Применение DDD и шаблонов проектирования

Проблемно-ориентированное проектирование приложений с примерами на C# и .NET

Domen_title.indd 1 21.08.2007 15:11:26

Applying Domain-Driven Design and Patterns

with Examples in C# and .NET

Jimmy Nilsson

Addison-Wesley

Domen_title.indd 2 21.08.2007 15:11:27

Применение DDD и шаблонов проектирования

Проблемно-ориентированное проектирование приложений с примерами на C# и .NET

Джимми Нильссон

Москва • Санкт-Петербург • Киев
2008

Domen_title.indd 3 21.08.2007 15:11:27

ББК 32.973.26-018.2.75
Н66
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *И.В. Берштейна*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com
115419, Москва, а/я 783; 03150, Киев, а/я 152

Нильссон, Джимми.

Н66 Применение DDD и шаблонов проектирования: проблемно-ориентированное проектирование приложений с примерами на C# и .NET. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2008. — 560 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1296-1 (рус.)

Эта книга о разработке корпоративных программных приложений в среде .NET с применением шаблонов проектирования. В ней описаны: проблемно-ориентированные методы проектирования (DDD, или Domain Driven Design), разработка посредством тестирования (TDD, или Test-Driven Development), объектно-реляционное преобразование, т.е. методы, которые многие относят к ключевым технологиям разработки программного обеспечения. По мере развития и усложнения технологии все большее значение приобретают вопросы правильного применения методов проектирования. Ценность этой книги в том и состоит, что она помогает разобраться в этих вопросах. Хотя большинство примеров кода представлено на языке C#, материал книги может оказаться полезным и для тех, кто работает на платформе Java.

Книга адресована опытным разработчикам архитектуры и прикладного программного обеспечения уровня предприятий, в том числе и в среде .NET.

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 Addison-Wesley Publishing Company, Inc, Copyright © 2006 by Pearson Education, Inc.

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means—electronic, mechanical, photocopying, recording, or otherwise—without written permission from the publisher, except for the inclusion of brief quotations in a review.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2008

ISBN 978-5-8459-1296-1 (рус.)
ISBN 0-321-26820-2 (англ.)

© Издательский дом "Вильямс", 2008
© Pearson Education, Inc., 2006

Оглавление

Предисловия 27
Вступление: наведение мостов 31

Часть I. Основы 39
Глава 1. Переоценка ценностей 41
Глава 2. Переход к шаблонам 85
Глава 3. РПТ и рефакторинг кода 115

Часть II. Прикладное ППО 149
Глава 4. Новая используемая по умолчанию архитектура 151
Глава 5. Проектирование на основе модели предметной области 181
Глава 6. Подготовка к инфраструктуре 221
Глава 7. Порядок устанавливают правила 271

Часть III. Применение шаблонов архитектуры корпоративных программных приложений 319
Глава 8. Инфраструктура для сохраняемости 321
Глава 9. Приведение в действие преобразователя NHibernate 353

Часть IV. Что дальше 389
Глава 10. Рекомендуемые методы проектирования 391
Глава 11. Акцент на пользовательском интерфейсе 449

Часть V. Приложения 489
Приложение А. Другие виды моделей предметной области 491
Приложение Б. Перечень рассмотренных шаблонов 529
Список литературы 536
Предметный указатель 543

Содержание

Отзывы о книге **17**

Об авторе **26**

Предисловия **27**

Вступление: наведение мостов **31**

Благодарности **37**

От издательства **38**

Часть I. Основы **39**

Глава 1. Переоценка ценностей **41**

Общие ценности 41

Оценка стилей архитектуры 42

Акцент на модели 42

Акцент на прецедентах использования 43

Если сделан акцент на модели, следует применять шаблон модели предметной области 48

База данных требует аккуратного обращения 53

Потеря соответствия модели предметной области и реляционной базы данных 58

Распределение требует аккуратного обращения 63

Акцент на обмене сообщениями 65

Оценка составляющих процесса 67

Предварительное проектирование архитектуры 68

Проектирование на основе предметной области 71

Разработка посредством тестирования 71

Рефакторинг кода 75

Отдельные составляющие или их сочетание 77

Непрерывная интеграция 78

Проблема интеграции 79

Решение проблемы или радикальный шаг в правильном направлении 79

Уроки, которые я извлек из опыта работы в своей организации 80

Дополнительные сведения 81
Не следует забывать об эксплуатации 81
Когда механизм нужен 81
Некоторые примеры механизмов эксплуатации 82
Это не только оплошность разработчиков 83
Резюме 84

Глава 2. Переход к шаблонам 85

Вкратце о шаблонах 86
Предостережения в отношении шаблонов 88
Шаблоны проектирования 90
Пример применения шаблона State 90
Архитектурные шаблоны 98
Пример применения шаблона Layers 99
Шаблон Domain Model в качестве другого примера 100
Шаблоны проектирования для конкретных типов приложений 100
Пример применения шаблона Query Object 101
Шаблоны предметной области 107
Пример применения шаблона Factory 109
Резюме 114

Глава 3. РПТ и рефакторинг кода 115

Разработка посредством тестирования 115
Порядок выполнения РПТ 116
Пора приступать к демонстрации 116
Следствия для проектирования 123
Недостатки проектирования методом РПТ 126
Следующая стадия 127
Заглушки и имитации 128
Типичный блочный тест 128
Объявление независимости 129
Работа с трудными объектами 130
Замена взаимодействующих объектов тестовыми заглушками 130
Замена взаимодействующих объектов имитирующими объектами 133
Последствия для проектирования 135
Выводы 136
Дополнительные сведения 136

Рефакторинг кода 136
Чистка недоброкачественного кода 137
Резюме 148

Часть II. Прикладное ППО 149

Глава 4. Новая используемая по умолчанию архитектура 151

Основа новой используемой по умолчанию архитектуры 151
Смещение акцента с базы данных на модель предметной области 153
Более конкретный акцент на ППО 153
Разделение на слои в соответствии с ППО 154
Первый набросок 155
Задачи и возможности для примера предметной области 156
Решение задач по очереди 158
Модель предметной области на данном этапе 171
Первая попытка привязать пользовательский интерфейс к модели предметной области 173
Основная цель 173
Акцент на простой пользовательский интерфейс 173
Составление списка заказов клиента 173
Ввод заказа 175
Что получается в итоге 175
Еще одно измерение 176
Расположение модели предметной области 178
Изолирование или совместное использование экземпляров 178
Реализация модели предметной области с сохранением или без сохранения состояния 179
Полная или частичная реализация модели предметной области 179
Резюме 180

Глава 5. Проектирование на основе модели предметной области 181

Уточнение модели предметной области простым экспериментированием с РПТ 181
Первая задача: создание классов `Order` и `OrderFactory` 182
Немного логики для предметной области 187
Вторая задача: создание класса `OrderRepository` и ввод свойства `OrderNumber` 188

Восстановление объекта-сущности из состояния сохраняемости: задание значений извне 193
Выборка списка заказов 198
Пора обсудить объекты-сущности 200
Назад к основному ходу рассуждений 200
Общая перспектива 202
Подделка класса `OrderRepository` 203
Несколько слов о сохранении 205
Общая сумма каждого заказа 206
Сведения о предыстории клиента 211
Жизненный цикл экземпляра объекта 214
Тип заказа 216
Ответственное лицо для заказа 216
Плавный интерфейс 218
Резюме 220

Глава 6. Подготовка к инфраструктуре 221

POCO как образ жизни 222
HC для объектов-сущностей и объектов-значений 223
Соблюдать HC или не соблюдать 228
HC во время выполнения или компиляции 228
Во что обходится HC для объектов-сущностей и объектов-значений 229
HC для хранилищ 231
Во что обходятся хранилища единственного ряда 237
Сценарии сохранения 238
Основания для принятых решений 240
Построение механизма подделки 243
Дополнительные свойства механизма подделки 244
Реализация подделки 246
Воздействие подделки на блочные тесты 247
Тестирование базы данных 251
Установка базы данных в исходное состояние перед каждым тестом 253
Сохранение состояния базы данных во время выполнения тестов 255
Сброс данных для теста перед его выполнением 255
Не забывайте о развитии схемы базы данных 256

Тестирование блока кода отдельно от проверки обращения к базе данных 257
Формирование запросов 261
Объекты запросов единственного ряда 262
Во что обходятся объекты запросов единственного ряда 263
Размещение запросов 266
Агрегаты как инструмент формирования запросов 267
Описания в качестве запросов 269
Другие методы формирования запросов 269
Резюме 270

Глава 7. Порядок устанавливают правила 271

Разделение правил на категории 271
Принципы для правил и их применения 272
Двухсторонняя проверка правил: необязательная (профилактическая) и обязательная (задним числом) 272
Сохранению подлежат все состояния — даже ошибочные 272
Правила должны применяться продуктивно 273
Правила должны допускать поправки для ввода специальных правил 273
Правила должны размещаться вместе с состоянием 273
Правила должны быть очень удобными для тестирования 273
Система должна препятствовать переходу в недействительное состояние 274
Начало создания интерфейса API 274
Контекст, контекст и еще раз контекст! 275
Ограничения, связанные с базой данных 276
Привязка правил к переходам, связанным с моделью предметной области или же с инфраструктурой 277
Уточнение принципа “Сохранению подлежат все состояния — даже ошибочные” 278
Требования к интерфейсу API для основных правил, связанных с сохраняемостью 280
Возврат к проблемам интерфейса API 281
Какая именно проблема возникла 281
Мы допустили неправильный переход 282
Если мы забыли выполнить проверку 282
Акцент на правилах, связанных с предметной областью 283

Правила, требующие взаимодействия 285
Поиск методов обработки на основе множеств 286
Обслуживание проверки достоверности по шаблону Service 289
Попытка сделать непредусмотренный переход 289
Идентификационный номер клиента 292
Как избежать осложнений 294
Агрегаты как инструмент проверки правил 295
Расширение интерфейса API 296
Запрос правил для настройки пользовательского интерфейса 296
Возможность для внесения правил 297
Уточнение реализации 297
Наивная реализация 297
Переход от самой наивной стадии к созданию классов правил 303
Составление списка правил 306
Использование списка правил 307
Составление подсписков 308
Усовершенствование интерфейса API 309
Специальная настройка 310
Предоставление пользователю метаданных 310
Проблема подстать шаблону 311
Как обращаться со сложными правилами 311
Привязка к абстрактному представлению сохраняемости 313
Внедряемый интерфейс проверки достоверности 313
Альтернативные решения для проверки достоверности задним числом при сохранении 314
Повторное использование метаданных преобразования 315
Обобщения и анонимные методы как спасение 315
Другие подходы 317
Резюме 317

Часть III. Применение шаблонов архитектуры корпоративных программных приложений 319

Глава 8. Инфраструктура для сохраняемости 321

Требования к инфраструктуре сохраняемости 322
Место для хранения данных 324
Оперативная память 324
Файловая система 327

Объектная база данных 327
Реляционные базы данных 328
Один или несколько администраторов ресурсов 329
Другие факторы 329
Выбор решения и переход к его воплощению 330
Подход 330
Написание вручную заказного кода 330
Генерация заказного кода 331
Преобразование метаданных (объектно-реляционное преобразование) 333
Повторный выбор 335
Классификация 335
Вид модели предметной области 336
Вид объектно-реляционного преобразователя 336
Отправная точка 337
Акцент на интерфейсе API 338
Тип языка запросов 339
Улучшенная поддержка базы данных 340
Другие функциональные возможности 341
Классификация по шаблонам инфраструктуры 342
Тип метаданных для преобразования 343
Поля идентификации 344
Преобразование внешних ключей 346
Внедренное значение 346
Решения проблемы наследования 347
Карта идентичности 348
Единица работы 349
Загрузка по требованию или энергичная загрузка 349
Управление параллельным выполнением операций 350
Резюме 351

Глава 9. Приведение в действие преобразователя NHibernate 353

Основания для выбора преобразователя NHibernate 353
Краткое введение в NHibernate 354
Подготовка 354
Метаданные для преобразования 356
Небольшой пример интерфейса API 362
Транзакции 365

Требования к инфраструктуре сохраняемости 366
Высокий уровень неведения сохраняемости 366
Некоторые желательные свойства жизненного цикла постоянных объектов-сущностей 367
Аккуратное обращение с реляционной базой данных 369
Классификация 370
Вид модели предметной области 371
Вид объектно-реляционного преобразователя 371
Отправная точка 372
Акцент на интерфейсе API 372
Тип языка запросов 372
Улучшенная поддержка базы данных 374
Другие функциональные возможности 375
Классификация по шаблонам инфраструктуры 377
Тип метаданных для преобразования 377
Поля идентификации 377
Преобразование внешних ключей 379
Внедренное значение 380
Решения проблемы наследования 381
Карта идентичности 382
Единица работы 383
Загрузка по требованию или энергичная загрузка 383
Управление параллельным выполнением операций 384
Дополнительные возможности: перехватчики проверки достоверности 384
NHibernate и ППО 385
Краткий обзор сборок 385
Интерфейс `ISession` и хранилища 386
Интерфейс `ISession`, хранилища и транзакции 386
Чего мы достигли 387
Резюме 388

Часть IV. Что дальше 389

Глава 10. Рекомендуемые методы проектирования 391

Все дело в контексте 392
Слои и разделы 392
Основания для разбиения на разделы 393
Ограниченный контекст 393

Взаимосвязь между ограниченными контекстами и разделами 394
Расширение масштабов проектов ППО 395
Основание для разбиения модели предметной области на разделы — ориентация на службы 395
Введение в AOC 396
Что собой представляет AOC 396
Назначение AOC 396
Отличия AOC 397
Что такое служба 397
Что входит в службу 398
О значении четырех руководящих принципов AOC 399
Еще раз о том, что такое служба 400
Место объектной ориентации в AOC 401
Архитектура “клиент-сервер” и AOC 401
Односторонний асинхронный обмен сообщениями 402
Как AOC улучшает масштабируемость 403
Структура службы AOC 403
Взаимодействие службы с другими службами 404
AOC и недоступные службы 407
Сложные процессы обмена сообщениями 408
Масштабирование служб 408
Выводы 409
Инверсия управления и внесение зависимостей 410
Ни один из объектов не изолирован 410
Фабрики, реестры и указатели служб 412
Внесение зависимостей в конструктор 415
Внесение зависимостей в механизм установки 418
Инверсия управления 419
Внесение зависимостей в среде Spring.NET 420
Автосвязывание в среде PicoContainer.NET 422
Вложенные контейнеры 424
Сравнение шаблонов Service Locator и Dependency Injection 426
Выводы 427
Аспектно-ориентированное программирование 429
Из-за чего столько шума 429
Терминология АОП 433
АОП в .NET 433
Выводы 447
Резюме 447

Глава 11. Акцент на пользовательском интерфейсе 449

Предварительный эпилог 449
Шаблон Model-View-Controller 451
Пример: обувной магазин Джо 452
Упрощение интерфейсов представления с помощью адаптеров 459
Развязка контроллера от представления 459
Сочетание представлений и контроллеров 460
Чего все это стоит 460
Разработка Web-формы посредством тестирования 461
Предпосылки 461
Пример 462
Модель предметной области 463
РПТ графического пользовательского интерфейса 463
Реализация Web-формы 470
Выводы 472
Имитация средствами NMock 472
Преобразование и заключение в оболочку 474
Преобразование и заключение объектов в оболочку 475
Заключение МПО в оболочку МП 476
Преобразование МП в МПО 477
Организация отношений между объектами 480
Вопросы состояния 484
Заключительные соображения 485
Резюме 486
Эпилог 486

Часть V. Приложения 489

Приложение А. Другие виды моделей предметной области 491

Объектно-ориентированная модель данных, уровень развитого обслуживания и документы 493
В начале 493
Объектная ориентация и реляционные структуры данных 494
Модель предметной области и объектно-реляционное преобразование 494
Уровень обслуживания 495
Объединение на одном уровне 496
Приложение Джимми и АОС 496

Структура уровня обслуживания 499
Предоставление данных 501
Степень структурирования 503
Несколько слов о транзакциях 505
Выводы 505
Модель базы данных как модель предметной области 506
Место обитания сущностей 506
Функциональный анализ как основа разработки приложений 511
Прагматизм и нетрадиционный подход 514
Предпосылки и история вопроса 515
Краткий обзор архитектуры 515
Пример приложения Джимми для обработки заказов 519
Бизнес-правила размещения заказов 522
Некоторые заключительные соображения 527
Резюме 527

Приложение Б. Перечень рассмотренных шаблонов 529

Список литературы 536

Предметный указатель 543

Отзывы о книге

“Я не знаю, как назвать то, чем я занимался до того, как пополнил свой инструментарий проектированием на основе предметной области и разработкой посредством тестирования, но теперь я склонен называть это не более чем хаотичным хакерством. Проектирование на основе предметной области и разработка посредством тестирования — это два метода, которыми я постоянно руководствуюсь в практическом применении принципов разработки программного обеспечения, что позволило мне осуществить поставленные цели в моих проектах с помощью надежного, устойчивого программного обеспечения. С точки зрения практики гибкой разработки программного обеспечения эта книга полностью оправдывает свое назначение. Она, вероятно, является одним из самых влиятельных руководств по успешной практике разработки программного обеспечения, предлагавшихся до сих пор сообществу разработчиков программного обеспечения средствами .NET”.

Скотт Беллуэр (Scott Bellware), авторитетный профессионал (Microsoft MVP) по C#, ведущий блог по проектированию на основе предметной области и разработке посредством тестирования

“Джимми Нильсон делает своим читателям большую услугу, показывая им, как применять основополагающие принципы проектирования и разработки приложений для предприятия, выработанные Эвансом, Фаулером и другими ведущими представителями данного направления в программировании. В книге представлены примеры с решением не только для пояснения, но и для демонстрации особенностей проектирования на основе предметной области, шаблонов архитектуры приложений для предприятия и разработки посредством тестирования. Глубокое понимание предмета и практический опыт Джимми делают чтение книги приятным и оставляют у читателя уверенность в том, что он научился чему-то полезному у большого специалиста-практика. Разработчики приложений для предприятия, стремящиеся овладеть этими принципами, обнаружат в книге весьма ценные наставления”.

Джек Гринфилд (Jack Greenfield), разработчик архитектуры инструментария для предприятия, Visual Studio Team System, Microsoft

“Хорошие разработчики архитектуры программного обеспечения должны постоянно совершенствоваться. Помимо целей внедрения текущей системы, они ищут пути совершенствования разработки и создания программного обеспечения. Эта книга является своего рода путевыми заметками, сделанными Джимми по ходу освоения

самых разных шаблонов, практических методов и технологий с пояснением эволюции его представлений о системах предприятия. Если вы идете той же дорогой, эта книга будет вам хорошим попутчиком".

Тим Евальд (Tim Ewald), главный разработчик архитектуры программного обеспечения в компании Foliage Software Systems и автор книги Transactional COM+: Building Scalable Applications

"Эта книга вносит большой вклад в осуществление крупных, но важных идей проектирования на основе предметной области".

Флойд Маринеску (Floyd Marinescu), автор книги EJB Design Patterns *и создатель Web-сайтов InfoQ.com и TheServerSide.com*

"Понимание принципов и движущих сил в предметной области имеет решающее значение для успешной разработки программного обеспечения. Джимми Нильсон почерпнул вдохновение в изучении шаблонов и проектирования на основе предметной области в течение последних десяти лет и воспроизвел свой опыт работы над конкретными проектами. Эта книга содержит замечательные примеры воплощения теории на практике. Она показывает глубокое понимание автором вопросов принятия проектных решений и компромиссов, связанных с объектно-ориентированной разработкой".

Андерс Хесселлунд (Anders Hessellund), Копенгагенский университет информационных технологий, Дания

"В этой книге рассматривается область, представляющая немалые трудности для освоения большинством разработчиков на платформе .NET. Мне как разработчику структуры шаблонов и практических методов, ставшему инициатором и приложившему немало усилий для выработки руководства по разработке приложений для предприятия в .NET, хорошо известно, насколько эта область важна для наших заказчиков, и я отчетливо осознаю те пробелы, которые все еще существуют в созданном нами руководстве.

Я был тронут тем, как Джимми делится своим опытом проектирования на основе предметной области и разработки посредством тестирования. Полагаю, что в настоящий момент эту тему лучше всего рассматривать с точки зрения простоты, шаблонов и осознания социальных аспектов создания приложений.

Я доверяю опыту Джимми и его знанию .NET, мне по душе его манера делиться принципами и примерами. Мне трудно представить, чтобы кого-то другой смог лучше раскрыть эту тему на примере платформы, с которой я работаю каждый день.

Настоятельно рекомендую книгу Джимми моим заказчикам, коллегам и всем инженерам из Microsoft.

Я надеюсь, что в конечном итоге наша отрасль только выиграет от выражения глубоких понятий в коде и достигнет прогресса в социальном процессе ясной формули-

ровки и эволюции общедоступных знаний. На пути к этому лежит проектирование на основе предметной области и разработка посредством тестирования".

Эдвард Ежерски (Edward Jezierski), разработчик структуры шаблонов и практических методов Microsoft

"Джимми связал вместе самые передовые методы — проектирование на основе предметной области, разработку посредством тестирования, рефакторинг кода и шаблоны проектирования — в убедительную альтернативу стилю информационной разработки, служившему оплотом приложений Microsoft. Книга демонстрирует воплощение методов, которые пропагандируются в проектировании на основе предметной области. Она имеет ярко выраженный практический характер и дает читателю представление о мыслительном процессе, на котором основывается применение испытанных методов и технологий".

Кристиан Кроухэрст (Christian Crowhurst), разработчик-аналитик

"Мне пришлось работать с Джимми над совместным проектом в течение 18 месяцев. Джимми действительно живет и работает по тем принципам, которые он проповедует. В книге он описывает свои действия на практике. В частности, он ежедневно пользуется объектно-реляционным преобразованием и преобразователем NHibernate, а разработка посредством тестирования помогает ему понять и вычленить бизнес-логику заказчика, а также убедиться в том, что заказчик получит проверенный, высококачественный код. Эта книга знакомит читателей с проверенными принципами. Пользуйтесь".

— Дан Бистрём (Dan Byström), разработчик программного обеспечения, `www.visual-design.se/blog`

"Показывая, как выполнять проектирование на основе предметной области и применять шаблоны приложений для предприятия, Джимми делает трудную тему доступной для широкого круга читателей. В этой книге найдется немало полезного как для опытного разработчика архитектуры программного обеспечения, так и для честолюбивого реализатора шаблонов. Разработчики программного обеспечения или его архитектуры, желающие усовершенствовать свои проекты и результаты внедрения программных решений, должны прочитать эту книгу".

Пер-Ола Нильссон (Per-Ola Nilsson), руководитель отдела разработки и разработчик архитектуры программного обеспечения в компании Luvit

"Постоянно находясь в курсе новейших тенденций в области разработки программного обеспечения, но не поддаваясь на рекламные провокации, Джимми профессионально отделяет зерна от плевел. Он обладает особым даром наставить и просветить читателя, руководствуясь практической применимостью. Джимми не предлагает

помощь как дар небес — он просто пытается убедить нас в том, что мы можем научиться чему-то совершенно новому у равного нам".

Матс Хеландер (Mats Helander), разработчик программного обеспечения, `www.matshelander.com`

"Как и предыдущая книга Джимми, эта книга является настоящим шедевром, идеально сочетающим в себе практические вопросы проектирования программного обеспечения с основательной теорией".

Франс Боума (Frans Bouma), создатель средства генерации кода LLBLGen Pro, `weblogs.asp.net/fbouma`

"Опираясь на солидное основание, Джимми рассматривает в своей книге практические вопросы прикладного проектирования на основе предметной области. Это руководство позволит разработчикам справиться с постоянно растущей сложностью новых приложений для предприятия, основанных на модели предметной области".

Поль Гиленс (Paul Gielens), старший консультант в компании Capgemini, `weblogs.asp.net/pgielens`

"Опираясь на свой опыт и наполняя в .NET конкретным содержанием такие сокращенные понятия, как DDD и TDD, можно создать весьма эффективную и адаптивную архитектуру программного обеспечения для предприятия. В книге показано, как и почему эти принципы составляют вместе столь эффективное целое. Таким образом, читатель наставляется на бесценный, но краткий путь к успешному созданию архитектуры программного обеспечения для предприятия. Эта книга, без сомнения, будет полезна разработчику такого программного обеспечения".

Мартин Розен-Лидхольм (Martin Rosén-Lidholm), разработчик архитектуры программного обеспечения в компании Exense Healthcare

"Если ваша задача — построить качественную систему предприятия с объектной ориентацией и реляционной базой данных, не упустите возможность прочитать эту книгу. Вы узнаете, каким образом проектирование на основе предметной области может стать движущей силой всего проекта и как объектно-ориентированная конструкция приводится в соответствие с базой данных — этот этап зачастую рассматривается в литературе очень лаконично, что уже само по себе окупает расходы на данную книгу. Кроме того, здесь вы найдете немало полезных советов и рекомендаций, которым можно последовать при проектировании".

Ингмар Лундберг (Ingemar Lundberg), разработчик программного обеспечения, `www.ingolundberg.com`

"Эта важная и своевременная книга должна стать настольной для всех, кто стремится разобраться в проектировании на основе предметной области на C#".

Грегори Янг (Gregory Young), авторитетный профессионал (Microsoft MVP) по C#, независимый консультант

"В книге рассматривается ряд важных понятий, применяемых в современной разработке программного обеспечения, в том числе разработка посредством тестирования, рефакторинг кода, шаблоны и, разумеется, проектирование на основе предметной области. Джимми Нильссон излагает эти понятия в удобном режиме диалога, и поэтому у вас создается впечатление, будто он сидит рядом, демонстрируя свои примеры и ведя с вами профессиональный разговор о достоинствах и недостатках различных решений".

Никлас Нильссон (Niclas Nilsson), разработчик программного обеспечения и преподаватель в компании Activa, он не ближний и даже не дальний родственник автора книги, а просто однофамилец

"В книге всесторонне описан процесс проектирования на основе предметной области (ППО — DDD) в практической плоскости посредством наведения мостов между различными абстрактными уровнями, начиная с общей бизнес-перспективы и заканчивая системным уровнем. Книга идеально подходит для разработчиков программного обеспечения и его архитектуры, поскольку написана в виде учебника или справочника. Автор отлично справился с задачей представления всех аспектов ППО не только в просветительской, но и в занимательной для чтения манере".

Гюнтер Ленц (Gunther Lenz), руководитель проекта и автор

"Не так давно я делился в блоге своими мыслями о том, что мы, разработчики архитектуры программного обеспечения, не даем остальным разработчикам достаточных наставлений относительно выбора оптимальных путей создания программного обеспечения. Ведь существует немало отличных идей, имеющих вполне аргументированное обоснование, практические примеры осуществления и горячих сторонников. Но лишь некоторые из них могут быть использованы в чистом виде для создания приложений, а тем более систем приложений и служб. Воплотить идеи в жизнь намного труднее, чем думают их поборники. С другой стороны, разработчики должны делать свое дело, и поэтому им зачастую неинтересно вникать во все детали, когда они приступают к созданию нового приложения. Это происходит даже в том случае, когда они действительно заинтересованы в тех преимуществах, которые таят новые подходы или идеи. Именно эту проблему и призвана решить книга Джимми.

Как воплотить в реальной, живой разработке замечательные идеи, выдвинутые за последние годы? Автор книги немало потрудился для того, чтобы подробно пояснить основные идеи с целью их последующей реализации на практике, а также для

группирования идей таким образом, чтобы помочь разработчикам увидеть весь путь, который они должны пройти от начала до конца. Проектирование на основе предметной области, разработка посредством тестирования, проверка зависимости, сохраняемость и многие другие практические вопросы рассматриваются в этой книге. По прочтении книги читатель сможет взять все эти важные идеи на вооружение, чтобы эффективно воспользоваться ими. Принимая во внимание большую ценность отдельных представленных идей, но ограниченное число людей, воплотивших эти идеи на практике, можно с уверенностью сказать, что эта книга является значительным достижением для всех нас. Поэтому я настоятельно рекомендую ее".

Филип Нельсон (Philip Nelson), главный научный сотрудник компании PAi, `xcskiwinn.org/community/blogs/panmanphil/defaulr.aspx`

"Радикальный переход от мышления, ориентированного на информацию, к более глубокому пониманию ООП, которое обнаруживается в проектировании на основе предметной области, разработке посредством тестирования, объектно-ориентированной сохраняемости и других гибких методах разработки, может оказаться непосильной, дезориентирующей задачей для непосвященных.

Терпеливо, настойчиво и наставительно Джимми помогает читателю совершить этот переход, исследуя соответствующие вопросы и возможные варианты и давая по ходу дела полезные советы. В этой книге показано, как различные методы и шаблоны вписываются в полностью согласованный подход к проектированию и созданию в .NET превосходно сопровождаемого программного обеспечения".

Джордж Хикс (George Hicks), старший разработчик компании Property Works

"Если вы когда-либо читали блог Джимми Нильссона, значит, вам должно быть известно, что он любит ставить под сомнение установившиеся "истины" в сообществе программирующих в .NET, пытаясь найти более совершенные подходы к разработке программного обеспечения. Он признает, что разработка программного обеспечения — трудное занятие и что решений на все случаи жизни не бывает, поэтому достоинства и недостатки каждого подхода должны быть уравновешены в контексте решаемой задачи перед выбором наиболее оптимального варианта. Для создания удобного сопровождаемого и проверяемого программного обеспечения со сложными бизнес-требованиями Джимми выбирает проектирование на основе предметной области и дополняет его рядом проверенных и установившихся принципов, шаблонов и практических методов.

Выбранный Джимми неформальный стиль изложения материала, демонстрации примеров и пояснения особенностей разработки посредством тестирования делает эту книгу очень доступной, особенно если учесть обширность и сложность рассматриваемых в ней вопросов. Будучи ведущим специалистом в области ППО, Джимми поясняет саму суть большинства ценных методов современной разработки програм-

много обеспечения и приводит там, где это необходимо, ссылки на многочисленные ресурсы. Эта книга — весьма ценное руководство по прикладному ППО, она может быть полезной для тех разработчиков, которые стремятся повысить свою общую квалификацию в проектировании".

Андреас Бринк (Andreas Brink), разработчик программного обеспечения

"Этой книгой Джимми Нильссон еще раз потвердил свое ведущее положение в области разработки прикладной архитектуры программного обеспечения. Ясно излагая свой личный взгляд на данный предмет и не оставляя ничего без внимания, Джимми знакомит читателя с большинством современных методов, которые должен знать каждый разработчик. Джимми показывает, как идеи других, ведущих в данной области специалистов, тяготеющих больше к теории, чем к практике, воплощаются в привлекальную и легко доступную структуру. Уверен, что эта книга обязательно займет свое место на книжной полке библиотеки каждого предприятия".

Микаэль Фрайдлитц (Mikael Freidlitz), вице-президент по программам совершенствования содержимого и знаний в компании IASA

"Проектирование на основе предметной области является важным методом, позволяющим создавать качественные бизнес-приложения, которые развиваются вместе с потребностями в выбранной сфере деятельности. В идеальном случае прикладное ППО мало чем отличается от ООП, но в действительности многие ограничения накладываются теми технологиями, с которыми нам приходится работать.

В этой книге рассматривается задача наведения мостов между принципами ППО и действиями, направленными на практическое воплощение этих принципов на платформе .NET. Джимми не только основательно разобрался в ППО и технологиях управления предприятием, но и извлек уроки из своего богатого опыта работы в данной отрасли, выбрав прагматичный, но не консервативный подход. Его книга весьма ценна".

