String city, String state) {
      return new Address(IdGenerator.INSTANCE.nextId(), version,
↳customer, line1, city, state);
   }

class AbstractMapper...

   public void insert(DomainObject object) {
      object.getVersion().insert();
```

Перед обновлением или удалением объекта домена **преобразователь данных** должен увеличить номер соответствующей версии.

```
class AbstractMapper...

   public void update(DomainObject object) {
      object.getVersion().increment();

class AbstractMapper...

   public void delete(DomainObject object) {
      object.getVersion().increment();
```

Поскольку речь идет об агрегате, объект покупателя удаляется вместе со всеми объектами адресов. Это позволит удалить объект версии сразу же после удаления объекта покупателя.

```
class CustomerMapper extends AbstractMapper...

   public void delete(DomainObject object) {
      Customer cust = (Customer) object;
      for (Iterator iterator = cust.getAddresses().iterator();
                   iterator.hasNext();) {
         Address add = (Address) iterator.next();
         MapperRegistry.getMapper(Address.class).delete(add);
      }
      super.delete(object);
      cust.getVersion().delete();
   }
```

Пример: общая пессимистическая автономная блокировка (Java)

Для применения общей блокировки в пессимистической схеме блокирования нужно подобрать некий блокируемый маркер, на который будут ссылаться все объекты взаимосвязанного множества. Как уже отмечалось, будем использовать **пессимистическую автономную блокировку** в качестве дополнения к **оптимистической автономной блокировке**, поэтому на роль блокируемого маркера прекрасно подойдет общий объект версии. Весь код для получения общего объекта версии остается тем же, что и в предыдущем примере.

Реализация нашей задачи связана с единственным спорным моментом. Как известно, для получения номера версии необходимо загрузить некоторые объекты. Если после загрузки данных применить **пессимистическую автономную блокировку**, как можно гарантировать, что в нашем распоряжении находятся самые свежие версии объектов? Самое простое, что можно предпринять в данной ситуации, — увеличить номер версии в той же системной транзакции, в которой происходит получение **пессимистической автономной блокировки**. Как только системная транзакция будет зафиксирована, пессимистическая блокировка вступит в силу. Таким образом, мы можем быть уверены, что используем последние версии объектов агрегата, независимо от того, в какой момент системной транзакции произошла их загрузка.

```
class LoadCustomerCommand...

      try {
         Customer customer = (Customer)
↳MapperRegistry.getMapper(Customer.class).find(id);
         ExclusiveReadLockManager.INSTANCE.acquireLock(
↳customer.getId(), AppSessionManager.getSession().getId());
         customer.getVersion().increment();
         TransactionManager.INSTANCE.commit();
      } catch (Exception e) {
         TransactionManager.INSTANCE.rollback();
         throw e;
      }
```

В реальных приложениях код увеличения номера версии рекомендуется встраивать в диспетчер блокировки или по крайней мере снабжать последний декоратором [20], который будет выполнять это увеличение. И разумеется, реальное приложение потребует намного более солидной обработки исключений и управления транзакциями, чем было показано в данном примере.

Пример: оптимистическая автономная блокировка корневого элемента (Java)

В данном примере используются почти все те же решения, что и в предыдущих примерах, включая **супертип слоя** домена и **преобразователи данных**. Как и раньше, у нас есть объект версии, однако на сей раз он не является совместно используемым. Данный объект просто реализует метод `increment()`, чтобы облегчить наложение **оптимистической автономной блокировки** за пределами **преобразователя данных**. Кроме того, для отслеживания изменений будет применяться **единица работы (Unit of Work, 205)**.

В качестве примера выбран агрегат, объекты которого связаны отношениями типа "родитель–потомок", поэтому для перехода к корневому элементу будем последовательно перемещаться по ссылкам потомков к вышестоящим родителям. Для этого понадобится немного изменить модель данных и модель домена.

```
class DomainObject...

   private Long id;
   private DomainObject parent;
   public DomainObject(Long id, DomainObject parent) {
      this.id = id;
      this.parent = parent;
   }
```

Прежде чем зафиксировать изменения агрегата, отслеживаемые **единицей работы**, необходимо заблокировать корневой элемент.

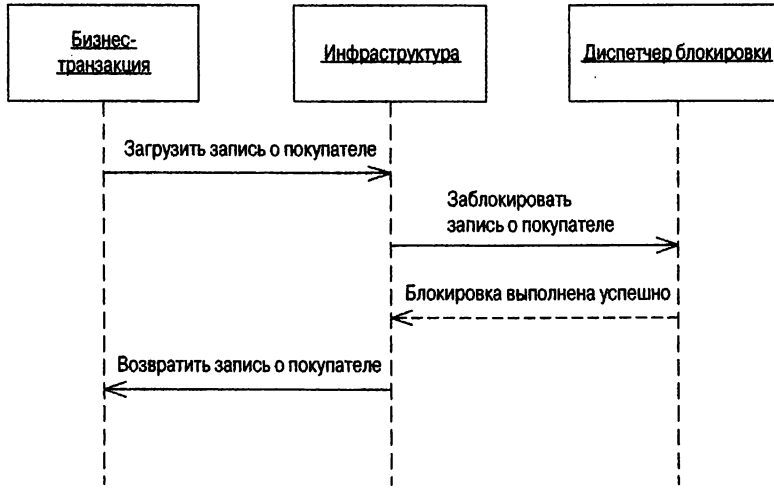
```
class UnitOfWork...

   public void commit() throws SQLException {
      for (Iterator iterator = _modifiedObjects.iterator();
                   iterator.hasNext();) {
         DomainObject object = (DomainObject) iterator.next();
         for (DomainObject owner = object; owner != null;
                   owner = owner.getParent()) {
            owner.getVersion().increment();
         }
      }
      for (Iterator iterator = _modifiedObjects.iterator();
                   iterator.hasNext();) {
         DomainObject object = (DomainObject) iterator.next();
         Mapper mapper = MapperRegistry.getMapper(
↳object.getClass());
         mapper.update(object);
      }
   }
```

Неявная блокировка (Implicit Lock)

Дейвид Райс

Предоставляет инфраструктуре приложения или супертипу слоя право накладывать автономные блокировки



Любая схема блокирования будет приносить пользу только тогда, когда в ее реализации нет "проколов". Стоит разработчику забыть вставить какую-нибудь строку кода, относящуюся к применению блокировки, и вся схема блокирования окажется совершенно бесполезной. Применение блокировки записи вместо блокировки чтения может привести к получению устаревших данных, а неправильное использование номера версии — к нежелательной перезаписи изменений, внесенных кем-то другим. Общее правило гласит: если элемент может быть заблокирован *где-нибудь*, он должен быть заблокирован *везде*. Игнорирование отдельной бизнес-транзакцией стратегии блокирования, применяемой в приложении, способно привести к появлению несогласованных данных. Несвоевременное снятие блокировки, разумеется, не приведет к порче данных, однако сведет на нет производительность приложения. Поскольку механизмы управления параллельными заданиями в автономном режиме достаточно сложно тестировать, подобные ошибки могут остаться незамеченными для всех используемых пакетов тестирования.

Пожалуй, наиболее очевидное решение этой проблемы состоит в том, чтобы не дать разработчикам совершить перечисленные ошибки. Выполнением наиболее важных процедур блокирования должны заниматься не разработчики, а само приложение. Таким образом, явное блокирование следует заменить неявным. Поскольку большинство корпоративных приложений в той или иной мере используют сочетание инфраструктуры, **супертипов слоя (Layer Supertype, 491)** и систем автоматической генерации кода, это предоставляет широкие возможности по реализации **неявной блокировки**.

Принцип действия

Для реализации **неявной блокировки** необходимо вынести код, который *никак нельзя пропустить* при описании схемы блокирования, в инфраструктуру приложения. За неимением лучшего термина будем использовать понятие "инфраструктура" для обозначения совокупности **супертипов слоя**, классов инфраструктуры и всех других конструкций, обеспечивающих жизнедеятельность приложения (подобно тому как водопровод и системы отопления обеспечивают жизнедеятельность в зданиях и городах). Обеспечить корректное блокирование могут также средства автоматической генерации кода. Несомненно, принцип вынесения кода блокировки в инфраструктуру приложения отнюдь не является революционным открытием. Полагаю, о нем задумывались все те, кому пришлось написать хотя бы несколько повторяющихся фрагментов механизма блокирования. Тем не менее, все воплощения этой идеи, встречавшиеся мне на практике, оказывались довольно слабы, поэтому уделим ей еще немного внимания.

Прежде всего для обеспечения **неявной блокировки** необходимо составить список процедур, которые являются обязательными в рамках конкретной стратегии блокирования. При использовании **оптимистической автономной блокировки (Optimistic Offline Lock, 434)** этот список будет содержать такие операции, как сохранение номера версии для каждой строки базы данных, включение проверки номера версии в критерии SQL-операторов `UPDATE` и `DELETE` и увеличение номера версии при изменении соответствующего объекта. В свою очередь, при использовании **пессимистической автономной блокировки (Pessimistic Offline Lock, 445)** список необходимых операций будет включать в себя применение блокировки перед загрузкой каждого необходимого объекта (как правило, это касается монопольной блокировки чтения и части блокировки чтения/записи, применяющейся для считывания объекта) и высвобождение всех блокировок по окончании сеанса или бизнес-транзакции.

Вы, должно быть, обратили внимание, что говоря о **пессимистической автономной блокировке**, я не упомянул ни одного типа блокировки, применяемого исключительно для редактирования данных, а именно: монопольной блокировки записи и части блокировки чтения/записи, предназначенной только для выполнения записи. Да, эти блокировки обязательно применяются в том случае, когда бизнес-транзакции нужно отредактировать данные. Тем не менее неудачные попытки наложения таких блокировок неявным способом могут привести к ряду проблем. Во-первых, те условия, в которых можно применить неявные блокировки записи (например, регистрация измененного объекта в **единице работы (Unit of Work, 205)**), не гарантируют аварийного завершения транзакции в самом начале работы пользователя. Приложение не сможет само определить, когда именно нужно применить такие блокировки. Несвоевременное прерывание транзакции в том случае, если предоставление блокировки невозможно, противоречит концепции **пессимистической автономной блокировки**, согласно которой пользователь не должен переделывать свою работу.

Второй, и столь же важный, факт состоит в том, что блокировки записи значительно ограничивают возможность параллельной работы в системе. В этом случае отказ от **неявной блокировки** принуждает разработчика задуматься о влиянии блокировок записи на степень параллелизма системы, переводя этот вопрос из чисто технической сферы в область бизнес-требований. Тем не менее нам все-таки нужно гарантировать, что все необходимые блокировки записи будут применены перед внесением соответствующих изменений. Эту проверку можно поручить инфраструктуре приложения. Отсутствие

блокировки на момент фиксации изменений является ошибкой программирования и, как минимум, должно привести к отказу подтверждения изменений. Впрочем, я рекомендую пропустить этап подтверждения и сразу же сгенерировать исключение о нарушении параллелизма, поскольку в реальных системах подтверждения могут быть отключены.

Говоря о **неявной блокировке**, следует сделать несколько предостережений. Ее применение позволяет разработчикам не думать о большей части процедур блокирования, однако не освобождает от необходимости предусмотреть все возможные последствия. Например, если разработчики используют **неявную блокировку** в пессимистической схеме блокирования, предполагающей ожидание блокировок, они все еще должны позаботиться о предотвращении взаимоблокировок. Как только разработчики перестают задумываться о применении блокировок, их бизнес-транзакции могут начать вести себя совершенно неожиданным образом — вот в чем состоит главный недостаток **неявной блокировки**.

Для того чтобы механизм неявного блокирования действительно приносил пользу, вынесение кода блокирования в инфраструктуру приложения необходимо реализовать оптимальным образом. Примеры неявного управления блокировками в оптимистической схеме приведены в начале главы. Возможностей удачной реализации **неявной блокировки** слишком много, чтобы их можно было продемонстрировать в рамках одной главы.

Назначение

Неявную блокировку следует применять во всех приложениях (за исключением, пожалуй, самых простых, не имеющих инфраструктуры). Задумайтесь: риск забыть какую-нибудь блокировку слишком велик, чтобы им можно было пренебречь.

Пример: неявная пессимистическая автономная блокировка (Java)

Рассмотрим систему, использующую монопольную блокировку чтения. Архитектура этой системы включает в себя **модель предметной области (Domain Model, 140)**, а для взаимодействия между объектами домена и реляционной базой данных применяются **преобразователи данных (Data Mapper, 187)**. При использовании монопольной блокировки чтения инфраструктура приложения должна блокировать объект домена, прежде чем позволить бизнес-транзакции совершать какие-либо действия над данным объектом.

Все объекты домена, используемые бизнес-транзакцией, определяются с помощью метода `find()` соответствующего преобразователя. Данная схема справедлива во всех случаях, независимо от того, как выполняется поиск объекта — непосредственным вызовом метода `find()` или перемещением по графу объекта. Для реализации неявной блокировки можно применить декоратор [20], чтобы добавить к поведению преобразователя требующуюся функциональность блокирования. Для этого напишем преобразователь `LockingMapper`, который будет применять блокировку перед попыткой найти объект.

```
interface Mapper...

   public DomainObject find(Long id);
   public void insert(DomainObject obj);
   public void update(DomainObject obj);
   public void delete(DomainObject obj);
```

```
class LockingMapper implements Mapper...

   private Mapper impl;
   public LockingMapper(Mapper impl) {
      this.impl = impl;
   }
   public DomainObject find(Long id) {
      ExclusiveReadLockManager.INSTANCE.acquireLock(
↳id, AppSessionManager.getSession().getId());
      return impl.find(id);
   }
   public void insert(DomainObject obj) {
      impl.insert(obj);
   }
   public void update(DomainObject obj) {
      impl.update(obj);
   }
   public void delete(DomainObject obj) {
      impl.delete(obj);
   }
```

Поскольку в течение сеанса поиск одного и того же объекта может выполняться несколько раз, перед применением метода поиска диспетчер блокировки должен проверить, нет ли у сеанса существующей блокировки на данный объект. Если бы вместо монопольной блокировки чтения использовалась монопольная блокировка записи, пришлось бы снабдить преобразователь декоратором, выполняющим проверку на наличие существующей блокировки вместо фактического применения последней перед обновлением или удалением соответствующего объекта.

Огромным преимуществом декораторов является то, что объект, к которому они применяются, даже не подозревает об изменении своей функциональности. Ниже показано, как изменяется поведение преобразователей, содержащихся в **реестре (Registry, 495)**.

```
LockingMapperRegistry implements MappingRegistry...

   private Map mappers = new HashMap();
   public void registerMapper(Class cls, Mapper mapper) {
      mappers.put(cls, new LockingMapper(mapper));
   }
   public Mapper getMapper(Class cls) {
      return (Mapper) mappers.get(cls);
   }
```

Когда бизнес-транзакция обращается к преобразователю, предполагается, что будет вызван стандартный метод обновления. Фактический же ход событий изображен на рис. 16.5.

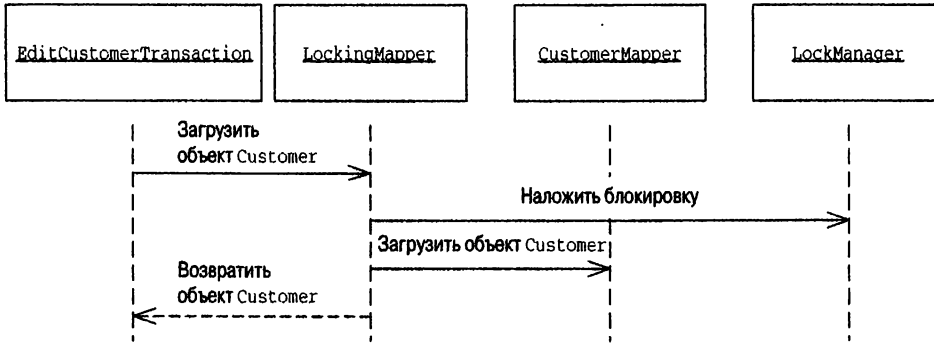


Рис. 16.5. Схема работы преобразователя LockingMapper

Глава 17

Типовые решения для хранения состояния сеанса

Сохранение состояния сеанса на стороне клиента (Client Session State)

Сохраняет состояние сеанса на стороне клиента

Принцип действия

Даже если дизайн корпоративной системы полностью ориентирован на использование серверных средств, часть сведений о сеансе (будь то его идентификатор или еще что-нибудь) приходится располагать на стороне клиента. В некоторых приложениях на сторону клиента помещают всю информацию о состоянии сеанса. В этом случае клиент передает серверу все сведения о сеансе вместе с каждым запросом, а сервер возвращает все сведения о сеансе вместе с каждым ответом. Подобная схема позволяет серверу полностью отказаться от сохранения состояний сеанса.

В большинстве случаев для перемещения данных между клиентом и сервером используется **объект переноса данных (Data Transfer Object, 419)**. Он может сериализовать свое содержимое для передачи по сети, тем самым позволяя перемещать весьма сложные структуры данных.

Клиент также нуждается в способе хранения данных. Если речь идет о толстом клиенте, хранение данных может осуществляться за счет собственных структур приложения, например полей его интерфейса (хотя я бы предпочел застрелиться, чем поступать таким образом). Гораздо более удачным решением является хранение состояний сеансов в совокупности невизуализированных объектов, например в модели домена или в самом **объекте переноса данных**. В любом случае, это не такая уж серьезная проблема.

С HTML-интерфейсами дело обстоит немного сложнее. Существует три основных способа сохранения состояния сеанса на стороне клиента: параметры адреса URL, скрытые поля и файлы cookie.

Использование параметров адреса URL хорошо подходит для работы с небольшим количеством данных. Вообще говоря, для отображения Web-страницы с результатами выполнения запроса все адреса URL принимают то или иное количество параметров

сеанса. Разумеется, объем сохраняемой информации существенно ограничен размерами адресов. Тем не менее данный метод прекрасно справляется с двумя-тремя параметрами, поэтому он весьма популярен для хранения на стороне клиента небольших значений наподобие идентификаторов сеансов. Некоторые платформы автоматически перезаписывают адреса URL, добавляя к ним идентификаторы сеансов. Изменение адреса Web-страницы может повлиять на работу закладок, поэтому данную схему хранения не рекомендуется использовать на коммерческих сайтах, связанных с обслуживанием потребителей.

Скрытое поле — это поле, значение которого передается обозревателю, но не отображается на Web-странице. Наличие скрытого поля задается дескриптором вида `<INPUT type = "hidden">`. Чтобы сохранить данные на стороне клиента, сервер сериализует состояние сеанса, помещает его в скрытое поле при отправке ответа и вновь считывает при получении следующего запроса. Как только что отмечалось, помещаемые в скрытое поле данные должны быть сериализованы. Обычно в качестве формата сериализации применяют XML, хотя, как известно, он слишком "многословен". Вместо этого данные можно сериализовать и в какой-нибудь другой текстовый формат. Не забывайте, однако, что значение скрытого поля скрыто только во время отображения; чтобы добраться к этому значению, достаточно просмотреть исходный код страницы.

Остерегайтесь сайтов, содержащих старые Web-страницы или страницы с фиксированными адресами. Переместившись на них, вы потеряете всю информацию о состоянии сеанса.

Последний, пожалуй наиболее спорный, метод хранения состояний сеанса — это использование файлов cookie, которые автоматически передаются от сервера к клиенту и наоборот. Как и при работе со скрытыми полями, помещаемые в файл cookie данные нужно сериализовать. Объем сохраняемой информации ограничивается размерами файлов — они не должны быть слишком большими. Кроме того, многим пользователям не нравится присутствие файлов cookie, поэтому их отключают, в результате чего сайт перестанет функционировать. Тем не менее в наше время все больше и больше сайтов основаны на использовании файлов cookie, поэтому подобные неприятности случаются нечасто. И конечно, наличие этих файлов не представляет никакой опасности для чисто "домашних" систем.

Не забывайте: файлы cookie безопасны не более чем другие способы хранения информации, а значит, несанкционированное использование данных возможно и здесь. Кроме того, файлы cookie работают только в пределах одного имени домена, поэтому, если ваш сайт разнесен по нескольким доменам, файлы cookie не смогут перемещаться между его частями.

Некоторые платформы автоматически определяют, разрешено ли на стороне клиента использование файлов cookie; если это не так, они применяют перезаписывание адресов URL. Данная схема позволяет легко сохранять на стороне клиента небольшие объемы сведений о сеансе.

Назначение

Типовое решение **сохранение состояния сеанса на стороне клиента** обладает массой преимуществ. В частности, оно поддерживает использование серверных объектов без состояний, обеспечивая максимальную степень кластеризации и устойчивости к отказам.

Разумеется, при отказе системы клиента все данные будут утеряны, однако в подобных ситуациях большинство клиентов ожидают именно такого исхода.

Количество аргументов против **сохранения состояния сеанса на стороне клиента** стремительно возрастает с увеличением объема сохраняемой информации. Все перечисленные схемы прекрасно подходят для хранения нескольких полей, однако при наличии больших объемов данных вопросы их размещения и временные расходы, вызванные передачей вместе с каждым запросом большого количества информации, становятся весьма критичны. Это особенно верно для HTTP-клиентов.

Применение данного типового решения связано и с проблемами безопасности. Сведения, отсылаемые клиенту, могут быть перехвачены и изменены. Единственный способ предотвратить подобные поползновения — воспользоваться шифрованием, однако шифрование и дешифровка каждого запроса может нанести большой урон производительности приложения. С другой стороны, все, что не было зашифровано, является потенциальной мишенью для любознательных глаз. Как правило, их хозяева обладают к тому же "пронырливыми" пальцами, поэтому не ожидайте, что сведения о состоянии сеанса вернутся в том же виде, в каком они были отосланы. Все данные, возвращаемые клиентом, должны быть подвергнуты тщательной проверке на правильность.

Без **сохранения состояния сеанса на стороне клиента** не обойтись при необходимости хранить идентификаторы сеансов. К счастью, идентификатор — это всего лишь число, поэтому проблем с его обработкой в перечисленных выше схемах быть не должно. Тем не менее попытки перехвата данных случаются и здесь, например если злоумышленник изменит идентификатор своего сеанса, чтобы перехватить сеанс другого пользователя. Для снижения подобного риска большинство платформ генерируют идентификаторы сеансов методом случайных чисел; если же такая возможность отсутствует, попробуйте воспользоваться хэшированием.

Сохранение состояния сеанса на стороне сервера (Server Session State)

Сохраняет сериализованное представление состояния сеанса на стороне сервера

Принцип действия

Самая простая форма данного типового решения предполагает размещение объекта сеанса в памяти сервера приложений. В этом случае для хранения объектов сеансов применяется коллекция, расположенная в оперативной памяти, а сами объекты индексированы по идентификатору сеанса. Все, что требуется от клиента, — передать идентификатор сеанса, после чего необходимый объект сеанса будет извлечен из коллекции и направлен на обработку запроса.

Подобная базовая схема основана на том предположении, что у сервера приложений достаточно памяти для выполнения таких задач. Кроме того, предполагается, что сервер приложений только один (т.е. кластеризация отсутствует) и что при внезапном отказе сервера приложений сеанс будет утерян, а вся проделанная работа, как говорится, "пойдет коту под хвост".

Некоторые корпоративные системы вполне довольствуются подобным набором предположений. Однако остальным он может не подойти. Существует несколько способов преодолеть названные ограничения, и эти способы подразумевают применение ходов, которые значительно усложняют данное типовое решение.

Первая проблема связана с использованием системных ресурсов, занятых объектами сеансов. Вообще говоря, чрезмерное использование ресурсов — самый большой недостаток **сохранения состояния сеанса на стороне сервера**. Очевидное решение данной проблемы состоит в том, чтобы не сохранять объекты сеансов в памяти, а сериализовать состояние сеанса в объект **мементо (Memento)** [20]. В связи с этим возникает два вопроса: в каком формате сохранять состояние сеанса на стороне сервера и где именно его сохранять?

Формат сериализации состояний сеанса должен быть как можно более простым, поскольку основной особенностью **сохранения состояния сеанса на стороне сервера** является именно простота программирования. Некоторые платформы снабжены простым механизмом, позволяющим сериализовать граф объектов в двоичный формат. Вместо этого объекты сеансов можно сериализовать в какой-нибудь текстовый формат, например в популярный сегодня XML.

Как правило, с двоичным форматом сериализации проще работать, потому что он практически не требует программирования, в то время как для обработки текстового формата необходимо написать хотя бы несколько строк кода. Кроме того, двоичные объекты занимают гораздо меньше места на диске; хотя в данном случае объем дискового пространства не является чем-то критичным, большие сериализованные объекты дольше активизируются в оперативной памяти.

Несмотря на все это, сериализация объектов в двоичный формат обладает двумя существенными недостатками. Во-первых, двоичный код нечитабелен, что может оказаться неудобным, если разработчик захочет просмотреть сериализованный объект. Во-вторых, применение двоичного формата чревато проблемами с использованием различных версий классов. Если после сериализации объекта к соответствующему классу будет добавлено новое поле, восстановить сериализованный объект будет невозможно. Разумеется, наличие разных версий программного обеспечения сервера вряд ли затронет многие объекты сеансов, если только речь не идет о круглосуточно функционирующем сервере, установленном на нескольких машинах, одни из которых содержат обновленное программное обеспечение, а другие — нет.

Данная проблема подводит нас ко второму вопросу: где именно реализовать **сохранение состояния сеанса на стороне сервера**? Наиболее очевидным местом для хранения объектов сеансов является сам сервер приложений. В этом случае объекты сеансов могут располагаться непосредственно в файловой системе либо в локальной базе данных. Это простое решение, однако оно не всегда обеспечивает возможность кластеризации и высокую устойчивость к отказам системы. Для поддержки всех этих требований объекты сеанса должны находиться где-нибудь в общедоступном месте, например на совместно используемом сервере. Это обеспечит возможность кластеризации и высокую отказоустойчивость за счет большего количества времени, необходимого для активизации сервера (хотя и эти расходы можно значительно уменьшить с помощью кэширования).

Как ни парадоксально, приведенные аргументы подводят нас к концепции сохранения сериализованного состояния сеанса в базе данных с использованием таблицы сеансов, индексированной по их идентификаторам. В этом случае для **сохранения состояния**

сеанса на стороне сервера в таблице базы данных потребуется применить **крупный сериализованный объект (Serialized LOB, 292)**. Различные базы данных по-разному обрабатывают крупные объекты, поэтому вопрос производительности во многом зависит от используемой базы данных.

На этом этапе обсуждения мы подошли к определению границы между **сохранением состояния сеанса на стороне сервера** и **сохранением состояния сеанса в базе данных (Database Session State, 479)**. Эта граница весьма условна. На мой взгляд, первое типовое решение переходит во второе тогда, когда данные вместо сериализации преобразуются в стандартный табличный формат.

Если сериализованное состояние сеанса хранится в базе данных, необходимо позаботиться об обработке утраченных или незаконченных сеансов. Это наиболее важно для Web-приложений, занимающихся обслуживанием покупателей. Одним из решений данной проблемы может стать применение демона, который будет обнаруживать и удалять сеансы, время жизни которых превышает установленный срок. К сожалению, этот подход может привести к большому количеству параллельных обращений к таблице сеансов. Каи Ю (Kai Yu) рассказал мне о другом подходе, который он успешно применяет на протяжении нескольких лет: разбить таблицу сеансов на 12 сегментов базы данных и каждые два часа выполнять циклическое чередование сегментов, удаляя содержимое самого старого сегмента и направляя к нему новые операции вставки сеансов. Данный подход подразумевает, что сеанс может существовать не более 24 часов, однако в действительности необходимость применения таких длительных сеансов возникает крайне редко.

Реализация всех перечисленных идей требует немалых усилий. К счастью, на рынке программного обеспечения появляется все больше и больше серверов приложений, автоматически поддерживающих данные возможности. Таким образом, вопросы реализации последних должны заботить не только и не столько разработчиков, сколько маленьких толстеньких производителей серверов приложений и прочих мелочей.

Особенности Java-реализации

Существует два распространенных способа реализации **сохранения состояния сеанса на стороне сервера**: использование HTTP-сеансов и использование компонентов сеанса с состоянием. Первый способ более прост и заключается в сохранении состояний сеанса на Web-сервере. В большинстве случаев это "привязывает" данные к серверу и делает их весьма чувствительными к отказам последнего. Некоторые производители реализуют возможность совместного использования HTTP-сеансов, что позволяет хранить параметры HTTP-сеанса в базе данных, доступной всем серверам приложений. (Разумеется, это можно сделать и вручную.)

Вместо этого для хранения сведений о сеансе можно воспользоваться компонентами сеанса с состоянием. Данный подход требует применения EJB-сервера. EJB-контейнер обрабатывает все действия, связанные с сохранением и пассивацией состояний сеанса, чем значительно облегчает программирование. К сожалению, данная схема хранения не исключает привязки к серверу. Впрочем, некоторые серверы приложений содержат средства, позволяющие избежать этой неприятной возможности. Один из них — сервер WebSphere от компании IBM — способен сериализовать компонент сеанса с состоянием в объект BLOB для размещения в базе данных DB2, что позволяет осуществлять доступ к состоянию сеанса нескольким серверам приложений.

Многие считают, что, поскольку компоненты сеанса без состояния лучше влияют на производительность приложения, их нужно *всегда* использовать вместо компонентов сеанса с состоянием. По правде говоря, это полная ерунда. Прежде чем отказываться от компонентов сеанса с состоянием, рекомендую протестировать свою среду разработки в условиях ожидаемой нагрузки и проверить, существенно ли различается производительность приложений при использовании компонентов сеанса с состоянием и без состояния. Компания ThoughtWorks провела серию тестирований с использованием нагрузки в несколько сотен параллельно работающих пользователей и не обнаружила какого-либо падения производительности, вызванного применением компонентов сеанса с состоянием. Таким образом, если последние не оказывают негативного влияния на производительность системы при ожидаемой степени нагрузки и намного легче поддаются обработке, использовать следует именно их. Остерегаться применения компонентов сеанса с состоянием можно по другим причинам (например, они могут затруднить обеспечение отказоустойчивости для некоторых серверов приложений), однако различие в производительности становится заметным только при поистине гигантских нагрузках.

И наконец, еще одной возможной альтернативой является использование компонентов сущностей. Вообще говоря, я не люблю работать с компонентами сущностей, однако в данном случае их можно использовать, чтобы хранить **крупные сериализованные объекты** с состояниями сеансов. Это довольно просто и, что приятно, позволяет избежать многих проблем, так часто связанных с компонентами сущностей.

Особенности .NET-реализации

Сохранение состояния сеанса на стороне сервера легко реализовать, используя встроенные средства .NET, предназначенные для работы с состояниями сеанса. По умолчанию .NET хранит сведения о сеансе в самом процессе сервера. Вместо этого для хранения данных можно использовать службу состояний, расположенную на локальном компьютере или любом другом узле сети. Применение службы состояний позволяет сохранить параметры сеанса даже при перезапуске Web-сервера. Переключение между названными способами хранения выполняется в файле настроек сервера, поэтому изменять само приложение вам не придется.

Назначение

Основным преимуществом **сохранения состояния сеанса на стороне сервера** является его простота. В большинстве случаев для реализации данного типового решения вообще не нужно писать отдельного кода. Разумеется, это зависит от того, устраивает ли вас хранение состояний сеанса в оперативной памяти, и, если это не так, от набора возможностей используемого сервера приложений.

Даже при отсутствии всех этих условий реализовать **сохранение состояния сеанса на стороне сервера** совсем несложно. Сериализация данных в объект BLOB и размещение его в базе данных требуют намного меньше усилий, чем преобразование тех же данных в табличный формат.

Самые большие объемы работ связаны с обеспечением функционирования сеансов, в частности если требуется организовать собственную поддержку кластеризации и устойчивости к отказам. В этом случае хранение состояний сеанса на стороне сервера может оказаться не самым удачным решением, особенно если параметров сеанса довольно мало или же если их проще преобразовать в табличный формат.

Сохранение состояния сеанса в базе данных (Database Session State)

Сохраняет состояние сеанса как обычное содержимое базы данных

Принцип действия

Когда клиент направляет серверу запрос, первое, что делает серверный объект, — обращается к базе данных и извлекает из нее данные, необходимые для удовлетворения запроса. Затем он выполняет всю необходимую работу и записывает обновленные данные обратно в базу данных.

Чтобы извлечь информацию из базы данных, серверному объекту нужны некоторые сведения о сеансе. Это можно осуществить только тогда, когда на стороне клиента будет храниться хоть какая-то информация о сеансе, как минимум его идентификатор. Впрочем, в большинстве случаев эта информация представляет собой не более чем набор ключей, необходимых для извлечения соответствующих данных.

Данные, используемые в ходе выполнения запроса, как правило, представляют собой комбинацию промежуточных данных сеанса, имеющих смысл только в контексте текущего взаимодействия с базой данных, и постоянных данных, имеющих смысл во всех взаимодействиях.

Ключевым моментом в этой схеме является то, что промежуточные данные сеанса рассматриваются как локальные по отношению к текущему сеансу и не должны применяться остальными частями системы до тех пор, пока результаты выполнения сеанса не будут зафиксированы в базе данных. Таким образом, если вы работаете с объектом заказа в контексте сеанса и хотите сохранить промежуточное состояние этого объекта в базе данных, вам нужно обрабатывать его отдельно от заказа, сохраненного в базе данных в конце этого же сеанса. В противном случае в базе данных появится масса "незаконченных" заказов, что особенно неприятно в таких ситуациях, как проверка на наличие определенных книг и подсчет ежедневной выручки.

Итак, как же отделить данные сеанса от основного содержимого базы данных? Пожалуй, самое очевидное решение — добавить к каждой строке базы данных, которая может содержать промежуточные данные сеанса, специальное поле. В качестве последнего можно использовать логическое поле с именем наподобие `isPending`. Однако гораздо лучше применить поле, содержащее идентификатор сеанса, что позволит легко отобрать все данные, принадлежащие конкретному сеансу. В этом случае всем запросам, извлекающим постоянные данные, понадобится отбросить промежуточные данные с помощью таких выражений, как `sessionID is not NULL`, или воспользоваться необходимыми фильтрами.

Применять поле с идентификатором сеанса довольно неудобно, потому что всем приложениям, имеющим дело с записями базы данных, понадобится знать о присутствии

этого поля, чтобы по ошибке не извлечь промежуточные данные сеанса. Иногда это неудобство можно устранить с помощью представлений и фильтров, однако применение представлений связано со своими расходами.

Возможной альтернативой является создание отдельного набора таблиц с промежуточными данными. Таким образом, если в базе данных есть таблицы для хранения заказов и пунктов заказов, понадобится добавить еще две таблицы для хранения промежуточных состояний заказов и промежуточных состояний пунктов заказов. Промежуточные данные будут сохраняться в промежуточных таблицах, а их окончательные варианты — в обычных, "настоящих" таблицах. Это позволит избежать многих неудобств, связанных с разделением данных, однако потребует реализации в слое отображения дополнительной логики выбора тех или иных таблиц, что, несомненно, усложнит структуру системы.

Зачастую постоянные данные таблиц подчиняются определенным правилам целостности, которые не распространяются на промежуточные данные. В этом случае вы можете отключить правила целостности в применении к промежуточным таблицам и активизировать их, если они все-таки понадобятся. К промежуточным данным не применяется и проверка на логическую правильность. Разумеется, на определенных этапах сеанса может понадобиться провести ту или иную проверку на правильность, однако это обычно определяется в логике серверного объекта.

Промежуточные таблицы должны представлять собой точные копии настоящих таблиц. В этом случае логика отображения будет самой простой. Используйте в промежуточных таблицах те же имена полей, что и в настоящих таблицах, а затем снабдите их дополнительным полем с идентификаторами сеанса, чтобы легко отбирать данные, принадлежащие определенному сеансу.

При сохранении промежуточных данных необходимо реализовать механизм, который будет удалять записи о брошенных или прерванных сеансах. Используя идентификатор сеанса, вы можете найти и уничтожить все ненужные данные. Если же пользователь может покинуть сеанс без уведомления сервера, понадобится организовать некоторую обработку времени ожидания. Для этого можно воспользоваться демоном, который будет запускаться каждые несколько минут и выполнять проверку наличия устаревших данных. Подобная схема обработки требует наличия в базе данных специальной таблицы, которая будет фиксировать время последнего взаимодействия с указанным сеансом.

Реализовать откат обновлений гораздо сложнее. Как выполнить откат сеанса, если в течение этого сеанса была обновлена существующая запись о заказе? Одно из возможных решений — не допускать отмену подобных сеансов. Все обновления существующих данных должны фиксироваться в таблице только в конце выполнения запроса. Эта схема очень проста и вполне соответствует видению процесса обновления самими пользователями. Все остальные альтернативы менее удобны, независимо от того, что вы используете — промежуточные поля или промежуточные таблицы. Для реализации описанной схемы при наличии промежуточных таблиц все обновляемые данные можно скопировать в промежуточные таблицы, выполнить в них необходимые изменения и перенести данные обратно в таблицы с постоянными записями по окончании текущего сеанса. При использовании промежуточных полей подобные действия возможны только тогда, когда идентификатор сеанса является частью ключа записи. В этом случае в одной и той же таблице могут одновременно находиться значения старого и нового идентификаторов, что способно крайне затруднить работу с таблицей.

Если промежуточные таблицы будут считываться только объектами, обрабатывающими сеанс, необходимость представления состояния сеанса в табличном формате крайне мала. В этом случае имеет смысл воспользоваться **крупным сериализованным объектом (Serialized LOB, 292)**, в результате чего **сохранение состояния сеанса в базе данных** превращается в **сохранение состояния сеанса на стороне сервера (Server Session State, 475)**.

При желании вы можете избежать всех этих трудностей, просто отказавшись от использования промежуточных данных. Другими словами, приложение можно спроектировать таким образом, чтобы все его данные рассматривались как постоянные. Разумеется, это далеко не всегда возможно и зачастую способно привести систему в состояние полной неразберихи (а потому наталкивает на мысль о необходимости явной обработки промежуточных данных). Тем не менее, если подобная возможность у вас есть, она значительно облегчит реализацию **сохранения состояния сеанса в базе данных**.

Назначение

Сохранение состояния сеанса в базе данных является одним из возможных вариантов обработки состояний сеанса наряду с типовыми решениями **сохранение состояния сеанса на стороне сервера** и **сохранение состояния сеанса на стороне клиента**. Каждое из них имеет свои преимущества и недостатки.

Одним из наиболее спорных аспектов **сохранения состояния сеанса в базе данных** является производительность. Использование серверных объектов без состояния разрешает применение пула объектов и облегчает выполнение кластеризации, что, разумеется, оказывает благоприятное влияние на производительность приложения. С другой стороны, необходимость извлечения и записи данных при выполнении каждого запроса влечет за собой определенные временные затраты. Для снижения этих затрат извлекаемые данные можно кэшировать, однако расходы на запись сократить нельзя.

Не менее важным аспектом является трудоемкость работ. Большая часть усилий программистов затрачивается на обработку состояний сеанса. Таким образом, если у вас нет состояний сеанса и промежуточные данные допускается сохранять наряду с постоянными, предпочтение следует отдать именно **сохранению состояния сеанса в базе данных**, поскольку оно не требует дополнительных затрат на кодирование и не приводит к падению производительности приложения (если извлекаемые объекты кэшируются).

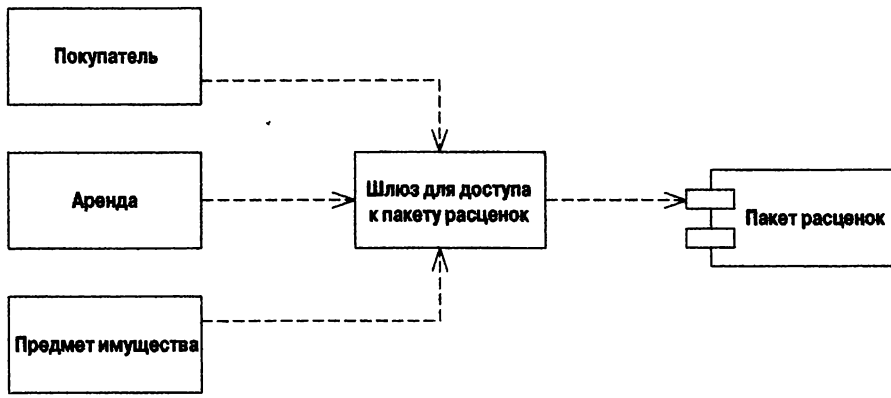
Выбирая между **сохранением состояния сеанса в базе данных** и **сохранением состояния сеанса на стороне сервера**, обратите внимание на то, как используемый сервер приложений поддерживает кластеризацию и отказоустойчивость. В большинстве стандартных решений реализовать кластеризацию и устойчивость к отказам легче, если состояние сеанса хранится в базе данных.

Глава 18

Базовые типовые решения

Шлюз (Gateway)

Объект, инкапсулирующий доступ к внешней системе или источнику данных



Программное обеспечение редко функционирует само по себе. Даже самые типичные объектно-ориентированные системы должны взаимодействовать со структурами, не являющимися объектами, например с таблицами реляционных баз данных, транзакциями CICS и документами XML.

В большинстве случаев для доступа к внешним источникам применяются интерфейсы API. К сожалению, они довольно сложны, так как учитывают характер источника. Каждый, кто работает с источником, должен понимать и его интерфейс — JDBC или SQL для реляционных баз данных, W3C или JDOM для XML и т.п. Это не только затрудняет понимание программного обеспечения, но и значительно усложняет потенциальную замену источника данных с реляционной СУБД документом XML или наоборот.

Решение данной проблемы настолько просто и очевидно, что его вряд ли стоит озвучивать. Весь специализированный код API помещается в класс-шлюз, интерфейс которого не отличается от интерфейса обычного объекта. После этого, чтобы получить доступ

к удаленному источнику, объекты приложения будут обращаться к **шлюзу**, который преобразует простые вызовы методов в вызовы специализированного API.

Принцип действия

Концепция типового решения **шлюз** очень проста. Возьмем внешний источник данных. Какие действия должно совершать приложение по отношению к этому источнику? Создайте простой API, наполните его всеми необходимыми методами и воспользуйтесь **шлюзом**, чтобы преобразовать вызовы его методов в обращения к внешнему источнику.

Одно из главных назначений **шлюза** — обеспечить основу для реализации **фиктивных служб (Service Stub, 519)**. Иногда для более удобного применения **фиктивной службы** в структуру **шлюза** приходится вносить дополнительные изменения. Не бойтесь это делать: удачное размещение **фиктивных служб** способно значительно облегчить тестирование, а значит, и написание системы.

Не стоит излишне усложнять **шлюз** — оставьте его настолько простым, насколько это возможно. Реализуя данное типовое решение, сосредоточьте усилия на адаптации внешней службы к особенностям приложения и обеспечении хорошей основы для поддержки фиктивных служб. **Шлюз** должен обрабатывать указанные задания и одновременно обладать как можно меньшей функциональностью. Вся более сложная логика должна содержаться в клиентах **шлюза**.

Зачастую для создания **шлюзов** применяют системы автоматической генерации кода. Описав структуру внешнего источника, вы можете сгенерировать для него соответствующий **шлюз**. Чтобы сконструировать оболочку для таблицы реляционной базы данных, можно воспользоваться реляционными метаданными, а чтобы сгенерировать **шлюз** для доступа к документу XML — схемами XML или шаблонами DTD. Разумеется, полученные **шлюзы** не будут отличаться особым изяществом, однако выполнять свои функции будут безотказно. Все остальные решения обычно оказываются более сложными.

Иногда реализацию **шлюза** рассматривают в терминах двух объектов — прикладной части (back end) и интерфейсной части (front end). Прикладная часть **шлюза** представляет собой минимальную оболочку функций доступа к внешнему источнику и вообще не упрощает использование API. В свою очередь, интерфейсная часть преобразует неудобный API в класс, оптимально приспособленный для применения конкретным приложением. Данную стратегию хорошо применять в случаях, когда построение оболочки для доступа к внешней службе и приспособление этой оболочки к нуждам приложения довольно сложны, а потому требуют обработки каждого из этих заданий в отдельных классах. Напротив, если построение оболочки для доступа к внешней службе не отличается особой сложностью, все перечисленные действия может выполнять и один класс.

Назначение

Типовое решение **шлюз** рекомендуется применять во всех случаях, когда интерфейс доступа к внешнему источнику слишком неудобен. Наличие **шлюза** позволяет сконцентрировать все неудобные обращения в одном месте вместо того, чтобы распространять их по всей системе. Применение **шлюза** не несет побочных эффектов, а код приложения становится более читабельным и понятным.

Как уже отмечалось, **шлюз** значительно облегчает тестирование, поскольку представляет собой потенциальную точку внедрения **фиктивных служб**. Даже если с интерфейсом

внешней системы все в порядке, использование **шлюза** позволяет сделать первый шаг в направлении реализации **фиктивной службы**.

Не менее важным преимуществом **шлюза** является возможность легко переключаться между источниками данных. Для перехода к другому источнику достаточно просто изменить класс **шлюза** — оставшейся части системы это не коснется. Таким образом, **шлюз** представляет собой простое и мощное средство инкапсуляции изменений. Иногда потребность в наличии подобной степени гибкости, а следовательно, и в реализации **шлюза** кажется спорной. Тем не менее, даже если вы не собираетесь менять источник данных в обозримом будущем, вы несомненно выиграете от простоты написания и тестирования кода, которую обеспечивает данное типовое решение.

В качестве альтернативного варианта изолирования приложений от внешних источников может применяться **преобразователь (Mapper, 489)**. Однако он имеет более сложную структуру, нежели **шлюз**, а потому я предпочитаю использовать именно последний.

Некоторое время меня одолевали сомнения относительно того, стоит ли выделять данную схему в самостоятельное типовое решение, противоположное существующим типовым решениям **интерфейс (Facade)**, **адаптер (Adapter)** и **медиатор (Mediator)** [20]. В конце концов я решил описать **шлюз** как отдельное типовое решение, поскольку, на мой взгляд, оно обладает рядом существенных отличий.

- Типовое решение **интерфейс** также упрощает работу с интерфейсом API, однако оно создается самим разработчиком внешней службы и предназначено для общего употребления. В свою очередь, **шлюз** разрабатывается клиентом для использования конкретным приложением. Кроме того, интерфейс доступа, предоставляемый объектом **интерфейса**, всегда отличается от интерфейса стоящего за ним объекта, в то время как интерфейс **шлюза** может представлять собой точную копию инкапсулируемого интерфейса (для тестирования или замены источника данных суррогатным объектом).
- Типовое решение **адаптер** изменяет интерфейс некоторого объекта для достижения соответствия с интерфейсом другого объекта. В отличие от него, интерфейс **шлюза** не нужно подстраивать под какой-либо существующий интерфейс. Вообще говоря, типовое решение **адаптер** можно применить, чтобы отобразить реализацию некоторого объекта на интерфейс **шлюза**. В этом случае объект **адаптера** будет представлять собой составную часть реализации **шлюза**.
- Типовое решение **медиатор** разделяет множество объектов так, чтобы они не знали о существовании друг друга, но были осведомлены о наличии объекта **медиатора**. **Шлюз** разделяет только два объекта, причем источник данных не знает о существовании **шлюза**.

Пример: создание шлюза к службе отправки сообщений (Java)

Обсуждая концепцию **шлюза** с моим коллегой Майком Реттигом (Mike Rettig), я узнал, как он использовал данное типовое решение для обработки доступа к внешним интерфейсам в приложениях EAI (Enterprise Application Integration — интеграция корпоративных систем). Мы решили, что опыт Майка может послужить прекрасной основой для рассмотрения примера использования **шлюза**.

Как всегда, наш пример отличается невероятно простым положением дел. Мы создадим шлюз к интерфейсу, который отсылает сообщение посредством службы отправки сообщений. Сам интерфейс представляет собой единственный метод:

```
int send(String messageType, Object [] args));
```

Первый аргумент данного метода — это строка, указывающая на тип сообщения, а второй — перечень аргументов самого сообщения. Система отправки сообщений позволяет отсылать сообщения любого типа, поэтому ей нужен подобный универсальный интерфейс. При настройке системы необходимо указать тип отсылаемых сообщений, а также количество и типы аргументов этих сообщений. Таким образом, можно настроить систему для отправки подтверждающих сообщений, указав в качестве типа сообщения слово `"CNFRM"`, а также задав аргумент `orderID` ("идентификатор заказа") типа `String`, аргумент `amount` ("количество товара") типа `Integer` и аргумент `symbol` ("код товара") типа `String`. Система отправки сообщений проверяет типы передаваемых аргументов и генерирует ошибку при попытке отослать сообщение неправильного типа или сообщение правильного типа с неправильными типами аргументов.

Описанная схема обеспечивает высокую (и, кстати, весьма необходимую) степень гибкости, однако универсальный интерфейс крайне неудобен в использовании. Из определения универсального интерфейса не ясно, какие типы сообщений допускается отсылать, а также сколько и каких аргументов должно быть в каждом сообщении. Вместо этого нам нужен интерфейс с методами наподобие следующего:

```
public void sendConfirmation(String orderID, int amount,
↳String symbol);
```

В этом случае для отправки подтверждающего сообщения объектом домена необходимо поступить так, как показано ниже.