Род Джонсон (Rod Johnson), основатель компании Spring Framework, генеральный директор компании Interface21

"Это отличная книга. Она является практическим путеводителем в область проектирования на основе предметной области. В ней рассматривается, поясняется и представляется в соответствующем контексте большое число важных вопросов, что может послужить читателю солидным основанием для самостоятельной работы с системой на основе модели предметной области. В книге описываются конкретные методы разработки, а также практические методы работы, как, например, внедрение разработки посредством тестирования".

Тронд-Эйрик Коллоэн (Trond-Eirik Kolloen), разработчик программного обеспечения и его архитектуры

“Может быть, мы уделим, наконец, внимание домашним проектам?”

Лотта, жена

“Ааа, дии, дии, дии, пии..” (Произношение заглавных букв книги на шведском языке)

Лео, четырехлетний сын

“Отец, ты действительно считаешь, что кто-то будет это читать?”

Тим, восьмилетний сын

Лотте, Тиму и Лео — центрам моей вселенной.

Proekt_book.indb 25 27.08.2007 21:32:21

Об авторе

Джимми Нильссон — владелец и управляющий шведской компании JNSK AB, он также автор двух книг и многочисленных статей в технических периодических изданиях. Джимми обучает специалистов и выступает с докладами на конференциях, но прежде всего, он разработчик почти с 20-летним стажем (`www/jnsk.se/weblog/`).

Предисловия

Создание программного обеспечения для предприятия — непростое дело. Несмотря на наличие множества инструментов и базовых средств разработки, упрощающих эту задачу, нам по-прежнему приходится решать, как использовать эти средства наилучшим образом. Для этого имеется немало подходов, но самое главное — знать, какой из них применять в конкретной ситуации, и вряд ли один подход пригодится на все случаи жизни. За последние несколько лет появилось целое поколение людей, занимающихся поиском подходов к проектированию приложений для предприятия и их документированию в виде шаблонов (соответствующий обзор можно посмотреть по адресу `http://martinfowler.com/articles/enterprisePatterns/html`). Те, кто занимается данной проблемой, в том числе и я, пытаются найти общие подходы и показать в своих трудах, как и когда их лучше всего применять. Результаты такой работы сильно разнятся, и поэтому читатель оказывается перед очень широким выбором.

Когда я приступил к работе над книгой *Patterns of Enterprise Application Architecture* (изд-во Addison-Wesley, 2002 год. *Архитектура корпоративных программных приложений*, пер. с англ. ИД "Вильямс", 2004 г.), то искал подобного рода рекомендации для проектирования в литературе по программным продуктам Microsoft. Я пытался найти хоть что-нибудь достойное внимания, но в конечном итоге обнаружил лишь одну редкую работу на данную тему — предыдущую книгу Джимми. Мне понравилась информативная манера изложения и стремление автора вникнуть в понятия, которые многие другие рассматривали весьма поверхностно. Именно Джимми решил воспользоваться идеями, выдвинутыми мной и другими авторами, и показать, как воплощать их в приложениях, создаваемых в .NET.

Основное внимание тех, кто занимается шаблонами для предприятий, сосредоточено на документировании удачных проектов. Мы увлечены и другим интересным направлением, являясь большими поклонниками гибких методов, охватывающих в том числе разработку посредством тестирования (РПТ — Test-Driven Development (TDD)) и рефакторинг кода. Эти идеи Джимми изложил в своей книге. Многие находят определенное противоречие в том, что те, кто занимается шаблонами, уделяют основное внимание проектированию, а те, кто занимается РПТ, — эволюции. Тот факт, что деятельность и тех и других во многом пересекается, свидетельствует о том, что это не так, и Джимми попытался в своей книге связать воедино оба эти направления.

В итоге получилась книга о гибких методах проектирования в среде .NET с применением готовых шаблонов для предприятия. В этой книге показано, как следует применять такие методы, как РПТ, объектно-реляционное преобразование и проек-

тирование на основе предметной области, в проектах, выполняемых в .NET. Если вы еще не сталкивались с подобными понятиями, эта книга послужит вам в качестве введения в методы, которые многие разработчики считают основными для разработки программного обеспечения в будущем. Если же вы знакомы с этими идеями, книга поможет вам поделиться опытом со своими коллегами.

Многие считают, что в сообществе разработчиков приложений на платформе Microsoft не так хорошо, как у остальных, распространяются полезные советы по проектированию приложений для предприятия. По мере усложнения и расширения возможностей применяемой технологии все более актуальной становится проблема ее эффективного применения. И эта книга представляет собой важный шаг на пути к пониманию данной проблемы.

Мартин Фаулер (Martin Fowler), `http://martinfowler.com`

Самый лучший способ научиться проектированию на основе предметной области (ППО — Domain-Driven Design (DDD)) — сесть рядом с дружески настроенным, терпеливым, опытным практиком и поэтапно решать вместе с ним конкретные задачи. Именно так и происходит при чтении этой книги.

Эта книга не открывает ничего особенно нового. Она лишь беспристрастно сообщает о личном опыте специалиста-практика и ряде современных практических методов, которые привлекли его внимание.

Джимми Нильссон повторяет то, что многие из нас уже говорили не раз: ряд современных направлений, в частности, ППО, шаблоны архитектуры приложения для предприятия (PoEAA) и разработка посредством тестирования (РПТ) не являются альтернативой друг другу, а взаимно дополняют друг друга как элементы успешной разработки.

Более того, все три направления оказываются сложнее, чем кажется на первый взгляд. Они требуют обширных и глубоких знаний. В этой книге уделяется некоторое внимание пропагандированию данных подходов, но бо́льшая ее часть все же посвящена подробностям их осуществления на практике.

Эффективное проектирование не сводится к набору заученных методов. Это способ мышления. Рассматривая конкретный пример, Джимми раскрывает перед нами ход своих мыслей. Он не только показывает и поясняет свое решение, но и дает нам возможность увидеть, как он к нему пришел.

Когда я что-то проектирую, в моей голове мелькают десятки соображений по этому поводу. Если эти мысли относятся к факторам, с которыми мне часто приходится иметь дело, они проносятся в моей голове так быстро, что я едва ли успеваю их осознать. Если же они касаются областей, где я чувствую себя не столь уверенно, я останавливаюсь на них дольше. Я вполне допускаю, что такой ход мыслей типичен для разработчиков, но передать его другому человеку не так-то просто. Поясняя примеры, Джимми как бы замедляет ход своих мыслей до такой степени, чтобы за ними можно было уследить. В каждое мгновение этого процесса две или три альтер-

нативы представляются, взвешиваются и отбрасываются в пользу одной, выбираемой окончательно.

Допустим, что требуется смоделировать объекты, реализуемые без осложнений, связанных с технологией сохраняемости. Каковы же те восемь способов, которыми базовая структура сохраняемости способна усложнить реализацию объекта предметной области? Какие соображения позволяют прийти к компромиссу в ряде этих вопросов? И какие ограничения накладывают современные базовые средства разработки, включая .NET?

Джимми мыслит прагматично. Принимая проектное решение, он обращается к своему опыту, который, в конечном итоге, приводит его к желанной цели. При этом он придерживается более основательного принципа, чем выбор, который слишком похож на хрестоматийный пример, а все его решения имеют предварительный характер.

Главный принцип проектирования, которого Джимми придерживается в первую очередь, составляет основную цель ППО — проектирование, отражающее глубокое понимание решаемой экономической задачи в такой форме, которая допускает приспособляемость к новым идеям. Так зачем столько рассуждать о технической структуре и архитектуре?

Типичное и, вероятно, естественное заблуждение состоит в том, что такое предпочтение предметной области не требует особых инженерных способностей и технических навыков. Если бы это было так на самом деле, стать компетентным разработчиком приложений для предметной области было бы нетрудно. Как ни странно, для воплощения четких и пригодных принципов предметной области в программном обеспечении, причем так, чтобы они не оказались по спудом массы технических вопросов, требуется особое умение применять конкретную технологию. Мои наблюдения показывают, что те, кто лучше всего владеет техническими и архитектурными принципами, зачастую знают, как применять технологию к месту, и относятся к числу наиболее эффективных создателей моделей предметной области.

Я имею в виду не знание всех особенностей сложных инструментов, а овладение теми знаниями, которые изложены книге *Архитектура корпоративных программных приложений* Мартина Фаулера, поскольку наивное применение технологии, как ни странно, приводит к ее насильственному проникновению в приложение.

Эта книга позволит многим читателям восполнить пробел в реализации выразительных объектных моделей на практике. Я лично извлек для себя ряд полезных способов продумывания вариантов применения технических структур и укрепил свои представления об особенностях выполнения ППО в среде .NET.

Помимо технической архитектуры, Джимми уделяет в своей книге много внимания написанию тестов. РПТ по-разному дополняет ППО. В отсутствие должного внимания к уточнению модели с целью ее улучшения, РПТ находит фрагментарное применение, где целенаправленное решение каждой проблемы в отдельности приводит к появлению негибкой системы. Полный тестовый набор фактически замедляет продвижение группы разработчиков к конечной цели, но именно в этом и заключается наибольшая ценность РПТ.

В наилучшем варианте тестовый набор представляет собой лабораторию для модели предметной области и ее технического выражения на универсальном языке. Особого рода тесты продвигают процесс моделирования вперед и сохраняют его сосредоточенный характер. В книге рассматриваются примеры поэтапной разработки таких текстов.

Джимми Нильссону присуще редкое сочетание уверенности в себе и скромности — свойств, которые, по моим наблюдениям, характерны для самых лучших разработчиков. Когда он рассказывает нам, читателям, как и почему изменились его представления о том, во что он раньше верил, мы можем проследить, как он пришел к своим нынешним представлениям, и это помогает нам пропустить делали применяемых методов и перейти непосредственно к основополагающим принципам. Скромность Джимми делает его подверженным самым разным влияниям, а мы получаем в итоге сплав идей из различных источников. Он испробовал много разного, результаты этих проб и его опыт стали для него руководством к действию. Его выводы отображают нынешнее его представление о предмете, причем с ясным пониманием того, что наши знания никогда не будут полными. Все это делает его рекомендации еще более полезными для читателя. Ответственное отношение к делу как нельзя лучше характеризует его как ведущего и успешного специалиста в области разработки программного обеспечения.

Эрик Эванс (Eric Evans)

Вступление: наведение мостов

На обложке этой книги представлена фотография моста через пролив Орезунд, соединяющего Швецию и Данию. Похоже, что на обложках всех книг по архитектуре программного обеспечения должны быть изображения мостов, но в данной книге для этого имеются дополнительные причины.

Этот мост заменил паром, которым я переправлялся еще ребенком. Мне очень нравится каждый раз ездить по нему, хотя я проделывал это десятки раз.

Между прочим, мой отец был среди тех, кто возводил самые высокие участки этого моста.

Но самое главное, что эта книга во многом сродни наведению мостов — в данном случае между пользователями и разработчиками, конкретной предметной областью и программным обеспечением, логикой и хранением информации, теми, кто организует и ведет базы данных, и теми, кто занимается объектно-ориентированным программированием.

Для того чтобы это вступление не выглядело слишком заумным, я, пожалуй, воздержусь от аналогии с шаблоном моста (Bridge) [GoF Design Patterns].

Назначение книги

Основное назначение этой книги состоит в том, чтобы показать, как строится четкая, но удобная с точки зрения сохраняемости модель предметной области (Domain Model). В ней также показано, как может выглядеть решение сохраняемости для такой модели предметной области и, в частности, как навести мост между такой моделью и базой данных.

Иными словами, приступая к этой книге, я поставил перед собой цель перенести содержание книг *Domain-Driven Design* Эрика Эванса [Evans DDD] и *Архитектура корпоративных программных приложений* Мартина Фаулера [Fowler PoEAA] в определенный контекст.

Проектирование на основе предметной области (ППО — Domain-Driven Design (DDD)) можно воспринимать как нечто абстрактное. Поэтому для пояснения, в частности, сохраняемости требуются вполне конкретные примеры. Мои примеры могут показаться довольно простыми, но они закладывают основание в качестве отправной точки. В этой книге не только поясняется, как пользоваться шаблонами, но и объясняется, как это можно делать, например, в объектно-реляционных преобразователях (O/R Mappers).

Для меня стало вполне очевидно, что разработать архитектуру приложений на все случаи жизни невозможно. В этой связи шаблоны оказались достаточно обобщенными, чтобы неоднократно применять их в разных контекстах.

Основное внимание в книге уделяется не самим шаблонам, а их применению в качестве инструмента и языка для обсуждения различных аспектов проектирования. Но даже те читатели, которые не знакомы с шаблонами, получат в ходе чтения книги некоторое представление о них и, возможно, заинтересуются ими.

Это же относится и к разработке посредством тестирования (РПТ — Test-Driven Development (TDD)). Пока еще не все разработчики заинтересовались этим методом. И особенно это касается сообщества программирующих в .NET, где РПТ, как, впрочем, и шаблоны, считается в лучшем случае удобным, специальным методом или же вообще неизвестна. Читатели этой книги смогут научиться РПТ.

Причины для написания книги

Написание моей первой книги [Nilsson NED] превратилось в трудное занятие в дополнение ко всем моим обычным проектам и обязанностям. В этой связи у меня появилась твердая уверенность, что я вряд ли буду писать еще одну книгу, но прошло время и у меня возникла потребность поделиться тем, что осталось еще невысказанным.

Я изменил свое решение после того, как недавно прочитал две книги. Первой из них была книга *Архитектура корпоративных программных приложений* Мартина Фаулера [Fowler PoEAA]. Эта книга вдохновила меня на очередное опробование модели предметной области после ряда предыдущих неудачных попыток.

Затем я прочитал книгу *Domain-Driven Design* Эрика Эванса [Evans DDD]. Эта книга дала мне представление о том, как мыслить и действовать в процессе разработки, уделяя основное внимание предметной области, и, в какой-то степени, как применять шаблон модели предметной области.

Еще одно влияние на мое решение оказали курсы по шаблонам, которые я вел в течение двух лет. В процессе общения с учащимися и работы с учебным материалом у меня сложилось определенное представление о данном предмете.

Мои взгляды на ППО претерпели изменения в ходе работы над весьма честолюбивым (и, к счастью, уже завершившимся) открытым проектом под названием Valhalla, который я разрабатывал вместе с Кристофером Скьолдборгом (Christopher Skjoldborg). (Большую часть работы выполнил Кристофер.)

Таким образом, я посчитал, что нужно написать книгу, которая должна быть в большей степени посвящена практике применения, чем теории, но в то же время под нее следует заложить твердое основание, которым могли бы послужить две упомянутые выше книги. Мне больше по сердцу именно практическое применение, поскольку я считаю себя, прежде всего, разработчиком.

Кому адресована книга

Эта книга адресована широкому кругу читателей. Она окажется полезной тем, у кого имеются определенные знания в следующих областях.

- Объектная ориентация
- .NET или аналогичная платформа
- C# или аналогичный язык программирования
- Реляционные базы данных, например, SQL Server

Тем не менее, интерес и увлеченность читателя способны восполнить недостающие знания и опыт работы в перечисленных выше областях.

Попробую подробнее пояснить, почему эта книга рассчитана на широкий круг читателей. Прежде всего, нужно подумать о том, чтобы разработчики не замыкались в рамках используемых платформ. Тем, кто работает в .NET, книга поможет освоить более фундаментальный подход, чем метод типа "перетаскивай, пока не опустишь" (если можно позволить себе такое поверхностное обобщение). А те, кто работает с Java, должны извлечь для себя пользу из описания и примеров совместного использования ППО и объектно-реляционного преобразования.

На мой взгляд, выбор языка проектирования или платформы имеет все меньшее значение, поэтому было бы не совсем логично ориентировать книгу только на тех, кто работает с .NET или Java. Намного полезнее попытаться выяснить круг читателей, подойдя к этому вопросу с другой меркой. И в этом отношении книга адресована руководителям проектов и разработчикам программного обеспечения и его архитектуры.

Эта книга также может оказаться полезной для читателей как промежуточного, так и продвинутого уровня подготовки, а в какой-то степени и для начинающих.

Как организована книга

Эта книга состоит из следующих четырех частей.

Часть I. Основы

В этой части архитектура приложений и процессы рассматриваются в общих чертах. Основное внимание уделяется моделям предметной области и ППО [Evans DDD]. Кроме того, в ней вводятся понятия шаблонов и РПТ. В эту часть включены следующие главы.

- Глава 1. Переоценка ценностей

  В этой главе рассматриваются свойства оцениваемой архитектуры приложений и процесса для получения качественных результатов с точки зрения разработки системы. Тема для обсуждения навеяна экстремальным программированием.

- Глава 2. Переход к шаблонам
  В этой главе приведены примеры и рассмотрены шаблоны из разных семейств, включая шаблоны проектирования, архитектурные шаблоны и шаблоны предметной области.
- Глава 3. РПТ и рефакторинг кода
  В главе 1 уделяется достаточно внимания РПТ и рефакторингу кода, а в этой главе данные вопросы рассматриваются более углубленно и приводятся довольно обширные примеры и разновидности РПТ.

Часть II. Прикладное ППО

В этой части речь идет о прикладном ППО, а также о подготовке модели предметной области для инфраструктуры и соответствующих правилах.

- Глава 4. Новая используемая по умолчанию архитектура
  В этой главе перечисляются требования к примеру приложения, а также создается первая пробная модель в качестве основания для материала последующих глав. При этом используется архитектура на основе применяемой модели.
- Глава 5. Проектирование на основе модели предметной области
  Требования, сформулированные в предыдущей главе, используются в этой главе в качестве основания для того, чтобы постепенно, выполняя РПТ, приступить к построению модели предметной области в стиле ППО.
- Глава 6. Подготовка к инфраструктуре
  Несмотря на то что рассмотрение различных аспектов инфраструктуры мы пытаемся отложить на как можно более дальний срок, о них все же целесообразно подумать заранее и подготовить модель предметной области к инфраструктуре с учетом ее потребностей. В этой главе подробно рассматриваются достоинства и недостатки моделей предметной области без учета сохраняемости.
- Глава 7. Порядок устанавливают правила
  В этой главе речь идет о бизнес-правилах в форме проверки достоверности, а также об учете в решении на основе модели предметной области потребностей в этих правилах с отсылкой на требования, сформулированные в главе 4.

Часть III. Применение шаблонов архитектуры корпоративных программных приложений

В этой части некоторые шаблоны из книги *Архитектура корпоративных программных приложений* Мартина Фаулера [Fowler PoEAA] переносятся в контекст дискуссии о том, что требуется от инфраструктуры для поддержки сохраняемости в модели предметной области. Выполнение подобных требований рассматривается на примере отдельного инструмента.

- Глава 8. Инфраструктура для сохраняемости
  Когда уже имеется довольно удачная модель предметной области, самое время подумать об инфраструктуре, а в данной книге это, главным образом, инфраструктура для сохраняемости. В этой главе рассматриваются различные свойства решений сохраняемости и отнесение определенного решения к конкретной категории.
- Глава 9. Приведение в действие преобразователя NHibernate
  Разделение решений на категории, выполненное в предыдущей главе, используется на примере решения сохраняемости, а именно преобразователя NHibernate [NHibernate].

Часть IV. Что дальше

В этой части основное внимание уделяется другим методам проектирования, на которые следует обратить внимание, а затем и применять. Кроме того, здесь показано, как действовать на уровне представления, когда требуется навести мосты между ним и моделью предметной области и как организовать тестирование пользовательского интерфейса на стадии разработки. Эта часть почти полностью написана приглашенными авторами.

- Глава 10. Рекомендуемые методы проектирования
  После краткого изложения ограниченного контекста в этой главе рассматриваются методы проектирования, на которые следует обратить внимание, включая ориентацию на обслуживание, внесение зависимостей, инверсию управления, аспектную ориентацию.
- Глава 11. Акцент на пользовательском интерфейсе
  В этой главе основное внимание уделяется сопряжению пользовательского интерфейса с моделью предметной области, а также повышению проверяемости пользовательского интерфейса при использовании модели предметной области как в насыщенных клиентских приложениях, так и в Web-приложениях.

Приложения

В двух приложениях к книге представлены дополнительные примеры видов модели предметной области и перечень шаблонов с кратким их описанием.

Основания для примеров на C#

Эту книгу ни в коем случае не следует рассматривать как учебник по C#. Тем не менее этот язык программирования понадобился мне для примеров.

Я выбрал этот язык, главным образом, потому, что пользуюсь в настоящее время именно им, а большинство программирующих на VB.NET и Java могут свободно читать исходный текст и на C#.

Что же касается конкретной версии, то бо́льшая часть кода в приведенных примерах вполне работоспособна в версиях C# 1.1 и 2.0, хотя в редких случаях отдельные фрагменты кода относятся к версии 2.0.

Темы, не охваченные в книге

За рамками этой книги осталось немало тем, но, на мой взгляд, в ней не были рассмотрены две важные темы: распределение и усовершенствованное моделирование.

Распределение

Первоначально я планировал включить в книгу материал, посвященный распределению, но затем пришел к выводу, что книга получится слишком распыленной. Тем не менее распределение упоминается в отдельных местах книги.

Усовершенствованное моделирование

Из заглавия книги читатель может сделать вывод, что в ней он найдет интересные примеры усовершенствованного моделирования в некоторых предметных областях. Но это не совсем так. Основное внимание уделяется РПТ и дополнению ППО соответствующей инфраструктурой.

finally{}

Нетрудно догадаться, что такая книга является плодом труда не одного человека, а целого коллектива. В приведенном ниже разделе “Благодарности” перечислены все те, кто принимал участие в работе над книгой и ее выпуске. Тем не менее, все ошибки, которые не удалось обнаружить до сдачи книги в печать, являются моими и только моими, как основного автора.

Дополнительную информацию о книге можно найти по адресу `ww.jnsk.se/adddp`.

Если вернуться к наведению мостов, то фотография моста через пролив Орезунд была сделана моим другом Магнусом фон Шенком (Magnus von Schenk) во время одной из его прогулок по морю под парусом.

Написать эту книгу оказалось легче, чем первую, тем не менее, она стоила мне немало крови, пота и слез. Надеюсь, что она поможет вам сберечь немного того, другого и третьего. Желаю вам получить удовольствие от книги. Удачи!

Джимми Нильссон
`www/jmsk.se/weblog/`
Листерби, Швеция
сентябрь 2005 года

Благодарности

Рискуя забыть кого-нибудь из тех, кого следует непременно здесь упомянуть (в этом случае я заранее приношу свои извинения и благодарю их), я выражаю свою признательность следующим лицам.

- Авторам предисловий: Мартину Фаулеру и Эрику Эвансу.
- Приглашенным авторам: Франсу Боуме, Дану Бистрёму, Уди Дахану (Udi Dahan), Эрику Дёрненбургу (Eric Dürnenburg), Матсу Хеландеру, Ингмару Лундбергу, Филипу Нельсону, Клаудио Перроне (Claudio Perrone), Александару Сеовичу (Aleksandar Seović) и Кристоферу Скьолдборгу.
- Рецензентам, которые дали мне немало полезных советов: Уильяму Булли (William Bulley), Марку Берхопу (Mark Burhop), Дану Бистрёму, Рассу Кондику (Russ Condick), Энди Конраду (Andy Conrad), Кристиану Кроухэрсту, Майку Дёрфлеру (Mike Dürfler), Стиву Айхерту (Steve Eichert), Эрику Эвансу, Мартину Фаулеру, Полю Гиленсу, Крису Хаддаду (Chris Haddad), Киму Хардингу Кристенсену (Kim Harding Christensen), Матсу Хеландеру, Нираю Гупте (Neeraj Gupta), Андерсу Хесселлунду, Роджеру Йоханссону (Roger Johansson), Роджеру Кратцу (Roger Kratz), Тронду-Эйрику Коллоэну, Ингмару Лундбергу, Патрику Лёвендалю (Patric Lüwendahl), Маркусу Макиннесу (Marcus MacInnes), Филипу Нельсону, Перу-Ола Нильссону, Фредрику Нормену (Fredrik Normén), Йохану Нормену (Johan Normén), Майклу О'Брайену (Michael O'Brien), Майклу Платту (Michael Platt), Себастьену Росу (Sébastien Ros), Мартину Розену-Лидхольму, Александару Сеовичу, Кристоферу Скьолдборгу, Джорджу Вишу II (George Vish II), Грегори Янгу и Кристену Окессону (Christen Åkesson).
- Художнику-оформителю книги Кьеллу Варнквисту (Kjell Warnquist)
- Редактору Лидии Уэст (Lydia West Zahn)
- Редактору-консультанту Крису Зану (Chris Zahn)
- Редактору проекта Майклу Терстону (Michael Thurston)
- Литературному редактору Элизе Уолтер (Elise Walter)
- Управляющему сбытом Керту Джонсону (Curt Johnson)
- Рецензенту издательства Карен Геттман (Karen Gettman), а также Сондре Скотт (Sondra Scott), которая первой уговорила меня взяться за этот проект.

Без всех этих людей книга не увидела бы свет или, по крайней мере, была бы ужасного качества. Благодарю, друзья. Я перед вами в долгу!

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. 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 свои замечания там. 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 электронные адреса:

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

Наши почтовые адреса:
в России: 115419, Москва, а/я 783
в Украине: 03150, Киев, а/я 152

Proekt_book.indb 38 27.08.2007 21:32:23

Часть I

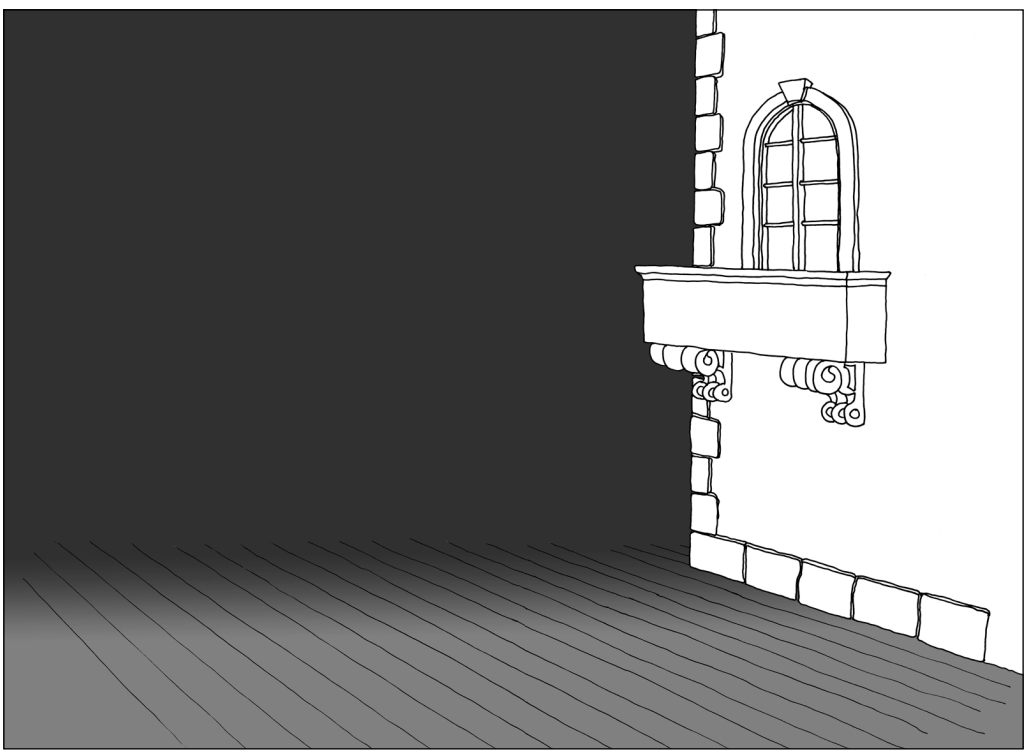
Основы

Глава 1. Переоценка ценностей

Глава 2. Переход к шаблонам

Глава 3. РПТ и рефакторинг кода

В этой части архитектура приложений и процессы рассматриваются в общих чертах. Основное внимание уделяется моделям предметной области (Domain Models) и *проектированию на основе предметной области* (ППО — Domain-Driven Design (DDD)) [Evans DDD]. Кроме того, в этой части вводятся понятия *шаблонов* и *разработки посредством тестирования* (РПТ — Test-Driven Development (TDD)).



Первая часть посвящена подготовке сцены. У Ромео и Джульетты должна быть сцена

Глава 1

Переоценка ценностей

Или беспорядочные мысли по поводу эволюции собственных представлений за последние годы

Назначение этой главы — подготовить сцену. С этой целью я намерен оглянуться в прошлое на несколько лет назад, чтобы проанализировать свои представления о различных понятиях и показать, как они менялись со временем.

Нам предстоит пройти немалый путь и охватить много вопросов, но общая идея состоит в том, чтобы осуществить *переоценку ценностей* в отношении архитектуры и процессов для разработки программного обеспечения.

На этом пути будут введены и описаны многие понятия, которые рассматриваются более подробно далее в этой книге.

Итак, мотор... Начали!

Общие ценности

В прошлом я немало преуспел, планируя заранее. Я нередко вводил функциональные возможности, структуры и механизмы в свои проекты с упреждением. Это обычно приносило свои плоды, но я часто забывал о сопутствующих артефактах, которые никогда не приносили особой пользы. Разумеется, их накапливалось очень много. Плата за это была довольно высокой как с точки зрения времени разработки, так и в смысле повышения сложности приложений.

За последние несколько лет нас поощряли применять другой подход: “Искать самое простое решение, которое могло бы стать работоспособным.” Этот принцип исходит, главным образом, из направления, называемого *экстремальным программированием* (ЭП — Extreme Programming (XP) [Beck XP]. Его можно выразить и по-другому: “Это вам не понадобится” (You Aren’t Going to Need It — YAGNI), что неплохо помогает не выходить за определенные границы. К этому, на мой взгляд, можно отнести и следующий принцип: “Не будь глупцом и упрощай” (Keep It Simple, Stupid — KISS).

Оба упомянутых выше подхода — своего рода крайности (попробуйте найти аргументы против идеи искать самое простое решение), но мне кажется, что им чего-то не хватает в том отношении, что они никак не решают проблему компромиссов, которые они создают. На вопрос: "Является ли то или иное решение самым лучшим?" можно ответить: "Как сказать". Все дело в компромиссах. Я лично предпочитаю искать решение где-то посредине, двигаясь в том или ином направлении в зависимости от ситуации. Слово "лагом" означает по-шведски "в самый раз", т.е. "ни больше ни меньше". Таким образом, лагом, или сохранение равновесия, а также связи с контекстом наряду с постоянным обучением относятся к тем общим ценностям, которым я отдаю предпочтение.

А теперь рассмотрим подробнее более конкретные категории ценностей (составляющие архитектуры и процесса), начиная с тех, что имеют непосредственное отношение к архитектуре, чтобы настроиться на нужный лад.

Оценка стилей архитектуры

Разработчикам программного обеспечения и его архитектуры приходится принимать проектные решения на основании требований к проекту. Я не собираюсь приводить здесь полный перечень ценностей, имеющих отношение к стилям архитектуры, но постараюсь подробнее остановиться на самых главных из них, включая акцент на модели, модели предметной области, базы данных, распределение и обмен сообщениями.

Акцент на модели

Мне уже давно пришелся по душе объектно-ориентированный принцип, но лишь недавно я полностью перешел на него. Прежде у меня возникали трудности с применением этого принципа на отдельных платформах, но теперь платформы стали достаточно зрелыми.

На заметку

Как верно заметил один из рецензентов книги, зрелость зависит от рассматриваемой платформы. Это звучит вполне обоснованно для тех, кто программирует на VB, но для тех, кто программирует на Java, SmallTalk, C# и т.д., созревание платформы происходило довольно долго.

Более десяти лет назад я попытался использовать визуальную модель, чтобы передать свое представление о требованиях к системе, которую я собирался построить, и воспользовался для этого некоторыми упрощенными структурами MMO [Rumbaugh OMT]. (MMO означает *метод моделирования объектов* — Object Modeling Method

(OMT) с соответствующим процессом и записью. Запись оказалась очень похожей на ту, что имеется в *унифицированном языке моделирования* (Unified Modeling Language — UML), что нельзя назвать простым совпадением, поскольку запись в UML была составлена под непосредственным влиянием MMO.) При этом мы обсуждали множественность среди классов, к чему должно относиться поведение и прочие вопросы. После нескольких совещаний я осознал, что ошибочно применяю свой метод обсуждения насущных вопросов с опытными пользователями вместо их метода. Они отвечали на мои вопросы произвольно и не видели в таком обсуждении никакой пользы, не говоря уже о том, чтобы придать ему хоть какую-то ценность. (В целом система была построена правильно, и хотя она используется до сих пор как базовая, тем не менее процесс ее разработки происходил не так гладко, как следовало бы. Впоследствии я изменил свой метод.)

Акцент на прецедентах использования

Я не раз размышлял об этом опыте общения с пользователями и подшучивал над тем, каким наивным я был. В последующие годы я старался действовать по методике целевых групп, т.е. по методике использования для пользователей и методике разработки программного обеспечения для разработчиков. Для общения обеих сторон вполне подошел метод прецедентов использования [Jacobson OOSE]. (По крайней мере, он хорошо зарекомендовал себя в неформальной форме историй пользователей в ЭП [Beck XP], т.е. коротких текстовых описаний, которыми я пользовался в своей работе. Подобные истории вкратце описывают часть функций системы. Примером тому может служить “Регистрация заказа для клиента компании”.)

Такой подход оказался вполне естественным для пользователей и мог стать таковым и для разработчиков, поэтому он многие годы служил мне ценным инструментом для наведения мостов. А наводил я мосты таким образом, чтобы на каждый прецедент использования приходился один класс в программе.

Я обратил внимание, что, применяя по-своему прецеденты использования, я стал мыслить вполне процедурными категориями. При этом я занимался разработкой, чем-то похожей на создание сценария транзакции (Transaction Script) [Fowler PoAAA], но при этом старался сохранить равновесие, как можно больше обобщая поведение объектов (по крайней мере, когда это было уместно).

Я ознакомился с идеями Ивара Якобсона (Ivar Jacobson) о прецедентах использования и был весьма поражен, когда осознал, что он не инкапсулировал прецеденты использования в отдельных классах, как я предполагал и делал долгое время.

В этом методе меня беспокоил еще один вопрос: постоянная борьба, которую мне приходилось вести с моим коллегой-разработчиком проекта Valhalla, Кристофером Скьолдборгом, когда мы работали на шаблоном модели предметной области [Fowler PoAAA]. Он не видел никакой пользы в классах прецедентов использования, утверждая, что они могут даже стать помехой при комбинировании отдельных частей прецедентов использования.

Различные шаблоны для основной логики

Прежде чем продолжать дальше, мне хотелось бы остановиться вкратце на упоминавшихся выше понятиях сценария транзакции и модели предметной области.

В своей книге *Архитектура корпоративных программных приложений* [Fowler PoEAA] Мартин Фаулер рассматривает три способа структурирования основной логики в архитектуре приложения: сценарий транзакции (Transaction Script), модуль таблицы (Table Module) и модель предметной области (Domain Model).

Сценарий транзакции аналогичен пакетным программам в том отношении, что все функциональные возможности описываются в нем от начала и до конца в отдельном методе. Сценарий транзакции довольно прост и удобен для решения простых задач, но он оказывается бесполезным, когда дело доходит до приложений большой сложности. При этом трудно избежать дублирования. Тем не менее, многие довольно крупные системы построены именно таким образом. Мой опыт построения таких систем показывает, что дублирование становится явным, как бы ни стараться его сократить. Он все равно проявляется.

Модуль таблицы инкапсулирует шаблон Recordset [Fowler PoEAA], после чего вызываются методы из этого модуля для получения информации, например, о заказчике с идентификационным номером 42. Для получения имени заказчика вызывается метод и передается идентификационный номер 42 в качестве его параметра. Для ответа на запрос внутри модуля таблицы используется шаблон Recordset. Это, безусловно, достоинство данного модуля, особенно в средах с достаточно полной реализацией шаблона Recordset. Но дело в том, что среди разных модулей таблиц наблюдается все та же тенденция к дублированию. Еще один недостаток таких модулей состоит в том, что они не позволяют использовать полиформистские решения для отдельных задач, поскольку заказчик вообще не видит идентичности объектов, а лишь идентичности, характеризуемые значением. Это в какой-то мере похоже на применение реляционной модели вместо объектно-ориентированной.

Напротив, в модели предметной области используется объектная ориентация для того, чтобы максимально приблизить ее описание к выбранным абстракциям. Она отлично подходит для приложений большой сложности, поскольку максимально использует возможности объектной ориентации и позволяет легче добиться подлинного соответствия предметной области. Разумеется, использование шаблона модели предметной области не лишено трудностей. К типичным трудностям такого рода относится непростой путь обучения эффективному использованию этого шаблона.

Акцент на проектировании на основе предметной области

Что касается применения прецедентов использования, то для меня явилось полным откровением проектирование на основе предметной области (ППО — Domain-Driven Design (DDD) [Evans DDD]. Подробнее о ППО речь пойдет далее, а пока достаточно сказать, что его суть — в сосредоточении основного внимания на предметной области, с тем чтобы она оказывала значительное влияние на программное обеспечение. Я по-прежнему считаю, что применение метода прецедентов использования отлично подходит для общения с пользователями, но с другой стороны, я подумал о том, что ППО может оказать немалую помощь, если суметь вовлечь пользователей в активное обсуждение базовой модели. Оно позволяет избежать ошибок на самой ранней стадии и помогает разработчикам лучше понять предметную область.

На заметку

"Модель" относится к одним из самых перегруженных понятий в программировании. У каждого из нас, на мой взгляд, имеется интуитивное представление о том, что это такое, но мне бы хотелось пояснить свое представление о модели. Вместо того чтобы пытаться сформулировать определение модели, я ради простоты (поскольку на эту тему можно было бы написать отдельную книгу) приведу ряд свойств модели, чтобы вы могли сравнить их с собственным представлением о модели. Итак, модель это:

- Нечто частное
- Имеющее определенное назначение
- Система абстракций
- Познавательный инструмент

Кроме того,

- Модель имеет несколько представлений (например, языковое, кодовое, схемное).
- В системе обычно действуют несколько моделей.

На тему моделей написано немало специальной литературы. Начинающим рекомендую первые страницы упоминавшейся ранее книги Эрика Эванса [Evans DDD].

Модель служит отличным инструментом для общения разработчиков с пользователями, и чем лучше это общение налажено, тем более качественным получается программное обеспечение как в краткосрочной, так и в долгосрочной перспективе.

Мы (и я в том числе), конечно, пользовались моделями всегда, но отличие между одной из моих старых моделей и теми, что я построил на принципах ППО, заключается в том, что в моих старых моделях акцент делался, главным образом, на инфраструктуру и техническую концепцию. А мои новые модели в значительной степени

свободны от подобных отвлеченностей и, напротив, полностью сосредоточены на базовой предметной области, ее основных понятиях и текущих задачах. Это большой прогресс в мышлении.

Иными словами, технические вопросы (например, доводка пользовательского интерфейса) появляются и исчезают, а предметная область остается. А когда меняется базовая предметная область, нам приходится вносить изменения как в саму модель, так и в программное обеспечение.

Разумеется, ничего нового я до сих пор не открыл, и, возможно, ломлюсь в открытые двери. Тем не менее, я считаю, что данный подход составляет основу методов достижения эффективности программного обеспечения и, как показывает мой опыт, он применяется редко. Под этим я подразумеваю акцент на модели, наличие модели (с разными представлениями), применяемой как пользователями, так и разработчиками, а также способность не отвлекаться на ненужные детали.

Основания для акцента на модели

Обратимся к двум крайним примерам. Какой разработчик лучше всего подойдет для построения вертикальной системы в специальной области, допустим, в финансовой? Безусловно, тот, кто имеет хорошую техническую подготовку, высокую квалификацию и значительный опыт, обширную сеть социальных связей и т.д. Кроме того, было бы неплохо, если бы он достаточно хорошо разбирался в самой предметной области, например, работал биржевым маклером с десяток лет, если система создается именно в таких целях.

Найти такого разработчика, конечно, можно, но, как показывает мой опыт, это скорее исключение, чем правило. Это исключение становится еще более очевидным, когда разработчик переходит к созданию новой системы в другой предметной области после запуска в эксплуатацию системы управления финансами. Ведь у него нет такого же десятилетнего опыта работы, например, в материально-техническом снабжении, здравоохранении, страховании и прочих сферах деятельности.

С другой стороны, можно дать пользователям, хорошо знающим предметную область, самим разработать программное обеспечение. Об этом многие мечтали уже давно и в какой-то степени эти мечты постепенно становятся явью. В то же время на это у пользователей уходит немало времени, которое они могли бы с большей пользой провести на своей основной работе. Кроме того, им придется преодолеть немало технических трудностей.

А есть ли другой, лучший выход? Ответ очевиден (по крайней мере, с современной точки зрения): разработчик должен узнать как можно больше о предметной области, с которой ему приходится работать, и привлечь к проекту пользователей, чтобы они могли не только передать свой опыт работы в предметной области коллективу разработчиков, но и активно и конструктивно участвовать в самом проекте. Ведь пользователи не только формулируют требования, хотя и это очень важно, но фактически помогают разрабатывать базовую систему. Если удастся создать такую атмосферу, то вряд ли найдется более квалифицированный специалист, чем пользователь, имею-

щий опыт работы в предметной области, когда придется решать, какой должна быть базовая модель, каковы ее основные абстракции и т.д.

Разумеется, разработчики тоже нужны. Но вся работа над проектом выполняется в тесном сотрудничестве с пользователями. То, что заказчикам показалось мелкой деталью, для разработчиков может оказаться очень важным в модели предметной области. Например, заказчики могут сказать: "Да, иногда бывает и так", и тогда вся картина совершенно меняется, но поскольку это нетипичный случай, они посчитают его не столь важным.

Время для повторного обсуждения модели с заказчиками

Зачем продолжать обсуждение модели с заказчиками, если предыдущие попытки ни к чему не привели? С тех пор многое изменилось. Во-первых, важно иметь ясное понимание контекста и возможностей проектной группы. Во-вторых, прецеденты использования могут оказать большую помощь в начале работы над проектом, после чего основное внимание можно уделить самой базовой модели. В-третьих, построение систем — это сфера деятельности, работать в которой или изучать которую приходится все большему числу людей. И в-четвертых, строго говоря, модель, представленная в графическом, схематическом, UML-подобном виде, не так важна для работы с пользователями, хотя такое ее представление по-прежнему более желательно для разработчиков, чем для пользователей. С этой целью осуществляется поиск *универсального языка* [Evans DDD].

Универсальный язык не похож на UML, XML Schema или C#. Это естественный, но очень емкий язык для предметной области. Именно на нем разработчики общаются с пользователями при описании решаемой задачи и модели предметной области. Вместо того чтобы слушать пользователей и пытаться перевести их истории на язык, понятный разработчикам, универсальный язык дает меньше оснований для непонимания, а пользователям становится легче улавливать упрощенные схемы системы, и они на самом деле помогают разработчикам исправить ошибки и узнать что-то новое о предметной области. Если это уместно, универсальный язык можно обсуждать в контексте графического или кодового представления моделей. В противном случае самое главное все равно сделано, если универсальный язык стал понятным.

На заметку

Эрик Эванс так прокомментировал приведенные выше соображения: "Следует ясно осознавать, что универсальный язык — это не жаргон специалистов в конкретной предметной области. Ведь у жаргона имеется слишком много неоднозначностей и допущений, а возможно, и слишком большая область действия. А универсальный язык подразумевает сотрудничество между специалистами в конкретной предметной области и специалистами по программному обеспечению. (Разумеется, он будет напоминать подмножество жаргона в данной предметной области.)"

Над универсальным языком придется немало поработать, чтобы сохранить его четко определенным и согласованным с программным обеспечением. Например, изменения в таком языке могут привести к изменениям в программном обеспечении, и наоборот. Оба эти артефакта должны оказывать влияние друг на друга.

Если сделан акцент на модели, следует применять шаблон модели предметной области

Если мы согласны с тем, что нужно делать акцент на глубоко проработанной модели и использовать объектную ориентацию, то, на мой взгляд, вполне естественным результатом этого должно стать применение шаблона модели предметной области [Fowler PoEAA] для структурирования базовой логики приложений и служб. Пример небольшой модели предметной области приведен на рис. 1.1.

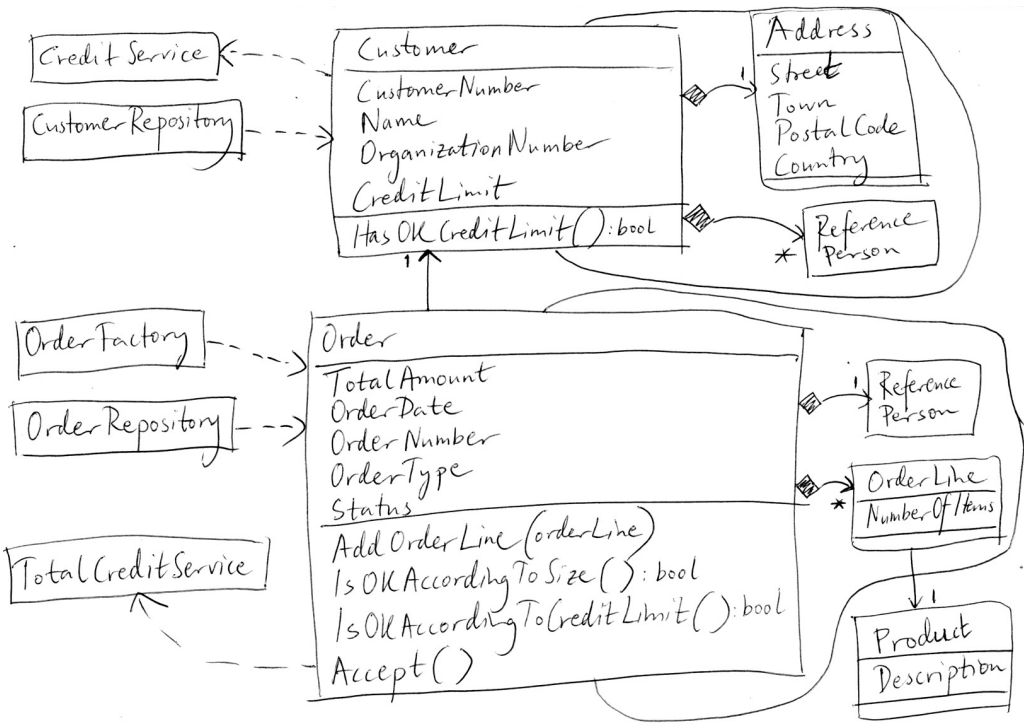


Рис. 1.1. Пример модели предметной области в виде первоначальной упрощенной схемы для примера приложения, рассматриваемого в главе 4

Несмотря на то что на рис. 1.1 показана лишь первоначальная упрощенная схема небольшой модели предметной области, нетрудно заметить, что эта модель выражает понятия предметной области без отвлечения на технические детали. Кроме того, она состоит из нескольких взаимодействующих мелких частей, образующих вместе единое целое.

На заметку

Если вы еще не освоили язык UML, рекомендуется прочитать хорошую книгу *UML. Основы* [Fowler UML Distilled]. Знать язык UML не обязательно, но, на мой взгляд, полезно иметь представление об его основах, поскольку в настоящей книге UML используется в качестве инструмента для составления упрощенных схем (разумеется, он может пригодиться вам и в других случаях, а не только при чтении данной книги).

На заметку

Если вас интересует, почему рис. 1.1 сделан от руки, следует иметь в виду, что это лишь набросок модели. Конкретные детали в коде будут исследованы и разработаны, в частности, с использованием разработки посредством тестирования (РПТ), далее в этой главе, а также в главе 3.

Прежде чем переходить к конкретному стилю, использованному мной для модели предметной области, вернемся немного назад.

История применения модели предметной области в свете моего прошлого опыта

Я все еще помню, как в первый раз попытался использовать модель предметной области, хотя тогда я так ее не называл. Это было где-то в 1991 году, и тогда я не мог понять, как согласовать модель с пользовательским интерфейсом. Я уже забыл, какое это было приложение, но помню, что решил выяснить, как должны выглядеть меню, чтобы иметь объектно-ориентированный характер. Эту первую преграду мне тогда так и не удалось преодолеть. Похоже, что я искал близкое соответствие между пользовательским интерфейсом и моделью предметной области и не мог найти способ добиться его.

Несколько лет до этого мне пришлось работать над приложением, в котором применялись объектно-ориентированные принципы, в частности, для создания пользовательского интерфейса. Больших достижений это, может быть, и не принесло, но кое-чего мне все же удалось добиться. Так, вместо того чтобы сначала выбрать функцию, а затем решить, к какому объекту ее применить (например, к папке), пользователь сначала выбирал объект, а затем делал выбор между разными функциями, доступными для выбранного объекта. В то время это был совершенно иной принцип, чем тот, по которому работает пользовательский интерфейс большинства современных бизнес-приложений.

О голых объектах

Возможно, я и не совсем ошибался в своих намерениях. Вероятно, я пытался добиться чего-то вроде *голых объектов* [Pawson/Matthews Naked Objects], но на самом деле я достиг этим не очень многого.

Не вдаваясь в подробности, нужно вкратце пояснить, что собой представляют голые объекты. Основная идея заключается в том, что не только разработчики, но и пользователи предпочитают объектную ориентацию. Она нравится им даже больше, чем мы думаем, и поэтому они довольно часто оказываются довольны пользовательским интерфейсом, который очень близко соответствует самой модели предметной области, причем настолько близко, что пользовательский интерфейс может быть создан автоматически с помощью структуры, основанной непосредственно на модели предметной области.

Система на основе голых объектов автоматически предоставляет пользователю форму, содержащую реквизиты, раскрывающие свойства в классе модели предметной области.

Для решения более сложных задач (как всегда) требуется некоторая специальная настройка, но принцип остается тем же: использовать структуру, "понимающую", какой именно пользовательский интерфейс требуется создать из фрагментов модели предметной области. Следовательно, когда готова модель предметной области, то более или менее готов и пользовательский интерфейс.

В настоящее время у меня нет опыта применения данного принципа и метода, но меня привлекает сама идея: пользователи получают реальную возможность почувствовать модель, что очень полезно для наведения мостов между ними и разработчиками.

После этого мне пришлось не раз опробовать на практике подходы к модели предметной области. И я столкнулся с определенными трудностями при ее применении, и, в частности, это касается снижения производительности (особенно в вариантах распределенных систем). Некоторые из моих реальных приложений были созданы именно в этом стиле, особенно самые простые. Следует, однако, заметить, что несмотря на все мои старания построить весьма эффективные, правильно сконструированные модели предметной области, они получались не такими, какими я предполагал их построить.

Но, как оказалось, это была не единственная трудность...

Старые истины могут оказаться ложными

Я верил в модели предметной области и пытался в прошлом применить их несколько раз, но большинство моих попыток реализовать их в объектно-ориентированном виде привели меня к заключению, что такой способ не действует. К примеру, основной недостаток моих приложений COM+, созданных в Visual Basic 6 (VB6), состоял в слишком заметном снижении производительности.

Когда я писал свою первую книгу [Nilsson NED], где описывалась используемая по умолчанию архитектура приложений .NET, я обратился к своему опыту работы с VB6/COM+ и вообще не касался архитектуры на основе модели предметной области.

Чуть позже я начал новые эксперименты, пытаясь выяснить, удастся ли мне сократить хотя бы на 10% время реакции в типичных ситуациях (например, при выборке одного заказа, выборке списка заказов для клиента или при сохранении заказа), когда в .NET применяется модель предметной области в сравнении со сценариями транзакций и шаблоном RecordSet. К своему удивлению я обнаружил, что мне удалось сократить такое снижение. Таким образом, старая истина оказалась ложной. Архитектура на основе модели предметной области оказывается теперь намного более реальной в .NET, чем в VB6. Объясняется это, в частности, сокращением времени создания экземпляра на несколько порядков величины. Поэтому при наличии очень большого числа экземпляров время реакции оказывалось небольшим. Еще одна причина состояла в том, что в .NET очень просто писать компоненты типа `Marshal By Value` (Маршализация по значению). При этом перемещается не только ссылка на экземпляр, но и весь экземпляр (по крайней мере, концептуально). Сделать то же самое в среде COM было непросто, а в VB6 — практически невозможно.

На заметку

Компонент `Marshal By Value` несколько утратил свое значение, поскольку теперь модель предметной области можно не пересылать в неизменном виде по каналам связи, но в своих ранних пробах я делал это.

Еще одна причина состоит в том, что в .NET лучше поддерживается объектная ориентация, поскольку эта платформа обладает намного более совершенным набором инструментов.

На заметку

Следует также упомянуть, что в сообществе разработчиков приложений на платформе Microsoft не уделялось должного внимания модели предметной области в прошлом. Совсем другое отношение к ней сложилось в сообществе разработчиков приложений на платформе Java. Я помню, как посещал семинар, на котором в основном присутствовали разработчики, программирующие на Java. Я спросил их тогда, какую структуру они предпочитают для логики, высказав предположение, что они, вероятно, пользуются шаблоном Recordset (введенным в версию 3 JDBC). Они посмотрели на меня так, как будто не поняли вопроса. Я быстро сообразил, что модель предметной области уже стала фактическим стандартом в их среде.

Для того чтобы внести ясность в данный вопрос, я хотел бы уточнить два момента. Во-первых, я *не* настаиваю на том, что шаблон модели предметной области следует непременно выбирать из соображений производительности. Напротив, такой шаблон *можно* зачастую выбирать, не навлекая осложнений, связанных с производительностью.

Во-вторых, я *не* утверждаю, что для сосредоточения основного внимания на модели требуется определенная технология. Для того чтобы выразить модель в программном обеспечении с точным соответствием предметной области, годятся в той и или иной степени разные технологии. Разумеется, одной самой лучшей технологии на все случаи жизни не существует. Разные предметные области и их задачи определяют разные критерии пригодности технологии. Поэтому технологию я рассматриваю как некое средство реализации. В этом смысле одни технологии оказываются более подходящими средствами реализации, чем другие.

Таким образом, эффективность применения шаблона модели предметной области во многом определяется самим проектированием.

В этом есть свои преимущества, поскольку по ходу дела можно получить немало полезной информации. Проектирование на основе предметной области и его стиль структурирования моделей предметной области оказывает в этом неоценимую помощь, и именно этому вопросу посвящена бо́льшая часть данной книги.

Одна корневая структура

Итак, сосредоточение основного внимания на модели предметной области означает, что все участники проекта привлекаются к построению модели. И прежде всего это, конечно, касается разработчиков.

Вполне возможно, что даже администраторы баз данных согласятся рассматривать модель предметной области как корневую структуру, несмотря на то что проектирование баз данных осуществляется несколько иначе. В таком случае можно считать, что администраторов баз данных также удалось уговорить перейти на универсальный язык общения! (На самом деле чем больше вам удастся убедить администратора баз данных стать сторонником модели предметной области и придерживаться ее, тем легче вам будет ее реализовать. Ведь тогда придется создавать меньше кода преобразования, если проектирование базы данных не будет радикально отличаться от модели предметной области, а станет лишь другим представлением той же самой модели.) И даже пользователи могут быть привлечены к построению модели предметной области. Конечно, у разных заинтересованных сторон разные требования к такой модели и разные представления, например, об уровне детализации, тем не менее, это одна корневая структура, с которой им предстоит уживаться, развивать и изменять ее...

В этой книге основное внимание уделяется рассмотрению решений, основанных на модели предметной области, но прежде чем сделать это, мне бы хотелось остановиться вкратце на других архитектурных ценностях. Оставим на время предметную область и обратимся к некоторым вопросам технического свойства. Прежде всего, следует выяснить, как обращаться с базой данных.

База данных требует аккуратного обращения

В прошлом я обращал особое внимание на производительность при построении систем. В частности, я, как правило, организовывал доступ к базе данных с помощью написанных вручную хранимых процедур. Это обычно позволяло добиться достаточно высокой производительности во время выполнения, особенно в отсутствие поведения CRUD (Create, Read, Update, Delete — Создание, Чтение, Обновление, Удаление) для каждого экземпляра (например, клиента) в отдельности, но при наличии логически более развитых хранимых процедур; например, для выполнения ряда операций и воздействия на множество строк в каждом вызове.

Подходящая эффективность

Несмотря на рост производительности аппаратных средств по закону Мура, проблема эффективности по-прежнему актуальна. Ведь одновременно с увеличением производительности аппаратных средств усложняются и задачи, которые нам приходится решать программными средствами.

Как показывает мой опыт, проблемы производительности чаще всего связаны с неэффективностью кода доступа к базе данных, неудачной структурой базы данных и другими подобными аспектами. Одна из основных причин этого обычно кроется в том, что практически ничего не было сделано для настройки базы данных, а основное внимание было уделено лишь безукоризненности объектной ориентации. Это, в свою очередь, привело к чрезмерному количеству циклических обращений к базе данных, неэффективному и даже неправильному коду транзакции, неудачным схемам индексации и прочим осложнениям.

Рассмотрим эту проблему на более конкретном примере, в котором объектной ориентации было уделено больше внимания, чем самой базе данных. Допустим, что имеется класс клиента `Customer` со списком заказов `Order`. У каждого экземпляра `Order` имеется список экземпляров строк заказа `OrderLine`. Схематически это выглядит так, как показано на рис. 1.2.

Допустим далее, что нам очень важна модель и что нас не очень интересует, каким образом данные выбираются и сохраняются в используемой реляционной базе данных. В данном случае возможна следующая (хотя и несколько наивная) схема: возложить на каждый класс (`Customer`, `Order` и `OrderLine`) ответственность за собственную сохраняемость/несохраняемость. Для этого можно ввести класс типа Layer Supertype (Супертип слоя) [Fowler PoEAA], называемый `PersistentObject`, как показано в примере рис. 1.3. В этом классе реализуется, например, метод `GetById()`, и от него наследуются классы модели предметной области (см. рис. 1.3).

А теперь допустим, что требуется выбрать клиента (`Customer`) и все его заказы (`Order`), а каждом заказе — все строки заказа (`OrderLine`). Для этого можно воспользоваться следующим кодом:

```
// Потребитель
Customer c = new Customer();
c.GetById(id);
```

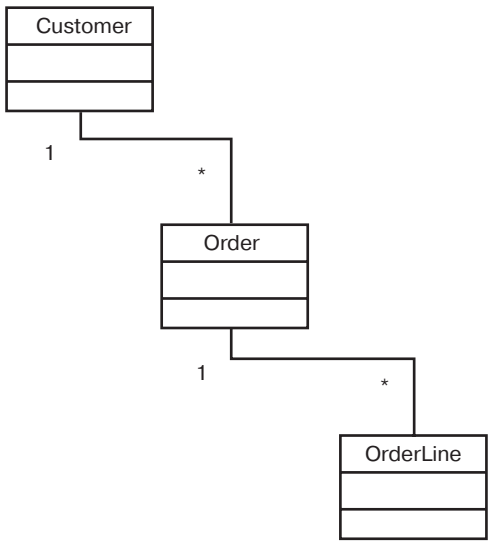


Рис. 1.2. UML-схема для примера клиента и его заказов

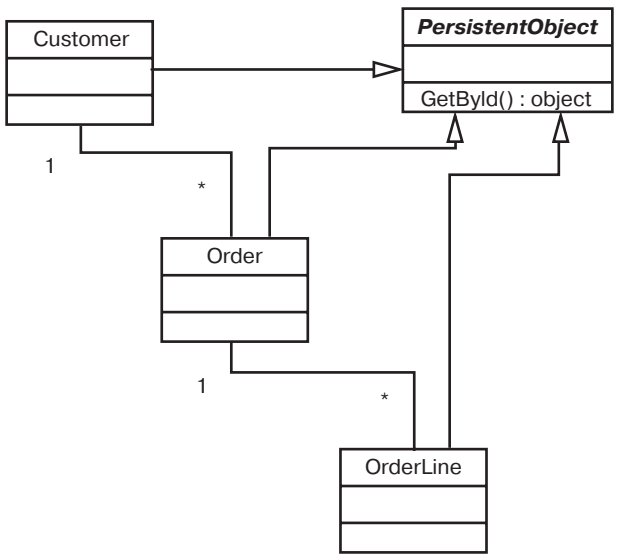


Рис. 1.3. UML-схема для примера клиента и его заказов с классом типа Layer Supertype

На заметку

Комментарий `// Потребитель` в приведенном выше фрагменте кода используется для того, чтобы показать имя класса (а возможно, и его методы) в подобных фрагментах кода ради ясности. Иногда имя класса не имеет особого значения (как в данном случае), и тогда вместо него указывается обобщенное имя.

В C++ этот принцип является встроенным, поскольку имена методов записываются следующим образом: `ClassName::MethodName`, но я считаю, что небольшой комментарий позволяет добиться того же результата.

В итоге будут выполняться следующие операторы SQL:

```
SELECT CustomerName, PhoneNumber, ...
FROM Customers
WHERE Id = 42
```

Далее метод `GetById()` осуществляет выборку списка заказов, но при этом будут выбираться только ключи посредством вызова метода `GetIdsOfChildren()`, который будет выполнять нечто следующее:

```
SELECT Id FRON Orders
WHERE CustomerId = 42
```

После этого класс `Customer` создает один экземпляр `Order` за другим, повторно вызывая процедуру `DataReader` с идентификаторами `Order` и поручая всю конкретную работу методу `GetById()` класса `Order` следующим образом:

```
// Customer.GetById()
...
Order o;

while theReader.Read()
{
  o = new Order();
  o.GetById(theReader.GetInt32(0));
  c.AddOrder(o);
}
```

Теперь для каждого заказа (`Order`) следует выбрать идентификаторы всех строк заказа (`OrderLine`) — в общем, картина должна быть ясна. Таким образом, в данном примере основное внимание было уделено объектной ориентации (по крайней мере, таковы были намерения разработчика), а не множествам, на которых, собственно, и простроены реляционные базы данных. Вследствие этого число циклических обращений к базе данных оказалось чрезмерно большим (по одному на каждую выбираемую строку плюс еще несколько), а эффективность упала ниже критического уровня.

На заметку

Следует заметить, что в конкретной ситуации может потребоваться именно такое поведение, какое описано в приведенном выше примере, чтобы, например, избежать массовой загрузки данных. Ведь нельзя с уверенностью сказать, что это *всегда* плохо — все зависит от контекста.