```
class Order...

   public void confirm() {
      if (isValid()) Environment.getMessageGateway().
↳sendConfirmation(id, amount, symbol);
}
```

В приведенном фрагменте кода имя метода указывает на тип отсылаемого сообщения, а для всех аргументов метода `sendConfirmation` явно заданы имена и типы. Разумеется, данный метод вызывать намного легче, чем метод универсального интерфейса. В этом и состоит смысл применения **шлюза** — обеспечить более удобный интерфейс для доступа к внешней системе. Правда, при каждом добавлении или изменении типа отсылаемых сообщений понадобится внести изменения и в класс шлюза, однако при отсутствии последнего изменения коснулись бы вызывающего кода, поэтому общая сумма изменений остается прежней. В этом плане применение **шлюза** более предпочтительно, так как позволяет определить клиентов и перехватить потенциальные ошибки еще на этапе компиляции.

Существует и другая проблема. При обнаружении ошибки универсальный интерфейс возвращает ее код. Нуль означает отсутствие ошибок, а любое число, отличное от нуля,

указывает на определенный тип ошибки (при этом разным ошибкам соответствуют разные числа). Данная схема уведомления об ошибках вполне естественна для языка C, однако в Java все обстоит иначе. Здесь для обработки ошибок применяются исключения, поэтому методы **шлюза** должны генерировать исключения, а не возвращать коды ошибок.

Вместо того чтобы рассматривать все возможные типы ошибок, сконцентрируем внимание только на двух из них: отправка сообщения неизвестного типа, а также отправка сообщения, один из аргументов которого равен NULL. Возвращаемые коды ошибок определены в интерфейсе системы отправки сообщений.

```
public static final int NULL_PARAMETER = -1;
public static final int UNKNOWN_MESSAGE_TYPE = -2;
public static final int SUCCESS = 0;
```

Природа первой и второй ошибок неодинакова. Ошибка, связанная с отправкой сообщения неизвестного типа, указывает на неполадки в классе шлюза; клиент не может привести к появлению подобной ошибки, поскольку вызывает только явные методы с заранее заданными типами сообщений. Тем не менее клиент может передать в качестве аргумента значение NULL и привести к появлению второй ошибки, а именно NULL_PARAMETER. Данная ошибка не нуждается в собственном исключении, так как является ошибкой программирования, — ситуации, для которых не пишут специальных обработчиков. Вообще говоря, следить за появлением значений NULL мог бы и сам **шлюз**, но повторять действия системы отправки сообщений не имеет смысла.

Исходя из этих доводов, реализуемый нами **шлюз** должен преобразовывать вызовы методов явного интерфейса в таковые универсального интерфейса, а коды ошибок — в соответствующие исключения.

```
class MessageGateway...

   protected static final String CONFIRM = "CNFRM";
   private MessageSender sender;
   public void sendConfirmation(String orderID, int amount,
↳String symbol) {
      Object [] args = new Object[]{orderID, new
↳Integer(amount), symbol};
      send(CONFIRM, args);
   }
   private void send(String msg, Object[] args) {
      int returnCode = doSend(msg, args);
      if (returnCode == MessageSender.NULL_PARAMETER)
         throw new NullPointerException("Null Parameter passed
↳for msg type:" + msg);
      if (returnCode != MessageSender.SUCCESS)
         throw new IllegalStateException(
↳"Unexpected error from messaging system #:" + returnCode);
   }
   protected int doSend(String msg, Object[] args) {
      Assert.notNull(sender);
      return sender.send(msg, args);
   }
```

Пока назначение метода doSend весьма туманно, не так ли? Между тем не будем забывать еще об одной не менее важной функции **шлюза** — облегчении тестирования. При наличии шлюза тестирование объектов можно проводить без обращения к реальной службе отправки сообщений. Для этого нужно создать **фиктивную службу**. В данном примере мы определим класс MessageGatewayStub, который наследует класс MessageGateway и переопределяет его метод doSend.

```
class MessageGatewayStub...

   protected int doSend(String messageType, Object[] args) {
      int returnCode = isMessageValid(messageType, args);
      if (returnCode == MessageSender.SUCCESS) {
         messagesSent++;
      }
      return returnCode;
   }
   private int isMessageValid(String messageType,
↳Object[] args) {
      if (shouldFailAllMessages) return -999;
      if (!legalMessageTypes().contains(messageType))
         return MessageSender.UNKNOWN_MESSAGE_TYPE;
      for (int i = 0; i < args.length; i++) {
         Object arg = args [i];
         if (arg == null) {
            return MessageSender.NULL_PARAMETER;
         }
      }
      return MessageSender.SUCCESS;
   }
   public static List legalMessageTypes() {
      List result = new ArrayList();
      result.add(CONFIRM);
      return result;
   }
   private boolean shouldFailAllMessages = false;
   public void failAllMessages() {
      shouldFailAllMessages = true;
   }
   public int getNumberOfMessagesSent() {
      return messagesSent;
   }
```

Подсчет количества отправленных сообщений нужен для того, чтобы проверить, правильно ли работает шлюз. Для выполнения проверки можно воспользоваться приведенными ниже тестами.

```
class GatewayTester...

   public void testSendNullArg() {
      try {
         gate().sendConfirmation(null, 5, "US");
         fail("Didn't detect null argument");
      } catch (NullPointerException expected) {
```

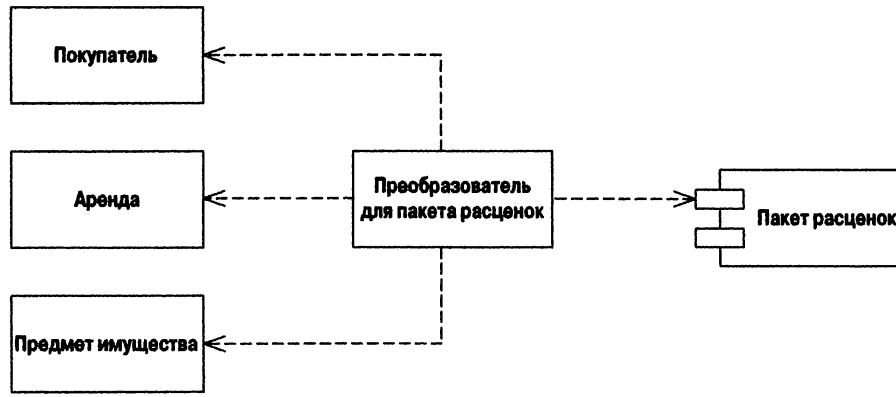
```
        }
        assertEquals(0, gate().getNumberOfMessagesSent());
    }
private MessageGatewayStub gate() {
    return (MessageGatewayStub) Environment.getMessageGateway();
    }
    protected void setUp() throws Exception {
        Environment.testInit();
    }
```

Как правило, **шлюз** располагают в таком месте, чтобы другим объектам было легко его найти. В этом примере я воспользовался статическим интерфейсом окружения. Переключение между реальной и фиктивной службами можно осуществить во время настройки системы посредством типового решения **дополнительный модуль (Plugin, 514)**. Кроме того, для активизации **фиктивной службы** можно воспользоваться методом `setUp` класса `GatewayTester`, выполняющим инициализацию окружения.

В данном примере для замены реальной службы отправки сообщений я воспользовался классом, производным от класса шлюза. В качестве возможной альтернативы можно рассмотреть создание класса, производного от самого класса службы, или же новую реализацию последней. При выполнении тестирования шлюз подсоединяется к **фиктивной службе** отправки сообщений; данная схема срабатывает тогда, когда реализовать службу заново не слишком сложно. Следует также отметить, что вместо замены службы можно осуществить замену шлюза. Более того, некоторые разработчики умудряются заменять и шлюз и службу, используя фиктивный шлюз для тестирования клиентов шлюза, а фиктивную службу — для тестирования самого шлюза.

Преобразователь (Mapper)

Объект, устанавливающий взаимодействие между двумя независимыми объектами



Иногда разработчику необходимо установить взаимодействие между двумя подсистемами, которые не должны знать о существовании друг друга. Чаще всего это происходит, если подсистемы не могут быть изменены или же их не следует связывать зависимостями (напрямую или даже опосредованно через некоторый отдельный элемент).

Принцип действия

Типовое решение **преобразователь** представляет собой некий "изоляционный" слой, проложенный между двумя подсистемами. Он управляет взаимодействием подсистем, причем ни одна из них об этом даже не догадывается.

Зачастую преобразователь перемещает данные из одного слоя в другой. После активизации преобразователя понять принцип его работы совсем несложно. Самый непростой аспект использования преобразователя — это его запуск, поскольку преобразователь не может быть напрямую вызван одной из подсистем, которые он отображает друг на друга. Иногда управление отображением, а значит, и вызов преобразователя возлагают на некоторую третью подсистему. Вместо этого преобразователь можно реализовать в виде типового решения **обозреватель (Observer)** [20], выполняющего роль "наблюдателя" за одной из обслуживаемых систем. В этом случае преобразователь активизируется при перехвате событий, сгенерированных этой системой.

Принцип работы преобразователя зависит от природы отображаемых слоев. Наиболее распространенной разновидностью преобразователя является рассмотренный нами ранее **преобразователь данных (Data Mapper, 187)**, примеры использования которого можно найти в главе 10 этой книги.

Назначение

Основное назначение **преобразователя** состоит в отделении друг от друга различных частей программной системы. Похожие функции выполняет и **шлюз (Gateway, 483)**. Последний применяется гораздо чаще, чем **преобразователь**, поскольку он намного проще и в написании, и в последующем использовании.

Таким образом, **преобразователь** необходимо использовать только тогда, когда ни одна из отображаемых систем не должна зависеть от взаимодействия с другой системой. Это действительно важно только в том случае, когда структура взаимодействия особенно сложна и практически не связана с основным назначением каждой системы. Поэтому в корпоративных приложениях главной областью применения **преобразователя** является обслуживание взаимодействий с базой данных. Соответствующая разновидность преобразователя получила название **преобразователя данных**.

Концепция **преобразователя** имеет немало общего с типовым решением **медиатор (Mediator)** [20] в том плане, что оба они применяются для разделения независимых объектов. Тем не менее между данными типовыми решениями есть принципиальное различие. Объекты, использующие **медиатор**, знают о его наличии, даже если им неизвестно о существовании друг друга. В свою очередь, объекты, разделенные **преобразователем**, не знают о наличии последнего.

Супертип слоя (Layer Supertype)

Тип, выполняющий роль суперкласса для всех классов своего слоя

Довольно часто одни и те же методы дублируются во всех объектах слоя. Чтобы избежать повторений, все общее поведение можно вынести в **супертип слоя**.

Принцип действия

Концепция **супертипа слоя**, а следовательно, и само типовое решение крайне просты. Все, что от вас требуется, — это создать суперкласс для всех объектов слоя (например, класс `DomainObject`, являющийся суперклассом для всех объектов домена в **модели предметной области (Domain Model, 140)**). После этого в созданный суперкласс может быть вынесено все общее поведение наподобие сохранения и обработки **полей идентификации (Identity Field, 237)**. Точно так же все **преобразователи данных (Data Mapper, 187)**, образующие слой отображения, могут иметь общий суперкласс, работающий с суперклассом объектов домена.

Если в рассматриваемом слое приложения находятся объекты нескольких различных типов, может понадобиться создать несколько **супертипов слоя**.

Назначение

Супертип слоя используется тогда, когда все объекты соответствующего слоя имеют некоторые общие свойства или поведение. Поскольку в моих приложениях объекты слоев имеют множество общих черт, применение **супертипа слоя** вошло у меня в привычку.

Пример: объект домена (Java)

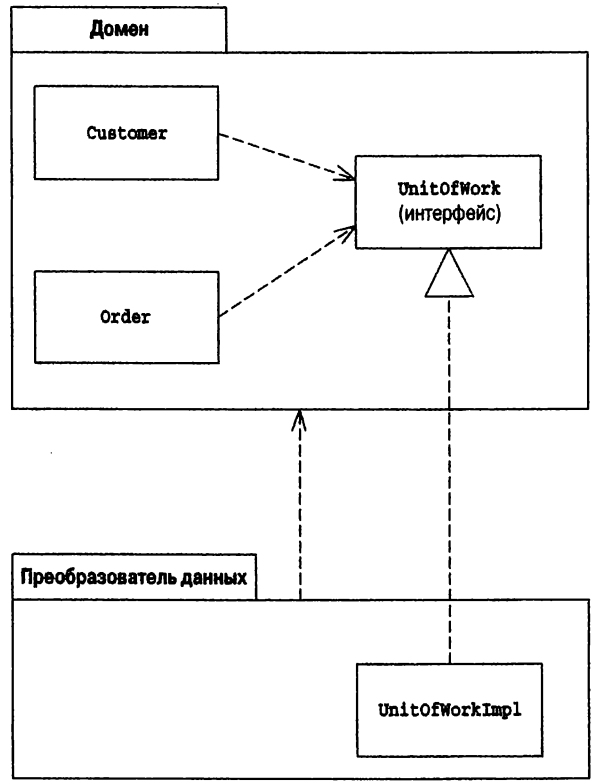
Ниже приведен простенький суперкласс объектов домена, выполняющий обработку их идентификаторов.

```
class DomainObject...