Если же обратиться к другой крайности — написанным и оптимизированным вручную хранимым процедурам, то код для рассмотренного выше примера будет выглядеть следующим образом:

```
CREATE PROCEDURE GetCustmerAndOrders(@customerId INT) AS
    SELECT Name, PhoneNumber,...
    FROM Customers
    WHERE Id = @customerId

    SELECT Id,...
    FROM Orders
    WHERE CustomerId = @customerId

    SELECT Id,...
    FROM OrderLines
    WHERE OrderId IN
        (SELECT Id
        FROM Orders
        WHERE CustomerId = @customerId)
```

Как упоминалось выше, хранимые процедуры зачастую оказываются эффективными во время выполнения, но весьма неэффективными при сопровождении, поскольку содержат немало кода для ручного сопровождения. Более того, хранимые процедуры, например, в Transact SQL (T-SQL — диалекте SQL, используемом для СУРБД Sybase SQL Server и Microsoft SQL Server) не вполне пригодны для обычных методов борьбы с дублированием, поэтому дублирование кода будет значительным.

На заметку

Догадываюсь, некоторые читатели скажут, что можно найти достаточно эффективное решение для приведенного выше примера и без хранимых процедур или что хранимые процедуры в любом случае — далеко не самое лучшее решение. В данном случае я просто попытался привести два крайних примера реализации с точки зрения эффективности, а именно: неудачно разработанный код, в котором используется динамический SQL, в сравнении с более удачно разработанными и написанными хранимыми процедурами.

В свете всего вышеизложенного возникает вопрос: является ли эффективность во время выполнения определяющим фактором при выборе метода проектирования?

Акцент на удобстве сопровождения

Если бы мне пришлось в настоящее время выбирать наиболее важную из возможностей программного обеспечения, ею оказалось бы удобство сопровождения. И дело совершенно не в том, что удобство сопровождения — это все, что требуется, а в том, что оно зачастую оказывается важнее, чем, например, масштабируемость.

При удобном сопровождении можно просто и эффективно по затратам реализовать и другие возможности программного обеспечения. Это, конечно, сильное упрощение, но от этого оно не теряет своего значения.

К данному вопросу можно подойти и по-другому: сравнить общие затраты на создание новой системы с общими затратами на ее сопровождение в течение всего срока ее службы. Как показывает мой опыт, затраты на сопровождение оказываются намного выше для большинства успешно внедренных систем.

И наконец, я считаю, что в настоящее время следует уделять должное внимание базе данных. К ней следует относиться дружественно, а не враждебно. Но в то же время написание вручную всего кода, связанного с базой данных, чаще всего не является "правильным" подходом. Впрочем, это имеет мало (или вообще не имеет никакого) отношения к модели, на которой мне хотелось бы сосредоточить все внимание. Если обратиться к известному и часто приводимому в литературе высказыванию Дональда Кнута (Donald Knuth): "Корень всех зол — в преждевременной оптимизации", то оптимизации лучше избегать до тех пор, пока показатели работы системы не укажут на необходимость ее выполнить.

Итак, подытоживая, можно сказать, что подобающего проектирования наряду с объектно-реляционным преобразованием (O/R Mapping) зачастую оказывается вполне достаточно, а в противном случае придется прибегнуть к ручной настройке. Объектно-реляционное преобразование в какой-то мере сродни оптимизации серверов баз данных — как правило, оно обладает достаточно развитой логикой, но в некоторых случаях его приходится выполнять вручную.

На заметку

Вопросы подобающего проектирования и объектно-реляционного преобразования более подробно рассматриваются далее в этой книге, а до тех пор определим объектно-реляционное преобразование как метод наведения мостов между объектной ориентацией и реляционными базами данных. Для этого достаточно описать взаимосвязь между вашей объектно-ориентированной моделью и реляционной базой данных, а объектно-реляционный преобразователь (O/R Mapper) сделает все остальное за вас.

Итак, один из подходов состоит в том, чтобы сначала применить такие инструменты, как объектно-реляционные преобразователи, к большей части кода доступа к базам данных, а затем написать вручную код, который приходится разрабатывать именно таким образом ради эффективности его исполнения.

А теперь рассмотрим более подробно проблему, с которой, скорее всего, придется иметь дело объектно-реляционному преобразователю, а именно: преобразование между двумя разными моделями — объектно-ориентированной и реляционной.

Потеря соответствия модели предметной области и реляционной базы данных

Как уже отмечалось, преобразование к пользовательскому интерфейсу относится к одной из характерных проблем применения моделей предметной области. Другая известная проблема — преобразование к реляционным базам данных. Эта проблема обычно называется *потерей соответствия* двух моделей: объектно-ориентированной и реляционной.

Мне бы хотелось изложить свое понимание понятия "потеря соответствия", хотя оно и может оказаться несколько популярным. Более формальное его определение дается в книге Каттела [Cattel ODM].

Во-первых, при использовании реляционной базы данных и модели предметной области появляются два типа систем. Подобная проблема вызвана отчасти тем, что оба типа систем находятся в разных адресных пространствах (если не на разных машинах), и поэтому приходится переносить данные из одного пространства в другое.

Во-вторых, не все примитивные типы совершенно одинаковы. Например, тип `string` в .NET имеет переменную длину, тогда как в Microsoft SQL Server к строковым обычно относятся типы `varchar`, `char` или `text`. Если используется тип `varchar` или `char`, приходится принимать решение относительно его максимальной длины. Если же используется тип `text`, программная модель окажется для него совершенно иной, чем для других строковых типов в SQL Server.

Другим примером может служить тип `DateTime`. В .NET этот тип очень похож на аналогичный тип в SQL Server, но между ними все же имеются отличия. В частности, представление даты в .NET осуществляется с точностью до 100 наносекунд, а в SQL Server — "только" с точностью до 3/1000 секунды. Другое примечательное отличие заключается в том, что если установить значение `DateTime.MinValue` типа `DateTime` в .NET, это приведет к исключительной ситуации при попытке сохранить данное значение в поле типа `DateTime` СУРБД SQL Server.

Еще одно отличие состоит в способности сохранять пустые значения. Так, в .NET нельзя сохранить пустое значение `null` в переменной типа `int`, а в SQL Server это вполне допустимо.

На заметку

Упомянутые выше затруднения существуют независимо от применения модели предметной области.

Большие отличия существуют в обращении с отношениями. В реляционной базе данных отношения образуются посредством дублирования значений. Основной ключ родителя (например, `Customers.Id`) дублируется как внешний в потомках (например, `Orders.CustomerId`), по существу, позволяя порожденным строкам "указывать" на соответствующие родительские строки. Таким образом, в реляционной модели все относится к данным, — даже отношения. А в объектно-ориентированной модели отношения могут быть установлены самыми разными способами (например, с помощью таких же значений, как и в реляционной модели, хотя это не типично). Чаще всего для этой цели используются встроенные идентификаторы объектов, позволяющие родителю иметь ссылки на идентификаторы порожденных им объектов (т.е. потомков). Как видите, это совершенно другая модель.

Перемещение по реляционной модели осуществляется двумя способами. Сначала можно воспользоваться первичным ключом, а затем — запросом для поиска всех потомков, внешние ключи которых имеют такое же значение, как и у первичного ключа родителя. После этого можно воспользоваться первичным ключом в новом запросе всех потомков каждого потомка и т.д. Другой и, вероятно, более типичный способ перемещения по реляционной модели состоит в применении реляционных соединений между родительским множеством и порожденным множеством. А типичный способ перемещения по объектно-ориентированной модели состоит в простом прослеживании отношений между экземплярами.

Ниже приведены два фрагмента кода, где имеется заказ под номером 42, и требуется посмотреть имя клиента по данному заказу.

C#:

```
anOrder.Customer.Name
```

SQL:

```
SELECT Name
FROM Customers
WHERE Is IN
     (SELECT CustomerId
     FROM Orders
     WHERE OrderNumber = 42)
```

На заметку

В коде SQL можно было бы с тем же успехом применить операцию `JOIN`, но, на мой взгляд, в данном случае более уместен подзапрос.

Еще одно отличие, связанное с перемещением по модели, состоит в том, что на уровне объектов перемещение оказывается односторонним. Если же требуется двухстороннее перемещение, оно организуется с помощью двух разных механизмов. А перемещение по реляционной модели всегда двухстороннее.

На заметку

Вопрос направленности перемещения по модели может быть рассмотрен и иначе, чем было пояснено выше, поскольку в реляционной базе данных имеется лишь "указатель" в одном направлении. Но я все же считаю, что этот "указатель" двухсторонний, поскольку им можно пользоваться для прохождения в обоих направлениях, а это самое главное.

Как уже отмечалось, реляционная модель основана на множествах. Каждая операция в ней связана с множествами. (Конечно, множество может состоять из одной строки, но оно по-прежнему остается множеством.) А в объектно-ориентированной модели приходится иметь дело с каждым объектом по очереди.

Более того, данные в реляционной модели являются "глобальными", тогда как в объектно-ориентированной модели приходится прилагать немало усилий для сохранения частного характера данных.

Когда же дело доходит до проектирования, то степень детализации оказывается совершенно разной. Для пояснения обратимся к примеру. Допустим, что нас интересует отслеживание одного номера домашнего телефона и одного номера рабочего телефона определенных лиц. В реляционной модели для этого, как правило, достаточно иметь одну таблицу под названием `People` (Люди; имена во множественном числе уже стали нормой в проектировании баз данных, чтобы сделать еще более очевидным тот факт, что мы имеем дело с множеством).

На заметку

Строго говоря, вместо слова "таблица" следует употреблять слово "отношение", когда речь идет о реляционной модели.

Эта таблица состоит из трех столбцов для первичного ключа и двух номеров телефонов. Столбцов может быть и больше, но до сих пор речь шла только о номерах телефонов, хотя если потребуется разделить номера телефонов на два столбца для междугородного телефонного кода и местного номера, это число придется увеличить до пяти, а если еще учесть и код страны, то до семи (рис. 1.4).

Самое главное, что даже для взаимно-однозначного соответствия все столбцы, как правило, определяются в одной таблице. В объектно-ориентированной модели для такой таблицы обычно создаются два класса: `People` и `PhoneNumber` (Номер

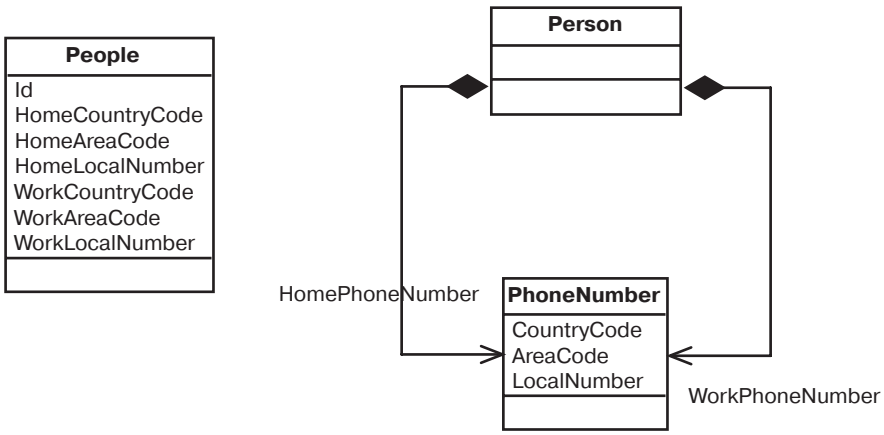


Рис. 1.4. Одна и та же модель, выраженная в реляционном и объектно-ориентированном виде

телефона). Затем создается экземпляр `Person` (Лицо) в виде комбинации двух экземпляров `PhoneNumber`. То же самое можно было бы сделать и в реляционной модели, но, как правило, это не имеет никакого смысла. Мы постараемся не использовать определения в реляционной модели многократно, в особенности потому что поведение не связано с табличными определениями. Это совершенно противоположно тому, что имеет место в объектно-ориентированной модели. Иными словами, в реляционной модели мы не увеличим удовлетворенную нормальную форму, если переместим номера телефонов (`PhoneNumber`) в отдельную таблицу. При этом можно увеличить лишь издержки. Таким образом, реляционная модель предназначена для работы с табличными, примитивными данными, и это имеет свои достоинства и недостатки. А объектно-ориентированная модель также предназначена для работы с данными, но со сложными.

На заметку

Что касается повторного использования определений, то у реляционной модели имеется очень четкая концепция, называемая *доменом*. К сожалению, эта концепция не достаточно полно поддерживается в современных программных продуктах.

С другой стороны, во многих программных продуктах поддерживаются сложные типы данных, но такая поддержка пока еще находится на стадии становления.

Выше был рассмотрен лишь один пример степени детализации, где реляционная модель оказалась на уровне более крупных структурных единиц, чем объектно-ориентированная модель. К этому вопросу можно подойти и совершенно иначе.

Так, у заказа (`Order`) в реляционной модели может быть множество строк заказа (`OrderLines`), но последние оказываются как бы сами по себе. Каждая строка заказа (`OrderLine`) — это просто строка таблицы, а каждый заказ (`Order`) — еще несколько таких строк. Между ними существует отношение, но строки таблицы являются структурными единицами. В объектно-ориентированной модели было бы правильно представить заказ (`Order`) в качестве структурной единицы и в то же время составить его из строк заказа (`OrderLines`). На этот раз реляционная модель оказалась на уровне менее крупных структурных единиц, чем объектно-ориентированная модель.

На заметку

Выше не имелось в виду, что у модели предметной области не может быть класса `OrderLine`. Напротив, он должен быть. Просто для работы в качестве структурной единицы служит `Order`, а `OrderLines` — это части `Order`.

И последнее, но не менее важное замечание: в реляционной модели не поддерживается наследование (опять же, это, по крайней мере, не характерно для большинства распространенных программных продуктов). В то же время наследование положено в основу объектно-ориентированной модели. Конечно, наследование можно сымитировать в реляционной модели, но оно окажется не более чем имитацией. Выбор среди разных видов имитации — это своего рода компромисс, и в любом случае он приводит к издержкам памяти, быстродействия или же к нарушению реляционного принципа.

На заметку

Глубокую и естественную интеграцию XML в базы данных следует, по-видимому, отнести к новейшим попыткам как-то преуменьшить проблему потери соответствия. Но ведь XML — это фактически третья, иерархическая модель, которой присуща аналогичная потеря соответствия с объектно-ориентированной и реляционной моделями.

В своих приложениях я, как правило, использовал реляционные базы данных, и поэтому потеря соответствия всегда создавала немалые трудности.

Для решения данной проблемы может быть использован шаблон Data Mapper [Fowler PoEAA]. В этом шаблоне описывается взаимосвязь между моделью предметной области и базой данных, после чего процедура перестановки выполняется автоматически. К сожалению, шаблон Data Mapper трудно применять, особенно на платформе .NET, где объектно-реляционный преобразователь реализован в конкретных продуктах лишь совсем недавно. На платформе Java имеется ряд зрелых продуктов

и даже стандартизированная спецификация, называемая JDO [Jordan/Russel JDO]. В этом смысле обе платформы совершенно отличаются друг от друга, какими бы сходными они ни были в остальном.

Итак, оставим тему моделей предметной области и баз данных и обратимся к вопросам распределения в конце этого раздела, посвященного стилям архитектуры приложений.

Распределение требует аккуратного обращения

Возможно, вам знаком такой синдром: "Появился новый инструмент, надо бы им воспользоваться." Я лично подвержен ему время от времени. И хотя с ним могут быть связаны всякие осложнения, я все же привык видеть в нем не только отрицательные, но и положительные стороны. К концу концов он может служить признаком здорового любопытства, нормального прогрессивного мышления и неуемного стремления к совершенству.

А теперь я хотел бы поведать несколько ужасных историй из своего прошлого, за что я заранее прошу прощения у читателя. Около десяти лет назад был выпущен продукт Microsoft Transaction Server (MTS), благодаря которому всем разработчикам приложений на основе архитектуры COM стало намного легче создавать распределенные системы. Что же в итоге произошло? Было создано немало распределенных систем, и я тоже приложил к этому руку.

На заметку

Здесь и далее будет употребляться название MTS, но в равной степени можно было бы выбрать и следующие названия: COM+, COM+ Component Services или Enterprise Services.

Свойство распределения оказалось весьма подходящим для систем. У переноса обработки со стороны клиента или базы данных на сторону сервера приложений имелись свои преимущества, в том числе совместное использование ресурсов (например, организация связного пула на стороне сервера, с тем чтобы сотни клиентов могли быть обслужены посредством лишь нескольких соединений с базой данных). В таких преимуществах очень нуждались некоторые системы, а обеспечить их стало намного легче.

Злоупотребление никогда не бывает во благо

К сожалению, эти преимущества были настолько привлекательны, что ими в конечном итоге стали злоупотреблять. Например, работу на стороне сервера можно было распределить среди нескольких разных серверов приложений. Но это совсем не означало клонирование серверов приложений, с тем чтобы все клиенты могли воспользоваться любым из них и получить любые услуги, которые им требовались. Вместо

этого клиент обращался только к одному серверу приложений, этот сервер вызывал другой сервер, а тот — третий и т.д. Таким образом, незначительные преимущества были сведены на нет существенными недостатками, обусловленными увеличением времени ожидания и снижением доступности серверов.

Другая ошибка заключалась в том, что простые приложения, рассчитанные только на двух одновременно работающих пользователей, были написаны и предназначались для выполнения в MTS. Это яркий пример того, как из мухи делают слона, поскольку возлагались большие надежды на то, что приложение может найти широкое распространение в не столь отдаленном будущем и будет одновременно применяться тысячами пользователей. Главный недостаток такого подхода состоял не в усложнении операционной среды, а в том, что проектирование приложения имело очень серьезные последствия, если требовалось полностью удовлетворить требованиям MTS. Характерным тому примером послужила борьба с избыточным обменом информацией между некоторыми (а еще хуже, между всеми) уровнями. Эти последствия привели к значительному усложнению проектирования. Злоупотребление, действительно, никогда не бывает во благо.

Возможно, вы обратили внимание на то, что я избегаю упоминания о типичных скрытых препятствиях, которые ожидают начинающих разработчиков распределенных систем, это касается также избыточного обмена информацией. Даже если, так или иначе, удастся обойти все эти препятствия и избежать связанных с ними ошибок, все равно останутся другие, менее очевидные осложнения.

Все эти осложнения и вынудили Мартина Фаулера сформулировать первый закон распределенного объектного проектирования: “Не распределять объекты!” [Fowler PoEAA]. Это означает, что если нет абсолютной необходимости в распределении, то делать его не следует. В противном случае это может обойтись увеличением сложности приложений свыше всякой нормы. Таков закон природы.

Распределение может быть полезным

У распределения все же имеются свои положительные свойства, которые могут оказаться полезными как в настоящее время, так и в перспективе, в частности:

- **Отказоустойчивость.**

  Если все приложения выполняются на одной машине, то при ее отказе возникают серьезные осложнения. Для одних приложений это несущественно, а для других — просто неприемлемо.

- **Безопасность.**

  По поводу повышения безопасности вследствие распределения приложений по нескольким машинам ведется горячая полемика. Так или иначе, забота о безопасности является частой причиной такого распределения.

- **Масштабируемость.**

  Для некоторых приложений нагрузка на одну машину оказывается настолько большой, что она не может с ней справиться. Кроме того, по финансовым

соображениям приобретение самой дорогой машины может оказаться в какой-то момент нецелесообразным. Вместо этого, при повышении нагрузки лучше добавить к старым машинам новые, а не выбрасывать старые.

Если действительно требуется распределенная система, в таком случае чрезвычайно важно неспешно и тщательно продумать весь проект, поскольку впоследствии трудности проектирования могут вызвать не меньшие трудности, связанные, например, с пропускной способностью системы. Когда речь идет о распределении, то очень важно принимать во внимание и такой аспект, как обмен сообщениями.

Акцент на обмене сообщениями

В этой книге речь идет, главным образом, о разработке *одного* приложения или *одной* службы, а не об организации нескольких служб. Тем не менее, при проектировании целесообразно обратить внимание и на обмен сообщениями (по крайней мере, начать думать о нем), поскольку оно, на мой взгляд, будет приобретать в будущем все большее значение.

Сообщения как основной элемент программной модели

Раньше я считал, что было бы неплохо абстрагироваться от сети или же скрыть ее в распределенных системах. Благодаря этому разработчику клиентского приложения не нужно было знать, должен ли вызов метода передаваться по сети или же это просто вызов локальной функции. Мне нравился такой подход, поскольку я стремился упростить работу разработчика клиентского приложения, чтобы он сосредоточил основное внимание на вопросах создания удобного пользовательского интерфейса и не отвлекался на вопросы, связанные с передачей данных по сети.

Обратимся к простому примеру. Допустим, что имеется класс `Customer`, как показано на рис. 1.5.

Рис. 1.5. Класс `Customer`

Экземпляры класса `Customer`, как правило, существуют (как можно надеяться) отдельно в том же самом адресном пространстве, что и программа клиента, поэтому когда клиент запрашивает имя `Customer`, это лишь локальный вызов.

Но вам ничто не мешает сделать `Customer` шаблоном Proxy (Заменитель) [GoF Design Patterns], который, в свою очередь, пересылает вызовы экземпляру `Customer`, находящемуся на сервере приложений. (Конечно, это несколько упрощенное представление, но именно так все приблизительно и происходит незаметно для вас, если вы, например, настроите класс `Customer` в MTS на сервере приложений.)

В своей книге Шиперски [Szyperski Component Software] отмечает, что прозрачность местоположения — это и достоинство, и недостаток. Достоинством она является, потому что все виды обмена информацией (внутри процесса, между процессами и между машинами) приводятся к одной абстракции — вызову процедуры, а недостатком — потому что она скрывает существенные отличия в затратах на разные виды вызовов. Как правило, эти отличия составляют несколько порядков величины времени выполнения.

На мой взгляд, современная тенденция сделать сообщения основным элементом программной модели, чтобы стали очевидными сообщения, требующие больших затрат, является противоположной прозрачности местоположения. Я поддерживаю развитие такой тенденции, просто потому что разработчику клиентского приложения должно быть ясно, что предстоит сетевой вызов, хотя это совсем не означает, что его должно быть трудно сделать. Например, можно организовать фабрики сообщений, но все равно при данном подходе намного проще понять, что может отнять время, а что не может.

Возникает резонный вопрос: каким образом обмен сообщениями оказывает влияние на модель предметной области? Потребность в модели предметной области не отпадает, хотя некоторые изменения, скорее всего, будут. Первым примером, который приходит на ум, являются вставки вместо обновлений, как в системном журнале — даже для модификаций. Для этой цели лучше всего подходит асинхронная передача данных.

Организация последовательности выполнения

Одно из самых важных преимуществ обмена сообщениями состоит в том, что оно делает выполнение намного более гибким. В прошлом я часто пользовался пакетными заданиями для длительного выполнения, если это не нужно было делать в реальном масштабе времени. Этот старый и, на мой взгляд, не часто используемый прием существенно увеличивает время реакции для той части задания, которая выполняется в реальном масштабе времени, и поэтому длительное выполнение зачастую происходит в период малой загрузки, как правило, в ночное время.

До настоящего времени я разработал слишком много приложений, чтобы все они были синхронными. Если часть функций требует длительного выполнения и не обязательно в реальном масштабе времени, в таком случае эти функции приходится переводить в режим пакетной обработки. Таким образом, выполнение части функций переносится из режима реального времени в режим запроса.

Более эффективное решение данной задачи состоит в планировании асинхронных сообщений с самого начала. В этом случае функции могут выполняться в реальном масштабе времени, если это уместно, или же поставлены в очередь сообщений для выполнения как можно раньше или же через заданные интервалы времени. (Пакетная обработка должна быть предусмотрена как вариант с самого начала.)

Решение, основанное на асинхронных сообщениях, может потребовать совсем иного мышления при построении пользовательского интерфейса, но раз уж прихо-

дится решать разные задачи проектирования, то, на мой взгляд, асинхронность может оказаться вполне приемлемым вариантом, к которому стоит обращаться чаще, чем это считалось в прошлом.

И в заключение хотелось бы поделиться следующими соображениями: сначала имеет смысл сосредоточить основное внимание на самой базовой модели, независимо от условий выполнения, а затем применить ее в самых разных ситуациях по мере надобности.

На этом оставим архитектурные ценности и перейдем к оценке самого процесса.

Оценка составляющих процесса

Я не большой знаток процесса, но мне хотелось бы поделиться некоторыми соображениями по этому поводу. Однако прежде необходимо уточнить, что я подразумеваю под термином "процесс".

Классическим примером процесса служит так называемый водопад. Каждая стадия такого процесса наступает в строгой линейной последовательности без каких-либо возвратов назад. Вкратце это выглядит следующим образом: сначала составляются технические условия проекта, а затем по этим условиям строится, испытывается и развертывается система.

За последние годы был внедрен ряд других процессов, имеющих свои достоинства и недостатки. Совсем недавно один такой процесс, называемый экстремальным программированием (ЭП — Extreme Programmning (XP)) [Beck XP] и отнесенный к категории процессов гибкой разработки программного обеспечения [Cockburn Agile], может быть описан как нечто, совершенно противоположное водопаду. Основной принцип ЭП состоит в следующем: невозможно знать и предусмотреть сразу все, чтобы составить хорошие, подробные технические условия проекта. Знания накапливаются по ходу работы над проектом не потому что проходит время, а потому что постепенно строятся отдельные части системы, что очень удобно для углубленного понимания задач ее проектирования.

На заметку

Подробнее об ЭП см., например, [Beck XP]. А в этой книге рассматриваются такие методы, как РПТ и рефакторинг кода, которые коренятся в ЭП.

Следует также иметь в виду, что ЭП не всегда начинается с нуля. На форумах по ЭП можно обнаружить большой интерес к ЭП и устаревшим системам. См. также книгу *Object-Oriented Reengineering Patterns* [Demeyer/Ducasse/Nierstrasz].

Для конкретной ситуации я всегда стараюсь отыскать удачное сочетание мелких составляющих процесса. В настоящее время я отдаю предпочтение нескольким

разным составляющим: проектированию на основе предметной области, разработке посредством тестирования и рефакторинга кода, но прежде чем перейти к ним, остановимся сначала на предварительном проектировании архитектуры (Up-Front Architecture Design).

ЭП и акцент на пользователе

Для ЭП характерна еще одна новая тенденция: разработка, ориентированная на пользователя. Это означает, что пользователи должны быть привлечены к работе над проектом вплоть до его завершения.

Для Швеции этот принцип далеко не нов — разработка, ориентированная на пользователя, начала практиковаться задолго до ЭП.

Я, конечно, не хочу этим подчеркнуть, что Швеция чем-то лучше других стран, но у нас уже довольно давно применяются и водопадные процессы.

Предварительное проектирование архитектуры

Несмотря на то что мне по душе многие принципы гибкой разработки программного обеспечения, в том числе и те, что не допускают принятие предварительных и поспешных решений по вопросам, которые могут быть неизвестны на данной стадии проекта, я все же считаю, что работу над новым проектом нельзя начинать только с чистого листа. Ведь чаще всего в нашем распоряжении с самого начала имеется достаточно информации о производственной среде (и чем больше ее, тем лучше), ожидаемой нагрузке, сложности системы и т.д. Такую информацию нужно, по крайней мере, учитывать, а лучше использовать при первоначальном опробовании концепции предварительной архитектуры.

Заимствование идей из ранее построенных удачных архитектур

Мы не можем себе позволить начинать с нуля каждое новое приложение, и это особенно касается его архитектуры. Я обычно анализирую исходную архитектуру, которой отдаю предпочтение в настоящий момент, чтобы понять, насколько она соответствует конкретной ситуации и требованиям вновь создаваемого приложения. Я также оцениваю несколько приложений, разработанных в последнее время, чтобы выяснить, каким образом их можно усовершенствовать с точки зрения архитектуры — опять же, с учетом требований нового приложения. Поэтому следует всегда стремиться оценивать и пытаться улучшить что-то. Безусловно, это очень ценное качество.

Если допустить на какое-то мгновение, что вам пришлась по душе идея шаблона модели предметной области [Fowler PoEAA], то принимать архитектурные решения вам будет немного легче, поскольку бо́льшая часть вашего внимания будет сосредоточена именно на этой модели. Решение об использовании шаблона модели предметной области, по существу, является предварительным проектным решением архитек-

туры, и оно очень важно, поскольку оказывает большое влияние на последующую работу над проектом.

Следует также отметить, что предварительное решение не является окончательным. Это же относится и к решению об использовании шаблона модели предметной области.

На заметку

Некоторые ведущие специалисты в данной отрасли рекомендуют реализовывать шаблон Transaction Script [Fowler PoEAA] вместе с шаблоном Recordset [Fowler PoEAA] на как можно более длительный период и переходить к другому решению только в том случае, если в нем возникнет потребность. Я с этим не согласен. Конечно, может случиться самое худшее, но я не сторонник такого перехода на поздней стадии работы над проектом, поскольку это влечет за собой очень серьезные последствия.

Согласованность

Веским аргументом в пользу раннего архитектурного решения служит тот факт, что группа разработчиков будет работать в том же направлении. Поэтому ради согласованности действий необходимо заранее задать это направление.

Это же касается и отделов информационного обслуживания, занимающихся разработкой многих приложений для своей организации. Было бы очень полезно, если бы архитектуры таких приложений были сходными, поскольку это делает намного более простым и эффективным переход людей от одного проекта к другому.

Фабрики разработки программ

Изложенное выше приводит нас к концепции фабрик разработки программ (Software Factories) [Greenfield/Short SF], которая появилась под влиянием понятия архитектуры семейства программных продуктов (Product Line Architechture) [Bosch Product Line]. Эта концепция заключается в следующем: в фирме-разработчике программного обеспечения организуются две линии: одна — для разработки архитектур и базовых структур, предназначенных для семейства приложений, а другая — для создания приложений на основе разработок, выполненных на первой линии, благодаря чему сокращаются расходы на базовые структуры в нескольких проектах.

На заметку

Придумать базовую структуру не так-то просто, поэтому намного лучше заготовлять такие структуры про запас [Fowler HarvestedFramework].

Тем не менее наладить эффективную работу фабрик разработки программ можно только в крупных организациях. Мой коллега, имеющий опыт применения архитектур семейств программных продуктов, сказал, что для этой цели организации придется нанять тысячи, а не сотни или десятки разработчиков из-за крупных капиталовложений и больших накладных расходов. Кроме того, даже если организации и начали применять архитектуры семейств программных продуктов, это совсем не означает, что эти архитектуры будут применяться все время. Не следует также сбрасывать со счетов как финансовые, так и бюрократические издержки, которые с этим связаны. Достаточно представить себе мелкую базовую структуру, применяемую в нескольких приложениях, а затем намного более крупную базовую структуру, чтобы понять, во что обходится организация фабрик разработки программ.

Тем не менее идея фабрик разработки программ достаточно интересна.

В основу фабрик разработки программ положен *язык программирования предметной области* (Domain Specific Language — DSL) [Fowler LW]. Если объяснить популярно, то язык DSL предназначен для решения подзадач с помощью языков, специально выбранных для решения конкретной задачи. Эти языки могут быть представлены как в графическом, так и в текстовом виде. Кроме того, они могут быть базовыми, как, например, XML, UML и C#, специальными, подобно редактору WinForms в VS.NET, или же мелкими языками, определяемыми разработчиками самостоятельно для решения конкретных задач.

Другой подход, во многом сходный с DSL, состоит в разработке архитектуры, основанной на модели.

Архитектура, основанная на модели

Архитектура, основанная на модели (Model-Driven Architecture — MDA) [OMG MDA], сродни "программированию посредством составления блок-схем в UML". В связи с этим понятием возникла идея создания *платформно-независимой модели* (Platform Independent Model — PIM), которую затем можно преобразовать в *платформно-зависимую модель* (Platform Specific Model — PSM) и далее — в исполняемую форму.

Представить себе такое очень трудно. Это все равно, что написать новую программу полностью на таком языке третьего уровня, как C#. Для повышения уровня абстракции требуется время.

Многие усматривают один существенный недостаток архитектуры MDA в ее тесной связи с UML. Отчасти этот недостаток объясняется потерей точности при переходе от UML к коду, а отчасти тем, что UML является базовым языком со всеми присущими ему достоинствами и недостатками.

Оба упомянутых выше подхода (DSL и MDA) можно свести к одному понятию разработки на основе модели (Model-Driven Development). Можно с уверенностью сказать, что все мы двигаемся именно в этом направлении, поэтому у разработки на основе модели имеются неплохие перспективы. Даже в настоящее время с помощью доступных инструментов можно делать многое из того, что подразумевает разработка на основе модели.

Кроме того, оба подхода (DSL и MDA) вполне согласуются, на мой взгляд, с принципами проектирования на основе предметной области, особенно это касается сосредоточения всего внимания на модели.

Проектирование на основе предметной области

В предыдущем разделе, посвященном стилям архитектуры, уже шла речь об акценте на модели и *проектировании на основе предметной области* (ППО — Domain-Driven Design (DDD)), но мне бы хотелось рассмотреть ППО и с точки зрения процесса. Прикладное ППО [Evans DDD] в качестве процесса подразумевает сосредоточение основного внимания и энергии на построении модели и как можно более точной ее реализации в программном обеспечении.

С этой целью создается максимально простая модель, в которой воплощаются наиболее важные свойства предметной области приложения. Во время разработки процесс может быть описан как совместное добывание знаний разработчиками и специалистами в предметной области. Добытые таким образом знания наполняют модель.

Кратчайший путь — воспользоваться старыми знаниями

Разумеется, не все знания добываются с нуля. Немало знаний содержится в книгах, причем не только в специальной литературе из конкретной предметной области. Полезную информацию можно почерпнуть и в литературе по программному обеспечению. Я имею в виду не только книги *Data Model Patterns* [Hay Data Model Patterns] и *Analysis Patterns* [Fowler Analysis Patterns], но и книгу *Enterprise Patterns and MDA* [Arlow/Neustadt Archetype Patterns], посвященную шаблонам прототипов.

Рефакторинг кода для углубления знаний

Очень большое значение имеет постоянная оценка модели, которая вполне пригодна и для ППО. Нужно постоянно и конструктивно оценивать модель, и если удастся найти пути ее упрощения, то не следует бояться рефакторинга кода.

Это же касается и обнаружения чего-то нового. Благодаря нескольким простым процедурам рефакторинга кода новое свойство становится не только возможным, но и достижимым и вполне вписывающимся в модель.

Рефакторинг кода сам по себе может углубить знания, как в моменты озарения и новых открытий.

Мы еще вернемся к рефакторингу кода, а пока рассмотрим тесно связанные с ней вопросы разработки посредством тестирования.

Разработка посредством тестирования

Мне часто приходилось слышать жалобы разработчиков на то, что они не могут выполнять автоматическое блочное тестирование, поскольку у них нет для этого подходящих инструментов. Конечно, инструменты важны, но еще важнее склад ума.

Несколько лет назад я написал собственный инструмент (рис. 1.6) и применял его для регистрации и выполнения тестов хранимых процедур, компонентов COM и классов .NET. Благодаря этому инструменту мне удалось избежать форм с многочисленными кнопками, которые приходится нажимать в определенном порядке для выполнения тестов.

Рис. 1.6. Копия экрана моего старого инструмента под названием JnskTest

В дальнейшем, когда появился инструмент NUnit [NUnit], производная от других версий xUnit (рис. 1.7), я начал пользоваться именно им. Ведь мой инструмент не отражал, что именно проверяют действующие тесты. Поэтому соответствующую информацию приходилось регистрировать явно.

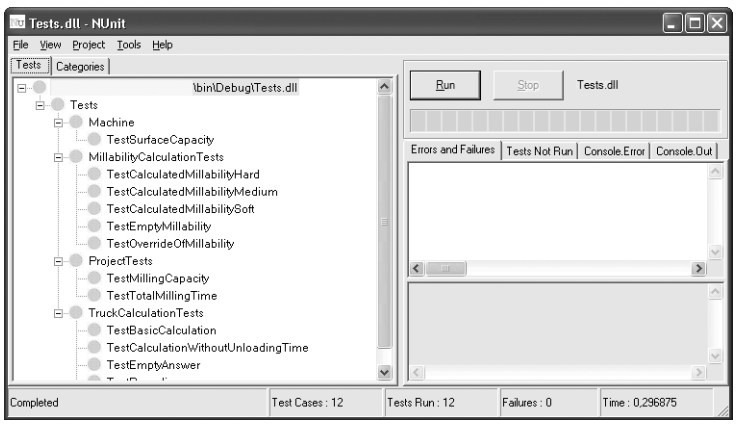


Рис. 1.7. Версия инструмента NUnit с графическим пользовательским интерфейсом

> ***На заметку***
>
> Теперь я пользуюсь другим инструментом под названием Testdriven.Net, но, как отмечалось выше, тип используемого инструмента не столь важен.

Автоматическое блочное тестирование можно выполнять независимо от используемого процесса. А для достижения еще большего положительного результата я настоятельно рекомендую выяснить, насколько вам подходит разработка посредством тестирования (РПТ — Test-Driven Development (TDD)).

Следующий уровень

РПТ заключается в написании тестов перед написанием настоящего кода. Благодаря этому тесты продвигают процесс не только проектирования, но и программирования.

На первый взгляд, РПТ напоминает нечто рутинное и малоприятное, и разработчики не ждут от нее ничего, кроме лишних хлопот. Но если бы они знали, как они неправы! Мой опыт показывает совершенно другое — РПТ весьма занимательна, что для меня стало полным откровением. Объясняется это, по-видимому, тем, что вы получаете мгновенную реакцию на вносимые вами изменения, а для профессионала нет большего удовольствия, чем создание качественных приложений.

Иными словами, РПТ связана не столько с тестированием, сколько с программированием и проектированием, т.е. с написанием более простого и надежного кода! Нельзя сбрасывать со счетов и "побочные эффекты" создаваемых автоматических блочных тестов!

Основания для РПТ

Первая причина, по которой я прибегнул к РПТ, состояла в том, что мне требовалось повысить качество своих проектов. Именно этого важного и очевидного результата позволяет добиться РПТ. Ведь мы не хотим создавать приложения, которые аварийно завершаются с первого же раза, стоит заказчикам воспользоваться ими, или приложения, которые выходят из строя, как только мы пытаемся их усовершенствовать. Это уже не приемлемо.

РПТ не помогает автоматически избавиться от программных ошибок в выпускаемых продуктах, но позволяет повысить их качество.

> **На заметку**
>
> Автоматические тесты сами по себе не служат основанием для РПТ. Они дают замечательные, но побочные эффекты. Если во главу угла поставлено качество, для достижения этой цели имеются другие формальные методы, но во многих случаях они считаются слишком "дорогими". Опять же, все зависит от контекста.

Для того чтобы воочию убедиться в повышении качества, достаточно написать тесты *после* настоящего кода. Это означает, что применять РПТ (т.е. писать тесты *до* настоящего кода) не обязательно. Нужно лишь соблюдать строгую дисциплину. С другой стороны, существует большой риск отказаться от написания тестов, когда сроки поджимают, что обычно случается на последней стадии работы над проектом. Именно благодаря РПТ осуществляются тесты.

Вторым ожидаемым результатом применения РПТ является реальное упрощение проекта. Как говорится, все прекрасное просто. И это очень важно, поскольку сложность порождает ошибки.

Вместо создания множества усовершенствованных планов, заранее охватывающих каждую деталь проекта, применяя РПТ, вы сосредоточиваете внимание на основных требованиях заказчика и воплощаете в проекте то, что требуется заказчику. При этом вы рассматриваете проект в большей степени с точки зрения заказчика, а не с технической точки зрения.

РПТ не подразумевает отказ от проектирования. Напротив, вы постоянно занимаетесь проектированием, применяя РПТ.

В прошлом я весьма преуспел в чрезмерном усложнении простых вещей. А теперь РПТ помогает мне сосредоточиться на том, что действительно требуется в *настоящий момент*. Для достижения такого результата (т.е. упрощения проекта) и требуется РПТ. Написать тесты задним числом для этого уже недостаточно.

Еще одним результатом применения РПТ является высокая производительность труда на протяжении *всей работы* над проектом. На первый взгляд, это кажется нелогичным. Когда начинается новый проект, то написание настоящего кода с самого начала кажется вполне продуктивным занятием. Поначалу это действительно так, но, как правило, производительность труда заметно падает к концу проекта. При этом появляются программные ошибки, заказчик решает внести ряд существенных изменений, что нарушает все планы, а исполнитель уже не все до конца понимает в проекте — в общем, картина ясна.

Тесты вынуждают выполнять требования к проекту и делать это сразу. Благодаря этому всякое недопонимание поставленных задач обнаруживается довольно рано. Столь же рано обнаруживаются и противоречивые или недостаточные требования к проекту.

Между прочим, спрашивать у заказчика, следует ли применять РПТ, не стоит — по крайней мере, в тот момент, когда вы просите дополнительной оплаты своих услуг, времени или чего-то другого. Если при анализе проекта от начала до конца выясняется, что РПТ влечет за собой дополнительные затраты, это, на мой взгляд, не должно вас останавливать. В конечном итоге заказчик будет доволен, если получит продукт того качества, на которое он рассчитывал.

На заметку

Оставим на время тему РПТ и обратимся к автоматическим тестам.

Мой коллега весьма скептически относился к необходимости проведения автоматических блочных тестов, создаваемых до или после настоящего кода. Он говорил мне, что за свою 20-летнюю карьеру он так и не извлек никакой пользы из автоматических блочных тестов. Но вскоре после этого разговора он, я полагаю, изменил свое отношение к ним. Мы работали вместе над проектом, где ему пришлось написать компонент COM на C++, а я написал тесты как таковые и

как спецификации. Когда все было готово, заказчик изменил одно из требований к проекту. Мой коллега внес небольшие изменения, но тесты выявили программную ошибку, проявившуюся только четыре раза на 1000 исполнений. Эта ошибка была обнаружена спустя лишь несколько секунд после начала тестирования по сравнению с многими часами, которые потребовались бы на ее обнаружение вручную. В последнем случае она могла быть вообще не обнаружена, но непременно проявилась бы во время эксплуатации в условиях производства.

Порядок выполнения РПТ

А теперь, когда я попытался убедить вас освоить РПТ, рассмотрим подробнее, как этот процесс происходит. (Мы еще вернемся к данному вопросу в главе 3, когда будем исследовать его более углубленно на реальном демонстрационном примере.) Итак, если иметь ясное представление о требованиях к проекту, то РПТ выполняется в следующем порядке.

Прежде всего, следует приступить к написанию теста. Необходимо сделать так, чтобы неудачное завершение теста имело какой-то смысл. Это дает уверенность в том, что тест действительно проверяет то, что нужно. Об этом простом правиле не следует забывать, хотя я лично нарушал его неоднократно и тем самым усложнял себе жизнь.

Второй этап состоит в написании максимально простого кода для прохождения теста.

На третьем этапе осуществляется рефакторинг кода в связи с обнаружением недоброкачественного кода (например, дублированного), а затем все начинается сначала и вводится еще один тест.

Если бы тест проводился с помощью инструмента NUnit, то результатом его выполнения на первом этапе стала бы светло-красная полоса (тест не прошел — положительный результат), а на втором этапе — светло-зеленая полоса (тест прошел — отрицательный результат).

Рефакторинг кода был упомянут на третьем этапе выполнения всего процесса РПТ, и поэтому теперь самое время дать хотя бы краткое пояснение этого понятия.

Рефакторинг кода

О том, что нужно постоянно учиться, мы знаем еще со школьной скамьи. И это как нельзя справедливо для рефакторинга кода [Fowler R], например, для того, чтобы улучшить модель.

Рефакторинг кода означает поэтапное внесение мелких, заранее известных изменений с целью улучшить конструкцию кода, т.е. сделать более удобным его сопровождение, не изменяя наблюдаемое поведение. Иными словами, это *качественное*, а не *принципиальное* изменение.

По существу, рефакторинг кода позволяет сделать качественным недоброкачественный код. Это означает, что стремиться с самого начала к идеальной конструкции кода совсем не обязательно, хотя бы потому, что добиться этого практически невозможно.

Основания для рефакторинга кода

Выявление недоброкачественного кода особых трудностей не вызывает — намного труднее выбрать удобный момент для его исправления. На мой взгляд, эту проблему следует решать, как только она возникает. Код реорганизуется потому, что без непрерывного сопровождения он становится вырожденным и раздробленным.

> **На заметку**
>
> Марк Берхоп сказал по этому поводу следующее: "Настоящий друг всегда придерживается свода правил разработки программного обеспечения. Одно из них гласит: код, оставленный неприкосновенным, служит источником программных ошибок".

Обратимся к аналогии ремонта дома. Если не заниматься им периодически, то со временем окна и двери не будут как следует закрываться, крыша начнет течь, краска на столярных изделиях облупится, и в конечном итоге дом станет непригодным для жилья и вообще развалится. Это непреложное правило, и вряд ли кто-то захочет доводить свой дом до такого состояния.

Разумеется, у программного обеспечения имеются свои отличия, поскольку оно не состоит из органических материалов и не подвержено воздействию стихии и ветра. Тем не менее мы интуитивно ощущаем, что со временем в программном обеспечении происходят какие-то изменения, если в процессе исправления ошибок и расширения программы ее код не реорганизуется.

Рефакторинг кода на практике

Код может быть реорганизован на всех стадиях жизненного цикла приложения, в том числе и в процессе разработки первой версии приложения. Следует, однако, признать, что вместо рефакторинга кода мы зачастую предпочитаем традиционный, обременительный процесс проектирования, который теперь называется *крупным проектированием наперед* (КПН — Big Design Up-Front (BDUF)). При этом мы тратим немало времени на подробное первоначальное проектирование и составление множества детальных UML-схем, надеясь на то, что разработка будет в итоге происходить гладко и быстро. Но даже если это происходит, по-прежнему существует риск, что код окажется недоброкачественным.

Вместо этого нужно просто признать тот факт, что нельзя заранее все предусмотреть и сделать правильно с первого раза. В этом случае уместен несколько иной под-

ход: перенести некоторые виды работ со стадии подробного первоначального проектирования на стадию разработки (тогда вся разработка, безусловно, превращается в проектирование) и быть готовым к непрерывному рефакторингу кода по мере изучения и выяснения особенностей проекта. Последнее обычно присуще стадии разработки и, на мой взгляд, такой подход дает в итоге более качественный код.

Таким образом, вместо того чтобы слишком много предполагать, следует больше изучать и проверять!

На заметку

Возможно, я был слишком категоричен в отношении КПН, но мне хотелось увязать этот вопрос с выяснением причин появления недоброкачественного кода. Ведь сделав однажды много предположений, можно в конечном итоге потерять немало времени из-за того, что они окажутся не более чем предположениями.

Мой коллега Джимми Экбёк (Jimmy Ekbäck) прокомментировал это следующим образом: "КПН может привести к еще более серьезным последствиям, чем напрасная трата времени из-за неверных предположений. Более того, КПН может просто накликать беду".

Рефакторинг кода плюс РПТ означает верный путь

Для надежного рефакторинга кода *необходимо* выполнять обширные тесты. В противном случае будут внесены ошибки или же будет решено не делать никаких изменений просто ради удобства сопровождения, поскольку слишком велик риск внесения программных ошибок. Но как только изменения будут прекращены ради удобства сопровождения, код станет медленно вырождаться.

На заметку

РПТ и рефакторинг кода более подробно рассматриваются на конкретных примерах в главе 3.

Кроме того, рефакторинг кода и РПТ полезны для исправления программных ошибок. Сначала такие ошибки выявляются в результате неудачного завершения теста, а затем они исправляются и тест завершается удачно, после чего код реорганизуется.

Отдельные составляющие или их сочетание

Я уверен, что многие из вас стоят на распутье: выбрать, например, РПТ или проектирование наперед?

На мой взгляд, проектирование наперед можно успешно сочетать с РПТ. В частности, можно предварительно разработать архитектуру, перейти к проектированию на основе предметной области и проработать проект для каждого вида поведения с помощью РПТ, включая и рефакторинг кода, а затем вернуться к архитектуре и изменить ее в соответствии с тем, что удалось узнать и выяснить по ходу дела, после чего вновь перейти к проектированию на основе предметной области и далее продолжать в том же духе.

На заметку

Должен признать, что я часто возвращаюсь к своей старой привычке выполнять подробное проектирование наперед. Но ведь анализ решаемой задачи под разными углами зрения (сверху вниз, снизу вверх, изнутри и снаружи) нередко оказывается наиболее эффективным приемом.

Хорошо известно, что с крупным проектированием наперед (КПН) связаны крупные осложнения. В то же время некоторые вопросы зачастую проясняются лишь со временем. Поэтому и нужно уметь сохранять равновесие.

И последнее замечание относительно ППО и РПТ: модели предметной области очень хорошо подходят для РПТ. Конечно, РПТ можно применять и к проектам, более ориентированным на базы данных, но мне лично не удавалось сделать это изящно и продуктивно, работая с моделями предметной области.

На заметку

Когда я обсуждал вопросы ППО и РПТ с Эриком Эвансом, он высказал следующее, на мой взгляд, очень точное соображение: “Я лично экспериментирую с моделью, составляя в то же время тесты. Написание тестов позволяет мне выяснить, к какому коду клиентского приложения может привести распределение различных обязанностей, а также уточнить имена методов и прочее, чтобы передать исходные намерения и добиться правильной последовательности операций”.

На чем бы ни был сделан акцент: на РПТ или на КПН, в любом случае имеется немало полезных методов. В завершение этой главы перейдем к вопросам эксплуатации, но прежде коснемся темы так называемой непрерывной интеграции.

Непрерывная интеграция

Многие из вас, наверное, признают, каким непреодолимым препятствием может оказаться ручная интеграция, если она выполняется, например, ежемесячно — даже в том случае, когда над проектом работают только два разработчика. Ситуация усугуб-

ляется еще больше, когда число разработчиков увеличивается. Этот процесс, помимо всего прочего, требует времени, не гарантирует защиту от ошибок и в итоге выполняется не по плану. И как всегда, для завершения проекта не хватает одного дня. Мой коллега Клаудио Перроне — именно тот человек, который способен доходчиво пояснить суть непрерывной интеграции.

Проблема интеграции

Клаудио Перроне

Рано или поздно все компоненты, созданные и видоизмененные разными разработчиками, должны быть построены и собраны в единую систему.

В прошлом я часто был свидетелем (и, разумеется, виновником) существенных задержек выполнения проектов вследствие интеграции в последний момент, когда неожиданные дефекты обнаруживались на последних стадиях цикла разработки.

Проблема интеграции возникает по самым разным причинам:

- плохо проверенные компоненты;
- неверные (а нередко и необоснованные) предположения относительно обработки исключительных ситуаций, пустых значений, параметров, глобальных переменных и т.д.;
- слабая конструкция;
- непредсказуемое поведение на разных платформах;
- неожиданные отличия в выпускаемых версиях и отладочных конструкциях;
- вопросы запутывания кода;
- вопросы установки и разрешения;
- ошибки в базовых структурах.

Следует заметить, что лишь некоторые из перечисленных выше причин могут быть устранены благодаря дисциплинированности и общительности разработчиков. В действительности группа разработчиков редко уделяет внимание интеграции, и если количество классов, используемых в системе, достаточно велико, то ошибки, накапливающиеся в результате совместной работы над проектом, можно искать до бесконечности. Это самый неприятный вариант, часто называемый “адом интеграции”.

Решение проблемы или радикальный шаг в правильном направлении

Для того чтобы свести к минимуму трудности интеграции, целесообразно формировать конструкции *автоматически* и применять стратегию пошаговой интеграции, где весь код перестраивается и проверяется хотя бы ежедневно, если нет возможности делать это непрерывно. Дело в том, что если введенный недавно код нарушает конструкцию, его следует исправить немедленно или же отменить изменения для восстановления системы в последнее состояние, известное как работоспособное.

Ниже перечислены основные части, из которых состоит система непрерывной интеграции:

- машина, специально выделенная для компоновки;
- система управления исходным кодом, выполняющая роль центрального хранилища всего исходного кода;
- служба текущего контроля, проверяющая систему управления исходным кодом на предмет изменений в исходном коде;
- механизм создания сценариев, запускаемый указанной выше службой и позволяющий строить конструкции;
- система отчетности, способная сразу же реагировать на результаты компоновки.

Имеется несколько продуктов интеграции, которые можно исследовать, чтобы остановить свой выбор на одном из них. В настоящее время я лично отдаю предпочтение продукту CruiseControl.NET [CC.NET], которым я пользуюсь совместно с NAnt — весьма распространенным механизмом компоновки на базе XML. Оба продукта являются открытыми. Для настройки CruiseControl.NET требуется приложить немало усилий, но как только это будет сделано, вам останется лишь вызывать NAnt всякий раз, когда происходят изменения в базе кода. При этом вы получаете уведомления о ходе построения и состоянии всех текущих конструкций, в которых применяются самые разные клиентские модули, включая небольшое приложение, использующее пиктограмму лотка из Windows, цветовое кодирование и всплывающие подсказки с уведомлениями для оперативного отображения итоговой информации.

Уроки, которые я извлек из опыта работы в своей организации

Когда мы первый раз рассматривали возможность внедрения системы непрерывной интеграции в компании InnerWorkings, большинство из нас нашло эту идею замечательной. Несмотря на сжатые сроки выполнения проекта, мы были уверены, что потери нескольких дней на внедрение такой системы вполне оправданы. Очевидно, что мы просто недооценили глубокое влияние подобной системы на качество программных продуктов и атмосферу доверия в группе разработчиков.

Теперь у нас имеется около 500 непрерывно интегрированных решений, причем их число постоянно растет. Треть из этих решений отличается наличием общей, подстраиваемой под заказчика, базовой структурой, которая также является интегрированной. К этапам интеграции всех этих решений относятся следующие: компиляция, запутывание кода, компоновка, тестирование и развертывание с использованием разных конфигураций и платформ.

Перед тем как приступать к данному проекту, нас предупредили, что мы не сможем интегрировать все свои решения непрерывно. Но я убежден, что выполнить данный проект иначе было бы просто невозможно и что такая интеграция оказалась одной из главных составляющих нашего успеха.

Дополнительные сведения

В качестве хорошей отправной точки для получения дополнительных сведений о непрерывной интеграции см. статью *Continuous Integration* Мартина Фаулера и Мэттью Фоммеля [Fowler/Foemmel CI].

Благодарю Клаудио за помощь! А теперь перейдем к некоторым аспектам эксплуатации.

Не следует забывать об эксплуатации

Не так давно у меня была беседа с представителями крупной шведской компании. Речь шла, в частности, о базовой структуре Valhalla и о ее внешнем виде в конкретный момент времени.

Меня спросили, каким образом мы реализовали такие механизмы эксплуатации, как регистрация, настройка, безопасность системы и прочее. Когда я ответил, что мы еще не ввели подобные механизмы, сначала наступило молчание, а затем последовал громкий смех. Оказывается, они потратили годы на решение подобных вопросов в своей базовой структуре, а мы даже и не думали о них.

Правда, мне удалось в какой-то мере оправдаться. На самом деле мы много думали об этом, но мне хотелось определить основные составляющие базовой структуры, прежде чем вводить механизмы эксплуатации. В конце концов, эти составляющие оказывают влияние на то, как должен выглядеть сам механизм. Кроме того, я сослался на свою первую книгу [Nilsson NED], где я посвятил несколько начальных глав таким механизмам, как трассировка, регистрация и конфигурация.

Когда механизм нужен

Почему так важны вопросы эксплуатации? Обратимся к примеру. Допустим, что трассировка приложения, введенного в эксплуатацию, оказалась недостаточной. (Это не выдуманный пример. Мне хорошо известно, что подобный аспект эксплуатации нередко упускается из виду. Я постоянно напоминал себе об этом последние годы, но так и не удосужился встроить трассировку в свои старые приложения, находящиеся в эксплуатации.) Когда возникает необычная ошибка, которая практически не фиксируется в журнале регистрации ошибок, ее причину очень трудно обнаружить, а саму ошибку — устранить.

Отсутствие трассировки на месте

Трассировку можно ввести в любой момент времени, но для этого потребуется как минимум два дня. Если же ошибка серьезная, то заказчик вправе рассчитывать на то, что ошибка будет устранена быстрее, чем за два дня.

Распространенный (и зачастую весьма неэффективный) выход из этого положения состоит во внесении эпизодических изменений в надежде на то, что проблема исчезнет сама.

Но вместо этого лучше всего выполнять эпизодическую трассировку в разных местах приложения. Разумеется, код приложения станет от этого менее изящным, а для выявления ошибки потребуется время. В следующий раз может возникнуть другая ошибка, хотя с тех пор мало что изменилось. И тогда придется начинать все с самого начала.

Кроме того, в производственной среде можно запустить на выполнение отладчик. Но в этом случае придется вмешиваться в работу других систем или расшифровывать исходный код, специально запутанный для того, чтобы его нельзя было отладить.

Кроме того, изменять код в процессе эксплуатации рискованно — даже если изменения не столь значительны, как введение трассировки. И хотя этот риск невелик, он все же существует.

На заметку

Если вы применяете метод аспектно-ориентированного программирования (АОП — Aspect-Oriented Programming (AOP)), то для введения трассировки впоследствии потребуется лишь несколько минут. Более подробно об АОП речь пойдет в главе 10.

Трассировка на месте

Если трассировка применяется на месте в виде готового решения, ее эффективность в отношении поиска ошибок должна быть хорошо известна. Вместо целого дня для выявления ошибок в данном случае потребуется несколько минут.

При этом очень важно быть внимательным и не следовать слишком часто принципу YAGNI (“Это вам не понадобится”), когда речь идет о механизмах эксплуатации. Ведь применение данного принципа обходится слишком дорого, когда возникает потребность во введении механизма эксплуатации. Согласно принципа YAGNI затраты на ввод какого-то компонента сначала и впоследствии оказываются практически одинаковыми, и поэтому можно подождать до тех пор, пока этот компонент не понадобится. Но когда затраты сперва малы, но велики впоследствии и, скорее всего, на них придется пойти, то потребуется иное решение.

Некоторые примеры механизмов эксплуатации

Ниже перечислен ряд механизмов эксплуатации, которые можно считать пригодными для большинства приложений масштаба предприятия.

- **Трассировка**
  Как уже упоминалось, целесообразно иметь возможность контролировать, что происходит в системе, когда пользователи работают по разным сценариям. Это не только очень эффективное решение для отслеживания программных ошибок, но и средство для выявления, например, узких мест в программном обеспечении.

- **Регистрация**
  Ошибки, предупреждения и информационные сообщения должны регистрироваться. Это особенно важно для выявления ошибок после того, как они произошли. Нельзя рассчитывать на то, что пользователи пришлют сообщения с точным описанием возникших проблем. Кроме того, информацию о работе системы лучше собирать, не показывая ее пользователям.
- **Конфигурация**
  Приходилось ли вам перекомпилировать приложения только из-за того, что сервер базы данных был перенесен на другую машину с новым именем? А мне приходилось. Безусловно, такого рода информация должна допускать настройку, и это может касаться большого количества информации, что, конечно, зависит от конкретного приложения.
- **Контроль производительности**
  Осуществление контроля производительности на основании информации о модели предметной области и других составляющих приложения оказывает большую помощь не только в отслеживании ошибок, но и в предупреждении неполадок в системе. Такой контроль позволяет без труда выяснить, что для выполнения отдельного сценария потребовалось на 30% больше времени, чем, к примеру, два месяца назад.
- **Безопасность**
  В настоящее время значение безопасности не нужно пояснять. Очевидно, следует внимательно рассмотреть такие вопросы, как аутентификация и установление полномочий. Кроме того, приложения должны быть защищены от разного рода атак.
- **Ревизия**
  Ревизию как один из аспектов безопасности очень важно проводить таким образом, чтобы затем можно было проверить, кто и когда это делал.

Это не только оплошность разработчиков

В защиту разработчиков следовало бы отметить, что на мои многочисленные вопросы, касающиеся требований к механизмам эксплуатации, специалисты по эксплуатации не смогли сказать ничего существенного. Вероятно, они не были избалованы такой поддержкой со стороны приложений.

В связи с этим желательно как можно раньше ознакомиться с типичными операциями и рассмотреть систему со стороны тех, кому предстоит ее эксплуатировать, чтобы выработать именно те механизмы эксплуатации, которые могут в итоге потребоваться. Обычный заказчик вряд ли сможет правильно сформулировать требования к механизмам эксплуатации. Они являются примером типичных нефункциональных требований, и поэтому обычный заказчик мало чем здесь поможет.

Гибкость механизмов эксплуатации может приобрести особое значение, поскольку разные заказчики пользуются разными рабочими платформами. Для этого

имеются стандарты, аналогичные Windows Management Instrumentation (WMI — Инструментальные средства управления средой Windows), но при построении базовой структуры благоразумнее руководствоваться принципом гибкости, чтобы иметь возможность без труда переходить, например, на другие форматы вывода для регистрации. Ведь один заказчик может пользоваться CA Unicenter, другой — Microsoft Operations Manager (MOM — Диспетчер операций от Microsoft), третий — еще одним продуктом, несовместимым с WMI, и т.д.

Резюме

В завершение этой главы мы коснулись ряда вопросов эксплуатации. Данная тема не рассматривается в этой книге глубоко, главное внимание уделяется предметной области как основной ценности для построения приложений.

Разумеется, в этой главе невозможно было охватить все вопросы, имеющие значение для современной разработки программного обеспечения. Я поставил перед собой задачу вкратце обсудить некоторые ценности, которые должны быть важны для каждого разработчика. В ходе этого обсуждения были введены понятия проектирования на основе предметной области, моделей предметной области, разработки посредством тестирования, шаблонов и многие другие понятия, касающиеся архитектуры и процессов.

Хотелось бы еще раз подчеркнуть три очень важных ценности: равновесие, учет контекста и постоянное обучение. Эти ценности важны не только для разработчиков программного обеспечения и его архитектуры, но и для каждого человека.

Итак, подготовив сцену, перейдем к рассмотрению разных видов шаблонов.

Глава 2

Переход к шаблонам

Постоянно сталкиваясь с новыми задачами проектирования и решая их, мы порой загоняем себя в угол. В одних случаях найденное решение страдает серьезными недостатками, а в других сами решения получаются неудачными. Если же повторно использовать удачные, проверенные и гибкие решения, то подобные риски сводятся к минимуму, а желаемый результат достигается быстрее. Подходящим инструментом для этой цели служат шаблоны. Благодаря высокому уровню абстракции, присущему шаблонам, мы можем приступить к более сложным задачам проектирования и успешно решить их. Знание шаблонов способствует также лучшему взаимопониманию между разработчиками, а это очень важный фактор разработки систем.