   private Long ID;
   public Long getID() {
      return ID;
   }
   public void setID(Long ID) {
      Assert.notNull("Cannot set a null ID", ID);
      this.ID =ID;
   }
   public DomainObject(Long ID) {
   this.ID = ID;
   }
   public DomainObject() {
   }
```

Отделенный интерфейс (Separated Interface)

Предполагает размещение интерфейса и его реализации в разных пакетах



По мере разработки системы может возникнуть желание улучшить структуру последней, уменьшая количество зависимостей между ее частями. Управлять зависимостями значительно удобнее, если классы будут сгруппированы в несколько пакетов. Выполнив группировку, вы сможете установить правила, определяющие, могут ли классы одного пакета обращаться к классам другого пакета, например правило, согласно которому классы слоя домена не должны вызывать методы классов слоя представления.

Несмотря на это, клиенту может понадобиться осуществить вызовы методов, противоречащие общей структуре зависимостей. В таком случае имеет смысл воспользоваться типовым решением **отделенный интерфейс**, определив интерфейс в одном пакете, а реализовав в другом. Таким образом, клиент, которому нужно установить связь с интерфейсом, будет оставаться полностью независимым от его реализации. **Отделенный интерфейс** представляет собой хороший объект для наложения **шлюза (Gateway, 483)**.

Принцип действия

Суть данного типового решения очень проста. Оно основано на том, что реализация зависит от своего интерфейса, но не наоборот. Это значит, что интерфейс и его реализацию можно разместить в разных пакетах, причем пакет, содержащий реализацию, будет зависеть от пакета, содержащего интерфейс. Все другие пакеты приложения могут зависеть от пакета, содержащего интерфейс, и при этом никак не зависеть от пакета, содержащего реализацию.

Разумеется, чтобы подобное приложение заработало во время выполнения, интерфейс должен иметь некоторую реализацию. Для этого можно воспользоваться отдельным пакетом, который будет связывать интерфейс и реализацию во время компиляции, или же связать их во время настройки приложения посредством **дополнительного модуля (Plugin, 514)**.

Интерфейс можно поместить в пакет клиента (как было показано на рисунке в начале раздела) или же в отдельный пакет (рис. 18.1). Если реализация имеет только одного клиента или все клиенты реализации находятся в одном пакете, интерфейс может быть размещен прямо в пакете клиента. В связи с этим нелишне задуматься о том, отвечают ли разработчики клиентского пакета за определение интерфейса? Размещение интерфейса в пакете клиента указывает на то, что данный пакет будет принимать обращения от всех пакетов, содержащих реализацию этого интерфейса. Таким образом, если у вас есть несколько клиентских пакетов, интерфейс лучше поместить в отдельный пакет. Это рекомендуется делать и тогда, когда определение интерфейса не входит в обязанности разработчиков клиентского пакета (например, когда определением интерфейса занимаются разработчики его реализации).

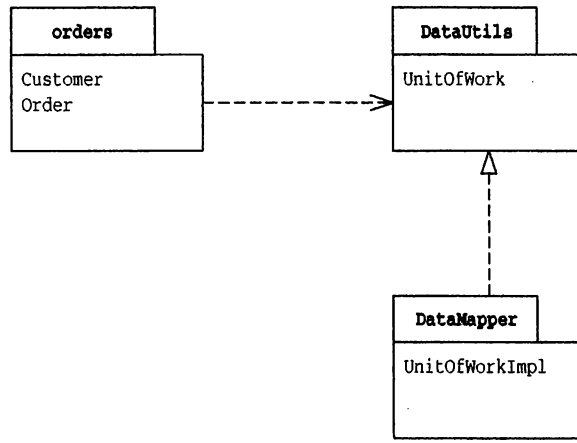


Рис. 18.1. Размещение **отделенного интерфейса** *в собственном пакете*

Помимо всего прочего, разработчику **отделенного интерфейса** необходимо выбрать, какими средствами языка программирования следует воспользоваться для описания интерфейса. Складывается впечатление, что при использовании языков наподобие Java и C#, содержащих специальные интерфейсные конструкции, проще всего применить ключевое слово `interface`. Как ни странно, это далеко не самый удачный выбор. В качестве интерфейса лучше использовать абстрактный класс, чтобы допустить наличие общего, но необязательного поведения.

Наиболее неприятным моментом в использовании **отделенного интерфейса** является создание экземпляра реализации. Как правило, чтобы создать экземпляр реализации, объект должен "знать" о классе последней. Зачастую в качестве такого объекта применяют отдельный объект-фабрику (factory object), интерфейс которого также реализован в виде **отделенного интерфейса**. Разумеется, объект-фабрика должен зависеть от реализации интерфейса, поэтому для обеспечения наиболее "безболезненной" связи применяют **дополнительный модуль**. Последний не только обеспечивает отсутствие зависимостей, но и позволяет отложить принятие решения о выборе класса реализации до момента настройки системы.

В качестве более простой альтернативы **дополнительному модулю** можно предложить использование еще одного пакета, знающего и об интерфейсе и о реализации, который будет создавать экземпляры нужных объектов во время запуска системы. В этом случае экземпляры всех объектов, использующих **отделенный интерфейс**, или же соответствующих объектов-фабрик могут быть созданы в момент запуска приложения.

Назначение

Отделенный интерфейс применяется для того, чтобы избежать возникновения зависимостей между двумя частями системы. Наиболее часто подобная необходимость возникает в описанных ниже ситуациях.

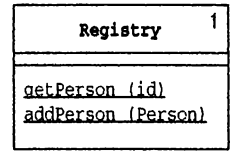
- Если вы разместили в пакете инфраструктуры абстрактный код для обработки стандартных случаев, который должен вызывать конкретный код приложения.
- Если код одного слоя приложения должен обратиться к коду другого слоя, о существовании которого он не знает (например, когда код домена обращается к **преобразователю данных (Data Mapper, 187)**).
- Если нужно вызвать методы, разработанные кем-то другим, но вы не хотите, чтобы ваш код зависел от интерфейсов API этих разработчиков.

Я часто встречал разработчиков, которые применяли **отделенный интерфейс** буквально к каждому классу приложения. Подобная предосторожность кажется мне чрезмерной. Отделение интерфейсов от реализаций и сохранение их в разных пакетах требует дополнительных трудозатрат, особенно если учесть необходимость написания объектов-фабрик (у которых есть собственные интерфейсы и реализации). Поэтому рекомендую использовать **отделенные интерфейсы** только тогда, когда нужно уничтожить зависимость между двумя подсистемами или обеспечить поддержку нескольких независимых реализаций. Вообще говоря, разделение интерфейса и реализации, находящихся в одном пакете, представляет собой не более чем простой рефакторинг, а потому вполне может быть отложено до тех пор, пока действительно не понадобится (если понадобится вообще).

На определенном этапе подобное управление зависимостями может оказаться не совсем уместным. Обычно зависимость от реализации применяют только для того, чтобы создать объект, а в остальных случаях обращаются к интерфейсу. В большинстве ситуаций этого достаточно. Неприятности наступают тогда, когда разработчику приходится искусственно применять правила установления зависимостей, например для выполнения проверки наличия зависимостей во время сборки. После этого все зависимости должны быть вновь уничтожены. В небольших приложениях применение подобных правил не так уж и важно, однако в более крупных системах правила установления зависимостей могут сыграть важную роль.

Реестр (Registry)

"Глобальный" объект, который используется другими объектами для поиска общих объектов или служб



Как следует поступить, чтобы найти какой-нибудь объект? Обычно мы обращаемся к другому объекту, который связан с искомым, и используем эту связь для перехода к последнему. Таким образом, если необходимо найти все заказы, сделанные заданным покупателем, следует обратиться к объекту покупателя и вызвать его метод для извлечения нужных заказов. К сожалению, иногда просто не с чего начать поиск: например, когда есть идентификатор покупателя, но нет ссылки на соответствующий объект. В этом случае понадобится какой-либо специальный метод поиска, однако куда же его поместить, чтобы он стал доступным из разных частей приложения?

Типовое решение **реестр** представляет собой глобальный объект, по крайней мере он выглядит как глобальный, даже если не является таковым в действительности.

Принцип действия

Проектирование **реестра**, как, впрочем, и любого другого объекта, должно рассматриваться в терминах интерфейса и реализации. Как и у всех других объектов, интерфейс и реализация **реестра** совершенно непохожи друг на друга, хотя многие почему-то думают, что они должны быть одинаковы.

В качестве интерфейса **реестров** рекомендую применять статические методы. Статический метод класса доступен всем объектам приложения. Более того, в статический метод можно поместить всю необходимую логику, включая делегирование полномочий другим методам, какими бы они не были — статическими или методами экземпляров объектов.

Эти идеи, однако, не означают, что данные **реестра** должны храниться в статических полях. Вообще говоря, я не использую статические поля, если только их значения не являются константами.

Прежде чем выбирать способ хранения данных **реестра**, подумайте об области их видимости. Эти данные могут фигурировать в различных контекстах выполнения. Одни из них являются глобальными по отношению ко всему процессу, другие — по отношению к потоку, а третьи и вовсе по отношению к сеансу. Разные области видимости требуют различных реализаций, однако интерфейс при этом может быть общий. Программист, пишущий код приложения, не должен знать, какие данные возвращает вызов статического метода — глобальные по отношению к процессу или по отношению к потоку. Для разных областей видимости можно предусмотреть разные **реестры**, но можно обойтись и одним, различные методы которого будут оперировать данными, имеющими разные области видимости.

Если определенные данные используются в контексте всего процесса, соответствующее поле **реестра** можно сделать статическим. Однако я редко использую статические поля для изменяемых данных, поскольку они не позволяют заменить **реестр** фиктивным объектом. Возможность замены **реестра** особенно важна в отношении тестирования (для этого можно воспользоваться **дополнительным модулем (Plugin, 514)**).

Реализуя **реестр**, глобальный по отношению к процессу, рекомендуется применять типовое решение **единственный элемент (Singleton)** [20]. Последнее гарантирует, что в системе будет существовать только один экземпляр соответствующего класса. В этом случае класс **реестра** будет состоять из единственного статического поля, содержащего экземпляр **реестра**. Зачастую при использовании объекта **единственный элемент** разработчики явно обращаются к его содержимому (посредством методов наподобие `Registry.soleInstance.getFoo()`), однако я предпочитаю применять статический метод, который скрывает от меня наличие единственного экземпляра объекта (`Registry.getFoo()`). Это особенно подходит для языков программирования, созданных на основе C, поскольку в них статические методы могут обращаться к закрытым данным экземпляра объекта.

Объекты с единственным экземпляром хорошо применять в однопоточных приложениях, а вот в многопоточных они могут стать настоящей проблемой. Манипулирование одним и тем же объектом в нескольких параллельных потоках зачастую приводит к совершенно непредсказуемым результатам. В качестве решения данной проблемы следовало бы применить синхронизацию, однако трудность написания кода синхронизации может окончательно свести вас с ума, прежде чем вам удастся ликвидировать все спорные моменты. Поэтому я не рекомендую использовать типовое решение **единственный элемент** для сохранения изменяемых данных в многопоточном окружении. Напротив, объекты с единственным экземпляром хорошо подходят для хранения неизменяемых данных, потому что невозможность изменения данных исключает возникновение конфликтов между параллельными потоками. Идеальным содержимым **реестра**, глобального по отношению к процессу, было бы нечто наподобие списка штатов США. Такие данные могут быть загружены в самом начале процесса и никогда не требуют изменений. Если же изменения все-таки случаются, они так редки, что обновление соответствующих данных можно реализовать посредством какого-либо радикального подхода, например прерывания процесса.

В большинстве случаев данные **реестра** являются глобальными по отношению к потоку. В качестве примера можно привести соединение с базой данных. Для проведения сеанса работы с базой данных многие среды разработки предоставляют хранилища, специфичные по отношению к потоку, наподобие Java-классов `ThreadLocal`. Вместо этого для хранения данных может применяться словарь, индексированный по потоку, элементами которого являются соответствующие объекты данных. В этом случае запрос на получение соединения приводит к выполнению поиска по значению текущего потока.

Манипулируя данными, глобальными по отношению к потоку, следует помнить, что внешне они ничем не отличаются от данных, глобальных по отношению к процессу. Метод наподобие `Registry.getDbConnection()` будет иметь одинаковый вид и для тех и для других данных.

Поиск по словарю может применяться и для данных, глобальных по отношению к сеансу. Для работы с такими данными необходимо иметь идентификатор сеанса, который в начале выполнения запроса может быть помещен в реестр, глобальный по отношению

к потоку. В этом случае для выполнения последующего доступа к данным сеанса объекты приложения могут проводить поиск в коллекции, индексированной по идентификаторам сеанса, используя значение идентификатора, хранящееся в реестре, глобальном по отношению к потоку.

Используя глобальный по отношению к потоку **реестр** со статическими методами, вы можете столкнуться с проблемами производительности, возникающими при попытках доступа к статическим методам **реестра** нескольких потоков. Избежать подобных проблем поможет прямое обращение к экземпляру потока.

Некоторым приложениям достаточно одного **реестра**, а некоторым может понадобиться сразу несколько. Как правило, **реестры** приложения группируются по слоям системы или же по контекстам выполнения. Я же предпочитаю группировать их по принципу использования, а не реализации.

Назначение

Несмотря на инкапсуляцию своих методов, содержимое **реестра** является глобальным по отношению к определенному контексту. Я никогда не любил работать с глобальными данными. В моих приложениях практически всегда встречается та или иная разновидность **реестра**, однако я совершенно искренне стараюсь избегать его применения и использую обычные переходы по связям между объектами там, где только возможно. Как правило, **реестр** следует применять только в случае крайней необходимости.

Существует несколько альтернатив использованию **реестра**. Одна из них состоит в том, чтобы передавать глобальные данные в виде параметров. К сожалению, этот подход приводит к тому, что параметры добавляются в вызовы методов, которым они совсем не нужны. Нередки ситуации, когда методы, действительно нуждающиеся в переданных параметрах, находятся в дереве вызовов на несколько слоев ниже тех, которым эти параметры были переданы. Передача практически ненужных параметров кажется мне пустой тратой времени, поэтому в подобных ситуациях я отдаю предпочтение **реестру**.

Еще одна альтернатива использованию **реестра** заключается в том, чтобы при создании экземпляров объектов добавлять к ним ссылки на необходимые глобальные данные. Разумеется, при этом в конструкторе объекта появляется лишний параметр, однако он по крайней мере не будет фигурировать в вызовах других методов. Как и предыдущий, этот подход имеет больше недостатков, чем преимуществ, но, если у вас есть глобальные данные, которые применяются только некоторым подмножеством классов, он может оказаться довольно полезным.

Существенным недостатком **реестра** является необходимость его изменения при добавлении новых объектов. Вследствие этого многие предпочитают хранить глобальные данные в коллекциях. Тем не менее я рекомендую использовать явные классы с явными методами, которые позволяют проследить, какие ключи применяются для поиска объекта. Чтобы понять принцип работы явного метода, достаточно взглянуть на исходный код или сгенерированную документацию. Использование коллекции лишено таких преимуществ. Чтобы понять, по какому ключу выполняется поиск, вам понадобится найти места системы, в которых происходит считывание или запись в коллекцию, либо обратиться к документации, которая, как известно, быстро становится неактуальной. Явный класс позволяет сохранить типовую безопасность в статически типизированных языках, а также инкапсулировать структуру **реестра**, чтобы при последующем росте системы ее можно было вынести в нужный класс или слой. Обычная коллекция не является

инкапсулированной, что значительно усложняет сокрытие реализации. Это особенно неудобно, если область видимости данных требуется изменить.

Как видите, необходимость в применении **реестра** все же возникает. Не забывайте, однако, что глобальных данных следует избегать до тех пор, пока их присутствие не станет неизбежным.

Пример: реестр с единственным экземпляром (Java)

Представьте себе приложение, которое считывает данные из базы данных и затем вносит в них изменения, чтобы получить нужную информацию. Для упрощения задачи наша система будет осуществлять доступ к данным посредством **шлюзов записи данных (Row Data Gateway, 175)**. Логика запросов к базе данных будет инкапсулирована в специальных методах поиска. Последние лучше реализовать в виде методов экземпляров объектов, чтобы при тестировании их можно было заменить **фиктивной службой (Service Stub, 519)**. Объекты поиска нужно где-нибудь разместить и, очевидно, наиболее подходящим для этого является **реестр**.

Реестр с единственным экземпляром объекта представляет собой очень простой пример типового решения **единственный элемент (Singleton)** [20]. Единственный экземпляр объекта реестра хранится в статическом поле.

```
class Registry...

   private static Registry getInstance() {
      return soleInstance;
   }
   private static Registry soleInstance = new Registry();
```

Все, что хранится в реестре, помещается в его экземпляр.

```
class Registry...

   protected PersonFinder personFinder = new PersonFinder();
```

Для облегчения глобального доступа открытые методы реестра сделаны статическими.

```
class Registry...

   public static PersonFinder personFinder() {
      return getInstance().personFinder;
   }
```

Чтобы заново инициализировать реестр, достаточно еще раз создать его единственный экземпляр.

```
class Registry...

   public static void initialize() {
      soleInstance = new Registry();
   }
```

Чтобы во время тестирования реестр можно было заменить **фиктивной службой**, я создам класс, производный от класса реестра.

```
class RegistryStub extends Registry...

   public RegistryStub() {
      personFinder = new PersonFinderStub();
   }
```

Вместо реального обращения к базе данных метод поиска **фиктивной службы** будет возвращать некий стандартный экземпляр **шлюза записи данных**, реализованного в виде объекта `Person`.

```
class PersonFinderStub...

   public Person find(long id) {
      if (id == 1) {
         return new Person("Fowler", "Martin", 10);
      }
      throw new IllegalArgumentException("Can't find id: " +
↳String.valueOf(id));
   }
```

Я поместил в реестр метод его инициализации в тестовом режиме. Впрочем, поскольку все суррогатное поведение находится в производном классе, я могу отделить весь код, необходимый для выполнения тестирования.

```
class Registry...

   public static void initializeStub() {
      soleInstance = new RegistryStub();
   }
```

Пример: реестр, уникальный в пределах потока (Java)

Мэттью Фоммел и Мартин Фаулер

Приведенный выше простой пример не подойдет для многопоточного приложения, в котором у каждого потока должен быть свой реестр. В языке Java существует типовое решение **хранилище потока (Thread Specific Storage)** [35], реализуемое с помощью локальных переменных потока `ThreadLocal`. Эти переменные могут быть использованы для создания реестра, уникального в пределах потока.

```
class ThreadLocalRegistry...

   private static ThreadLocal instances = new ThreadLocal();
   public static ThreadLocalRegistry getInstance() {
      return (ThreadLocalRegistry) instances.get();
   }
```

Для настройки **реестра** требуются методы, которые будут захватывать и высвобождать его по мере необходимости. Обычно время удержания реестра потоком совпадает с границами транзакции или сеанса.

```
class ThreadLocalRegistry...

   public static void begin() {
      Assert.isTrue(instances.get() == null);
      instances.set(new ThreadLocalRegistry());
   }
   public static void end() {
      Assert.notNull(getInstance());
      instances.set(null);
   }
```

Хранение объекта `PersonFinder` можно реализовать так же, как и в предыдущем примере.

```
class ThreadLocalRegistry...

   private PersonFinder personFinder = new PersonFinder();
   public static PersonFinder personFinder() {
      return getInstance().personFinder;
   }
```

Все обращения к реестру извне будут обрамляться методами `begin` и `end`.

```
      try {
         ThreadLocalRegistry.begin();
         PersonFinder f1 = ThreadLocalRegistry.personFinder();
         Person martin = Registry.personFinder().find(1);
         assertEquals("Fowler", martin.getLastName());
      } finally {ThreadLocalRegistry.end();
      }
```

Объект-значение (Value Object)

Небольшие простые объекты наподобие денежных значений или диапазонов дат, равенство которых не основано на равенстве идентификаторов

Работая с объектными системами различных типов, необходимо четко осознавать разницу между ссылочными объектами и **объектами-значениями**. Последние обычно представляют собой небольшие объекты; они напоминают элементарные типы данных, присутствующие во многих языках, которые не являются исключительно объектно-ориентированными.

Принцип действия

Сформулировать различие между ссылочным объектом и **объектом-значением** довольно сложно. В широком понимании этого слова **объектами-значениями** называют небольшие объекты наподобие денежных значений или дат, а ссылочными — более крупные объекты, например объект покупателя или объект заказа. Подобное определение довольно удобно, однако весьма неформально.

Ключевым отличием объекта-значения от ссылочного объекта является принцип проверки на равенство. Равенство ссылочных объектов определяется на основе равенства их идентификаторов, например встроенных идентификаторов объектно-ориентированных языков программирования или неких уникальных номеров, идентифицирующих объекты (таких, как первичные ключи реляционной базы данных). В свою очередь, равенство **объектов-значений** определяется на основе равенства значений полей соответствующего класса. Таким образом, два объекта дат считаются равными, если равны их значения дня, месяца и года.

Указанное различие отражается и в способе обработки объектов. Поскольку **объекты-значения** небольшие и легко создаются, их можно передавать в виде значения, а не в виде ссылки на объект. В действительности никого не волнует, сколько объектов, соответствующих 18 марта 2001 года, находится в системе в данный момент. Никого не волнует и то, ссылаются ли эти объекты на один и тот же физический объект даты или же представляют собой различные экземпляры объекта с одинаковыми значениями полей.

Большинство языков программирования не содержат специальных средств для работы с **объектами-значениями**. В этом случае для корректной обработки **объекты-значения** следует делать неизменяемыми, т.е. чтобы после создания объекта значения его полей не менялись. Это делается во избежание ошибок, связанных со ссылками на объекты. Подобные ошибки возникают тогда, когда два объекта совместно используют один и тот же объект-значение и один из владельцев последнего изменяет значение какого-нибудь поля. В качестве иллюстрации рассмотрим небольшой пример. Пусть Мартин был принят на работу 18 марта, и мы знаем, что Синди была принята на работу в тот же день. Мы можем установить дату принятия на работу Синди равной дате принятия на работу Мартина. Если теперь Мартин изменит эту дату на 18 мая, дата принятия на работу Синди также будет изменена. Естественно, это совсем не то, чего ожидают многие из нас. Обычно изменение даты принятия на работу или других небольших значений подразумевает полную замену старого объекта новым. Данному принципу полностью отвечает концепция неизменяемых объектов.

Объекты-значения не следует хранить в виде отдельных записей базы данных. Как правило, для хранения **объектов-значений** применяют **внедренные значения (Embedded Value, 288)** или **крупные сериализованные объекты (Serialized LOB, 292)**. Поскольку **объекты-значения** довольно небольшие, их лучше сохранять в виде **внедренного значения**, так как подобная форма хранения позволяет осуществлять запросы к содержимому **объекта-значения**.

Если в системе выполняется множество процедур сериализации в двоичный формат, следует подумать об оптимизации этого процесса для **объектов-значений** — она может существенно улучшить производительность приложения. Особенно это касается языков наподобие Java, которые не содержат специальных средств для обработки **объектов-значений**.

Примеры **объектов-значений** приведены в разделе, посвященном типовому решению **деньги (Money, 502)**.

Особенности .NET-реализации

Платформа .NET содержит все необходимые средства для обработки **объектов-значений**. В языке C#, чтобы пометить объект как **объект-значение**, достаточно указать в его объявлении ключевое слово `struct`, а не `class`. После этого обработка указанного объекта будет проводиться исходя из семантики значения.

Назначение

Типовое решение **объект-значение** следует применять для моделирования тех сущностей, равенство которых определяется по значениям полей, а не по идентификаторам объектов. Это особенно хорошо подходит для небольших, легких в создании объектов наподобие денежных значений или диапазонов дат.

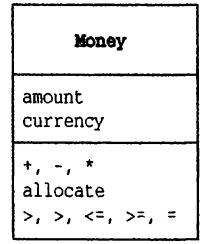
Совпадение названий

На протяжении довольно долгого времени данное типовое решение носило название **объекта-значения**. К сожалению, несколько лет назад в сообществе J2EE [3] термин *объект-значение (value object)* стал применяться для обозначения типового решения **объект переноса данных (Data Transfer Object, 419)**, что вызвало настоящую бурю среди разработчиков типовых решений. Подобная путаница с названиями — далеко не редкость в нашей сфере деятельности. Не так давно авторы книги [3] решили отказаться от использования термина "объект-значение" и заменили его термином *объект переноса (transfer object)*.

Я продолжаю использовать термин "объект-значение" в том смысле, в котором он применяется в контексте данной книги. По крайней мере, так я не вступаю в противоречие со своими предыдущими работами!

Деньги (Money)

Представляет денежное значение



Большая часть корпоративных приложений, да и компьютеров вообще оперирует денежными величинами. Меня всегда удивлял и продолжает удивлять тот факт, что в популярных языках программирования не содержится стандартных типов, предназначенных

для работы с деньгами. Отсутствие подобного типа приводит к возникновению многих проблем, одна из которых связана с использованием валют. Если все финансовые расчеты выполняются в одной валюте, это не так уж сложно; однако, если используемых валют несколько, вам наверняка захочется складывать доллары и иены с учетом различия в их курсах. Существует и более тонкий аспект, касающийся округления. Как правило, результаты денежных вычислений округляют до наименьшей единицы используемой валюты, что зачастую приводит к потере центов (или копеек, или других единиц вашей местной валюты) в результате ошибок округления.

Для обработки подобных ситуаций воспользуемся средствами объектно-ориентированного программирования и создадим типовое решение под названием **деньги**. Еще раз подчеркну: меня весьма удивляет, что ни одна из базовых библиотек распространенных языков программирования еще не содержит подобного класса.

Принцип действия

В основе типового решения **деньги** лежит идея создания класса, поля которого будут содержать величину денежной суммы, а также валюту, в которой указана эта сумма. Величина денежной суммы может быть представлена значением целочисленного типа или же действительным значением с фиксированным количеством десятичных знаков. В одних вычислениях проще использовать действительные значения, а в других — целочисленные. Применять числовые типы с плавающей точкой не стоит, потому что они вызывают проблемы с округлением, которых всячески следует избегать. Большую часть времени денежные величины будут округляться до наименьшей единицы текущей валюты, например до центов. Несмотря на это, для выполнения некоторых вычислений могут понадобиться дробные единицы. Проектируя приложение, следует четко обозначить, с каким видом денежных единиц — целочисленных или дробных — вы работаете в данный момент, особенно если приложение использует единицы обоих видов. Для обработки подобных случаев рекомендуется применять два различных типа данных, поскольку выполнение арифметических операций над каждым из них происходит по-своему.

Денежные величины представляют собой **объекты-значения (Value Object, 500)**, поэтому методы проверки на равенство и хэш-операции соответствующих объектов должны быть переопределены так, чтобы их выполнение основывалось на величине денежной суммы и используемой валюте.

Обработка денежных величин подразумевает выполнение арифметических вычислений, поэтому обращаться с деньгами так же просто, как если бы они были обычными числами. Несмотря на это, арифметические действия над денежными величинами имеют ряд существенных особенностей, что, несомненно, отличает объекты-деньги от обычных чисел. Прежде всего, операции сложения и вычитания должны выполняться с учетом используемой валюты, чтобы методы денежных объектов могли перехватить попытку сложения величин, приведенных в разных валютах, и соответствующим образом отреагировать на нее. Самое простое (и самое распространенное) решение данной проблемы заключается в том, чтобы попытка сложения величин в разных валютах рассматривалась как ошибка. В некоторых более сложных ситуациях можно воспользоваться идеей "кошелька" (money bag), принадлежащей Вэрду Каннинхэму (Ward Cunningham). Кошелек — это объект, содержащий денежные значения в разных валютах. Такой объект может участвовать в выполнении вычислений наряду с обычными

денежными объектами. Кроме того, все содержимое кошелька может быть переведено в какую-либо одну валюту.

Операции умножения и деления еще более сложны, потому что их выполнение неминуемо приводит к проблемам округления. При умножении денежных величин можно воспользоваться обычными числами: например, чтобы подсчитать величину пятипроцентного налога на сумму заказа, ее следует умножить на 0,05. Таким образом, выполнение операций умножения над денежными объектами подразумевает умножение объекта на скалярную величину, что не так сложно.

Настоящие проблемы округления возникают в ситуациях, когда заданную сумму денег необходимо распределить между несколькими различными местами. В связи с этим попробуйте решить простенькую задачку, предложенную Мэттью Фоммелем (Matthew Foemmel). Предположим, у меня есть бизнес-правило, в соответствии с которым я должен положить имеющуюся сумму денег на два банковских счета: 70% суммы на первый счет и 30% — на второй. Пусть эта сумма составляет 5 центов. В результате вычислений я получаю, что на первый счет нужно положить 3,5 цента, а на второй — 1,5 цента. Если я выполню стандартное округление до ближайшего целого числа, полтора цента превратятся в два, а три с половиной — в четыре. Как видите, в сумме у меня появился лишний цент. Если же я округлю полученные величины путем отбрасывания дробной части, то получу 4 цента, т.е. потеряю один цент. Таким образом, я не могу применить к обеим величинам одну и ту же схему округления, потому что все универсальные схемы приводят к потере или приобретению "лишних" центов.

Мне встречалось несколько вариантов решения данной проблемы.

- Наиболее распространенное решение — не обращать на это внимания (в конце концов, речь идет о каком-то там центе). Тем не менее большинство банковских клиентов были бы крайне возмущены подобной небрежностью по отношению к своим деньгам.
- Вычислять сумму, которую нужно положить на последний счет, путем вычитания из общей суммы денег тех сумм, которые были положены на все предыдущие счета. Это позволит избежать потери центов, однако в конце концов может привести к накоплению на последнем счете довольно большого количества центов, которые ему не принадлежат.
- Предоставить пользователям класса `Money` возможность самим выбирать схему округления при вызове нужного метода. В этом случае программист может указать, что 70% суммы следует округлить до следующего целого числа, а 30% суммы — до предыдущего. Разумеется, подобный метод весьма неудобен, если деньги нужно распределить не между двумя, а, скажем, между десятью счетами. Кроме того, об указании схемы округления легко забыть. Иногда во избежание подобных проблем метод умножения класса `Money` принудительно снабжают параметром округления. Это не только обяжет программиста выбрать схему округления, но и напомнит ему о необходимости написания тестов. К сожалению, данный подход весьма непрактичен при выполнении множества вычислений (например, при подсчете величин налогов), использующих одну и ту же схему округления.
- Идеальное, на мой взгляд, решение — создать специальную функцию-распределитель. В качестве параметра этой функции будет передаваться список чисел, определяющих пропорцию, согласно которой необходимо распределить деньги

(например, `aMoney.allocate([7,3])`). По окончании выполнения функция-распределитель будет возвращать список денежных значений. Чтобы гарантировать отсутствие лишних или утерянных центов, оставшиеся от округления центы распределяются между всеми счетами псевдослучайным способом. Применение функции-распределителя имеет свои недостатки: во-первых, о ней можно забыть, а во-вторых, она не позволяет применить более четкие правила распределения оставшихся центов.

Описанные подходы касаются фундаментального различия между выполнением операции умножения для определения величины, пропорциональной исходной сумме (например, налога на сумму заказа), или же для распределения исходной суммы по нескольким местам. Первую процедуру удобнее реализовать с помощью умножения, а вторую — с помощью распределения. Таким образом, при выполнении умножения или деления денежных величин необходимо тщательно следить за тем, для чего эта операция применяется.

При необходимости объект `Money` можно снабдить методами, выполняющими перерасчет из одной валюты в другую (например, методом `aMoney.convertTo(Currency.DOLLARS)`). Наиболее очевидный способ выполнения такого перерасчета — посмотреть курс валют и умножить денежную величину на соответствующий коэффициент. К сожалению, в некоторых ситуациях этот подход может не сработать. Правила перерасчета основных европейских валют подразумевают специфическое выполнение округлений, которое противоречит описанной выше простой схеме. В этом случае для инкапсуляции алгоритма перерасчета следует создать отдельный объект-преобразователь.

Для сортировки денежных величин можно использовать операции сравнения. При попытке сравнения значений, представленных в разных валютах, система может выдать ошибку или же выполнить автоматический перерасчет величин в одну из валют.

Помимо всего прочего, типовое решение **деньги** может инкапсулировать в себе выполнение отображения. Это позволяет отображать денежные значения в элементах управления пользовательских интерфейсов и при печати отчетов так, как это делается в реальной жизни. Кроме того, объект `Money` может включать в себя анализатор входной строки, чтобы обеспечить механизм ввода, специфичный для конкретной валюты. Для реализации подобного механизма можно воспользоваться библиотеками, прилагаемыми к вашей платформе. Сегодня все больше и больше платформ обеспечивают поддержку глобализации, предоставляя специальные средства форматирования для отображения валют определенных стран.

Хранение денежных объектов в базе данных также чревато определенными проблемами, поскольку базы данных (в отличие от своих производителей) не понимают всю значимость денег. Наиболее очевидное решение — хранить денежные величины в виде **внедренного значения (Embedded Value, 288)**. В этом случае вместе с каждым денежным значением будет сохраняться название соответствующей валюты, что может оказаться крайне неудобным, например если все суммы на счете клиента указаны в фунтах. В подобных ситуациях название валюты лучше хранить где-нибудь в одном месте и изменить код отображения на базу данных таким образом, чтобы при загрузке объектов соответствующий преобразователь извлекал и название валюты, в которой ведется счет.

Назначение

Я применяю типовое решение **деньги** для выполнения практически всех числовых вычислений в объектно-ориентированных средах разработки. Основной аргумент в пользу данного типового решения связан с необходимостью инкапсулировать обработку округления, что позволяет избежать большинства ошибок округления до фиксированного количества десятичных знаков. Еще одним преимуществом является возможность одновременного выполнения вычислений в нескольких валютах. Разумеется, наличие подобных объектов может сказаться на производительности, хотя я крайне редко слышал о случаях, когда применение данного типового решения существенно снижало производительность приложения. Впрочем, даже тогда инкапсуляция денежных величин в отдельных объектах значительно упрощала выполнение финансовых операций, ради чего стоило пойти на определенные жертвы.

Пример: класс Money (Java)

Мэттью Фоммел и Мартин Фаулер

Реализуя класс `Money`, нужно решить, какой тип данных будет применяться для хранения величины денежной суммы. Вы все еще убеждены в необходимости применения действительного типа с плавающей точкой? Тогда попробуйте запустить приведенный ниже код.