Мне уже не раз приходилось слышать, что шаблоны — теоретическая чепуха и элитарная выдумка, не приносящая никакой пользы. Эта глава опровергает такое мнение, поскольку оно очень далеко от истины. Шаблоны могут быть очень практичными, полезными для повседневной работы и чрезвычайно интересными для разработчиков. Возможно, вы обратили внимание на то, что я уже упоминал в предыдущей главе некоторые шаблоны. К ним, в частности, относится шаблон модели предметной области [Fowler PoEAA]. В этой главе рассмотрены три разные категории шаблонов: шаблоны проектирования (как обобщенные, так и прикладные), архитектурные шаблоны и шаблоны предметной области.

На заметку

Следует иметь в виду, что подобное разделение на категории весьма условно, хотя оно и не столь важно и интересно, как сами шаблоны. Поэтому не обращайте особого внимания на такое разделение шаблонов на категории, если оно никак не помогает вам лучше понять сами шаблоны.

Даже если вы знакомы с шаблонами, в этой главе вы все равно найдете для себя много интересного. Я постараюсь не пользоваться старыми пояснениями, но изложу свое представление о шаблонах. В частности, я намерен сосредоточить обсуждение темы шаблонов вокруг модели предметной области, что не совсем типично для рассмотрения шаблонов проектирования. Надеюсь, что хотя бы рассуждения на данную тему в разных разделах этой главы покажутся вам интересными.

Прежде всего, хотелось бы остановиться на самом понятии шаблонов и показать, почему их следует изучать.

Вкратце о шаблонах

Шаблоны предоставляют простые, изящные решения часто повторяющихся задач проектирования. Главное преимущество шаблонов — в гибкости, модульности и возможности создавать понятную и ясную конструкцию. Как видите, я даже не упомянул возможность повторного использования, хотя это не совсем справедливо. Шаблоны переносят акцент с повторного использования кода на повторное использование знаний. Поэтому шаблоны имеют непосредственное отношение к повторному использованию, но не в том смысле, в каком мы привыкли думать о нем.

На заметку

Грегори Янг заметил, что многие шаблоны имеют отношение к повторному использованию посредством развязывания. Характерным тому примером служит шаблон внесения зависимостей (Dependency Injection), более подробно рассматриваемый в главе 10.

В процессе изучения шаблонов у вас может возникнуть мысль о том, что они совсем не похожи на то, что вы уже привыкли делать. Самое интересное, что шаблоны были не *изобретены*, а скорее, *собраны* или *выделены* в виде *проверенных* решений. Но решение — это лишь часть шаблона. Они делятся на три части: *контекст*, *задачу* и *решение*.

Учиться лучше всего на чужих ошибках, но иногда нужно обращаться и к знаниям, накопленным другими, что служит веским основанием для изучения шаблонов. А теперь попробуем найти и другие основания.

Зачем изучать шаблоны

Наиболее очевидным основанием для этого может служить тот факт, что шаблоны являются хорошими абстракциями, предоставляющими конструктивные элементы для проектирования системы.

Если группа разработчиков осознает преимущества шаблонов, они становятся очень важной частью языка общения. Вместо того чтобы описывать во всех деталях каждую проектную идею, нередко оказывается достаточно произнести название шаблона, чтобы все остальные члены группы смогли оценить пригодность этой идеи для решения конкретной задачи. Ввод шаблонов в язык общения группы разработчиков служит сам по себе очень веским основанием для выбора шаблонов, поскольку они способствуют взаимопониманию, а также увеличению богатства и выразительности языка общения. Ведь общение играет очень важную роль в разработке программного обеспечения.

Еще одним основанием для изучения шаблонов служит тот факт, что умение использовать шаблоны может пригодиться надолго. Для сравнения я изучил SQL в 1988 году и могу до сих пор неплохо зарабатывать себе на жизнь, работая только с этим языком. Продукты и платформы, с которыми я работаю, менялись с тех пор не-

сколько раз, но базовые принципы остались неизменными. То же самое относится и к шаблонам. Книга *Design Patterns* [GoF Design Patterns] вышла в 1995 году и до сих пор очень актуальна. Следует также заметить, что шаблоны не зависят от конкретного языка, продукта или платформы. (На разных платформах могут поддерживаться определенные разновидности реализации, и это лишний раз свидетельствует в пользу книги *Design Patterns*, которая была написана с учетом объектной ориентации.)

Читая эту книгу, можно обнаружить, что шаблоны весьма согласуются с принципами хорошего объектно-ориентированного проектирования. А что собой представляет хорошее объектно-ориентированное проектирование? Некоторые его принципы рассматриваются в книге Роберта К. Мартина (Robert C. Martin) *Быстрая разработка программ: принципы, шаблоны и практические методы* (*Agile Software Development: Principles, Patterns, and Practices*, пер. с англ., ИД "Вильямс", 2003 г.) [Matrin PPP]. К ним, в частности, относятся *принцип единственной ответственности* (ПЕО — Single Responsibility Principle (SRP)), *принцип открытости-закрытости* (ПОЗ — Open-Closed Principle (OCP)) и *принцип подстановки Лискова* (ППЛ — Liskov Substitution Principle (LSP)).

Принципы объектно-ориентированного проектирования

Ниже дается краткое пояснение принципов объектно-ориентированного проектирования, сформулированных Мартином.

- **Принцип единственной ответственности (ПЕО)**
  У такого элемента, как класс, должна быть лишь одна, определяемая им ответственность. Если же класс отвечает и за представление данных и доступ к ним, это характерный пример нарушения принципа ПЕО.
- **Принцип открытости-закрытости (ПОЗ)**
  Класс должен быть закрыт для модификации, но открыт для расширения. При изменении класса всегда существует риск что-то нарушить. Но если вместо модификации класс расширяется подклассом, такое изменение менее рискованно.
- **Принцип подстановки Лискова (ППЛ)**
  Допустим, что существует иерархия наследования для классов `Person` (Лицо) и `Student` (Студент). При использовании класса `Person` должна быть возможность использовать класс `Student`, поскольку он является подклассом `Person`. На первый взгляд, это всегда происходит автоматически, хотя и не совсем очевидно в отношении рефлексии — метода, позволяющего проверять программными средствами тип экземпляра объекта, считывать и устанавливать его свойства и поля, а также вызывать его методы, ничего не зная заранее об этом типе. Так, в методе, использующем рефлексию для обращения к классу `Person`, может и не предполагаться подкласс `Student`.

Проблема рефлексии имеет отношение к синтаксису. Мартин приводит в большей степени семантический пример класса `Square` (Квадрат), который относится к классу `Rectangle` (Прямоугольник). Но когда используется метод задания ширины квадрата `SetWidth()` для класса `Square`, это не имеет смысла — по крайней мере, для этого нужно еще вызвать (внутренним образом) метод задания высоты квадрата `SetHeight()`. Такое поведение отличается от того, что требуется для класса `Rectangle`.

Безусловно, все эти принципы оказываются спорными в определенных контекстах. Ими следует руководствоваться лишь как рекомендациями, но не "истиной в последней инстанции". Например, применяя ПОЗ, нетрудно переусердствовать до такой степени, когда становится ясно, что для реализации метода лучше модифицировать класс, чем расширять его. И ввод метода в класс можно также рассматривать как расширение.

Шаблоны пригодны не только для проектирования наперед, но и очень полезны (возможно, даже в большей степени) во время рефакторинга кода, когда невольно возникает мысль: "Мой код становится все более неорганизованным! Пора воспользоваться шаблоном!" Это проблема, где трудно отделить причину от следствия, но я решил все же начать с шаблонов, а затем перейти к рефакторингу кода (в следующей главе).

Предостережения в отношении шаблонов

Откровенно говоря, я не вижу особых причин не изучать шаблоны, но существует, по крайней мере, один весьма распространенный отрицательный эффект, от которого следует предостеречь. Дело в том, что среди тех разработчиков, которые только ознакомились с шаблонами, весьма распространено предубеждение, будто они просто вынуждены втискивать по 17 шаблонов буквально в каждое решение. Но чаще всего этот эффект действует недолго.

Намного дольше остается риск переусердствовать при проектировании. Ведь кроме применения 17 шаблонов, нужно хорошенько обдумать решение самой задачи. И если первоначальное решение кажется неверным, это совсем не означает, что в нем недостает какого-то шаблона.

На заметку

Мой коллега поведал мне недавно о том, как он обсуждал задачу проектирования с разработчиками в одной компании. Ему потребовалось три минуты, чтобы найти очень простое решение этой задачи, которая действительно была несложной. Но других разработчиков такое решение не вполне устраивало, поэтому они потратили три дня на тщательный анализ его правильности.

На мой взгляд, *разработка посредством тестирования* (РПТ) — это именно тот метод, который позволяет избежать осложнений, связанных с чрезмерным проектированием. При этом намного больше внимания уделяется решению конкретной задачи — и только, а шаблоны вводятся по мере надобности в ходе рефакторинга кода.

У вас может создаться впечатление, будто шаблоны — это панацея. Но это всего лишь одно из средств в обширном арсенале разработчика.

Шаблоны нередко воспринимаются как слишком "простые", но они становятся очень сложными в контексте и в сочетании с другими шаблонами. Я уже не раз встречал в группах новостей сообщения, подобные следующему: "Я разобрался с шаблоном X, но когда я попытался применить его в своем приложении вместе с шаблонами Y и Z, это оказалось чрезвычайно трудно. Пожалуйста, помогите!" Но ведь это не повод, чтобы *не* изучать шаблоны. В этой книге, к сожалению, нет места для обсуждения данного вопроса, хотя ему и уделяется некоторое внимание. Прежде всего, необходимо рассмотреть шаблоны отдельно от вопросов проектирования.

В своей книге *Рефакторинг с использованием шаблонов* (*Refactoring to Patterns*, пер. с англ., ИД "Вильямс", 2006 г.) [Kerievsky R2P] Джошуа Кериевски неоднократно подчеркивает, что шаблоны чаще всего должны применяться не при проектировании наперед, а в ходе рефакторинга кода в направлении шаблонов.

Усвоение шаблонов

Вот мнение Грега Ирвина (Gregg Irwin) об усвоении шаблонов: "Как и многие другие понятия, шаблоны, на мной взгляд, следует изучать поэтапно:

1. Сначала вы пользуетесь ими, даже не осознавая этого.
2. Затем вы узнаете, читаете о них и пытаетесь в них разобраться.
3. Изучив шаблоны больше, вы начинаете применять их откровенно, если не наивно.
4. Воодушевившись, вы начинаете их проповедовать (возможно и такое).
5. Наконец до вас что-то "доходит".
6. Вы продолжаете изучать шаблоны и применять их "не так наивно" и менее откровенно.
7. Проходит время, и вы замечаете в них недостатки.
8. Далее вы подвергаете сомнению саму концепцию шаблонов (зачастую из-за неправильного ее применения).
9. После этого вы вообще забываете о шаблонах или же приобретаете больше знаний и опыта их применения (повторяя, если требуется, пп. 5–9).
10. И наконец, вы вновь пользуетесь ими, даже не осознавая этого".

А теперь перейдем к рассмотрению первой категории шаблонов и постараемся сделать так, чтобы они стали более понятными для начинающих.

Шаблоны проектирования

Одна лишь мысль об этой категории шаблонов вызывает ассоциации с книгой *Design Patterns* [GoF Design Patterns], которая не раз упоминается в данной книге. Это, конечно, не единственная книга о шаблонах проектирования, тем не менее, она считается авторитетным трудом на данную тему.

Шаблоны проектирования имеют довольно низкий уровень и в то же время абстрактны, поскольку они, с одной стороны, имеют специальный характер, а с другой — общий характер по отношению к предметной области. Такие шаблоны остаются полезными независимо от уровня и типа создаваемой системы.

Назначение шаблонов проектирования можно, в частности, сформулировать следующим образом: они служат для уточнения и улучшения подсистем или компонентов. При последующем рассмотрении двух других категорий шаблонов обнаружится следующий характерный факт: шаблоны проектирования часто используются в других шаблонах, а некоторые из них могут применяться особым образом.

На заметку

С низким уровнем шаблонов проектирования связана одна любопытная история. Меня как-то спросили, чем я отличаюсь от своего коллеги на профессиональном уровне. Я ответил, что мой коллега занимается программированием на языках низкого уровня, а я — на языках высокого уровня. Тот, кто задал мне этот вопрос, совершенно не разбирался в программировании, но сделал вид, очень напоминающий выражение лица самодовольного бахвала.

Книга *Design Patterns* [GoF Design Patterns] читается с большим трудом. Но всякий раз, когда я читаю ее, то узнаю и выясняю для себя что-то новое. В частности, по поводу шаблонов у меня часто возникали мысли о том, что это неверное, не самое лучшее и даже нелепое решение. Но по зрелом размышлении, я всякий раз приходил к выводу, что оно все-таки верное.

А теперь перейдем от общих рассуждений к конкретному пояснению шаблонов проектирования. Для этой цели я выбрал свой любимый шаблон проектирования, называемый State (Состояние).

Пример применения шаблона State

Обучение на конкретных примерах — весьма эффективный педагогический прием, поэтому я и воспользуюсь им, рассмотрев следующую задачу.

Задача

Заказ на закупку может находиться в одном из следующих состояний: `New Order` (Новый заказ), `Registered` (Зарегистрирован), `Granted` (Подтвержден), `Shipped` (Отгружен), `Invoiced` (Выписан счет-фактура) и `Cancelled` (Отменен). Имеются строгие правила, регламентирующие переход заказа из одного состояния в другое. Например, не разрешается непосредственный переход из состояния `Registered` в состояние `Shipped`.

Кроме того, имеются отличия в поведении в зависимости от конкретного состояния. Так, в состоянии `Cancelled` нельзя вызвать метод `AddOrderLine()`, чтобы ввести в заказ дополнительные позиции (это же, кстати, относится и к состояниям `Shipped` и `Invoiced`).

Следует также иметь в виду, что определенное поведение приводит к смене состояния. Так, если вызывается метод `AddOrderLine()`, происходит смена состояния `Granted` обратно на `New Order`.

Первое предлагаемое решение

Для решения этой задачи нужно описать граф состояний в коде. На рис. 2.1 приведен очень простой, классический граф состояний, схематически описывающий смену состояния кнопки при ее нажатии (`Down`) и отпускании (`Up`) пользователем.

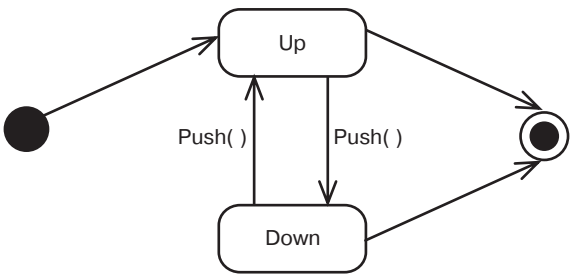


Рис. 2.1. Граф состояний кнопки

Если применить этот же метод к заказу (`Order`), то получится граф состояний, приведенный на рис. 2.2.

Одно из наиболее очевидных решений, вероятно, состоит в следующем применении оператора `enum`:

```
public enum OrderState
{
    NewOrder,
    Registered,
    Granted,
    Shipped
    Invoiced
    Cancelled
}
```

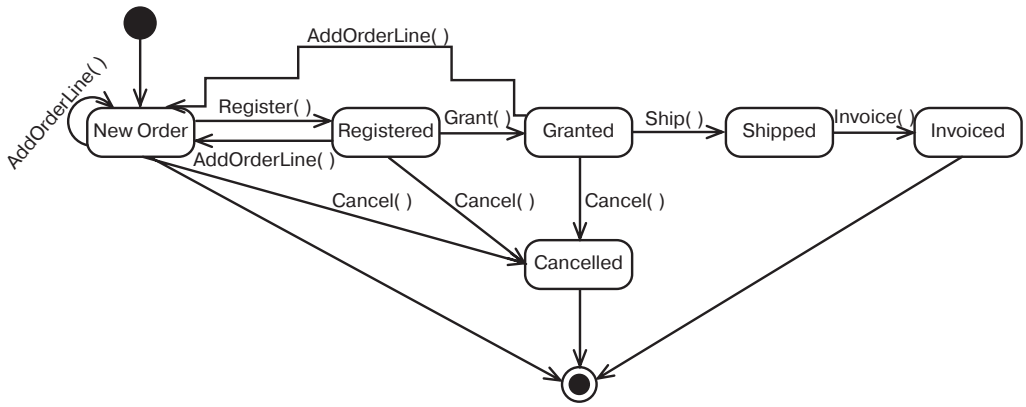


Рис. 2.2. Граф состояний заказа

Затем для хранения текущего состояния заказа можно воспользоваться закрытым полем в классе `Order` следующим образом:

```
private OrderState _currentState = OrderState.NewOrder;
```

После этого в отдельных методах необходимо в первую очередь проверить, может ли конкретный метод быть вообще вызван в данном состоянии, а также принять решение, следует ли осуществлять переход в новое состояние и каким оно должно быть. В методе `AddOrderLine()` это можно реализовать следующим образом:

```
private void AddOrderLine(OrderLine orderLine)
{
  if (_currentState == OrderState.Registered || _currentState ==
OrderState.Granted)
    _currentState == OrderState.NewOrder;
  else if (_currentState == OrderState.NewOrder)
    // Не выполнять переход.
  else
    throw new ApplicationException(...

  // Выполнить полезные действия...
}
```

Как следует из приведенного выше фрагмента кода, в данный метод было введено немало избыточного кода только ради соблюдения графа состояний. Аналогичный код предстоит рассредоточить по всему классу `Order`, чтобы распространить сведения о графе состояний среди разных методов. Это характерный пример едва заметного, но пагубного дублирования кода.

Даже в таких примерах, как этот, следует принять меры для сведения к минимуму дублирования и фрагментации кода. Попробуем это сделать.

Второе предлагаемое решение

Второе решение представляется несколько иным. В частности, создать закрытый метод смены состояния под названием `_ChangeState()` и вызывать его из метода `AddOrderLine()`. Этот метод может быть составлен из длинного оператора выбора следующим образом:

```
private void _ChangeState(OrderState newState)
{
  if (newState == _currentState)
    retun; // Не считать ошибкой переход состояния в самое себя

  switch (_currentState)
  {
    case OrderState.NewOrder:
      swith (newState)
      {
        case OrderState.Registered:
        case OrderState.Cancelled:
          _currentState = newState;
          break;

        default:
          throw new ApplicationException(...
          break;
      }
    case OrderState.Registered:
      switch (newState)
      {
        case OrderState.NewOrder:
        case OrderState.Granted:
        case OrderState.Cancelled:
          _currentState = newState;
          break;

        default:
          throw new ApplicationException(...
          break;
      }
...
// И так далее...
  }
}
```

Теперь метод `AddOrderLine()` выглядит следующим образом:

```
public void AddOrderLine(OrderLine orderLine)
{
    _ChangeState(OrderState.NewOrder);

    // Выполнить полезные действия...
}
```

В представленном выше фрагменте кода для метода `_ChangeState()` я поленился приводить полностью структуру оператора выбора, но, на мой взгляд, совершенно очевидно, что это весьма характерный пример недоброкачественного кода, особенно если учесть упрощенность данного примера, где вообще не рассматриваются подробно состояния заказа или особенности их смены.

> ***На заметку***
>
> Любой пример, в том числе и только что приведенный, может оказаться в некоторых случаях "верным решением". Мне просто хотелось бы подчеркнуть, что единственное решение задачи не всегда бывает верным.
>
> Что же касается недоброкачественного кода, то более подробно о нем речь пойдет в следующей главе.

Данное (т.е. второе предлагаемое) решение мне приходилось проводить в жизнь в нескольких проектах. Нельзя сказать, что мне по душе такое решение. На первый взгляд, оно кажется удачным, но по мере усложнения задачи оно доставляет все больше хлопот. Поэтому попробуем еще одно решение.

Третье предлагаемое решение

Третье решение основано на таблице, содержащей некоторую информацию о конфигурации и описывающую то, что должно происходить под определенным воздействием. Следовательно, вместо описания смены состояний в коде, как было предложено в первом и втором решениях, на этот раз подобная смена описывается в таблице (табл. 2.1).

Таблица 2.1. *Смена состояний заказа*

Текущее состояние	Новое допустимое состояние
`NewOrder`	`Registered`
`NewOrder`	`Cancelled`
`Registered`	`NewOrder`
`Registered`	`Granted`
`Registered`	`Cancelled`
...	...

Теперь в методе `_ChangeState()` достаточно проверить, является ли допустимым новое состояние, передаваемое в качестве параметра, если, например, текущим оказывается состояние `NewOrder`. В этом случае в качестве нового допускается только состояние `Registered` или `Cancelled`.

Данную таблицу можно дополнить еще одним столбцом (табл. 2.2).

Таблица 2.2. *Смена состояний заказа, пересмотренная*

Текущее состояние	Метод	Новое состояние
`NewOrder`	`Register()`	`Registered`
`NewOrder`	`Cancel()`	`Cancelled`
`Registered`	`AddOrderLine()`	`Cancelled`
`Registered`	`Grant()`	`Granted`
`Registered`	`Cancel()`	`Cancelled`
...		...

Теперь метод `_ChangeState()` должен воспринимать в качестве параметра не новое состояние, а имя соответствующего метода. После этого новое состояние определяется в данном методе по таблице.

Это довольно постое и ясное решение. Главное его преимущество заключается в простоте просмотра нескольких возможных вариантов смены состояний, а главный недостаток — вероятно, в том, что очень трудно учесть специальное поведение в зависимости от текущего состояния, а затем перейти в одно из нескольких возможных состояний при выполнении метода. Конечно, воплотить данное решение в жизнь не труднее, чем второе предлагаемое решение, но все же оно не очень удачное. В таблице можно было бы зарегистрировать информацию о тех делегатах (т.е. указателях на функцию со строгим контролем типов), которые должны выполняться при определенной смене состояний, а возможно, и распространить данный принцип на решение других задач, но, на мой взгляд, существует риск приведения кода в полный беспорядок, например, во время отладки.

А есть ли другие решения? Последуем принципу повторного использования знаний и попробуем применить шаблон проектирования под названием State.

Четвертое предлагаемое решение

Общая структура шаблона State приведена на рис. 2.3.

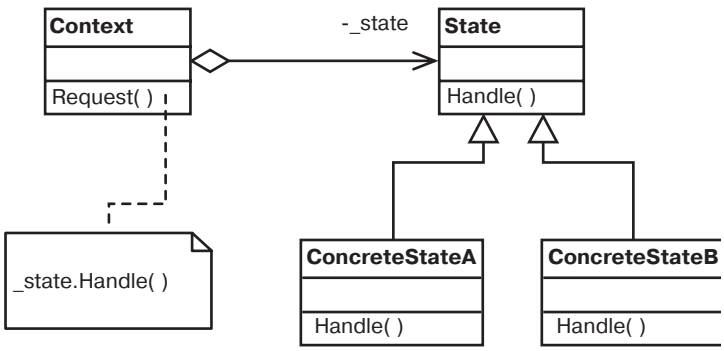


Рис. 2.3. Общая структура шаблона State

Идея заключается в том, чтобы инкапсулировать разные состояния в виде отдельных классов (см. `ConcreteStateA` и `ConcreteStateB`). Эти классы конкретных состояний наследуют от класса `State`. У класса `Context` имеется экземпляр в виде поля, и когда он получает вызов метода запроса `Request()`, то вызывает метод обработки `Handle()` экземпляра состояния. Метод `Handle()` по-разному реализуется для классов различных состояний.

Итак, это общая структура данного шаблона. А теперь посмотрим, как она будет выглядеть, если применить ее к решаемой задаче. На рис. 2.4 приведена UML-схема для рассматриваемого примера.

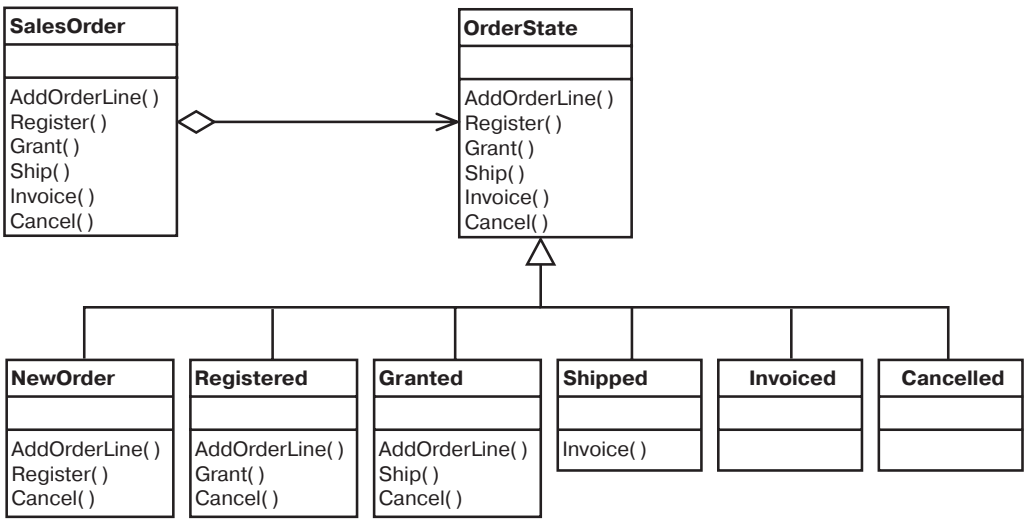


Рис. 2.4. Конкретный пример применения шаблона State

> ***На заметку***
>
> В данном примере было бы целесообразно ввести еще один абстрактный класс в качестве базового класса для состояний `NewOrder`, `Registered` и `Granted`. В этом классе могут быть реализованы методы `AddOrderLine()` и `Cancel()`.

В данном конкретном примере класс `Order` соответствует классу `Context` в общей структуре шаблона. Кроме того, у класса `Order` имеется поле `OrderState`, хотя на этот раз `OrderState` — не перечисление, а класс. В целях рефакторинга кода в старых тестах может предполагаться перечисление, и поэтому его стоит сохранить (во всяком случае как свойство проверки текущего экземпляра из иерархии наследования состояний в конкретной реализации), а следовательно, не вносить изменения во внешний интерфейс.

Вновь образовавшийся класс `Order` получает новый экземпляр состояния `NewOrder` при создании копии экземпляра и передает себя конструктору следующим образом:

```
internal OrderState _currentState = new NewOrder(this);
```

Как видите, данное поле объявляется как внутреннее. Дело в том, что класс состояния может сам изменить текущее состояние, поэтому класс `Order` передает все виды смены состояний классам разных состояний. (Можно было бы также сделать так, чтобы `OrderState` стал внутренним классом `Order`, тогда не пришлось бы указывать специально тип `internal`.)

На этот раз метод `Register()` выглядит в классе `Order` достаточно просто, например, следующим образом:

```
public void Register()
{
    _currentState.Register();
}
```

И в классе `NewOrder` метод `Register()` выглядит не намного сложнее. По крайней мере, в нем можно теперь уделить основное внимание определению его собственного состояния, и благодаря этому код становится четким и ясным. Во всяком случае он мог бы иметь следующий вид:

```
public void Register()
{
    _parent._Register();
    _parent._currentState = new Registered(_parent);
}
```

Перед сменой состояния происходит обратный вызов родителя (`_parent._Register()`), предписывающий ему подготовиться к смене состояния. Следует заметить, что обратный вызов передается внутреннему методу `_Register()` а не общедоступному методу `Register()`. Разумеется, это лишь один из возможных вариантов. Другие варианты состоят в том, чтобы поместить код в базовом классе `OrderState` или же в самом классе `NewOrder`. В итоге код должен оказаться в наиболее подходящем месте.

Таким образом, если требуется выполнить ряд операций до или после смены состояний, их нетрудно и очень удобно инкапсулировать. Если же требуется запретить определенную смену состояний в классе `NewOrder`, для этого достаточно пропустить реализацию соответствующего метода и воспользоваться для него реализацией базового класса `OrderState`. Эта реализация базового класса формирует исключение, указывая на недопустимую смену состояний, если требуется именно такое поведение. Другим типичным примером используемой по умолчанию реализации служит отсутствие каких-либо действий.

На заметку

Если требуются дополнительные исключения, зависящие от контекста, то соответствующие методы можно, конечно, реализовать и в подклассах, — даже если все они предназначены только для формирования исключений.

Это также означает, что вместо базового класса `OrderState` можно воспользоваться интерфейсом. Догадываюсь, что если бы книга *Design Patterns* [GoF Design Patterns] была написана ныне, во многих шаблонах вместо абстрактных базовых классов (или, по крайней мере, вместе с ними) использовались бы интерфейсы. Шаблон State не может служить характерным тому примером, хотя и в нем такое вполне возможно.

Дополнительные пояснения

Применяя шаблон State, мы на самом деле заменяем одно поле целым рядом отдельных классов. На первый взгляд, это далеко не самая блестящая идея, но в конечном итоге мы достигаем замечательного результата, перенося поведение на то место, которое ему принадлежит, что вполне согласуется с *принципом единственной ответственности* (ПЕО).

Разумеется, такому подходу присущи и недостатки, самый характерный из которых состоит в том, что программа может оказаться наполненной мелкими классами, когда применяется шаблон State.

Разумеется, каждое решение требует критического анализа, но, на мой взгляд, к применению шаблона State следует отнестись очень серьезно. Ведь может оказаться, что он даст возможность решить задачу при минимальном дублировании кода и разделении ответственности среди инкапсулированных и связующих структурных единиц — классов конкретных состояний. Следует, однако, иметь в виду, что шаблоном State легко злоупотребить, как, впрочем, и любым другим инструментом. Поэтому пользуйтесь им благоразумно!

Итак, мы рассмотрели пример применения шаблона проектирования, который относится к числу общих. Мы еще вернемся к шаблонам проектирования, но прежде обсудим другую категорию шаблонов.

Архитектурные шаблоны

Термин *архитектурные шаблоны* обычно ассоциируется с теми шаблонами, которые рассматриваются в книге *Pattern-Oriented Software Architecture* (Pattern-Oriented Software Architecture) [POSA 1] Бушмана и других (Buschmann et al.). В этой книге некоторые шаблоны отнесены к категории архитектурных. К ним, в частности, относятся шаблоны Pipes and Filters (Каналы и фильтры) и Reflection (Рефлексия).

Шаблон Pipes and Filters направляет потоки данных по каналам и обрабатывает их с помощью фильтров. Этот шаблон нашел признание среди разработчиков, пользующихся SOA, — архитектурой, ориентированной на службы, в качестве полезного средства для построения систем, ориентированных на обмен сообщениями.

Шаблон Reflection встроен в обе платформы Java и .NET. При этом вы получаете возможность писать программы, которые читают и записывают информацию в объекты в самом общем виде, используя метаданные объектов, т.е. ничего не зная заранее о типе объекта.

Если шаблоны проектирования предназначены для улучшения подсистем или компонентов, то архитектурные шаблоны служат для структурирования подсистем. А теперь перейдем к характерному примеру — шаблону Layers (Слои) [POSA 1].

Пример применения шаблона Layers

Слои или разделение на слои — это основной принцип разработки архитектуры, означающий перенос ответственности на отдельные связующие структурные единицы (группы классов) и определение зависимостей между этими единицами. Данный принцип хорошо знаком большинству разработчиков.

Шаблон Layers широко распространен и вполне понятен, поэтому приведем лишь краткий пример его применения, чтобы дать ясное представление о данной категории шаблонов.

Задача

Допустим, что построен ряд классов для приложения SalesOrder (Заказ на закупку), в том числе классы `Customer`, `Order`, `Product` и т.д. Эти классы инкапсулируют значение, которое они имеют для предметной области, а также способ их сохраняемости/несохраняемости и представления. Текущая реализация должна обеспечивать сохраняемость в файловой системе и представление объектов в виде фрагментов кода HTML.

К сожалению, выяснилось, что в приложении должна быть обеспечена возможность поддержки объектов в виде XML, а для сохраняемости следует использовать реляционную базу данных.

Первое предлагаемое решение: применить слои

Самое распространенное решение данной задачи состоит, вероятно, в том, чтобы перенести ответственность некоторых классов. Ведь новые требования проливают свет на то, что эти классы нарушают ПЕО.

Попробуем разделить классы таким образом, чтобы их отношение к ответственности оказалось связным с технологической точки зрения.

Таким образом старые классы теперь должны быть сосредоточены только на значении предметной области (назовем это слоем предметной области). А к ответственности представления (или, скорее, потребления) должен иметь отношение другой ряд классов (т.е. еще один слой — потребителя). И наконец, к ответственности сохраняемости должен иметь отношение еще один ряд классов (это слой сохраняемости).

В итоге самопроизвольно получается трехслойное решение: по одному слою на каждый вид ответственности. У двух новых слоев имеются две реализации, каждая из которых должна удовлетворять новым и старым требованиям.

Зависимости между слоями также определены, и в данном случае решено, что слой потребителя будет зависеть от слоя предметной области, а слой предметной области — от слоя сохраняемости. Таким образом, слою потребителя вообще ничего не известно о слое сохраняемости, что вполне приемлемо для данного конкретного проекта.

К вопросу разделения на слои мы еще вернемся в главе 4, рассмотрев его под несколько иным углом зрения.

Шаблон Domain Model в качестве другого примера

Шаблон модели предметной области (Domain Model) [Fowler PoEAA] мы уже упоминали в главе 1 и будем делать это на протяжении всей этой книги. Мне лично данный шаблон представляется удачным примером архитектурного шаблона.

Мы не будем рассматривать здесь пример применения шаблона Domain Model, поскольку данной теме посвящена львиная этой книги. Вместо этого продолжим обсуждение шаблонов, уделив внимание тому, насколько они зависят от предметной области. Итак, перейдем к шаблонам проектирования для конкретных типов приложений.

Шаблоны проектирования для конкретных типов приложений

Этот ряд шаблонов проектирования имеет не общий характер, как рассмотренные выше шаблоны, а специальный и предназначен для создания приложений масштаба предприятия.

Определить приложение для предприятия непросто, но его можно рассматривать как крупномасштабную информационную систему, обслуживающую многих пользователей или обрабатывающую большие объемы данных.

Основным источником по шаблонам данной категории служит книга *Архитектура корпоративных программных приложений* [Fowler PoEAA] Мартина Фаулера.

На первый взгляд, такие шаблоны могут показаться не такими привлекательными и замечательными, как некоторые другие шаблоны проектирования, но на самом деле они очень полезны, решают много насущных вопросов и воплощают в себе немало опыта и знаний. Как упоминалось выше, они имеют менее общий характер, чем остальные шаблоны проектирования, и сосредоточены, главным образом, на крупномасштабных информационных системах.

Их применяют к уже выбранной структуре логики, например, модели предметной области. Такие шаблоны определяют не структурирование *модели предметной области* (или любых других моделей для структурирования основной логики), а *инфраструктуру*, поддерживающую модель предметной области.

В качестве конкретного примера рассмотрим применение шаблона Query Object (Объект запроса) [Fowler PoEAA].

Пример применения шаблона Query Object

Допустим, что имеется модель предметной области для приложения SalesOrder, где имеются классы `Customer` и `Order`, причем класс `Order` состоит из целого ряда других классов, т.е. все очень просто и ясно.

Имеется также ряд других приложений, которые нужно выбрать для перемещения по модели предметной области. Одно из решений состоит в том, чтобы создать глобальный корневой объект со ссылками на корнеподобные совокупности объектов. Одним из примеров в данном случае может служить совокупность объектов покупателя. Итак, разработчик должен начать с глобального корневого объекта и перейти к совокупности объектов покупателя и просмотреть всю совокупность до тех пор, пока не обнаружит искомый объект, а возможно, перейти к заказам покупателя на закупку, если его интересует именно это.

По тому же принципу все совокупности объектов считаются глобальными, что дает возможность получить непосредственный доступ к совокупности объектов покупателя и просмотреть ее циклически.

Оба принципа легко понять и нетрудно применить, но им присущ один недостаток — с точки зрения распределенной системы им все же чего-то недостает. Допустим, что на стороне клиента действует модель предметной области (у каждого клиента имеется одна такая модель или, скорее, небольшое подмножество ее экземпляров, но не разделяемые экземпляры модели предметной области), а на стороне сервера — база данных (это довольно распространенная модель развертывания). Что произойдет, если попросить корневой объект предоставить совокупность объектов для миллиона покупателей? Все эти покупатели будут переданы клиенту, чтобы он смог циклически просмотреть совокупность объектов в локальном режиме. Это не самое изящное решение, поскольку придется ждать до тех пор, пока не будет передана вся совокупность объектов.

Другое решение состоит в том, чтобы ввести в действие сервер приложений и попросить его передавать на сторону клиента только *ссылку* на совокупность объектов. В этом случае объем передаваемых данных значительно сократится. Но с другой стороны, появится невероятное количество сетевых вызовов, когда клиент будет циклически просматривать список и запрашивать очередного покупателя по сети миллион раз. (А еще хуже, если маршализация самих экземпляров покупателей будет происходить не по значению, а только по ссылке.) В качестве еще одного решения можно было бы передавать совокупность объектов покупателя постранично, чтобы сервер предоставлял клиенту одновременно список из 100 покупателей.

Всем этим решениям присущ один общий недостаток. Ведь просматривать всех покупателей приходится нечасто, а для этого требуется некое подмножество, что приводит нас к следующей задаче.

Задача

Пользователям требуется форма, в которой им было бы удобно искать покупателей. У них должна быть возможность запрашивать всех покупателей, например, по следующим признакам:

- Имя, начинающееся с “aa” (скрытый маркетинг одной шведской автомобильной компании).
- Заказы за последний месяц.
- Заказы на общую сумму не более одного миллиона.
- Имя лица, ответственного за заказы “Stig” (Стиг).

Но в той же самой форме у них должна быть возможность запрашивать покупателей только в определенном регионе Швеции. И в этом случае форма поиска должна быть достаточно удобной.

Итак, рассмотрим три разных предлагаемых решения этой задачи, а именно: фильтрацию в модели предметной области, фильтрацию в базе данных по крупным спискам параметров и применение шаблона Query Object.

Первое предлагаемое решение: фильтрация в модели предметной области

Вернемся на шаг назад и допустим, что мы могли бы использовать любое из уже обсуждавшихся выше решений, чтобы воплотить где-нибудь совокупность объектов, а затем применить фильтр к каждому экземпляру этих объектов. Все экземпляры, удовлетворяющие критерию фильтрации, затем вводятся в новую совокупность объектов, что и дает искомый результат.

Это довольно простое решение, но оно оказывается практически непригодным во многих реальных ситуациях. При этом напрасно расходуется свободное пространство и время. Ведь приходится как-то материализовать сведения не только о миллионе покупателей, но и об их заказах. Такое решение, пожалуй, неосуществимо, не говоря уже о том, что сама задача может еще больше усложниться.

Разумеется, окончательный вывод в данном случае в значительной степени зависит от платформы выполнения. Напомним, что модель развертывания — это подмножество экземпляров данной модели у каждого клиента, база данных на сервере и отсутствие разделяемых экземпляров модели предметной области.

Если бы вместо этого имелось одно разделяемое подмножество экземпляров модели предметной области на сервере приложений (которому присущи свои недостатки — подробнее об этом в последующих главах книги), такое решение было бы приемлемым, но только для логики на стороне сервера. Что же касается клиентов, запрашивающих подмножество разделяемых экземпляров модели предметной области, то они должны как-то выразить свои критерии запроса.

Второе предлагаемое решение: фильтрация в базе данных по крупным спискам параметров

Базы данных, как правило, вполне подходят для хранения и запроса данных, поэтому воспользуемся ими в своих целях. В данном случае нужно выразить с помощью оператора SQL то, что требуется запросить, а затем преобразовать полученный результат в экземпляры из нашей модели предметной области.

Первую задачу мог бы решить следующий оператор SQL:

```
SELECT Id, CustomerName, ...
FROM Customers
WHERE CustomerName LIKE '%aa%'
AND Id IN
  (SELECT CustomerId
  FROM ReferencePersons
  WHERE FirstName = 'Stig')
AND Id IN
  (SELECT CustomerId
  FROM Orders
  WHERE TotalAmount > 1000000)
AND Id IN
  (SELECT CustomerId
  FROM Orders
  WHERE OrderDate BETWEEN '20040601' AND '20040630'
```

На заметку

Возможность объединения двух подвыборок, предназначенных для базы данных `Orders`, в одну подвыборку далеко не бесспорна, поскольку, на мой взгляд, это может привести к незначительному изменению значения по сравнению с поставленными требованиями. Но в данном случае это не так важно.

В этом решении мы просто материализуем экземпляры тех объектов, которые нас интересуют. Однако нет никакой необходимости в том, чтобы весь этот код SQL оказался в слое, содержащем модель предметной области. Каково же назначение этой модели в данном случае? Похоже, что в слое покупателей придется иметь дело с двумя моделями.

Итак, перед нами встает новая задача: как выразить в слое потребителя то, что ему требуется? Слой предметной области, который отвечает за преобразования из базы данных в модель предметной области и обратно, может снабдить слой потребителя подходящим методом поиска. Поэтому второе предлагаемое решение выглядит следующим образом:

```
public IList SearchForCustomers
(string customerNameWithWildCards
, bool mustHaveOrderedSomethingLastMonth
```

```
, int minimumOrderAmount
, string firstNameOfAtLeastoneReferencePerson)
```

Возможно, это решение и удовлетворит требованию первого запроса, но не второго. Для этого придется ввести ряд других параметров, как например:

```
public IList SearchForCustomers
(string customerNameWithWildCards
, bool mustHaveOrderedSomethingLastMonth
, int minimumOrderAmount
, string firstNameOfAtLeastoneReferencePerson)
, string country, string town)
```

Вероятно, вы уже поняли, к чему это ведет. Список параметров быстро разрастается и становится непригодным к употреблению, поскольку может потребоваться еще немало других параметров. Конечно, можно было бы реализовать редакторы, отображающие метки-заполнители для каждого параметра при вызове метода, но такой способ не исключает ошибки и малопригоден. Когда же потребуется еще один параметр, придется изменить все старые вызовы или, по крайней мере, организовать новую перегрузку методов.

Еще одна трудность заключается в том, как выразить определенные понятия в списке параметров этим беспомощным способом указания примитивных типов данных. Характерным тому примером служит параметр `mustHaveOrderedSomething LastMonth` (То, что должно было быть заказано за последний месяц). А как насчет предыдущего месяца или последнего года? Конечно, в качестве параметров можно было бы в данном случае использовать две даты и перенести ответственность за определение временного промежутка между ними на потребителя метода, но что если нас интересуют только покупатели в определенном городе? Как в таком случае должны выглядеть параметры даты? Полагаю, что можно было бы выбрать минимальную и максимальную даты, чтобы образовать как можно больший временной промежуток между ними, но это совсем не очевидный способ выражения “всех дат”.

На заметку

Грегори Янг так прокомментировал проблему выражения способа предшествования операторов в списке параметров, сославшись на следующий пример: (критерий1 и критерий2) или (критерий1 и критерий2)

От такого решения приходится быстро отказываться. Я пришел к этому выводу еще в те времена, когда работал в среде VB6, и поэтому остановил свой выбор на массиве. В первом столбце массива находилось имя поля (например, `CustomerName`), во втором столбце — оператор (например, оператор `Like` из перечисления), а в третьем столбце — критерий (например, “`*aa*`”). Для каждого критерия в массиве была выделена одна строка.

Такое решение позволило преодолеть некоторые трудности, связанные со списком параметров, но у него были свои трудности. С одной стороны, мне не нужно было менять старый код потребителя только потому, что может быть введен новый критерий — это было удобно. Но с другой стороны, данный способ оказался совершенно беспомощным в отношении более развитых критериев, и поэтому мне пришлось отступить на шаг назад и раскрыть схему базы данных для того, чтобы справиться, например, с таким критерием: "Имеются ли заказы на общую сумму не более одного миллиона?". Для критерия я тогда использовал полный оператор `IN`.

Решение, основанное на использовании массива, оказалось шагом в верном направлении, но оно могло бы стать еще более гибким, если бы массив был заменен объектами. К сожалению, в VB6 не было возможности создавать компоненты маршализации по значению. Конечно, можно было бы выбрать более гибкую структуру массива, но все это намного естественнее делается в .NET с помощью шаблона Query Object.

Третье предлагаемое решение: применение шаблона Query Object

Принцип, положенный в основу шаблона Query Object, состоит в том, чтобы инкапсулировать критерии в экземпляре объекта запроса (`Query`), а затем передать этот экземпляр в другой слой, где он преобразуется в требуемый запрос SQL. UML-схема общего решения может выглядеть так, как показано на рис. 2.5.

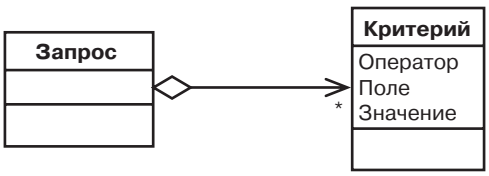


Рис. 2.5. Блок-схема класса для общего решения с помощью шаблона Query Object

В критерии можно было бы использовать другой запрос (даже если это не совсем очевидно из типичной блок-схемы на рис. 2.5). Подобным образом нетрудно сформировать аналог подзапроса в SQL.

А теперь попытаемся выработать язык для шаблона Query Object применительно к решаемой задаче. Но прежде допустим, что модель предметной области выглядит так, как показано на рис. 2.6.

Посмотрим, как вновь сформированный наивный язык запросов будет выглядеть на C#:

```
Query q = new Query("Customer");
q.AddCriterion("CustomerName", Op.Like, "*aa*");
Query sub1 = new Query("Order");
sub1. AddCriterion("TotalAmount", Op.GreaterThan, 1000000);

Query sub2 = new Query("Order");
```

```
sub2. AddCriterion("OrderDate", Op.Between,
DateTime.Parse("2004-06-01"), DateTime.Parse("2004-06-30"));
q.AddCriterion(sub2);

q.AddCriterion("ReferencePersons.FirstName",Op.Equal, "Stig");
```

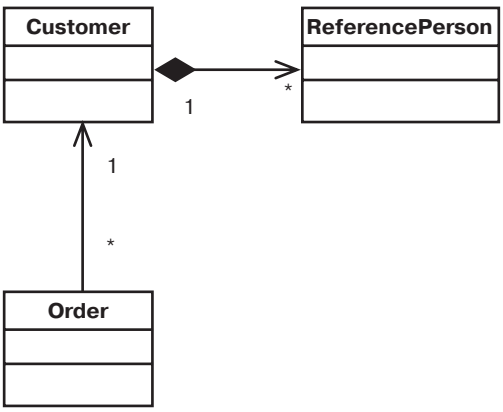


Рис. 2.6. Модель предметной области для рассматриваемого примера

На заметку

Параметр конструктора `Query` является не именем таблицы, а именем класса модели предметной области. Это же относится и к параметрами метода `AddCriterion()`. Под этим подразумеваются не столбцы таблицы, а свойства и поля класса. В данном случае имена свойств или полей используются в модели предметной области.

Следует также заметить, что в данном конкретном примере не потребовался подзапрос для критерия, связанного с поиском ответственных лиц (`ReferencePersons`), поскольку в модели предметной области можно перемещаться от объекта `Customer` к объекту `ReferencePerson`. С другой стороны, подзапросы потребовались для поиска заказов (`Orders`) по совсем другой причине.

Дополнительные пояснения

Если у вас имеется опыт программирования в SQL, то может сложиться впечатление, будто вариант в коде SQL более выразителен, легче читается и вообще лучше. Безусловно, SQL — эффективный язык запросов, но не следует забывать о цели, которая в данном случае преследуется. Ведь нам требуется как можно плотнее работать с моделью предметной области (в определенных пределах), чтобы добиться решения,

более удобного для сопровождения. Следует также заметить, что приведенный выше код C# оказался слишком многословным. Поэтому далее в этой книге будет показано, как может выглядеть синтаксис на примере написания "тонкого" слоя поверх общей реализации объекта запроса.

Итак, нам удалось добиться еще большей прозрачности кода по отношению к схеме базы данных. В целом, это не так уж плохо. Есть возможность в любой момент выйти за пределы этой маленькой "песочницы", если действительно понадобится составить запросы SQL, чтобы полностью использовать возможности базы данных, не прибегая к спасательным средствам.

Следует также подчеркнуть, что полноценная реализация шаблона Query Object быстро становится очень сложной, поэтому будьте внимательны, чтобы не переусердствовать.

В качестве положительного побочного эффекта следует отметить простоту использования объектов запроса для локальной фильтрации, например, для поддержания кэшированного списка всей продукции. В этом случае разработчик, пользующийся моделью предметной области, просто создает шаблон Query Object, как обычно, но затем применяет его немного иначе, не касаясь базы данных.

Предупреждение

Мне, конечно, хорошо известно, что кэширование не только приносит пользу, но и таит в себе опасность. Поэтому заранее предупреждаю: будьте осторожны, чтобы это благо не обернулось во зло.

Некоторые читатели, имеющие опыт ППО, вероятно, предпочли бы воспользоваться шаблоном Specification (Описание) [Evans DDD] для решения данной задачи, что приводит нас непосредственно к третьей и последней категории шаблонов, которую мы собираемся рассмотреть: шаблонам предметной области.

Шаблоны предметной области

У шаблонов предметной области совсем иное назначение, чем у архитектурных шаблонов. Их основное назначение — в структурировании самой модели предметной области, инкапсулировании в модели знаний о предметной области, применение универсального языка и в недопущении рассредоточения инфраструктуры по сторонам от самых важных ее частей.

По своему назначению эти шаблоны в чем-то пересекаются с шаблонами проектирования и, в частности, с шаблоном Strategy (Стратегия) [GoF Design Patterns], который может быть отнесен и к категории шаблонов предметной области. Причина такого пересечения заключается в том, что шаблоны, подобные Strategy, отлично подходят для структурирования модели предметной области.

Подобно шаблонам проектирования, шаблоны предметной области имеют как общий, так и специальный характер. В то же время они сосредоточены на самой сущности модели предметной области. Они призваны сделать эту модель более ясной, выразительной и содержательной, а также наполнить ее знаниями, добытыми из предметной области.

В конце предыдущего раздела упоминалось, что шаблон Specification относится к категории шаблонов предметной области и является альтернативой шаблону Object Query. Это, на мой взгляд, дает удобный повод для того, чтобы пояснить назначение шаблонов предметной области. Шаблон Object Query имеет специальное назначение, позволяя потребителю составить запрос с помощью синтаксиса, основанного на использовании объектов, чтобы так или иначе обнаружить любые объекты в модели предметной области. А шаблон Specification может также служить для формирования запросов, но вместо объекта общего запроса и поочередно задаваемых критериев в данном случае в качестве основной концепции используется описание, которое инкапсулирует знания о предметной области и сообщает свое назначение.

Например, для поиска покупателей золота можно воспользоваться как объектами запроса, так и описаниями, но решение в обоих случаях окажется разным. С помощью шаблона Object Query выражаются критерии определения покупателей золота, а с помощью шаблона Specification — класс, который может, в частности, называться `GoldCustomerSpecification` (Описание покупателя золота). В данном случае сами критерии не обнаруживаются и не дублируются, но инкапсулируются в классе, имеющем подходящее описательное имя.

Одним из источников по шаблонам предметной области служит книга [Arlow/Neustadt Archetype Patterns], но я выбрал пример шаблона предметной области из другого источника: книги *Domain-Driven Design* [Evans DDD] Эрика Эванса. Этим примером служит шаблон Factory (Фабрика).

На заметку

Здесь шаблон Factory рассматривается как шаблон предметной области, а в книге *Design Patterns* [GoF Design Patterns] также упоминаются шаблоны типа Factory. Но это не должно вводить в заблуждение читателей, знакомых с шаблонами, поскольку шаблоны проектирования сосредоточены в большей степени на специальном, техническом уровне, а шаблоны предметной области — на семантическом уровне предметной области.

Кроме того, они отличаются конкретными целями и типичными реализациями. В книге *Design Patterns* такие шаблоны называются Factory Method (Фабричный метод) и Abstract Factory (Абстрактная фабрика). В частности, шаблон Factory Method служит для переноса создания экземпляра “правильного” класса в подклассы, а шаблон Abstract Factory — для формирования семейств зависимых объектов.

В то же время шаблон Factory для ППО оказывается простым с точки зрения реализации и служит *только* для фиксации и инкапсуляции концепции создания определенного класса.

Пример применения шаблона Factory

Отрасль разработки программного обеспечения подвержена влиянию индустриализма. Это действительно так, но еще не известно, хорошо это или плохо. Ведь мы рассуждаем о проектировании как об удобном принципе разработки программного обеспечения, мы говорим об архитектуре, семействах программных продуктов и прочих понятиях, связанных с промышленным производством. Еще одним примером такого влияния может служить шаблон Factory. Но прежде чем рассматривать пример его применения, поставим соответствующую задачу.

Задача

В данном случае задача состоит в том, чтобы упростить составление заказа. Это нужно делать в двух разных случаях: во-первых, при составлении нового заказа, неизвестного базе данных; и во-вторых, при запросе старого заказа из базы данных для материализации в модели предметной области. В обоих случаях требуется создать экземпляр заказа, но на этом заканчивается их сходство в отношении составления заказа.

Кроме того, у заказа должен всегда быть клиент, иначе его составление теряет всякий смысл. В дополнение к этому требуется составлять кредитные поручения и повторные заказы.

Первое предлагаемое решение

Самое простое решение данной задачи состоит в использовании конструктора, подобного следующему:

```
public Order();
```

В этом случае после вызова данного конструктора потребителю придется задать свойства экземпляра посредством ввода или путем запроса соответствующих значений в базе данных.

К сожалению, такое решение чревато всякими неприятностями. В частности, можно было бы организовать отслеживание “запорченных” свойств, но не обязательно извещать об этом экземпляр, который был восстановлен из состояния сохраняемости как “запорченный”. Еще одно затруднение состоит том, как задать идентификацию, если она вообще не задается. Оба затруднения позволяет преодолеть рефлексия (по крайней мере, идентификатор не объявляется как `readonly` (только для чтения)), но должно ли все это беспокоить разработчика как потребителя модели предметной области? Вряд ли. Конечно, можно было бы проанализировать и менее явные решения,

но я уверен, что более распространенным и очевидным решением было бы использование параметризированных конструкторов.

В прошлом я посвятил немало времени программированию в VB6 и поэтому не избалован параметризированными конструкторами. Можете ли вы вообразить приложение без параметризированных конструкторов? Мне, во всяком случае, трудно себе это представить теперь.

Так или иначе, в C#, Java и прочих языках программирования у нас имеется возможность использовать параметризированные конструкторы, и в этом, собственно, заключается суть первого предлагаемого решения данной задачи. Итак, воспользуемся тремя общедоступными конструкторами класса `Order` следующим образом:

```
public Order(Customer customer);
public Order(Order order,
    bool trueForCreditFalseForRepeat);
public Order(int orderId);
```

Два первых конструктора используются при создании нового экземпляра объекта `Order`, пока еще отсутствующего в базе данных. Первый из них служит для создания нового, обычного заказа. Итак, пока все идет хорошо, но я умышленно отсрочил ввод требований, чтобы создать заказы в целях резервирования. При вводе этого требования первый конструктор должен будет измениться, чтобы его можно было использовать для двух разных целей.

Второй конструктор служит для создания кредитного поручения или повтора старого заказа. Определенно это выглядит не так ясно, как хотелось бы.

И последний конструктор используется при выборке старого заказа. Но выяснить, какой именно конструктор следует применять, можно только по параметру, что не совсем ясно. Кроме того, организация значительной обработки в конструкторе (в данном случае это особенно касается третьего конструктора) считается неудачным практическим приемом программирования, а непосредственное обращение к базе данных — и того хуже.

Второе предлагаемое решение

Как следует из книги *Effective Java* [Bloch Effective Java], первым и самым лучшим из 57 практических приемов программирования, которые следует взять на вооружение, считается замена конструкторов фабричными методами. В данном случае это может выглядеть следующим образом:

```
public static Order Create(Customer customer);
public static Order CreateReservation(Customer customer);
public static Order CreateCredit(Order orderToCredit);
public static Order CreateRepeat(Order orderToRepeat);
public static Order Get(int orderId);
```

Преимущество фабричного метода состоит в том, что у него есть имя, обнаруживающее его цель. Например, пятый метод в приведенном выше фрагменте кода

выглядит намного яснее, чем аналогичный конструктор из первого предлагаемого решения, в данном случае — третий конструктор. На самом деле это же можно сказать и об остальных фабричных методах, где требования резервирования и повторные заказы были введены без особых трудностей конструирования.

В упомянутой выше книге рассматриваются также статические фабричные методы, которые не обязательно создают новый экземпляр объекта всякий раз, когда они применяются, что может дать большие преимущества. Еще одно (более характерное) преимущество заключается в том, что эти методы могут возвращать экземпляр объекта любого подтипа возвращаемого ими типа.

А есть ли у них какие-нибудь недостатки? Их главный недостаток, на мой взгляд, состоит в нарушении ПЕО [Martin PPP], когда код создания находится в самом классе. В книге Эванса [Evans DDD] по этому поводу приводится удачное сравнение с автомобильным двигателем, которому ничего не известно о том, как он был создан, поскольку он за это не отвечает. Только представьте, насколько сложнее стал был автомобильный двигатель, если бы ему пришлось не только работать, но и создавать самого себя. Этот аргумент особенно справедлив в тех случаях, когда код создания оказывается сложным.

Аналогию с автомобильным двигателем можно дополнить такой деталью, как возможность перемещения двигателя из другого места, например, с полки на местном или центральном складе. Это означает, что старый экземпляр объекта должен быть воссоздан посредством выборки из базы данных и последующей его материализации, что делается совсем иначе, чем создание экземпляра автомобильного двигателя в действительности или объекта `Order` в программе.

Итак, мы подошли вплотную к вопросу применения шаблона. Воспользуемся решением, аналогичным второму предложенному, но вынесем созидательное поведение в отдельный класс, не обращая пока что внимания на выборку из базы данных, к которой имеет непосредственное отношение другой шаблон предметной области, называемый Repository (Хранилище) и более подробно рассматриваемый в последующих главах книги.

Третье предлагаемое решение

Итак, перейдем непосредственно к применению шаблона Factory в качестве шаблона предметной области. Начнем с его блок-схемы, приведенной на рис. 2.7.

Рис. 2.7. Блок-схема экземпляра для шаблона Factory

С точки зрения потребителя, код для получения готового экземпляра объекта `Order` может выглядеть следующим образом:

```
anOrder = OrderFactory.Create(aCustomer);
aReservation = OrderFactory.CreateReservation(aCustomer);

aCredit = OrderFactory.CreateCredit(anOldOrder);
aRepeat = OrderFactory.CreateRepeat(anOldOrder);
```

Для потребителя это не сложнее, чем использование обычного конструктора, хотя выглядит чуть более навязчиво, но и только.

Для того чтобы потребитель смог получить старый заказ, ему нужно сообщить еще кое-что (не только шаблон Factory, но и шаблон Repository). Это более ясный и выразительный способ, но о нем речь пойдет далее.

К сожалению, нельзя исключить создание экземпляров заказов посредством конструктора из других классов в модели предметной области в том случае, если код шаблона Factory находится во внешнем классе, но можно немного уменьшить вероятность наступления этого события, сделав конструктор внутренним. А поскольку в нем уже находится шаблон Factory, то разработчики сами догадаются, что это способ создать экземпляр класса, а не использовать непосредственно конструктор целевого класса.

На заметку

Эрик Эванс так прокомментировал предыдущий абзац: “Я надеюсь, что когда-нибудь языки программирования будут поддерживать такие концепции. Подобно тому, как это происходит теперь с конструкторами, в будущем они, возможно, будут допускать объявление фабрик и прочего”.

Дополнительные пояснения

Прежде всего, следует заметить, что иногда оказывается вполне достаточно и конструктора. Так, иерархия может не представлять особого интереса, клиенту требуется выбрать реализацию, конструкция оказывается довольно простой, а клиент имеет доступ ко всем свойствам. Важно понять, что фабрики следует создавать и применять не всякий раз, а только там, где они действительно помогают добиться большей ясности и раскрыть цели модели предметной области.

Для шаблона Factory весьма характерна установка экземпляра объекта в действительное состояние. В частности, мне пришлась по душе возможность устанавливать подэкземпляры с помощью шаблонов Null Object (Пустой объект) [Woolf Null Object], когда это уместно. Возьмем для примера заказ (`Order`). За отгрузку заказа отвечает транспортировщик (`Transpoter`).

Сначала заказ не был отгружен (и в этом случае отгрузка нас мало интересовала), поэтому мы не можем предоставить для него какой-либо объект `Transpoter`, но вместо того чтобы просто оставить пустым свойство `Transpoter`, мы устанавливаем это свойство в соответствии с пустым (скорее, “не выбранным”) объектом

Transpoter. Это всегда позволяет обнаружить описание свойства Transpoter аналогично следующему:

```
anOrder.Transpoter.Description
```

Если бы объект Transpoter был пустым, нужно было бы проверить сначала именно этот факт. Для подобных целей как нельзя лучше подходит шаблон Factory. (То же самое можно было бы сделать и с помощью конструктора, но во многих случаях приходится применять шаблоны Null Object, а решиться на это непросто, если имеются другие возможности. Однако не следует забывать об усложнении конструктора.)

Для того чтобы лучше управлять процессом создания за пределами шаблона Factory, можно, конечно, использовать несколько разных фабричных методов со многими параметрами.

Применяя шаблоны Factory, можно также скрывать требования к инфраструктуре, которых нельзя избежать.

Нередки случаи, когда термин *фабрика* применяется семантически не совсем корректно или, по меньшей мере, иначе, когда с помощью фабрик создаются все экземпляры объектов (как новые, так и "старые"). Например, в COM у каждого класса, как и у многих базовых структур, имеется своя фабрика класса. Но это не шаблон предметной области Factory: название и метод те же, но цели другие.

В качестве еще одного примера следует указать на возможность организовать в шаблоне Factory (косвенным образом) выборку значений из базы данных. И в этом случае следует избегать значительной обработки в конструкторах, поскольку это не самое лучшее место для размещения подобного рода логики.

На заметку

Порой возникает следующая проблема: задать некоторые свойства извне нельзя, но в то же время нужно извлечь из них значения. Это происходит при установке в шаблоне Factory значений по умолчанию (и выборке в шаблоне Repository старого экземпляра из базы данных). Возможно, такая ситуация покажется вам не очень приятной.

Мы еще вернемся к этому вопросу, а пока следует запомнить, что данная проблема нередко возникает из-за того, что для изменения внутреннего состояния нельзя использовать рефлексию, если даже это и разрешается в соответствии с настройками режима безопасности.