```
double val = 0.00;
for (int i = 0; i < 10; i++) val += 0.10;
System.out.println(val == 1.00);
```

Думаю, после выполнения этого простенького примера вы напрочь откажетесь от применения чисел с плавающей точкой. Оставшиеся альтернативы включают в себя числа с фиксированной точкой и целые числа. В Java им соответствуют типы данных `BigDecimal`, `BigInteger` и `long`. Использование целочисленных типов значительно упрощает выполнение внутренних арифметических операций. Кроме того, `long` является стандартным типом данных, а значит, существенно повышает читабельность математических выражений.

```
class Money...

   private long amount;
   private Currency currency;
```

В данном примере я использую целочисленный тип данных, поэтому денежные величины необходимо выражать в наименьших единицах текущей валюты (для простоты я назвал их центами). При использовании типа `long` для операций над очень большими числами может возникнуть ошибка переполнения. Таким образом, если вам понадобится оперировать суммами наподобие 92 233 720 368 547 758,09 доллара, подумайте о применении типа `BigInteger`.

Чтобы облегчить создание объектов, предоставим несколько конструкторов, принимающих в качестве аргумента различные типы данных.

```
public Money(double amount, Currency currency) {
   this.currency = currency;
   this.amount = Math.round(amount * centFactor());
}
public Money(long amount, Currency currency) {
   this.currency = currency;
   this.amount = amount * centFactor();
}
private static final int[] cents = new int[] { 1, 10, 100,
↳1000 };
private int centFactor() {
   return cents[currency.getDefaultFractionDigits()];
}
```

Денежные единицы различных валют делятся на разное количество более мелких единиц. В версии Java 1.4 количество элементарных единиц, на которые делится основная денежная единица, определяется классом `Currency`. Чтобы узнать, сколько элементарных единиц (например, центов) содержится в сумме, выраженной в основных денежных единицах (например, в долларах), ее нужно умножить на 10 в соответствующей степени. Впрочем, выполнение подобных операций в Java настолько неудобно, что вместо возведения в степень гораздо проще (и быстрее) воспользоваться массивом. Разумеется, нужно быть морально готовым к тому, что наше приложение откажется работать с гипотетическими валютами, в которых основная денежная единица состоит из 10 000 "копеек".

Хотя в большинстве случаев арифметические операции будут выполняться над самими объектами `Money`, нам может понадобиться осуществить доступ к содержимому последних.

```
class Money...

   public BigDecimal amount() {
      return BigDecimal.valueOf(amount,
↳currency.getDefaultFractionDigits());
   }
   public Currency currency() {
      return currency;
   }
```

Не забывайте: к использованию функций доступа следует обращаться только в самых крайних случаях. Практически всегда к содержимому объекта можно добраться более удачным способом, не нарушая его инкапсуляции. В качестве примера можно привести отображение на базу данных, рассмотренное в разделе, посвященном **внедренному значению** (см. главу 12).

Чтобы облегчить создание денежных объектов в определенной часто используемой валюте (например, в долларах), можно написать вспомогательный конструктор.

```
class Money...

   public static Money dollars(double amount) {
      return new Money(amount, Currency.USD);
   }
```

Поскольку **деньги** являются **объектом-значением (Value Object, 500)**, необходимо определить метод проверки на равенство.

```
class Money...

   public boolean equals(Object other) {
      return (other instanceof Money) && equals((Money)other);
   }
   public boolean equals(Money other) {
      return currency.equals(other.currency) && (amount ==
↳other.amount);
   }
```

Там, где есть метод проверки на равенство, не обойтись без хэширования.

```
class Money...

   public int hashCode() {
      return (int) (amount ^ (amount >>> 32));
   }
```

Пришло время определить арифметические операции, совершаемые над объектами `Money`. Начнем со сложения и вычитания.

```
class Money...

   public Money add(Money other) {
      assertSameCurrencyAs(other);
      return newMoney(amount + other.amount);
   }
   private void assertSameCurrencyAs(Money arg) {
      Assert.equals("money math mismatch", currency,
↳arg.currency);
   }
   private Money newMoney(long amount) {
      Money money = new Money();
      money.currency = this.currency;
      money.amount = amount;
      return money;
   }
```

Обратите внимание на использование закрытого метода-фабрики `NewMoney`, который не выполняет привычное преобразование долларов в центы. Этот метод будет несколько раз фигурировать во внутреннем коде объекта `Money`.

Имея метод сложения, определить вычитание совсем несложно.

```
class Money...

   public Money subtract(Money other) {
      assertSameCurrencyAs(other);
      return newMoney(amount - other.amount);
   }
```

В качестве базового метода сравнения будет выступать метод `compareTo`.

```
class Money...

   public int compareTo(Object other) {
      return compareTo((Money)other);
   }
   public int compareTo(Money other) {
      assertSameCurrencyAs(other);
      if (amount < other.amount) return -1;
      else if (amount == other.amount) return 0;
      else return 1;
   }
```

Это практически все, что можно выжать из классов Java в настоящее время. Тем не менее можно немного повысить читабельность кода, реализовав еще несколько методов сравнения наподобие приведенного ниже.

```
class Money...

   public boolean greaterThan(Money other) {
      return (compareTo(other) > 0);
   }
```

Теперь можно взяться и за умножение. В данном примере воспользуемся стандартным режимом округления, однако при необходимости можно определить какой-нибудь другой режим.

```
class Money...

   public Money multiply(double amount) {
      return multiply(new BigDecimal(amount));
   }
   public Money multiply(BigDecimal amount) {
      return multiply(amount, BigDecimal.ROUND_HALF_EVEN);
   }
   public Money multiply(BigDecimal amount, int roundingMode) {
      return new Money(amount().multiply(amount), currency,
↳roundingMode);
   }
```

Чтобы распределить сумму денег между несколькими счетами, не потеряв при этом ни цента, понадобится реализовать метод распределения. Самая простая схема распределения предполагает, что на все счета будут положены одинаковые (или почти одинаковые) денежные суммы.

```
class Money...

   public Money[] allocate(int n) {
      Money lowResult = newMoney(amount / n);
      Money highResult = newMoney(lowResult.amount + 1);
      Money[] results = new Money[n];
```

```
      int remainder = (int) amount % n;
      for (int i = 0; i < remainder; i++) results [i] =
↳highResult;
      for (int i = remainder; i < n; i++) results [i] =
↳lowResult;
      return results;
   }
```

Более сложный алгоритм распределения может поделить деньги в любой заданной пропорции.

```
class Money...

   public Money[] allocate(long[] ratios) {
      long total = 0;
      for (int i = 0; i < ratios.length; i++) total +=
↳ratios[i];
      long remainder = amount;
      Money[] results = new Money[ratios.length];
      for (int i = 0; i < results.length; i++) {
         results[i] = newMoney(amount * ratios[i] / total);
         remainder -= results[i].amount;
   }
   for (int i = 0; i < remainder; i++) {
      results[i].amount++;
   }
   return results;
   }
```

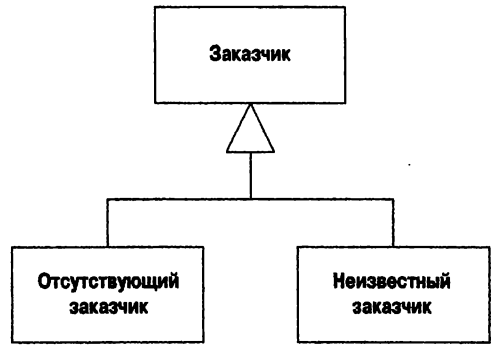
Данный алгоритм можно применить для решения задачки Мэттью Фоммела (Matthew Foemmel).

```
class Money...

   public void testAllocate2() {
      long[] allocation = {3,7};
      Money[] result = Money.dollars(0.05).allocate(allocation);
      assertEquals(Money.dollars(0.02), result[0]);
      assertEquals(Money.dollars(0.03), result[1]);
   }
```

Частный случай (Special Case)

Производный класс, описывающий поведение объекта в особых ситуациях



Значения NULL — настоящий бич объектно-ориентированных приложений. Потенциальное наличие таких значений напрочь лишает разработчика возможности воспользоваться преимуществами полиморфизма. Обычно методу можно передавать ссылки на переменные или же значения заданных типов, не беспокоясь о том, принадлежит ли переданное значение к указанному типу или же к производному классу. В строго типизированных языках проверку на правильность вызова можно проводить еще на этапе компиляции. К сожалению, если передаваемая переменная имеет значение NULL, при попытке доступа к последнему вы совершите ошибку времени выполнения и получите не более чем красивую и весьма "дружелюбную" трассировку стека.

Если переменная может принимать значение NULL, следует позаботиться о реализации проверки на наличие последнего, чтобы при его обнаружении приложение выполняло некие особые действия. Зачастую "особые действия" одинаковы во многих контекстах, что, несомненно, приводит к необходимости повторения одних и тех же фрагментов кода в различных местах приложения. Итак, снова дублирование?

Наличие значений NULL — наиболее распространенный, но далеко не единственный пример подобных проблем. Большинство математических приложений связаны с обработкой бесконечности. Сложение бесконечных величин выполняется по иным правилам, чем сложение обычных действительных чисел. В качестве еще одного примера могу привести случай из моего раннего опыта работы с бизнес-приложениями, касающийся обработки неизвестного или не полностью известного заказчика (мы называли его "захватчиком"). Все описанные выше ситуации имеют одну общую особенность: они предполагают изменение обычного поведения объектов данного типа.

Вместо того чтобы при возникновении особых ситуаций возвращать значение NULL или вообще что-нибудь непонятное, можно воспользоваться типовым решением **частный случай**. Последнее возвращает специальный объект с тем интерфейсом, который ожидает увидеть вызывающий метод.

Принцип действия

Основная идея данного типового решения состоит в том, чтобы создать производный класс для обработки частных, особых случаев. Таким образом, чтобы избежать проверки объекта `Customer` на наличие значений `NULL`, можно создать объект `NullCustomer`. Последний будет переопределять все методы объекта `Customer`, чтобы при возникновении особой ситуации приложение не начало вести себя совершенно непредсказуемым образом, а выполнило что-нибудь безобидное. В этом случае при обнаружении значения `NULL` вызывающий метод вместо экземпляра класса `Customer` будет возвращать экземпляр производного класса `NullCustomer`.

Обычно приложению не требуется проводить различие между экземплярами объекта `NullCustomer`, поэтому для его реализации можно воспользоваться типовым решением **приспособленец (Flyweight)** [20], предполагающим совместное использование одного экземпляра объекта в нескольких контекстах. К сожалению, иногда этот прием не срабатывает. В уже упоминавшемся бизнес-приложении подсчитывать долг заказчика следовало всегда — даже когда этот заказчик был неизвестен и сведений о его заказе практически не было. В этом случае каждому неизвестному заказчику должен был соответствовать свой экземпляр объекта `NullCustomer`.

Значения `NULL` могут иметь разную природу. Наличие объекта `NullCustomer` может означать, что заказчика нет или же что заказчик есть, но мы не знаем, кто он. В подобных случаях вместо применения общего объекта `NullCustomer` рекомендуется описать отдельные **частные случаи** для отсутствующего заказчика (например, `MissingCustomer`) и для неизвестного заказчика (например, `UnknownCustomer`).

Нередки ситуации, когда при переопределении методов в **частном случае** последний возвращает еще один **частный случай**, например при запросе последнего счета неизвестного покупателя соответствующий метод объекта `UnknownCustomer` может возвратить объект `UnknownBill`.

Определение арифметических операций над числами с плавающей точкой, приведенное в стандарте IEEE 754, содержит хорошие примеры **частных случаев**, касающихся обработки положительных бесконечных величин, отрицательных бесконечных величин и значений типа NaN (not-a-number — не число). При попытке деления на нуль подобные системы не генерируют исключение, а возвращают значение типа NaN, которое может участвовать в выполнении арифметических операций наравне с обычными числами.

Назначение

Типовое решение **частный случай** следует применять для описания повторяющегося поведения, связанного с обработкой значений `NULL` или других особых ситуаций после выполнения соответствующих проверок на их наличие.

Дополнительные источники информации

Я еще не встречал описания **частного случая** в виде отдельного типового решения. Тем не менее в [40] содержится описание типового решения **null-объект (Null Object)**, которое — надеюсь, вы простите мне небольшой каламбур — можно рассматривать как частный случай **частного случая**.

Пример: объект NullEmployee (C#)

Ниже приведен небольшой пример использования **частного случая** для обработки значений NULL.

У нас есть обычный класс Employee.

```
class Employee...

      public virtual String Name {
         get {return _name;}
         set {_name = value;}
      }
      private String _name;
      public virtual Decimal GrossToDate {
         get {return calculateGrossFromPeriod(0);}
      }
      public virtual Contract Contract {
         get {return _contract;}
      }
      private Contract _contract;
```

Поведение этого объекта может быть переопределено методами класса NullEmployee.

```
class NullEmployee : Employee, INull...

      public override String Name {
         get {return "Null Employee";}
         set {}
      }
      public override Decimal GrossToDate {
         get {return 0m;}
      }
      public override Contract Contract {
         get {return Contract.NULL;}
      }
```

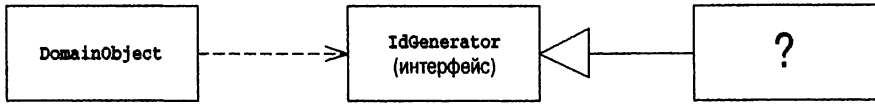
Обратите внимание, что при попытке извлечения значения контракта из объекта NullEmployee мы получаем значение NULL.

Приведенные выше переменные позволяют избежать реализации множества проверок на наличие значений NULL, если все эти значения имеют одну и ту же природу. По умолчанию повторяющиеся значения NULL обрабатываются объектом NullEmployee. Кроме того, проверку на наличие значений NULL можно выполнить и явно, снабдив объект Employee методом isNull или проверив тип объекта на предмет реализации соответствующего опознавательного интерфейса (marker interface).

Дополнительный модуль (Plugin)

Дейвид Райс и Мэттью Фоммел

Связывает классы во время настройки,
а не во время компиляции приложения



Типовое решение **отделенный интерфейс (Separated Interface, 492)** часто используют в приложениях, запускаемых в различных исполняющих средах, каждая из которых требует собственной реализации конкретного поведения. Большинство разработчиков обеспечивают выбор нужной реализации путем написания метода-фабрики. В связи с этим хочется рассмотреть небольшой пример. Предположим, вы создали генератор первичного ключа базы данных и воспользовались **отделенным интерфейсом**, чтобы для определения значений первичного ключа во время тестирования применялся простой счетчик, расположенный в оперативной памяти, а в реальных ситуациях значения ключа извлекались из последовательности номеров, хранящейся в базе данных. В этом случае метод-фабрика должен содержать некий условный оператор, который просматривает локальную переменную окружения, определяет, находится ли приложение в режиме тестирования, и возвращает нужный генератор первичного ключа. Чем больше методов-фабрик будет в приложении, тем больше проблем это принесет. Добавление новых конфигураций среды развертывания (например, "выполнять тестирование с использованием базы данных, расположенной в оперативной памяти, без управления транзакциями" или "работать в реальном режиме с использованием базы данных DB2 и полноценным управлением транзакциями") требует изменения условных выражений в каждом из методов-фабрик, новой сборки и нового развертывания приложения. В действительности процесс настройки не должен быть разбросан по всему приложению и уж никак не должен требовать новой сборки или нового развертывания. Чтобы избежать перечисленных проблем, можно воспользоваться типовым решением **дополнительный модуль**, которое обеспечивает централизованную настройку приложения во время выполнения.

Принцип действия

Вначале нужно определить **отделенный интерфейс** для каждого поведения, которое требует применения различных реализаций в зависимости от выбранной исполняющей среды. Помимо этого, воспользуемся базовой разновидностью типового решения **фабрика (Factory)**, несколько видоизменив его в соответствии с нашими требованиями. Реализация **дополнительного модуля** в виде объекта-фабрики требует, чтобы все правила связывания находились в единственной внешней точке. Это необходимо для облегчения настройки приложения. Кроме того, связывание интерфейса с реализацией должно выполняться не во время компиляции, а динамически, т.е. во время выполнения, чтобы избежать повторной сборки приложения.

Правила связывания можно хранить в обычном текстовом файле. Метод-фабрика **дополнительного модуля** считывает файл настроек, ищет строку, задающую реализацию запрошенного интерфейса и возвращает эту реализацию (рис. 18.2).

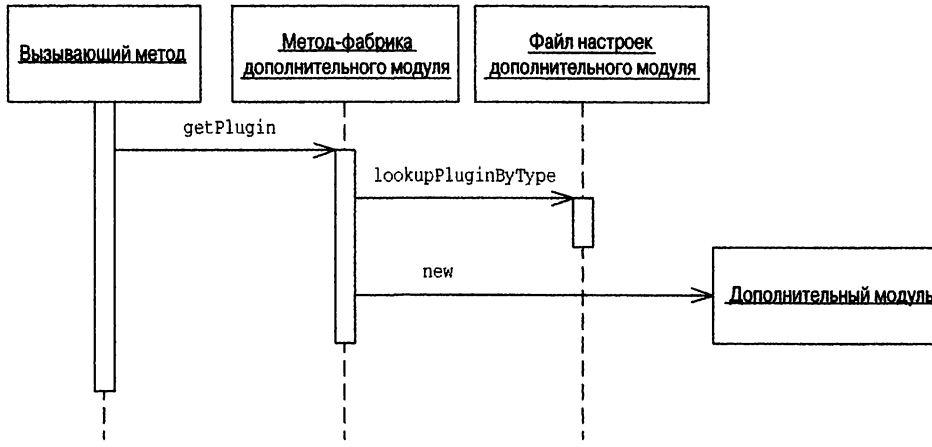


Рис. 18.2. Получение вызывающим методом нужной реализации отделенного интерфейса

Применение **дополнительного модуля** наиболее предпочтительно в языках, поддерживающих отражение, поскольку объект-фабрика может создавать объекты реализаций без необходимости устанавливать зависимости во время компиляции. При использовании метода отражения файл настроек должен содержать отображения имен интерфейсов на имена классов реализаций. В этом случае объект-фабрика может находиться в пакете инфраструктуры, никак не связанном с пакетами, в которых находятся классы реализаций, и не нуждается в изменениях при добавлении в файл настроек новых отображений.

Даже если используемый язык программирования не поддерживает отражение, наличие централизованной точки настройки значительно упрощает работу с приложением. В этом случае правила связывания также могут быть помещены в текстовый файл, с той лишь разницей, что для отображения интерфейса на нужный класс реализации объект-фабрика будет применять условную логику. Разумеется, в этом случае объект-фабрика должен учитывать каждую существующую реализацию. Впрочем, на практике это не так сложно: просто добавьте к объекту-фабрике новый параметр при добавлении нового класса реализации. Чтобы зависимости между слоями и пакетами устанавливались только во время сборки приложения, объект-фабрику необходимо поместить в отдельный пакет.

Назначение

Типовое решение **дополнительный модуль** применяется тогда, когда одно и то же поведение может иметь несколько реализаций в зависимости от выбранной исполняющей среды.

Пример: генератор идентификаторов (Java)

Как уже отмечалось, генерация ключей или идентификаторов предусматривает поведение, реализация которого может различаться в зависимости от среды развертывания (рис. 18.3).

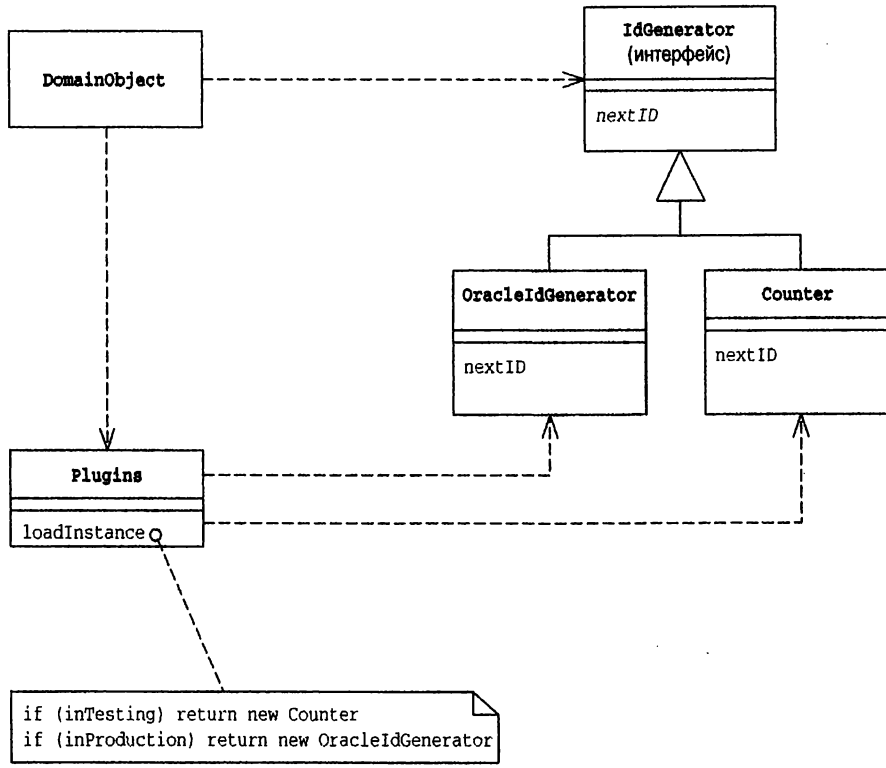


Рис. 18.3. Различные генераторы идентификаторов

Вначале опишем **отделенный интерфейс** `IdGenerator`, а также все необходимые реализации.

```
interface IdGenerator...

   public Long nextId();

class OracleIdGenerator implements IdGenerator...

   public OracleIdGenerator() {
      this.sequence = Environment.getProperty("id.sequence");
      this.datasource = Environment.getProperty("id.source");
   }
```

Реализация метода `nextId()` в классе `OracleIdGenerator` возвращает следующий незанятый номер из заранее определенной последовательности номеров, хранящейся в заданном источнике данных.

```
class Counter implements IdGenerator...

   private long count = 0;
   public synchronized Long nextId() {
      return new Long(count++);
   }
```

Итак, у нас появились классы реализаций. Теперь можно написать объект-фабрику **дополнительного модуля**, который будет выполнять отображение интерфейса на необходимые реализации.

```
class PluginFactory...
   private static Properties props = new Properties();

   static {
     try {
             String propsFile = System.getProperty("plugins");
             props.load(new FileInputStream(propsFile));
     } catch (Exception ex) {
             throw new ExceptionInInitializerError(ex);
     }
   }

   public static Object getPlugin(Class iface) {
      String implName = props.getProperty(iface.getName());
      if (implName == null) {
         throw new RuntimeException("implementation
↳not specified for " + iface.getName() + " in PluginFactory
↳properties.");
      }
      try {
         return Class.forName(implName).newInstance();
      } catch (Exception ex) {
         throw new RuntimeException("factory unable to construct
↳instance of " + iface.getName());
      }
   }
```

Обратите внимание, что для загрузки нужной конфигурации мы обращаемся к системному свойству `plugins`, значением которого является имя файла, содержащего правила связывания. Существует много различных способов определения и хранения правил связывания, однако в данном примере мы воспользовались простым файлом настроек. Помещение имени файла настроек в системное свойство вместо явного указания пути в классе `PluginFactory` позволяет легко выбрать нужную конфигурацию приложения из любого места компьютера. Это очень удобно при перемещении составных частей приложения между средами разработки, тестирования и функционирования.

Как могут выглядеть файлы настроек, предназначенные для работы приложения в режиме тестирования и в реальном режиме, показано ниже.

```
config file test.properties...

   # test configuration
   IdGenerator=TestIdGenerator

config file prod.properties...

   # production configuration
   IdGenerator=OracleIdGenerator
```

Давайте вернемся к интерфейсу `IdGenerator` и добавим к нему статический атрибут `INSTANCE`, значение которого будет определяться путем обращения к объекту `PluginFactory`. Данный подход позволяет скомбинировать **дополнительный модуль** с типовым решением **единственный элемент (Singleton)**, чтобы метод получения идентификатора выглядел как можно проще и читабельнее.

```
interface IdGenerator...

   public static final IdGenerator INSTANCE =
      (IdGenerator) PluginFactory.getPlugin(IdGenerator.class);
```

Теперь можно описать процесс получения идентификатора, зная, что **дополнительный модуль** сам выберет нужный генератор в зависимости от исполняющей среды.

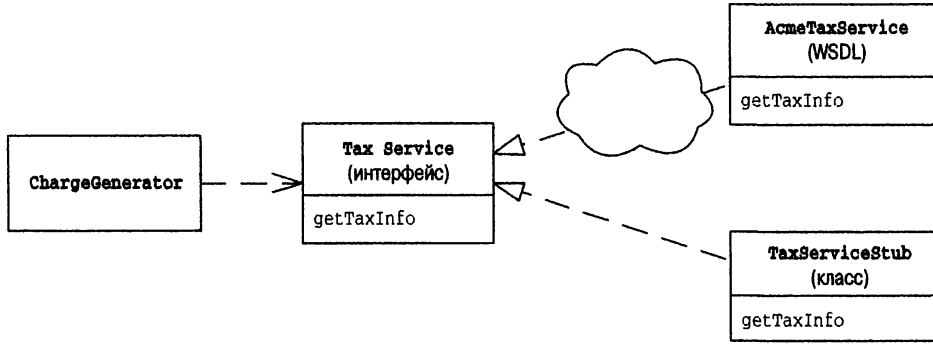
```
class Customer extends DomainObject...

   private Customer(String name, Long id) {
      super(id);
      this.name = name;
   }
   public Customer create(String name) {
      Long newObjId = IdGenerator.INSTANCE.nextId();
      Customer obj = new Customer(name, newObjId);
      obj.markNew();
      return obj;
   }
```

Фиктивная служба (Service Stub)

Дейвид Райс

Устраняет зависимость приложения от труднодоступных или проблемных служб на время тестирования



Функционирование большинства корпоративных приложений зависит от наличия доступа к сторонним службам наподобие систем оценки кредитоспособности, определения налоговых ставок и ценообразования. Спросите любого разработчика корпоративных систем, и он с негодованием расскажет вам, как это плохо — зависеть от ресурсов, которые полностью контролируются кем-то другим. Кто знает, что случится с этими ресурсами в будущем? Кроме того, поскольку большинство подобных служб являются удаленными, это не может не повлиять на стабильность и производительность приложения.

Все эти проблемы, как минимум, замедляют процесс разработки приложения. Нередки ситуации, когда разработчики вынуждены сидеть сложа руки и ждать, пока нужная служба вновь вернется к работе, или вставлять суррогатные фрагменты кода, чтобы компенсировать отсутствующие возможности службы, которые появятся в будущем. Что еще хуже (и к сожалению, весьма вероятно), зависимость приложения от внешних служб может сделать невозможным проведение тестов, когда эти службы недоступны, что полностью прервет процесс разработки.

Для ускорения и повышения качества процесса разработки реальную службу на время тестирования можно заменить **фиктивной службой**, которая будет выполняться в локальной системе, а значит, позволит избежать проблем с наличием доступа и его скоростью.

Принцип действия

Вначале нужно установить доступ к внешней службе с помощью **шлюза (Gateway, 483)**. Последний следует реализовать не как класс, а в виде **отделенного интерфейса (Separated Interface, 492)**, имеющего одну реализацию для обращения к реальной службе и, как минимум, еще одну реализацию, являющуюся **фиктивной службой**. Нужная реализация **шлюза** должна загружаться с помощью **дополнительного модуля (Plugin, 514)**. Не забывайте главный принцип написания **фиктивной службы**: она должна быть настолько

простой, насколько это возможно; излишняя сложность замедлит процесс разработки, чего мы, собственно, и пытаемся избежать.

В качестве примера рассмотрим процесс замены службы, определяющей величину налога с продажи. Такая служба принимает на вход место проживания покупателя, тип товара и стоимость покупки, а возвращает ставку налога с продажи, принятую в соответствующем штате США, и величину налога на указанную сумму. Самый простой способ заменить подобную службу **фиктивной службой** — написать две-три строки кода, которые будут высчитывать налог с продажи, используя единую ставку для всех товаров и покупателей.

Разумеется, настоящие законы налогообложения далеко не так просты. В некоторых штатах отдельные типы товаров освобождаются от налогов, поэтому определить комбинацию товара и штата, для которой не следует подсчитывать величину налога, можно только с помощью реальной службы. Между тем большая часть функциональности нашего приложения зависит от того, облагается налогом данный тип товара или нет, поэтому нужно усовершенствовать **фиктивную службу**, чтобы она учитывала подобные тонкости. Стараясь не усложнять задачу, добавим к **фиктивной службе** условное выражение, которое будет исключать из списка товаров, облагаемых налогом, одну фиксированную комбинацию товара и штата, и затем используем эту комбинацию во всех остальных вариантах тестирования. Таким образом, количество строк кода нашей фиктивной службы все еще не превышает числа пальцев на одной руке.

Более динамичная **фиктивная служба** могла бы поддерживать список комбинаций товаров и штатов, не облагаемых налогом, с возможностью добавления новых комбинаций во время тестирования. Тем не менее даже для такой реализации **фиктивной службы** понадобилось бы не более 10 строк кода. Не забывайте: наша цель состоит именно в том, чтобы ускорить процесс разработки, поэтому при любых обстоятельствах **фиктивная служба** должна оставаться достаточно простой.

Применение динамической **фиктивной службы** вызывает весьма интересный вопрос, связанный с ее зависимостью от вариантов тестирования. Пополнение списка товаров, не облагаемых налогом, осуществляется посредством метода, которого нет в интерфейсе **шлюза**, применяемого для доступа к реальной службе. Чтобы **фиктивную службу** можно было загрузить при помощи **дополнительного модуля**, упомянутый метод должен быть добавлен к **шлюзу**. Думаю, добавление нескольких строк кода никак не отяготит **шлюз**, особенно если это делается во имя тестирования. Обязательно убедитесь, что реализация **шлюза**, методы которой вызывают реальную службу, генерирует ошибки подтверждения во всех упомянутых методах во время тестирования.

Назначение

Фиктивная служба используется тогда, когда зависимость приложения от конкретной внешней службы значительно затрудняет процесс разработки и тестирования.

Следует отметить, что многие приверженцы экстремального программирования употребляют термин *объект-имитатор (mock object)*, имея в виду не что иное, как **фиктивную службу**. Мы считаем нужным употреблять именно термин "фиктивная служба", так как он используется дольше, нежели термин "объект-имитатор".

Пример: служба определения величины налога (Java)

Приложение, о котором идет речь в данном примере, использует службу определения величины налога, развернутую в виде Web-службы. Вначале создадим **шлюз**, чтобы коду домена не приходилось сталкиваться со всеми странностями, присущими Web-службам. Для облегчения загрузки будущих **фиктивных служб** шлюз будет реализован в виде **отделенного интерфейса**. И наконец, загрузка необходимой реализации службы будет осуществляться **дополнительным модулем**.

```
interface TaxService...

   public static final TaxService INSTANCE =
      (TaxService) PluginFactory.getPlugin(TaxService.class);
   public TaxInfo getSalesTaxInfo(String productCode, Address
↳addr, Money saleAmount);
```

Простая **фиктивная служба**, определяющая величину налога на основании единой процентной ставки вне зависимости от штата и типа товара, будет выглядеть как показано ниже.

```
class FlatRateTaxService implements TaxService...

   private static final BigDecimal FLAT_RATE = new BigDecimal(
↳"0.0500");
   public TaxInfo getSalesTaxInfo(String productCode, Address
↳addr, Money saleAmount) {
      return new TaxInfo(FLAT_RATE, saleAmount.multiply(
↳FLAT_RATE));
      }
```

Приведенная ниже **фиктивная служба** будет исключать из списка товаров, облагаемых налогом, определенную комбинацию товара и штата.

```
class ExemptProductTaxService implements TaxService...

   private static final BigDecimal EXEMPT_RATE = new BigDecimal(
↳"0.0000");
   private static final BigDecimal FLAT_RATE = new BigDecimal(
↳"0.0500");
   private static final String EXEMPT_STATE = "IL";
   private static final String EXEMPT_PRODUCT = "12300";
   public TaxInfo getSalesTaxInfo(String productCode, Address
↳addr, Money saleAmount) {
      if (productCode.equals(EXEMPT_PRODUCT) &&
↳addr.getStateCode().equals(EXEMPT_STATE)) {
         return new TaxInfo(EXEMPT_RATE, saleAmount.multiply(
↳EXEMPT_RATE));
      } else {
         return new TaxInfo(FLAT_RATE, saleAmount.multiply(
↳FLAT_RATE));
      }
   }
```

И наконец, рассмотрим более динамичную **фиктивную службу**, методы которой позволяют пополнять или же очищать список комбинаций товаров и штатов во время тестирования. Обратите внимание: при наличии подобных методов в приведенной ниже **фиктивной службе** нужно вернуться назад и добавить реализации этих же методов в более простые **фиктивные службы**, а также в реализацию, которая обращается к реальной Web-службе. Все методы, не используемые во время тестирования, должны генерировать ошибки подтверждения.

```
class TestTaxService implements TaxService...

   private static Set exemptions = new HashSet();
   public TaxInfo getSalesTaxInfo(String productCode, Address
↳addr, Money saleAmount) {
      BigDecimal rate = getRate(productCode, addr);
      return new TaxInfo(rate, saleAmount.multiply(rate));
   }
   public static void addExemption(String productCode, String
↳stateCode) {
      exemptions.add(getExemptionKey(productCode, stateCode));
   }
   public static void reset() {
      exemptions.clear();
   }
   private static BigDecimal getRate(String productCode, Address
↳addr) {
      if (exemptions.contains(getExemptionKey(productCode,
                  addr.getStateCode()))) {
         return EXEMPT_RATE;
      } else {
         return FLAT_RATE;
      }
   }
```

В данном примере не показана реализация **шлюза**, которая обращается к реальной службе определения величины налогов. Связывание интерфейса `TaxService` с упомянутой реализацией выполняется конфигурацией **дополнительного модуля**, предназначенной для работы в реальных условиях. Еще одна конфигурация **дополнительного модуля**, предназначенная для работы в режиме тестирования, связывает интерфейс `TaxService` с приведенной выше **фиктивной службой**.

Как уже отмечалось, доступ объектов приложения к службе определения величины налогов должен осуществляться посредством **шлюза**. Ниже приведен код класса `ChargeGenerator`, который вычисляет базовую стоимость покупки и затем обращается к службе определения величины налогов, чтобы подсчитать сумму налога на купленный товар.

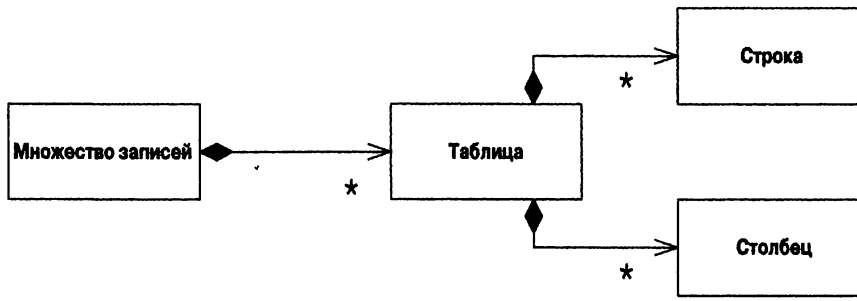
```
class ChargeGenerator...

   public Charge[] calculateCharges(BillingSchedule schedule) {
      List charges = new ArrayList();
      Charge baseCharge = new Charge(
↳schedule.getBillingAmount(), false);
      charges.add(baseCharge);
```

```
      TaxInfo info = TaxService.INSTANCE.getSalesTaxInfo(
↳schedule.getProduct(), schedule.getAddress(),
↳schedule.getBillingAmount());
      if (info.getStateRate().compareTo(
↳new BigDecimal(0)) > 0) {
         Charge taxCharge = new Charge(info.getStateAmount(),
↳true);
         charges.add(taxCharge);
      }
      return (Charge[]) charges.toArray(
↳new Charge[charges.size()]);
   }
```

Множество записей (Record Set)

Представление табличных данных в оперативной памяти



На протяжении последних 20 лет основным способом долгосрочного хранения данных являются таблицы реляционных баз данных. Наличие горячей поддержки со стороны больших и малых производителей баз данных, а также стандартного (в принципе) языка запросов привело к тому, что практически каждая новая разработка опирается на реляционные данные.

На гребне успеха реляционных баз данных рынок программного обеспечения пополнился массой инфраструктур, предназначенных для быстрого построения пользовательских интерфейсов. Все эти инфраструктуры основываются на использовании реляционных данных и предоставляют наборы элементов управления, которые значительно облегчают просмотр и манипулирование данными, не требуя написания громоздкого кода.

К сожалению, подобные средства имеют один существенный недостаток. Предоставляя возможность легкого отображения и выполнения простых обновлений, они не содержат нормальных элементов, в которые можно было бы поместить бизнес-логику. Наличие каких-либо правил проверки, помимо простейших "правильны ли данные?", а также любых бизнес-правил и вычислений здесь просто не предусмотрено. Разработчикам не остается ничего иного, как помещать подобную логику в базу данных в виде хранимых процедур или же внедрять ее прямо в код пользовательского интерфейса.

Типовое решение **множество записей** значительно облегчает жизнь разработчиков, предоставляя структуру данных, которая хранится в оперативной памяти и в точности напоминает результат выполнения SQL-запроса, однако может быть сгенерирована и использована другими частями системы.

Принцип действия

Обычно **множество записей** не приходится конструировать самому. Как правило, подобные классы прилагаются к используемой программной платформе. В качестве примера можно привести объекты `DataSet` библиотеки ADO.NET и объекты `RowSet` библиотеки JDBC 2.0.

Первой ключевой особенностью **множества записей** является то, что его структура в точности копирует результат выполнения запроса к базе данных. Это значит, что вы можете использовать классический двухуровневый подход выполнения запроса и помещения данных прямо в соответствующие элементы пользовательского интерфейса с теми же преимуществами, которые предоставляют подобные двухуровневые средства. Вторая ключевая особенность заключается в том, что вы можете создать собственное **множество записей** или же использовать **множество записей**, полученное в результате выполнения запроса к базе данных, и легко манипулировать им в коде логики домена.

Несмотря на наличие готовых классов для реализации **множества записей**, их можно создать и самому. Однако следует отметить, что данное типовое решение имеет смысл только при наличии средств пользовательского интерфейса, предназначенных для отображения и считывания данных, которые также приходится создавать самостоятельно. В любом случае в качестве хорошего примера реализации **множества записей** можно назвать построение списка коллекций, весьма распространенное в динамически типизированных языках сценариев.

При работе с **множеством записей** очень важно, чтобы последнее было легко отсоединить от источника данных. Это позволяет передавать **множество записей** по сети, не беспокоясь о наличии соединения с базой данных. Более того, если **множество записей** легко поддается сериализации, оно может выступать в качестве **объекта переноса данных (Data Transfer Object, 419)**.

Необходимость отсоединения приводит к очевидной проблеме: как выполнять обновление **множества записей**? Все больше и больше платформ реализуют **множество записей** в виде **единицы работы (Unit of Work, 205)**, благодаря чему все изменения **множества записей** могут быть внесены в отсоединенном режиме и затем зафиксированы в базе данных. Как правило, источник данных может воспользоваться **оптимистической автономной блокировкой (Optimistic Offline Lock, 434)**, чтобы проверить параллельные транзакции на наличие конфликтов и в случае отсутствия последних записать все внесенные изменения в базу данных.

Явный интерфейс

Большинство реализаций **множества записей** используют *неявный интерфейс (implicit interface)*. В этом случае, чтобы извлечь из **множества записей** требующиеся сведения, необходимо вызвать универсальный метод с аргументом, указывающим на то, какое поле нам нужно. Например, чтобы получить сведения о пассажире, забронировавшем билет на указанный авиарейс, необходимо воспользоваться выражением наподобие `aReservation["passenger"]`. Применение явного интерфейса требует реализации отдельного

класса `Reservation` с конкретными методами и атрибутами, а извлечение сведений о пассажире выполняется с помощью выражений типа `aReservation.passenger`.

Неявные интерфейсы обладают чрезвычайной гибкостью. Универсальное **множество записей** может быть использовано для любых видов данных, что избавляет от необходимости написания нового класса при определении каждого нового типа **множества записей**. Несмотря на подобные преимущества, я считаю неявные интерфейсы Очень Плохим Решением. Как я узнаю, какое слово нужно использовать для того, чтобы добраться к сведениям о пассажире, забронировавшем билет, — "passenger" ("пассажир"), "guest" ("клиент") или, может быть, "flyer" ("летящий")? Единственное, что я могу сделать, — это "прочесать" весь исходный код приложения в поисках мест, где создаются и используются объекты бронирования. Если же у меня есть явный интерфейс, я могу открыть определение класса `Reservation` и посмотреть, какое свойство мне нужно.

Данная проблема еще более критична для статически типизированных языков программирования. Чтобы узнать фамилию пассажира, забронировавшего билет, мне придется прибегнуть к кошмарному выражению вида `((Person)aReservation ["passenger"]).lastName`. Поскольку компилятор утрачивает всю информацию о типах, для извлечения нужных сведений тип данных приходится указывать вручную. В отличие от этого, явный интерфейс сохраняет информацию о типе, поэтому для извлечения фамилии пассажира я могу воспользоваться гораздо более простым выражением `aReservation.passenger.lastName`.

В связи с этим я предпочитаю избегать неявных интерфейсов (а также не менее ужасного приема передачи данных с использованием словарей). Я не приветствую применение неявных интерфейсов и к **множествам записей**, однако в данном случае ситуация облегчается тем, что элементы **множества записей** обычно соответствуют существующим столбцам таблицы базы данных. Более того, имена столбцов фигурируют в SQL-выражениях, создающих **множество записей**, поэтому найти имя нужного свойства отнюдь не так сложно, как кажется на первый взгляд.

Как бы там ни было, явные интерфейсы все-таки предпочтительнее. В библиотеке ADO.NET возможность применения явного интерфейса обеспечивают строго типизированные объекты `DataSet` — сгенерированные классы, которые предоставляют явный и полностью типизированный интерфейс к **множеству записей**. Поскольку объект `DataSet` может содержать в себе множество таблиц и отношений, в которых они находятся, строго типизированные объекты `DataSet` включают в себя свойства, использующие информацию об отношениях между таблицами. Генерация классов `DataSet` выполняется на основе определений схем XML (XML Schema Definition — XSD).

В настоящее время более распространены неявные интерфейсы, поэтому в примерах данной книги я использовал нетипизированные объекты `DataSet`. Несмотря на это, для написания реальных приложений с использованием библиотеки ADO.NET я рекомендую применять типизированные объекты `DataSet`. Если же используемая среда разработки не поддерживает ADO.NET, предлагаю воспользоваться генерацией кода для создания собственных **множеств записей** с явными интерфейсами.

Назначение

Как мне кажется, настоящая ценность **множества записей** раскрывается в тех средах разработки, которые могут применять его в качестве стандартного способа манипулирования данными. Большинство элементов управления пользовательского интерфейса способны работать с **множествами записей**, и это является неоспоримым аргументом в пользу применения их самих. При наличии подобных сред разработки для организации логики домена следует применять **модуль таблицы (Table Module, 148)**: извлеките **множество записей** из базы данных, передайте его **модулю таблицы** для подсчета необходимой информации, отошлите пользовательскому интерфейсу для отображения и редактирования и затем снова передайте **модулю таблицы** для проверки правильности сделанных изменений. Затем обновление **множества записей** следует зафиксировать в базе данных.

Нельзя не отметить, что громкий успех средств, основанных на использовании **множества записей**, по большей части был обусловлен извечным господством реляционных структур данных и языка запросов SQL и, что гораздо важнее, отсутствием каких-либо серьезных альтернатив. К счастью, в наши дни ситуация сдвинулась с мертвой точки. Причиной этому стало широкое распространение языка XML и появление языка запросов XPath, и я думаю, что в будущем на рынке программного обеспечения появится множество средств, использующих иерархические структуры данных точно так же, как сегодняшние средства используют **множество записей**. Вероятно, **множество записей** следует рассматривать как частный случай более универсального типового решения: нечто наподобие **универсальной структуры данных (Generic Data Structure)**. Впрочем, я оставлю разработку этого типового решения до тех пор, пока оно не приобретет реальный смысл.

Список основных источников информации

1. Alexander et al. *A Pattern Language* — Oxford, 1977.

   Источник вдохновения многих людей, принявших деятельное участие в продвижении концепции типовых решений. Я не так восхищаюсь этой книгой, как другие. Тем не менее, она стоит того, чтобы ее прочитали хотя бы ради знакомства с историческим подходом, давшим толчок такому большому количеству идей.

2. Alpert, Brown, and Woolf. *Design Patterns Smalltalk Companion.* — Addison-Wesley, 1998.

   Малоизвестная за пределами сообщества программистов, работающих со Smalltalk, эта книга расширяет и разъясняет целый ряд классических типовых решений.

3. Alur, Crupi, and Malks. *Core J2EE Patterns: Best Practices and Design Strategies.* — Prentice Hall, 2001.

   Одна из книг "новой волны", вдохнувших новую жизнь в старые формы. Хотя описанные в ней типовые решения разработаны специально для платформы J2EE, многие из них могут применяться и в других местах.

4. *http://www.ambysoft.com/mappingObjects.html*

   Масса полезных идей касательно объектно-реляционного отображения.

5. Beck. *Extreme Programming Explained.* — Addison-Wesley, 2000. (Бек, К. *Экстремальное программирование. Библиотека программиста.*: Пер. с англ.— СПб.: Питер, 2002.)

   Альфа и омега экстремального программирования. Эту книгу должен прочитать каждый, кто интересуется разработкой программного обеспечения.

6. Beck. *Smalltalk Best Practice Patterns.* — Prentice Hall, 1997.

   Слово "Smalltalk" в названии этой книги отпугивает многих потенциальных читателей, и совершенно напрасно. Она содержит больше хороших советов по объектно-ориентированным языкам программирования, чем многие

книги, написанные специально для этих целей. Данный шедевр имеет единственный недостаток: после того как он прочитан, становится ясно, как много мы теряем, не программируя на Smalltalk.

7. Beck. *Test-Driven Development: By Example.* — Addison-Wesley, 2003.

   Эта книга Кента Бека представляет собой справочник по тесно связанным между собой процессам тестирования и рефакторинга, правильное чередование которых позволяет значительно улучшить структуру приложения.

8. Bernstein and Newcomer. *Principles of Transaction Processing.* — Morgan Kaufmann, 1997.

   Отличное введение в загадочный мир транзакций.

9. Brown et al. *Enterprise Java Programming with IBM Websphere.* — Addison-Wesley, 2001.

   Две трети этой книги представляют собой руководство по использованию программного обеспечения, зато оставшаяся треть содержит больше хороших советов по проектированию, чем многие толстенные тома, целиком посвященные этой теме.

10. *http://members.aol.com/kgb1001001/Chasms.htm*

    Одна из самых первых и самых лучших статей, посвященных объектно-реляционному отображению.

11. Cockburn. *Writing Effective Use Cases.* — Addison-Wesley, 2001.

    Один из лучших справочников по вариантам использования.

12. Cockburn. *Prioritizing Forces in Software Design.* — In: Pattern Languages of Program Design 2 / Vlissides, Coplien, and Kerth (eds.). — Addison-Wesley, 1996.

    Дискуссия на тему границ приложения.

13. Coleman, Arnold, and Bodoff. *Object-Oriented Development: The Fusion Method, Second Edition.* — Prentice Hall, 2001.

    Книга была написана еще до появления UML, а потому сегодня большая часть материала представляет лишь исторический интерес. Тем не менее содержащееся в ней обсуждение модели интерфейса может весьма пригодиться разработчикам слоя служб.

14. *http://martinfowler.com/apsupp/spec.pdf*

    Описание типового решения **спецификация (Specification)**.

15. Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* — Addison-Wesley, 2004.

    Книга посвящена разработке моделей предметной области — одному из самых важных и самых сложных аспектов разработки корпоративных приложений.

16. *http://martinfowler.com/ap2/timeNarrative.html*

Типовые решения для сохранения истории объектов, состояние которых меняется с течением времени.

17. Fowler. *Analysis Patterns.* — Addison-Wesley, 1997.

Типовые решения для моделирования предметной области.

18. Fowler. *Refactoring.* — Addison-Wesley, 1999.

Метод улучшения структуры существующего кода.

19. *http://martinfowler.com/articles/continuousIntegration.html*

В этом очерке описано, как выполнять автоматическую сборку программного обеспечения несколько раз в течение дня.

20. Gamma, Helm, Johnson, and Vlissides. *Design Patterns.* — Addison-Wesley, 1995. (Гамма Э., Хелм Р., Джонсон Р., Влиссидес Д. *Приемы объектно-ориентированного программирования. Паттерны проектирования.* — СПб. : Питер, 2003.)

Знаменитая книга о типовых решениях с множеством конструктивных идей.

21. Hay. *Data Model Patterns.* — Dorset House, 1995.

Типовые решения для проектирования концептуальных моделей данных с реляционной точки зрения.

22. Jacobson et al. *Object-Oriented Software Engineering.* — Addison-Wesley, 1992.

Одна из первых книг по объектно-ориентированному проектированию. Содержит описание вариантов использования модели "интерфейс–контроллер–сущность" и подхода к проектированию с ее применением.

23. *http://www.objectarchitects.de/ObjectArchitects/orpatterns/index.htm*

Чудесный источник по объектно-реляционному отображению.

24. Kirtland. *Designing Component-Based Applications.* — Microsoft Press, 1998.

Описание архитектуры Windows DNA (Distributed interNet Architecture).

25. Knight and Dai. *Objects and the Web* // IEEE Software. — March/April 2002.

Чудесная статья, посвященная системе "модель–представление–контроллер", ее формированию и использованию в Web-приложениях.

26. Larman. *Applying UML and Patterns, Second Edition.* — Prentice Hall, 2001. (Ларман К. *Применение UML и шаблонов проектирования, 2-е изд.* — К. : Диалектика, 2002.)

Сегодня это моя любимая книга по введению в объектно-ориентированное проектирование.

27. Lea. *Concurrent Programming in Java, Second Edition.* — Addison-Wesley, 2000.

Обязательно прочитайте эту книгу, если хотите заняться написанием многопоточных приложений.

28. Marinescu. *EJB Design Patterns.* — New York : John Wiley, 2002.

Сравнительно новая книга по типовым решениям для платформы Enterprise Java Beans.

29. Martin and Odell. *Object Oriented Methods: A Foundation (UML Edition).* — Prentice Hall, 1998.

Данная книга описывает объектное моделирование с концептуальной точки зрения, а также исследует основы моделирования вообще.

30. Nilsson. *.NET Enterprise Design with Visual Basic .NET and SQL Server 2000.* — Sams, 2002.

Серьезная книга, посвященная новой архитектуре для платформы Microsoft.

31. Vlissides, Coplien, and Kerth (eds.). *Pattern Languages of Program Design 2.* — Addison-Wesley, 1996.

Сборник статей, посвященных типовым решениям.

32. Martin, Buschmann, and Riehle (eds.). *Pattern Languages of Program Design 3.* — Addison-Wesley, 1998.

Сборник статей, посвященных типовым решениям.

33. Buschmann et al. *Pattern-Oriented Software Architecture.* — Wiley, 2000.

Лучшая книга по типовым решениям более общего характера.

34. Riehle, Seberski, Baumer, Megert, and Zullighoven. *Serializer.* — In: Pattern Languages of Program Design 3 / Martin, Buschmann, and Riehle (eds.). — Addison-Wesley, 1998.

Детальное описание сериализации объектных структур в различные форматы.

35. Schmidt, Stal, Rohnert, and Buschmann. *Pattern-Oriented Software Architecture, Volume 2.* — New York : John Wiley, 2000.

Типовые решения для моделирования распределенных систем и систем с параллельной обработкой заданий. Данная книга предназначена скорее для проектировщиков серверов приложений, чем для их пользователей, однако небольшие знания на эту тему все же не помешают.

36. Snodgrass. *Developing Time-Oriented Database Applications in SQL.* — Morgan-Kaufmann, 1999.

Рассматривается, как отслеживать историю данных в реляционной СУБД.

37. *http://jakarta.apache.org/struts/*

Описание набирающей популярность инфраструктуры Jakarta для платформы Java, предназначенной для представления данных в Web.

38. Waldo, Wyant, Wollrath, and Kendall. *A Note on Distributed Computing.* — SMLI TR-94-29. — Sun Microsystems, 1994. *http://research.sun.com/technical-reports/1994/smli-tr-94-29.pdf.*

Классическая статья на тему угрожающей бессмысленности выражения "прозрачная реализация распределенных объектов".

39. *http://c2.com/cgi/wiki*

Оригинальная система общения Wiki Wiki Web, разработанная Вэрдом Каннинхэмом (Ward Cunningham). Достаточно сумбурный, тем не менее очаровательный Web-сайт, открытый для самых сумасшедших людей с самыми сумасшедшими идеями.

40. Woolf. *Null Object.* — In: Pattern Languages of Program Design 3 / Martin, Buschmann, and Riehle (eds.). — Addison-Wesley, 1998.

Описание типового решения **null-объект (Null Object)**.

41. *http://www.joeyoder.com/Research/objectmappings*

Хороший источник по объектно-реляционным типовым решениям.

Предметный указатель

.

.NET, 125

A

ACID, 96
ADO.NET, 172

E

EJB 2.0, 124
Enterprise Application Integration (EAI), 485
Enterprise Java Beans, 124

G

Globally Unique Identifier (GUID), 241

J

J2EE, 124

W

Web-службы, 126; 414

А

Автономный параллелизм, 87; 100; 87; 100
Активная запись (Active Record), 182
  назначение, 184
  пример: простой класс Person (Java), 184
  принцип действия, 182

Б

Базы данных
  отображение на объектную модель, 59
    архитектурные решения, 59
    взаимное отображение объектов и реляционных структур, 67
    другие проблемы, 78
    использование метаданных, 75
    реализация отображения, 73
    соединение с базой данных, 76
    считывание данных, 66
    функциональные проблемы, 64
Бизнес-логика, 46
Бизнес-транзакция, 99
Блокировка
  оптимистическая, 92
  пессимистическая, 92
Блокировка с низкой степенью детализации (Coarse-Grained Lock), 457
  назначение, 460
  пример: общая оптимистическая автономная блокировка (Java), 460
  пример: общая пессимистическая автономная блокировка (Java), 466
  пример: оптимистическая автономная блокировка корневого элемента (Java), 467
  принцип действия, 457

В

Взаимоблокировка, 94
Виртуальный прокси-объект, 221
Внедренное значение (Embedded Value), 288
  дополнительные источники информации, 290
  назначение, 289
  пример:простой объект-значение (Java), 290
  принцип действия, 289
Волнообразная загрузка, 222

Д

Двухэтапное представление (Two Step View), 383
  назначение, 385
  пример: двухэтапное применение XSLT (XSLT), 390
  пример:страницы JSP и пользовательские дескрипторы (Java), 393
  принцип действия, 383
Деньги (Money), 502
  назначение, 506

пример: класс Money (Java), 506
принцип действия, 503
Диспетчер значения, 222
Длинная транзакция, 96
Дополнительный модуль (Plugin), 514
назначение, 515
пример: генератор идентификаторов (Java), 516
принцип действия, 514
Достоверность, 89

Е

Единица работы (Unit of Work), 205
назначение, 211
пример: регистрация посредством изменяемого объекта (Java), 212
принцип действия, 206

Ж

Живучесть системы, 8

З

Загрузка по требованию (Lazy Load), 220
назначение, 223
пример: виртуальный прокси-объект (Java), 224
пример: инициализация по требованию (Java), 224
пример: использование диспетчера значения (Java), 226
пример: использование фиктивных объектов (C#), 227
принцип действия, 221
Запрос, 89

И

Изолированность данных, 91
Изолированные потоки, 90
Инициализация по требованию, 221
Интерфейс удаленного доступа (Remote Facade), 405
назначение, 410
пример: Web-служба (C#), 414
пример: использование компонента сеанса Java в качестве интерфейса удаленного доступа (Java), 410
принцип действия, 406
Интерфейсы
локального вызова, 112
удаленного вызова, 112
Источник данных
архитектурные типовые решения источников данных
активная запись (Active Record), 182
преобразователь данных (Data Mapper), 187
шлюз записи данных (Row Data Gateway), 175
шлюз таблицы данных (Table Data Gateway), 167
для модели предметной области, 122
для модуля таблицы, 122
для сценария транзакции, 121
логика, 46

К

Катализаторы сложности, 50
Ключ
простой, 238
составной, 238
Коллекция объектов (Identity Map), 216
назначение, 219
пример: методы для работы с коллекцией объектов (Java), 219
принцип действия, 216
Контекст выполнения, 89
Контроллер запросов (Front Controller), 362
дополнительные источники информации, 364
назначение, 364
пример: простое отображение (Java), 365
принцип действия, 362
Контроллер приложения (Application Controller), 397
дополнительные источники информации, 400
назначение, 400
пример: модель состояний контроллера приложения (Java), 400
принцип действия, 398
Контроллер страниц (Page Controller), 350
назначение, 352
пример: использование страницы JSP в качестве обработчика запросов (Java), 355
пример: обработка запросов страницей сервера с применением механизма разделения кода и представления (C#), 358
пример: простое отображение с помощью контроллера-сервлета и представления JSP (Java), 352
принцип действия, 351

Л

Логика домена, 46
выбор типового решения, 55
организация, 51

типовые решения
модель предметной области (Domain Model), 140
модуль таблицы (Table Module), 148
слой служб (Service Layer), 156
сценарий транзакции (Transaction Script), 133
уровень служб, 56

M

Метаданные, 75
Множество записей (Record Set), 523
назначение, 526
принцип действия, 524
Модель
Core J2EE, 128
Microsoft DNA, 128
Брауна, 127
Маринеску, 129
Нильссона, 129
Модель предметной области (Domain Model), 140
дополнительные источники информации, 143
источник данных, 122
назначение, 143
пример: определение зачтенного дохода (Java), 144
принцип действия, 140
Модель–представление–контроллер (Model View Controller), 347
назначение, 350
принцип действия, 348
Модуль таблицы (Table Module), 148
источник данных, 122
назначение, 151
пример: определение зачтенного дохода (C#), 152
принцип действия, 149

H

Наследование, 71
Наследование с одной таблицей (Single Table Inheritance), 297
назначение, 298
пример: общая таблица игроков (C#), 299
принцип действия, 298
Наследование с таблицами для каждого класса (Class Table Inheritance), 305
дополнительные источники информации, 307
назначение, 306
пример: семейство игроков (C#), 307
принцип действия, 305
Наследование с таблицами для каждого конкретного класса (Concrete Table Inheritance), 313
назначение, 315
пример: конкретные классы игроков (C#), 316
принцип действия, 314
Неповторяемое чтение, 98
Несогласованное чтение, 88
предотвращение возможности, 93
Неявная блокировка (Implicit Lock), 468
назначение, 470
пример: неявная пессимистическая автономная блокировка (Java), 470
принцип действия, 469

O

Объект запроса (Query Object), 335
дополнительные источники информации, 337
назначение, 337
пример: простой объект запроса (Java), 337
принцип действия, 336
Объект переноса данных (Data Transfer Object), 419
дополнительные источники информации, 424
назначение, 424
пример: передача информации об альбомах (Java), 425
пример: сериализация с использованием XML (Java), 429
принцип действия, 419
Объект-значение (Value Object), 500
назначение, 502
принцип действия, 501
Операции чтения с временными признаками (Temporal Reads), 94
Оптимистическая автономная блокировка (Optimistic Offline Lock), 434
назначение, 439
пример: слой домена с преобразователями данных (Java), 439
принцип действия, 435
Отделенный интерфейс (Separated Interface), 492
назначение, 494
принцип действия, 493
Отображение внешних ключей (Foreign Key Mapping), 258
назначение, 261
пример: коллекция ссылок (C#), 266
пример: многотабличный поиск (Java), 265
пример: однозначная ссылка (Java), 262
принцип действия, 258

Отображение зависимых объектов (Dependent Mapping), 283
назначение, 285
пример: альбомы и композиции (Java), 285
принцип действия, 283
Отображение метаданных (Metadata Mapping), 325
назначение, 327
пример: использование метаданных и метода отражения (Java), 328
принцип действия, 326
Отображение с помощью таблицы ассоциаций (Association Table Mapping), 269
назначение, 270
пример: загрузка сведений о нескольких служащих посредством одного запроса (Java), 278
пример: использование SQL для непосредственного обращения к базе данных (Java), 274
пример: служащие и профессиональные качества (C#), 271
принцип действия, 270
Отсроченная транзакция, 97

П

Перенос сеанса, 109
Пессимистическая автономная блокировка (Pessimistic Offline Lock), 445
назначение, 450
пример: простой диспетчер блокировки (Java), 450
принцип действия, 446
Повторяемое чтение, 98
Поле идентификации (Identity Field), 237
дополнительные источники информации, 243
назначение, 242
пример: использование составного ключа (Java), 246
пример: использование таблицы ключей (Java), 244
пример: числовой ключ (C#), 243
принцип действия, 237
Поток
изолированный, 90
определение, 90
Представление по шаблону (Template View), 368
назначение, 372
пример: использование страницы JSP в качестве представления с вынесением контроллера в отдельный объект (Java), 373
пример: страница сервера ASP.NET (C#), 375
принцип действия, 369
Представление с преобразованием (Transform View), 379
назначение, 380
пример: простое преобразование (Java), 381
принцип действия, 379
Преобразователи наследования (Inheritance Mappers), 322
назначение, 324
принцип действия, 323
Преобразователь (Mapper), 489
назначение, 490
принцип действия, 490
Преобразователь данных (Data Mapper), 187
назначение, 192
пример: отделение методов поиска (Java), 198
пример: простой преобразователь данных (Java), 193
пример: создание пустого объекта (Java), 201
принцип действия, 187
Привязка к серверу, 109
Пример
Web-служба (C#), 414
альбомы и композиции (Java), 285
виртуальный прокси-объект (Java), 224
выбор стратегий хранилища (Java), 345
генератор идентификаторов (Java), 516
двухэтапное применение XSLT (XSLT), 390
загрузка сведений о нескольких служащих посредством одного запроса (Java), 278
запись о сотруднике (Java), 178
инициализация по требованию (Java), 224
использование SQL для непосредственного обращения к базе данных (Java), 274
использование диспетчера данных для объекта домена (Java), 181
использование диспетчера значения (Java), 226
использование компонента сеанса Java в качестве интерфейса удаленного доступа (Java), 410
использование метаданных и метода отражения (Java), 328
использование объектов ADO.NET DataSet (C#), 172
использование составного ключа (Java), 246
использование страницы JSP в качестве обработчика запросов (Java), 355
использование страницы JSP в качестве представления с вынесением контроллера в отдельный объект (Java), 373
использование таблицы ключей (Java), 244
использование фиктивных объектов (C#), 227

класс Money (Java), 506
класс PersonGateway (C#), 170
коллекция ссылок (C#), 266
конкретные классы игроков (C#), 316
методы для работы с коллекцией объектов (Java), 219
многотабличный поиск (Java), 265
модель состояний контроллера приложения (Java), 400
неявная пессимистическая автономная блокировка (Java), 470
обработка запросов страницей сервера с применением механизма разделения кода и представления (C#), 358
общая оптимистическая автономная блокировка (Java), 460
общая пессимистическая автономная блокировка (Java), 466
общая таблица игроков (C#), 299
объект NullEmployee (C#), 513
объект домена (Java), 491
однозначная ссылка (Java), 262
определение зачтенного дохода (C#), 152
определение зачтенного дохода (Java), 136; 144; 161; 136; 144; 161
оптимистическая автономная блокировка корневого элемента (Java), 467
отделение методов поиска (Java), 198
передача информации об альбомах (Java), 425
поиск подчиненных заданного сотрудника (Java), 344
простое отображение (Java), 365
простое отображение с помощью контроллера-сервлета и представления JSP (Java), 352
простое преобразование (Java), 381
простой диспетчер блокировки (Java), 450
простой класс Person (Java), 184
простой объект запроса (Java), 337
простой объект-значение (Java), 290
простой преобразователь данных (Java), 193
регистрация посредством изменяемого объекта (Java), 212
реестр с единственным экземпляром (Java), 498
реестр, уникальный в пределах потока (Java), 499
семейство игроков (C#), 307
сериализация иерархии отделов в формат XML (Java), 294
сериализация с использованием XML (Java), 429
слой домена с преобразователями данных (Java), 439
служащие и профессиональные качества (C#), 271
служба определения величины налога (Java), 521
создание пустого объекта (Java), 201
создание шлюза к службе отправки сообщений (Java), 485
страница сервера ASP.NET (C#), 375
страницы JSP и пользовательские дескрипторы (Java), 393
числовой ключ (C#), 243
Процесс, 90

Р

Расширение блокировки, 97
Реестр (Registry), 495
  назначение, 497
  пример: реестр с единственным экземпляром (Java), 498
  пример: реестр, уникальный в пределах потока (Java), 499
  принцип действия, 495
Ресурсы транзакций, 96

С

Сеанс, 89
  без состояния, 105
  с состоянием, 105
  состояние, 107
Сервер
  без состояния, 105
  с состоянием, 106
Сериализованный крупный объект (Serialized LOB), 292
  назначение, 294
  пример: сериализация иерархии отделов в формат XML (Java), 294
  принцип действия, 292
Системная транзакция, 99
Слой служб (Service Layer), 156
  дополнительные источники информации, 160
  назначение, 160
  пример: определение зачтенного дохода (Java), 161
  принцип действия, 157
Составная сущность (Composite Entity), 125
Состояние сеанса, 107

Сохранение состояния сеанса в базе данных (Database Session State), 479
  назначение, 481
  принцип действия, 479
Сохранение состояния сеанса на стороне клиента (Client Session State), 473
  назначение, 474
  принцип действия, 473
Сохранение состояния сеанса на стороне сервера (Server Session State), 475
  назначение, 478
  принцип действия, 475
Стратегии распределенных вычислений, 111
  интерфейсы локального и удаленного вызова, 112
  интерфейсы распределения, 116
  когда без распределения не обойтись, 114
  соблазны модели распределенных объектов, 111
  сужение границ распределения, 115
Супертип слоя (Layer Supertype), 491
  назначение, 491
  пример: объект домена (Java), 491
  принцип действия, 491
Схема
  поток на запрос, 103
  процесс на запрос, 103
  процесс на сеанс, 102
Сценарий транзакции (Transaction Script), 133
  задача определения зачтенного дохода, 135
  источник данных, 121
  назначение, 135
  пример: определение зачтенного дохода (Java), 136
  принцип действия, 133

Т

Таблица ключей, 241
Типовые решения
  Composite Entity, 125
  архитектурные, 59
  архитектурные типовые решения источников данных
    активная запись (Active Record), 182
    преобразователь данных (Data Mapper), 187
    шлюз записи данных (Row Data Gateway), 175
    шлюз таблицы данных (Table Data Gateway), 167
  базовые
    деньги (Money), 502
    дополнительный модуль (Plugin), 514
    множество записей (Record Set), 523
    объект-значение (Value Object), 500
    отделенный интерфейс (Separated Interface), 492
    преобразователь (Mapper), 489
    реестр (Registry), 495
    супертип слоя (Layer Supertype), 491
    фиктивная служба (Service Stub), 519
    частный случай (Special Case), 511
    шлюз (Gateway), 483
взаимное отображение объектов и реляционных структур, 67
  наследование, 71
  отображение связей, 67
входной контроллер, 86
граница приложения (Application Boundary), 160
задача автономного параллелизма
  блокировка с низкой степенью детализации (Coarse-Grained Lock), 457
  неявная блокировка (Implicit Lock), 468
  оптимистическая автономная блокировка (Optimistic Offline Lock), 434
  пессимистическая автономная блокировка (Pessimistic Offline Lock), 445
интерфейс доступа посредством компонентов сеанса (Session Facade), 159
логика домена
  модель предметной области (Domain Model), 140
  модуль таблицы (Table Module), 148
  слой служб (Service Layer), 156
  сценарий транзакции (Transaction Script), 133
моделирование поведения
  единица работы (Unit of Work), 205
  загрузка по требованию (Lazy Load), 220
  коллекция объектов (Identity Map), 216
моделирование структуры
  внедренное значение (Embedded Value), 288
  наследование с одной таблицей (Single Table Inheritance), 297
  наследование с таблицами для каждого класса (Class Table Inheritance), 305
  наследование с таблицами для каждого конкретного класса (Concrete Table Inheritance), 313
  отображение внешних ключей (Foreign Key Mapping), 258
  отображение зависимых объектов (Dependent Mapping), 283

отображение с помощью таблицы ассоциаций (Association Table Mapping), 269
поле идентификации (Identity Field), 237
преобразователи наследования (Inheritance Mappers), 322
сериализованный крупный объект (Serialized LOB), 292
наблюдатель (Observer), 165
операции чтения с временными признаками (Temporal Reads), 94
отображение с использованием метаданных
объект запроса (Query Object), 335
отображение метаданных (Metadata Mapping), 325
хранилище (Repository), 341
представление, 84
представление данных в Web
двухэтапное представление (Two Step View), 383
контроллер запросов (Front Controller), 362
контроллер приложения (Application Controller), 397
контроллер страниц (Page Controller), 350
модель–представление–контроллер" (Model View Controller), 347
представление по шаблону (Template View), 368
представление с преобразованием (Transform View), 379
распределенная обработка данных
интерфейс удаленного доступа (Remote Facade), 405
объект переноса данных (Data Transfer Object), 419
сохранение состояния сеанса
сохранение состояния сеанса в базе данных (Database Session State), 479
сохранение состояния сеанса на стороне клиента (Client Session State), 473
сохранение состояния сеанса на стороне сервера (Server Session State), 475
Транзакции, 95
бизнес-транзакции, 99
длинные, 96
запроса, 97
отсроченные, 97
системные, 99
Транзакция запроса, 97

У

Удаленный вызов процедур, 116
Упорядочиваемая транзакция, 97
Управление параллельными заданиями, 87
ACID-свойства транзакций, 96
изолированность и устойчивость данных, 91
контексты выполнения, 89
параллельные операции и серверы приложений, 102
предотвращение возможности несогласованного чтения данных, 93
проблемы параллелизма, 88
разрешение взаимоблокировок, 94
ресурсы транзакций, 96
системные транзакции и бизнес-транзакции, 99
стратегии блокирования, 91
типовые решения задачи обеспечения автономного параллелизма, 101
транзакции, 95
уровни изоляции, 97
Устойчивость данных, 91
Утраченные изменения, 88

Ф

Фантомные записи, 98
Фиктивная служба (Service Stub), 519
назначение, 520
пример: служба определения величины налога (Java), 521
принцип действия, 519
Фиктивный объект, 222

Х

Хранилище (Repository), 341
дополнительные источники информации, 344
назначение, 343
пример: выбор стратегий хранилища (Java), 345
пример: поиск подчиненных заданного сотрудника (Java), 344
принцип действия, 342
Хранимые данные, 107
Хранимые процедуры, 126

Ч

Частный случай (Special Case), 511
дополнительные источники информации, 512
назначение, 512
пример: объект NullEmployee (C#), 513
принцип действия, 512

Чтение мусора, 98
Чтение нефиксированных данных, 98
Чтение фиксированных данных, 98

Ш

Шлюз (Gateway), 483
  назначение, 484
  пример: создание шлюза к службе отправки сообщений (Java), 485
  принцип действия, 484
Шлюз записи данных (Row Data Gateway), 175
  назначение, 176
  пример: запись о сотруднике (Java), 178
  пример: использование диспетчера данных для объекта домена (Java), 181
  принцип действия, 175
Шлюз таблицы данных (Table Data Gateway), 167
  дополнительные источники информации, 169
  назначение, 168
  пример: использование объектов ADO.NET DataSet (C#), 172
  пример: класс PersonGateway (C#), 170
  принцип действия, 167

Список типовых решений

Активная запись (Active Record, 182). Объект, выполняющий роль оболочки для строки таблицы или представления базы данных. Он инкапсулирует доступ к базе данных и добавляет к данным логику домена.

Блокировка с низкой степенью детализации (Coarse-Grained Lock, 457). Блокирует группу взаимосвязанных объектов как единый элемент.

Внедренное значение (Embedded Value, 288). Отображает объект на несколько полей таблицы, соответствующей другому объекту.

Двухэтапное представление (Two Step View, 383). Выполняет визуализацию данных домена в два этапа: вначале формирует некое подобие логической страницы, после чего преобразует логическую страницу в формат HTML.

Деньги (Money, 502). Представляет денежное значение.

Дополнительный модуль (Plugin, 514). Связывает классы во время настройки, а не во время компиляции приложения.

Единица работы (Unit of Work, 205). Содержит список объектов, охватываемых бизнес-транзакцией, координирует запись изменений в базу данных и разрешает проблемы параллелизма.

Загрузка по требованию (Lazy Load, 220). Объект, который не содержит все требующиеся данные, однако может загрузить их в случае необходимости.

Интерфейс удаленного доступа (Remote Facade, 405). Предоставляет интерфейс с низкой степенью детализации для доступа к объектам, имеющим интерфейс с высокой степенью детализации, в целях повышения эффективности работы в сети.

Коллекция объектов (Identity Map, 216). Гарантирует, что каждый объект будет загружен из базы данных только один раз, сохраняя загруженный объект в специальной коллекции. При получении запроса просматривает коллекцию в поисках нужного объекта.

Контроллер запросов (Front Controller, 362). Контроллер, который обрабатывает все запросы к Web-сайту.

Контроллер приложения (Application Controller, 397). Точка централизованного управления порядком отображения интерфейсных экранов и потоком функций приложения.

Контроллер страниц (Page Controller, 350). Объект, который обрабатывает запрос к конкретной Web-странице или выполнение конкретного действия на Web-сайте.

Множество записей (Record Set, 523). Представление табличных данных в оперативной памяти.

Модель–представление–контроллер (Model View Controller, 347). Распределяет обработку взаимодействия с пользовательским интерфейсом между тремя участниками.

Модель предметной области (Domain Model, 140). Объектная модель домена, охватывающая поведение (функции) и свойства (данные).

Модуль таблицы (Table Module, 148). Объект, охватывающий логику обработки всех записей хранимой или виртуальной таблицы базы данных.

Наследование с одной таблицей (Single Table Inheritance, 297). Представляет иерархию наследования классов в виде одной таблицы, столбцы которой соответствуют всем полям классов, входящих в иерархию.

Наследование с таблицами для каждого класса (Class Table Inheritance, 305). Представляет иерархию наследования классов, используя по одной таблице для каждого класса.

Наследование с таблицами для каждого конкретного класса (Concrete Table Inheritance, 313). Представляет иерархию наследования классов, используя по одной таблице для каждого конкретного класса этой иерархии.

Неявная блокировка (Implicit Lock, 468). Предоставляет инфраструктуре приложения или супертипу слоя право накладывать автономные блокировки.

Объект-значение (Value Object, 500). Небольшие простые объекты наподобие денежных значений или диапазонов дат, равенство которых не основано на равенстве идентификаторов.

Объект запроса (Query Object, 335). Объект, представляющий запрос к базе данных.

Объект переноса данных (Data Transfer Object, 419). Применяется для переноса данных между процессами в целях уменьшения количества вызовов.

Оптимистическая автономная блокировка (Optimistic Offline Lock, 434). Предотвращает возникновение конфликтов между параллельными бизнес-транзакциями путем обнаружения конфликта и отката транзакции.

Отделенный интерфейс (Separated Interface, 492). Предполагает размещение интерфейса и его реализации в разных пакетах.

Отображение внешних ключей (Foreign Key Mapping, 258). Отображает ассоциации между объектами на ссылки внешнего ключа между таблицами базы данных.

Отображение зависимых объектов (Dependent Mapping, 283). Передает некоторому классу полномочия по выполнению отображения для дочернего класса.

Отображение метаданных (Metadata Mapping, 325). Хранит описание деталей объектно-реляционного отображения в виде метаданных.

Отображение с помощью таблицы ассоциаций (Association Table Mapping, 269). Сохраняет множество ассоциаций в виде таблицы, содержащей внешние ключи таблиц, связанных ассоциациями.

Пессимистическая автономная блокировка (Pessimistic Offline Lock, 445). Предотвращает возникновение конфликтов между параллельными бизнес-транзакциями, предоставляя доступ к данным в конкретный момент времени только одной бизнес-транзакции.

Поле идентификации (Identity Field, 237). Сохраняет идентификатор записи базы данных для поддержки соответствия между объектом приложения и строкой базы данных.

Представление по шаблону (Template View, 368). Преобразует результаты выполнения запроса в формат HTML путем внедрения маркеров в HTML-страницу.

Представление с преобразованием (Transform View, 379). Представление, которое поочередно обрабатывает элементы данных домена и преобразует их в код HTML.

Преобразователи наследования (Inheritance Mappers, 322). Структура, предназначенная для организации преобразователей, которые работают с иерархиями наследования.

Преобразователь (Mapper, 489). Объект, устанавливающий взаимодействие между двумя независимыми объектами.

Преобразователь данных (Data Mapper, 187). Слой преобразователей (Mapper, 764), который осуществляет передачу данных между объектами и базой данных, сохраняя последние независимыми друг от друга и от самого преобразователя.

Реестр (Registry, 495). "Глобальный" объект, который используется другими объектами для поиска общих объектов или служб.

Сериализованный крупный объект (Serialized LOB, 292). Сохраняет граф объектов путем их сериализации в единый крупный объект (Large Object — LOB) и помещает его в поле базы данных.

Слой служб (Service Layer, 156). Схема определения границ приложения посредством слоя служб, который устанавливает множество доступных действий и координирует отклик приложения на каждое действие.

Сохранение состояния сеанса в базе данных (Database Session State, 479). Сохраняет состояние сеанса как обычное содержимое базы данных.

Сохранение состояния сеанса на стороне клиента (Client Session State, 473). Сохраняет состояние сеанса на стороне клиента.

Сохранение состояния сеанса на стороне сервера (Server Session State, 475). Сохраняет сериализованное представление состояния сеанса на стороне сервера.

Супертип слоя (Layer Supertype, 491). Тип, выполняющий роль суперкласса для всех классов своего слоя.

Сценарий транзакции (Transaction Script, 133). Способ организации бизнес-логики по процедурам, каждая из которых обслуживает один запрос, инициируемый слоем представления.

Фиктивная служба (Service Stub, 519). Устраняет зависимость приложения от труднодоступных или проблемных служб на время тестирования.

Хранилище (Repository, 341). Выступает в роли посредника между слоем домена и слоем отображения данных, предоставляя интерфейс в виде коллекции для доступа к объектам домена.

Частный случай (Special Case, 511). Производный класс, описывающий поведение объекта в особых ситуациях.

Шлюз (Gateway, 483). Объект, инкапсулирующий доступ к внешней системе или источнику данных.

Шлюз записи данных (Row Data Gateway, 175). Объект, выполняющий роль шлюза (Gateway, 753) к отдельной записи источника данных. Каждой строке таблицы базы данных соответствует свой экземпляр шлюза записи данных (Row Data Gateway).

Шлюз таблицы данных (Table Data Gateway, 167). Объект, выполняющий роль шлюза (Gateway, 753) к базе данных.

Шпаргалка

Предупреждение: рассмотрение приведенных ниже вопросов слишком упрощено!

1. **Как структурировать логику домена?**

   Логика простая — *сценарий транзакции (Transaction Script, 133)*.

   Логика сложная — *модель предметной области (Domain Model, 140)*.

   Логика не слишком простая и не слишком сложная, а среда разработки содержит достаточно средств для манипулирования множеством записей (Record Set, 523) — *модуль таблицы (Table Module, 148)*.

2. **Как снабдить логику домена более отчетливым интерфейсом API?**

   Слой служб (Service Layer, 156).

3. **Как структурировать процесс представления данных в Web?**

   Модель–представление–контроллер (Model View Controller, 347).

4. **Как организовать обработку HTTP-запросов?**

   Поток функций приложения довольно прост, а каждый (или почти каждый) адрес URL соответствует отдельному документу Web-сервера — *контроллер страниц (Page Controller, 350)*.

   Поток функций приложения довольно сложен — *контроллер запросов (Front Controller, 362)*.

   Мне нужна поддержка пользователей из других стран или гибкие политики безопасности — *контроллер запросов (Front Controller, 362)*.

5. **Как управлять форматированием Web-страниц?**

   Я предпочитаю отредактировать HTML-код страницы и вставить в него маркеры для отображения динамических данных — *представление по шаблону (Template View, 368)*.

   Я рассматриваю Web-страницу как результат преобразования данных домена (возможно, находившихся в формате XML) — *представление с преобразованием (Transform View, 379)*.

   Я хочу подвергнуть глобальным изменениям внешний вид и поведение своего сайта — *двухэтапное представление (Two Step View, 383)*.

   Я хочу, чтобы один и тот же логический экран имел несколько различных представлений — *двухэтапное представление (Two Step View, 383)*.

6. **Как управлять сложным потоком функций приложения?**

   Контроллер приложения (Application Controller, 397).

7. **Как взаимодействовать с базой данных?**

   Я использую сценарий транзакции (Transaction Script, 133) — *шлюз записи данных (Row Data Gateway, 175)*.

   Я использую сценарий транзакции (Transaction Script, 133), и моя платформа обеспечивает хорошую поддержку множества записей ((Record Set, 523) — *шлюз таблицы данных (Table Data Gateway, 167)*.

   Структура модели предметной области (Domain Model, 140) в точности соответствует таблицам базы данных — *активная запись (Active Record, 182)*.

   Модель предметной области (Domain Model, 140) довольно сложна — *преобразователь данных (Data Mapper, 187)*.

   Я использую модуль таблицы (Table Module, 148) — *шлюз таблицы данных (Table Data Gateway, 167)*.

8. **Как гарантировать, что одни и те же данные не будут загружены в несколько разных объектов приложения?**

   Коллекция объектов (Identity Map, 216).

9. **Как сохранить связь объектов домена с соответствующими записями базы данных?**

   Поле идентификации (Identity Field, 237).

10. **Как сократить количество кода, описывающего отображение объектов домена на базу данных?**

    Отображение метаданных (Metadata Mapping, 325).

11. **Как сформулировать запрос к базе данных в терминах модели предметной области (Domain Model, 140)?**

    Объект запроса (Query Object, 335).

12. **Как сохранить связи между объектами в базе данных?**

    У меня есть ссылка на один объект — *отображение внешних ключей (Foreign Key Mapping, 258).*

    У меня есть ссылка на коллекцию объектов — *отображение внешних ключей (Foreign Key Mapping, 258).*

    У меня есть отношение типа "многие ко многим" — *отображение с помощью таблицы ассоциаций (Association Table Mapping, 269).*

    У меня есть коллекция объектов, которые используются только в контексте другого объекта — *отображение зависимых объектов (Dependent Mapping, 283).*

    У меня есть поле, в котором хранится объект-значение (Value Object, 500) — *внедренное значение (Embedded Value, 288).*

    У меня есть сложная сеть объектов, которые не используются другими частями базы данных — *сериализованный крупный объект (Serialized LOB, 292).*

13. **Как избежать загрузки в оперативную память всего содержимого базы данных?**

    Загрузка по требованию (Lazy Load, 220).

14. **Как сохранить структуры наследования в реляционной базе данных?**

    Наследование с одной таблицей (Single Table Inheritance, 297).

    Наличие общей таблицы для всей иерархии наследования приведет к созданию высокой конкуренции за право доступа к таблице — *наследование с таблицами для каждого класса (Class Table Inheritance, 305).*

    Создание общей таблицы для всей иерархии наследования приводит к бессмысленной трате свободного места — *наследование с таблицами для каждого класса (Class Table Inheritance, 305).*

    Выполнение запросов требует слишком большого числа соединений, но применять общую таблицу все-таки не хочется — *наследование с таблицами для каждого конкретного класса (Concrete Table Inheritance, 313).*

15. **Как отслеживать считывание и изменение объектов?**

    Единица работы (Unit of Work, 205).

16. **Как зафиксировать сделанные изменения в базе данных в рамках одного клиентского запроса?**

    Оптимистическая автономная блокировка (Optimistic Offline Lock, 434).

    Я не могу допустить, чтобы пользователь потерял результаты проделанной работы — *пессимистическая автономная блокировка (Pessimistic Offline Lock, 445).*

17. **Как наложить общую блокировку на группу взаимосвязанных объектов?**

    Блокировка с низкой степенью детализации (Coarse-Grained Lock, 457).

18. **Как гарантировать выполнение всех необходимых действий по наложению и снятию блокировки?**

    Неявная блокировка (Implicit Lock, 468).

19. **Как осуществлять удаленный доступ к объектам, имеющим интерфейс с высокой степенью детализации?**

    Интерфейс удаленного доступа (Remote Facade, 405).

20. **Как передать содержимое нескольких объектов в одном удаленном вызове?**

    Объект переноса данных (Data Transfer Object, 419).

21. **Как сохранить промежуточное состояние сеанса в процессе выполнения бизнес-транзакции?**

    Нужно сохранить небольшое количество данных — *сохранение состояния сеанса на стороне клиента (Client Session State, 473).*

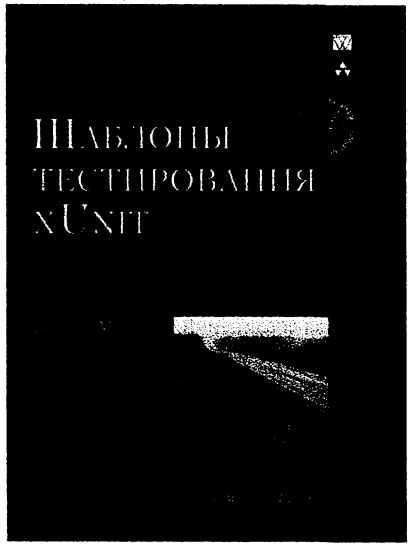
    Нужно сохранить большое количество данных — *сохранение состояния сеанса на стороне сервера (Server Session State, 475).*

    Промежуточные результаты работы могут быть сохранены в базе данных — *сохранение состояния сеанса в базе данных (Database Session State, 479).*

ШАБЛОНЫ ТЕСТИРОВАНИЯ XUNIT

РЕФАКТОРИНГ КОДА ТЕСТОВ

Джерард Месарош

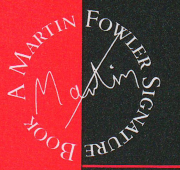


www.williamspublishing.com

В данной книге показано, как применять принципы разработки программного обеспечения, в частности шаблоны проектирования, инкапсуляцию, исключение повторений и описательные имена, к написанию кода тестов. Книга состоит из трех частей.
В первой части приводятся теоретические основы методов разработки тестов, описываются концепции шаблонов и "запахов" тестов (признаков существующей проблемы). Во второй и третьей частях книги приводится каталог шаблонов проектирования тестов, "запахов" и других средств обеспечения большей прозрачности кода тестов. Кроме этого, в третьей части книги сделана попытка обобщить и привести к единому знаменателю терминологию тестовых двойников и подставных объектов, а также рассмотрены некоторые принципы их применения при проектировании как тестов, так и самого программного обеспечения.
Книга ориентирована на разработчиков программного обеспечения, практикующих гибкие процессы разработки.

ISBN 978-5-8459-1448-4

в продаже



Объектные технологии

ШАБЛОНЫ КОРПОРАТИВНЫХ ПРИЛОЖЕНИЙ

Очередной подъем в сфере разработки корпоративных приложений связан с появлением новых многообещающих технологий. Все большую популярность приобретают объектно-ориентированные платформы, предназначенные для построения мощных многоуровневых приложений, в частности Java и .NET. Последние открывают перед разработчиками невиданные доселе возможности, однако реализовать их не так легко. Многие существующие приложения грешат одними и теми же недостатками, поскольку разработчики корпоративных приложений не имеют того багажа знаний, которым обладают опытные проектировщики объектных систем.

Книга *Шаблоны корпоративных приложений* представляет существенный интерес для разработчиков бизнес-приложений, так как затрагивает повседневные проблемы, с которыми они сталкиваются. Ее автор, известный проектировщик объектно-ориентированных систем Мартин Фаулер, отмечает, что, несмотря на эволюцию технологий — от Smalltalk через CORBA до Java и .NET, глобальные принципы, применяющиеся для решения распространенных проблем проектирования, остались прежними. С помощью группы экспертов Фаулеру удалось выделить более сорока стандартных приемов и организовать их в типовые решения. Результатом этого сотрудничества стало появление книги, материал которой может и должен быть применен ко всем платформам разработки корпоративных систем.

Данная книга состоит из двух частей, каждая из которых, по сути, может рассматриваться как отдельная книга. Первая часть представляет собой краткий учебный курс по разработке корпоративных приложений. Рекомендуем прочитать ее от начала и до конца, чтобы хорошо ориентироваться в области знаний, охваченной автором. Оставшиеся главы книги содержат подробные описания конкретных типовых решений. К каждому типовому решению прилагаются сведения о его использовании и основных аспектах реализации, а также практические примеры исходного кода на языках Java и C#. Для лучшего понимания изложенных концепций книга иллюстрирована многочисленными схемами UML.

Вооружившись этой книгой, вы получите целый ряд проверенных методик и сможете принимать важные архитектурные решения касательно всевозможных аспектов построения корпоративных приложений.

В книге рассматриваются следующие темы:

- Разделение корпоративного приложения на слои
- Основные подходы к организации бизнес-логики
- Детальное изучение объектно-реляционного отображения
- Организация представления данных в Web с использованием системы “модель-представление-контроллер”
- Параллельная обработка заданий, охватывающих несколько системных транзакций
- Проектирование интерфейсов распределенного доступа к объектам

Мартин Фаулер — старший научный консультант компании ThoughtWorks, занимающейся разработкой и интеграцией корпоративных приложений. В 80-х годах прошлого столетия он был в числе первых разработчиков, которые начали применять объектные технологии для построения многоуровневых корпоративных приложений. Мартин является автором книг *Analysis Patterns*, *UML Distilled*, *Planning Extreme Programming* и *Refactoring*, вышедших в издательстве Addison-Wesley.

www.martinfowler.com
www.awprofessional.com

ВИЛЬЯМС
www.williamspublishing.com

ADDISON WESLEY
www.pearsoneduc.com

ISBN 978-5-8459-1611-2
10018
9 785845 916112