Я лично считаю, что программистов нельзя полностью удержать от неблаговидных действий, когда они выступают в качестве основных потребителей модели предметной области, но следует принять меры против необдуманных действий, совершаемых по ошибке, и следить за тем, чтобы неблаговидные действия не совершались.

В общем, я не вижу в этом, как правило, особой проблемы.

В целом применение шаблона Factory ясно показало, что некоторые сложности создания экземпляров объектов были перенесены из самого объекта `Order` в его собственную концепцию. Это позволило также несколько прояснить модель предметной области. Кроме того, это ясно указывает на интересный и сложный характер логики создания экземпляров.

Резюме

В этой главе были вкратце представлены шаблоны, и я попытался возбудить в вас интерес к ним. Надеюсь, что мне это удалось, поскольку я буду обращаться к шаблонам как к важным инструментам в последующих главах данной книги.

А теперь пора перейти к рассмотрению особенностей применения PPT на конкретных примерах.

Глава 3

РПТ и рефакторинг кода

Удачный пример — это очень сильное средство. Думаю, что вы согласитесь с этим. *Разработку посредством тестирования* (РПТ) можно, в частности, описать как применение тестов или *примеров* для установления поведения. Такие примеры могут неоднократно служить самым разным целям. Например, в процессе разработки они позволяют обнаружить недостающие или неверно выполняемые требования, при рефакторинге кода — служат для подстраховки, а по завершении разработки — помогают составить качественную документацию.

В этой главе речь пойдет не только о тестах и примерах, но и о самой РПТ, рассматриваемой на ряде примеров. На мой взгляд, это самый лучший способ раскрыть сущность данного метода.

После первоначального обсуждения особенностей РПТ и демонстрации тестирования по состоянию, мой коллега Клаудио Перроне представит некоторые методы создания примеров в диалоговом режиме, а также написания тестов с использованием заглушек и имитаций.

Рефакторинг кода является главным принципом, применяемым во время РПТ. Но даже рефакторинг кода ориентирован на примеры. Это можно представить себе, как написание сначала примера отдельной части решения, а затем рефакторинг кода для уточнения и улучшения данного примера до тех пор, пока не будет достигнут искомый результат. Такой подход избавляет от необходимости заранее создавать совершенную конструкцию, что очень удобно, поскольку добиться совершенства сразу практически невозможно. Рефакторинг кода будет также рассмотрен на ряде примеров.

Итак, начнем с основных положений РПТ.

Разработка посредством тестирования

Имея уже опыт применения РПТ в течение нескольких лет, я по-прежнему увлечен этим методом. Я все больше и больше убеждаюсь, что РПТ — это единственный и самый важный способ стать более умелым программистом. Сильно сказано, но требует аргументации. Вы можете, конечно, убедиться в этом сами, но прежде рассмотрим саму суть данного метода. Для этого вернемся к порядку выполнения РПТ, описанному в главе 1.

Порядок выполнения РПТ

Начинать следует с написания теста. Необходимо сделать так, чтобы неудачное завершение теста имело какой-то смысл. Это дает уверенность в том, что тест действительно проверяет то, что нужно.

Второй этап состоит в написании максимально простого кода для прохождения теста.

На третьем этапе осуществляется рефакторинг кода.

Затем все начинается сначала и вводится еще один тест.

Если бы тест проводился с помощью инструмента NUnit, то результатом его выполнения на первом этапе стала бы светло-красная полоса, а на втором этапе — светло-зеленая полоса.

Вкратце последовательность действий при РПТ можно сформулировать как заклинание: красная полоса, зеленая полоса, рефакторинг кода; красная полоса, зеленая полоса, рефакторинг кода...

Пора приступать к демонстрации

Здесь предполагается, что вы знакомы с Testdriven.net [Testdriven.net], NUnit [NUnit] или рядом других аналогичных инструментов (в противном случае вам придется выбрать подходящий инструмент и освоить его самостоятельно). Поэтому вместо описания самих инструментов обратимся непосредственно к применению принципа РПТ.

Я, конечно, знаю, что на тему применения РПТ написано немало хорошей и полезной литературы, в том числе [Beck TDD], [Astels TDD], [Martin PPP], но мне бы хотелось продемонстрировать его по-своему. Для этой цели я воспользуюсь не доморощенным, а реальным примером из своего приложения.

Несколько лет назад компания Dynapac обратилась ко мне с предложением написать приложение, которое предполагалось включить в комплектацию новой серии планировщиков, выпускавшихся этой компанией. Планировщики применяются для удаления старого асфальта перед покрытием нового. Прокатка асфальта служит для восстановления поверхности его покрытия с заданным профилем. При этом удаляются выбоины, выравниваются ямки и прочие неровности поверхности, которая становится ровной и текстурированной.

Данное приложение предполагалось использовать для расчета самых разных параметров, помогавших оптимизировать применение планировщиков. В частности, приложение должно было допускать расчет твердости старого асфальта, времени, необходимого для планировки определенной проекции, количества грузовиков, необходимых для выполнения прокатки при минимальных затратах средств, и многое другое.

Это реальное приложение я и собираюсь взять в качестве примера для демонстрации РПТ, чтобы дать представление о порядке его выполнения. При этом основное внимание будет уделено расчету количества грузовиков.

Для перевозки старого асфальта должно быть выбрано правильное количество грузовиков. Если их окажется слишком мало, планировщик вынужден будет всякий раз останавливаться или двигаться медленнее, а слишком большое их количество приведет к простоям как самих грузовиков, так и рабочей силы. В обоих случаях возрастают затраты на выполняемые работы.

Итак, приступим к решению данной задачи методом РПТ. Сначала составим следующий список всех необходимых функций в виде текстового файла:

```
Рассчитать вальцуемость
Рассчитать объем прокатки
Рассчитать необходимое количество грузовиков
```

Как упоминалось выше, основное внимание в данном примере уделяется расчету количества грузовиков, поэтому отметим третью строку в данном текстовом файле, чтобы знать, с чего следует начать. Как только нам (или заказчику) потребуется новая функция, ее можно будет ввести в этот файл. Таким образом, текстовый файл дает возможность все учесть и в то же время сосредоточиться на каждой решаемой задаче по очереди.

На заметку

Тесты и реорганизованный код можно записывать в одном и том же файле, но я лично предпочитаю вместо этого писать тесты, помечая атрибутом `Ingnore` (Игнорировать) те тесты, на которых не сосредоточено теперь мое внимание. Рефакторинг кода, потребности в котором возникают в ходе работы, но в данный момент неактуальны, можно отложить до более удобного момента. Когда же потребности в ней возникнут вновь, тогда и следует уделить ей внимание.

Далее следует подумать о тестах для расчета количества грузовиков. Поначалу мне казалось, что сделать это будет нетрудно, но чем больше я об этой задаче думал, тем сложнее мне представлялось ее выполнение.

Итак, начнем с простого теста, который должен проверять следующее: если нет никакой входной информации, то грузовики вообще не требуются. Прежде всего, создадим новый проект под названием `Tests`, введем класс `TruckCalculationTests` (Тесты расчета количества грузовиков) и подготовим все необходимое для тестовой конфигурации в соответствии с требованиями инструмента NUnit, включая задание ссылки на `nunit.framework`, ввод оператора использования `NUnit.Framework` и обрамление класса атрибутом `[TestFixture]`. Затем напишем первый тест, который должен выглядеть следующим образом:

```
[Test]
public void WillGetZeroAsResultWhenNoInputIsGiven()
{
```

```
  TruckCalculation tc = new TruckCalculation();

  Assert.AreEqual(0, tc.NeedNumberOfTrucks);
}
```

На самом деле при написании этого очень простого теста я принял два мелких проектных решения. Во-первых, я решил, что мне потребуется класс `TruckCalculation`. И во-вторых, я решил, что у этого класса должно быть свойство `NeedNumberOfTrucks` (Необходимое количество грузовиков).

Разумеется, тестовый проект нельзя компилировать, поскольку еще не написан проверяемый им код, поэтому продолжим работу, перейдя к "реальному", так сказать, проекту под названием `Dynapac.PlanPave.DomainModel`. Введем в этом проекте следующий класс:

```
public class TruckCalculation
{
  public int NeedNumberOfTrucks
  {
      get {return -1;}
  }
}
```

В этом фрагменте кода явно сымитирован возврат значения –1. Это сделано намеренно для того, чтобы тест не прошел. Напомним, что начинать всегда следует с неудачно завершающегося теста, а для данного конкретного теста возврат нулевого значения не означал бы его непрохождение. Неудачное завершение теста должно иметь какой-то смысл, хотя начать с простого теста и придумать для него настоящий смыл невозможно (ни теперь, ни при выполнении реального проекта), поэтому будет достаточно и этого.

Затем вернемся к проекту `Tests` и установим ссылку на проект `Dynapac.PlanPave.DomainModel`. Кроме того, введем оператор использования и скомпонуем все решение в целом. После этого выполним тесты (в настоящий момент это лишь один тест) в проекте `Tests`. Будем надеяться, что тест не пройдет. После этого вернемся к коду свойства и изменим его таким образом, чтобы оно возвращало нулевое значение, еще раз выполним тест. Теперь он должен пройти.

Требуется ли рефакторинг кода? Вряд ли, поскольку этот код достаточно прост и вполне удовлетворяет условиям теста.

Таким образом, мы сделали небольшой шаг в правильном направлении, начав работу над новой функцией для расчета количества грузовиков, и создали простой, небольшой, но удачный тест.

А теперь сделаем еще один небольшой шаг, опять-таки предоставив тестам направлять ход нашей работы. Для этого нам потребуется еще один тест. В проверяемой области требуется показать как округленный, так и более точный результат — с точностью хотя бы до одной десятичной цифры. Округленный результат должен всегда округляться до целого числа, поскольку использовать половину грузовика

трудно (здесь не идет речь о грузовиках разной грузоподъемности). Это необходимый и подобающий тест, но не будем на этом останавливаться, а сделаем еще один, более интересный шаг, чтобы ввести тест на округление с атрибутом `Ignore` следующим образом:

```
[Test, Ignore("Заняться этим тестом позднее")]
public void CannotRoundDecimalTruckDown()
{
}
```

Итак, перейдем непосредственно к расчету. Сделать такой расчет нетрудно. В нем необходимо учесть расстояние, скорость перевозки, грузоподъемность грузовика, объем прокатки, время погрузки и разгрузки, а возможно, и ряд других факторов. Ради простоты не будем пока что принимать во внимание факторы перевозки и время погрузки, не говоря уже о пока еще неизвестных нам факторах. В данный момент нас, прежде всего, интересует объем прокатки, грузоподъемность грузовика и время разгрузки. Теперь мы можем написать следующий тест:

```
[Test]
public void CanCalculateWhenOnlyCapaciltyIsDealWith()
{
  TruckCalculation tc = new TruckCalculation();

  tc.MillingCapacity = 20;
  tc.TruckCapacity = 5;
  tc.UnloadaingTime = 30;

  Assert.AreEqual(2, tc.NeedNumberOfTrucks)
```

В этом фрагменте кода указано следующее: объем прокатки составляет 20 тонн/час, каждый грузовик может перевезти за один раз 5 тонн, что означает 4 разгрузки. Кроме того, для одной разгрузки требуется 30 минут, поэтому каждый грузовик может быть использован лишь 2 раза в час. Таким образом, нам потребуется 2 грузовика, что и следует проверить.

На заметку

Проведенный до сих пор расчет может показаться не совсем корректным, поскольку в нем учитывается слишком мало факторов. Но ведь это только начало. В данный момент намного важнее не сам расчет, а процесс продвижения с помощью РПТ, поэтому обращайте основное внимание именно на это, а не на расчет.

Если попытаться скомпилировать проект `Test`, то из этого ничего не получится, поскольку в новом тесте предполагается проверить три новых свойства. Введем эти свойства в класс `TruckCalculation` следующим образом:

```
// Класс TruckCalculation
public int TruckCapacity = 0;
public int MillingCapacity = 0;
public int UnloadingTime = 0
```

Это, пожалуй, не свойства, а общедоступные поля. Впрочем, данный вопрос мы обсудим далее, а до тех пор скомпонуем решение и, возможно, получим положительный результат выполнения теста (красную строку).

Действительно, получили то, что и ожидали. Теперь нужно внести изменения в свойство `NeedNumberOfTrucks`, чтобы сделать расчет более реальным.

```
// Класс TruckCalculation
  public int NeedNumberOfTrucks
  {
    get
    {
      return MillingCapacity /
      (TruckCapacity * 60 / UnloadingTime);
    }
}
```

Возможно, это слишком большой шаг вперед, но зато он уверенный.

Вновь скомпонуем решение и еще раз выполним тесты. Один тест проходит (зеленая полоса), а другой — нет (красная полоса). В чем же дело? Оказывается, неудачно завершился не последний, а старый тест. Дело в том, что первый тест не дает никаких значений для свойств `TruckCapacity` и `UnloadingTime`, поэтому возникает исключительная ситуация, обусловленная делением на нуль. И хотя это глупая ошибка, мы можем быть довольны результатом тестирования. Ведь даже столь мелкий и простой тест позволил нам обнаружить программную ошибку сразу же после того, как она была введена в код. Сработала довольно жесткая, но эффективная обратная связь.

Теперь нужно внести некоторые поправки в расчет следующим образом:

```
// Класс TruckCalculation
  public int NeedNumberOfTrucks
  {
    get
    {
      if (TruckCapacity != 0 && UnloadingTime != 0)
          return MillingCapacity /
          (TruckCapacity * 60 / UnloadingTime);
      else
          return 0;
    }
  }
```

Итак, скомпонуем решение, выполним тесты и получим отрицательный результат (зеленая полоса). Прекрасно — это еще один, надежный шаг (благодаря тестам) в правильном направлении.

А требуется ли рефакторинг кода? Уверен, что многим из вас пришлось не по нраву применение общедоступных полей в данном примере. Раньше они мне были тоже не по душе, но с тех пор я изменил свое мнение. Ведь общедоступные поля ничем не хуже свойств, поскольку допускают чтение и запись значений без особого вмешательства. С одной стороны, они проще свойств, но с другой — не допускают вмешательства, если таковое требуется при задании одного из значений. Рефакторингом кода мы займемся, когда в этом действительно возникнет потребность, но не раньше.

На заметку

Что касается рефлексии, то между общедоступными полями и общедоступными свойствами имеются отличия, что может привести к осложнениям, если будет решено заменить одни другими.

Один из рецензентов книги обратил внимание на еще одно отличие, о котором я и не подозревал: общедоступные свойства не могут быть использованы в качестве аргументов `ref`, в отличие от общедоступных полей.

Конечно, все зависит от используемого языка программирования. Так, в C# и VB.NET общедоступное поле и общедоступное свойство используются потребителем одинаково, но совсем другое дело, например, C++ и Java.

Если уж говорить о рефакторинге кода, то можно было бы ввести в область `[SetUp]` тестового класса метод, создающий экземпляр члена расчета на уровне экземпляра. И это верное замечание. Нельзя забывать о тестах, размышляя о рефакторинге кода. Но пока что вносить подобные изменения еще рано. Конечно, они позволят немного сократить дублирование кода, но в то же время сделают тесты менее ясными. А ясность имеет для тестов зачастую наибольшее значение.

И еще одно неприятное обстоятельство: некоторая беспорядочность кода для расчета. На мой взгляд, было бы лучше перенести проверку маловероятной ситуации появления нулевых значений в оператор защиты. Такое изменение не имеет большого значения и является, скорее, делом личных предпочтений, но благодаря ему код становится чуть более удобным для чтения. Так или иначе, выполним рефакторинг кода под названием Replace Conditional with Guard Clause (Замена вложенного условного выражения оператором защиты) [Fowler R]:

```
// Класс TruckCalculation
  public int NeedNumberOfTrucks
  {
    get
    {
      if (TruckCapacity = 0 || UnloadingTime = 0)
          return 0;

          return MillingCapacity /
```

```
            (TruckCapacity * 60 / UnloadingTime);
    }
  }
```

На заметку

Как и имена шаблонов, названия видов рефакторинга кода могут быть использованы как средства общения среди разработчиков.

Итак, скомпонуем тесты и выполним их. Они по-прежнему дают отрицательные результаты (зеленую полосу), а код становится более понятным, хотя его можно улучшить. Для этой цели вполне подойдет рефакторинг кода типа Consolidate Conditional Expression (Объединение условного выражения) [Fowler R]:

```
// Класс TruckCalculation
  public int NeedNumberOfTrucks
  {
    get
    {
      if (_IsNotCalculatable()
          return 0;

          return MillingCapacity /
          (TruckCapacity * 60 / UnloadingTime);
    }
  }
```

И раз уж мы коснулись этого фрагмента кода, то следует обратить внимание на не совсем четкий вид формулы расчета. Ее назначение станет более понятным, если изменить одну ее часть на вызов свойства, используя рефакторинг кода типа Extract Method (Метод извлечения) [Fowler R]:

```
// Класс TruckCalculation
  public int NeedNumberOfTrucks
  {
    get
    {
      if (_IsNotCalculatable()
          return 0;

          return MillingCapacity /
          _SingleTruckCapacityDuringOneHour;
    }
  }
```

На этом можно пока что остановиться, а далее ввести еще один тест. Однако этого небольшого описания процесса создания нового класса для расчета необходимого

количества грузовиков вполне достаточно — оно может дать ясное представление о порядке выполнения РПТ.

Возможно, вас удивит отсутствие при написании тестов помощи из системы интеллектуального управления, которая позволяет сократить объем вводимого вручную кода, "угадывая" его заранее. Мне также нравится такая система, но в данном случае можно вполне обойтись и без нее. Напомню, что здесь речь идет об интерфейсе, поэтому ради дополнительного контроля его можно написать и дважды.

В зависимости от того, насколько вы уверены в коде, который пишете в данный момент, вам нужно варьировать темп своей работы. Так, если вам кажется, что вы слишком доверяете коду и очень торопитесь, только усугубляя положение, сбавьте темп и пишите более мелкие тесты и фрагменты, чтобы вернуться к нормальному ритму продвижения вперед.

Следствия для проектирования

В ходе демонстрации особенностей РПТ можно было заметить, что написание тестов во многом напоминает проектирование. Ниже перечислены некоторые следствия, вытекающие из проектирования с помощью РПТ вместо детального проектирования наперед.

- **Больше контроля со стороны клиента**

  Раньше я считал, что конфигурацию следует как можно более тщательно скрывать внутри, чтобы классы настраивали сами себя. Но для применения РПТ тестам должна быть предоставлена возможность устанавливать экземпляры так, как это им нужно. На самом деле это не так уж и плохо, поскольку повторное использование классов намного упрощается. (Разумеется, нужно быть внимательным, чтобы не создавать условия для появления ошибок.)

- **Больше интерфейсов и фабрик**

  Для того чтобы стало возможным (или, по крайней мере, более простым) применение заглушек и имитирующих объектов, функциональные возможности следует объединять посредством интерфейсов. Это позволяет передавать тестовые, а не настоящие объекты, что может быть трудно организовать в среде тестирования. Если вам приходилось работать в среде COM, значит, вам хорошо известно, что у работы с интерфейсами имеется немало преимуществ.

> ***На заметку***
>
> Более подробно о заглушках и имитациях речь пойдет далее в этой главе, а пока достаточно сказать, что заглушки являются своего рода "подстановками", чуть более сложными, чем открытые интерфейсы, и предоставляющими готовые значения, созданные ради разработки и проверки потребителей заглушек. Имитации представляют собой еще одну разновидность "подстановок", предназначенных для проверки и устанавливаемых с некоторыми предположениями относительно порядка их вызова, после чего эти предположения проверяются.

- **Больше показываемых подытогов**

  Для того чтобы продвигаться мелкими шагами, нужно показывать подытоги, чтобы затем их проверять. (Если продолжить приведенный выше демонстрационный пример, то можно было бы показать больше подытогов, например, для перевозки и времени погрузки.) А поскольку подытоги предназначены только для чтения, особых осложнений они не вызывают. Ведь все равно их, как правило, приходится показывать в пользовательском интерфейсе. Нужно лишь быть внимательным, чтобы не раскрыть больше алгоритмов, чем требуется, аккуратно соблюдая меру в раскрытии и сокрытии информации.

- **Больше конкретности**

  Если начать приведенный выше демонстрационный пример с детального проектирования наперед, то можно с уверенностью сказать, что был бы создан класс грузовиков со множеством их свойств. Но поскольку основное внимание было уделено тому, что именно требуется для выполнения тестов, этот класс был благополучно опущен. Ведь для описания грузовиков требовалось только свойство грузоподъемности, которое и было задано и сохранено в классе расчета.

- **Меньше связывания**

  РПТ приводит к меньшему связыванию. Так, проверить пользовательский интерфейс с помощью автоматических тестов очень трудно, поэтому пользовательский интерфейс не должен в той или иной степени переплетаться с моделью предметной области. Кроме того, связывание сокращается в значительной степени благодаря тому, что больше используются интерфейсы.

Помимо следствий проектирования, применение РПТ имеет и другие положительные следствия, перечисленные ниже.

- **Вторая версия API**

  Когда потребитель применяет прикладной пользовательский интерфейс (API), то оказывается, что он уже один раз использовался в ходе РПТ. Возможно, теперь потребитель посчитает API более устойчивым и совершенным, чем это было бы в противном случае.

- **Сопровождение в перспективе**

  При обширном тестировании сопровождение кажется несбыточной мечтой. Когда потребуются изменения, тесты сразу же покажут, насколько это скажется на потребителях.

  Если требуются изменения, а обширных тестовых наборов нет, то можно создать собственные тесты перед тем, как вносить изменения. Конечно, это не очень интересное занятие и, к тому же, качественные тесты, возможно, не удастся создать спустя длительный промежуток времени после написания кода, но зачастую такой подход оказывается лучше, чем внесение изменений в надежде, что все обойдется. Когда же возникнет потребность в очередных изменениях, тесты уж будут под рукой.

Несмотря на применение РПТ, в рабочем коде все равно появляются программные ошибки. Когда это происходит, то процесс исправления ошибки оказывается аналогичным обычной процедуре РПТ: сначала пишется тест, выявляющий программную ошибку, а затем код, который приводит к успешному выполнению (новых или старых) тестов. И наконец, код, если необходимо, реорганизуется.

- **Документация**

  Создаваемые тесты уже сами по себе являются достаточно качественной документацией. Если примеры помогают лучше понять то или иное свойство разрабатываемого приложения, то тесты служат примерами того, что в нем должно и что не должно работать.

  С другой стороны, блочные тесты обнаруживают предположения одних разработчиков в отношении классов, создаваемых другими. Так, если тесты не проходят, это может произойти из-за того, что класс ведет себя не так, как предполагалось, а такая информация очень ценится во время разработки.

- **Компилятор с более развитой логикой**

  В прошлом я делал немало попыток сделать компилятор своим советником, подсказывающим мне, когда я делаю что-то неверно. Тесты позволяют добиться этого на более высоком уровне. В частности, они отлично подходят для обнаружения того, что упустил компилятор. Иными словами, компилятор прекрасно справляется с обнаружением синтаксических ошибок, а тесты обнаруживают семантические ошибки.

  Более того, типовая безопасность приобретает меньшую ценность при РПТ, поскольку тесты без труда вылавливают ошибки несоответствия типов. А это, в свою очередь, означает, что работу можно организовать в более динамичном стиле.

  Кроме того, при соблюдении принципов РПТ оказывается, что с отладчиком нужно работать намного меньше, чем это пришлось бы делать в противном случае.

На заметку

Иногда все же возникает потребность в проверке значений переменных и прочих аналогичных элементов. С этой целью можно воспользоваться вызовами метода `Console.WriteLine()` или же выполнить тесты в самом отладчике. Когда приходится прибегать к таким приемам, это, на мой взгляд, служит явным признаком того, что разработка зашла слишком далеко, а тестов было проведено слишком мало.

- **Возможность повторного использования в других тестах**

  Тесты, создаваемые во время разработки, могут пригодиться и для составления тестов интеграции. Мой опыт показывает, что многие осложнения во время

эксплуатации вызваны тем, что реальные условия эксплуатации отличаются от тех, что предполагались. Так, если имеется обширный набор тестов, его можно использовать и для проверки развертывания завершенного приложения. Таким образом, из созданных однажды тестов можно извлечь выгоду неоднократно.

Итак, подведем итог: проектирование, предполагающее использование тестов (или тестируемость), является в настоящее время очень важной проектной целью, а возможно, и главным фактором выбора проектных решений.

Недостатки проектирования методом РПТ

Описание РПТ было бы неполным, если бы не были указаны также некоторые недостатки, присущие данному методу.

- **Пользовательский интерфейс**

  Применение РПТ для создания пользовательского интерфейса связано с определенными трудностями. Данный метод лучше и легче сочетается с моделью предметной области и базовой логикой, хотя и это само по себе не так уж плохо. В частности, РПТ помогает отделить пользовательский интерфейс от модели предметной области, что считается одним из самых лучших практических приемов базового проектирования (делать это все равно придется, а РПТ только стимулирует подобные действия). Нужно лишь написать очень "тонкий" пользовательский интерфейс и вынести логику за пределы форм.

- **Данные постоянного хранения**

  Базы данных представляют особую трудность для РПТ, поскольку после записи информации в базу данных в одном тесте она оказывается в другом состоянии, когда дело доходит до другого теста, а это затрудняет точную локализацию ошибок. Конечно, можно написать код для установки базы данных в заранее известное состояние при переходе от одного теста к другому, но это усложняет работу тестирующего и замедляет выполнение тестов.

  С другой стороны, можно было бы написать тесты таким образом, чтобы на них не оказывала влияние смена состояния базы данных, хотя это, скорее всего, привело бы к снижению эффективности тестов. Еще одно решение состоит в использовании заглушек или имитирующих объектов с целью отдельно сымитировать состояние базы данных в ходе тестов интеграции, когда резервная копия базы данных восстанавливается перед выполнением тестов.

  И наконец, можно воспользоваться транзакциями исключительно в целях тестирования. Конечно, если тесты направлены на проверку семантики транзакций, такое решение вряд ли окажется пригодным.

На заметку

Более подробно о тестировании, связанном с базой данных, речь пойдет в главе 6.

- **Ложное чувство безопасности**

  Прохождение теста (зеленая полоса) совсем не означает полное отсутствие программных ошибок — оно лишь означает, что тесты не *обнаружили* никаких ошибок подобного рода. Следовательно, РПТ не должна вызывать у вас ложное чувство безопасности. Помните, что удачное завершение теста не гарантирует отсутствие программных ошибок — оно свидетельствует лишь о том, что при выполнении конкретного теста ошибки не были обнаружены.

  С другой стороны, несовершенство тестов совсем не означает, что их не следует выполнять. Они по-прежнему оказывают неоценимую помощь в повышении качества конечного продукта!

  Как однажды сказал мой коллега Матс Хеландер, ценность теста не в том, что он проходит, а в том, что он не проходит!

- **Потеря общего представления**

  РПТ можно рассматривать как метод восходящего проектирования, и поэтому время от времени не исключается потеря общего представления о проекте. В связи с этим целесообразно периодически представлять код, например, в виде UML-схемы, чтобы получить общую картину. При этом очень часто обнаруживаются резервы для рефакторинга кода, благодаря которой код подготавливается для дальнейших этапов разработки.

- **Больше кода для сопровождения**

  Вопрос сопровождения — обоюдоострый. С одной стороны, тесты отлично подходят для работ, связанных с сопровождением, а с другой, — для сопровождения оказывается больше кода.

Выше уже упоминалось о трудностях использования РПТ в связи с созданием пользовательского интерфейса и взаимодействием с базами данных, и это служит еще одним основанием для применения шаблона модели предметной области, поскольку в этом случае зачастую появляется возможность писать тесты, не касаясь базы данных и пользовательского интерфейса, по сравнению с тем, когда используются другие логические структуры! Более подробно о тестировании баз данных и пользовательского интерфейса речь пойдет в последующих главах этой книги.

Как бы там ни было, РПТ — это замечательный инструмент, который стоит иметь в своем арсенале средств проектирования!

Следующая стадия

Итак, выше были представлены лишь самые общие принципы РПТ. Если вы учтете все перечисленные особенности, достоинства и недостатки РПТ, ваша работа, возможно, пойдет более гладко, когда вы начнете применять РПТ.

Однако в какой-то момент вы, вероятно, обнаружите, что настройка некоторых тестов становится слишком сложной и зависит от тех частей проекта, которые имеют мало отношения к конкретному тесту. Это служит явным признаком того, что вы

достигли следующей стадии РПТ. Подробнее об этой стадии поведает мой коллега Клаудио Перроне в следующем разделе, посвященном заглушкам и имитациям.

Заглушки и имитации

Тестирование баз данных более подробно рассматривается далее в этой книге, тем не менее, пора дать первое представление о нем. В этом разделе Клаудио Перроне покажет, каким образом в тестировании применяются заглушки и имитации. Если в предыдущем разделе было представлено так называемое тестирование по состоянию, то в этом разделе — тестирование по взаимодействию.

При рассмотрении в этом разделе примеров, связанных с базами данных, будет применен классический подход, а о шаблоне предметной области речь пойдет уже в следующей главе.

Итак, передаю слово Клаудио.

Типичный блочный тест

Клаудио Перроне

Типичный подход к тестированию поведения объекта заключается в его настройке на информацию из соответствующего контекста, вызове его методов и записи ряда утверждений для проверки возвращаемого значения или метода, изменившего *состояние* среды предполагаемым образом.

В приведенном ниже примере проверяется метод `SaveUser()` бизнес-компонента `UseBC`, иллюстрируя данный подход:

```
[Test, Rollback] // Автоматический откат
                 // (использование служб без компонентов)
public void TestUserBCCanSaveUser()
{
  // Настройка информации из контекста (входных условий)
  UserInfo user = new UserInfo(
    "Claudio", "Perrone", "MyUniqueLogin", "MyPassword");
  // Это еще не тестирование, а только уточнение
  // исходного состояния
  Assert.IsTrue(user.IsNew);
  Assert.AreEqual(NEW_OBJECT_ID, user.ID);
  Assert.IsFalse(
    IsUserInsertedInDataBase(
    "Claudio", "Perrone", "MyUniqueLogin", "MyPassword");

  // Входные условия заданы — далее следует выполнение
  // проверяемого метода
  UserBC bcUser = new UserBC();
  bcUser.SaveUser(user);
```

```
  // Проверка выходных условий
  Assert.IsFalse(user.IsNew);
  Assert.IsTrue(user.ID != NEW_OBJECT_ID);
  Assert.IsTrue(
    IsUserInsertedInDataBase(
    "Claudio", "Perrone", "MyUniqueLogin", "MyPassword");
}
```

На заметку

Атрибут `Rollback` в текущей версии инструмента NUnit не предоставляется (хотя это и входит в планы его разработчиков). Поэтому я воспользовался модифицированной версией, написанной Роем Оширови (Roy Osherove; `http://weblogs.asp.net/rosherove/archive/2004/07/12/180189.aspx`).

Бизнес-компонент `UseBC` принимает бизнес-объект и сохраняет его в базе данных, поэтому в данном тесте проверяется его поведение, для чего создается соответствующий бизнес-объект, который передается в виде параметра методу `SaveUser()`, а затем проверяется сохраняемость этого бизнес-объекта на информационном складе.

Объявление независимости

Осложнения, которые может вызвать такой способ тестирования, связаны с тем, что объекты очень редко действуют обособленно. Ведь другие объекты, от которых зависит проверяемый нами объект, зачастую выполняют часть всей работы.

Для того чтобы проиллюстрировать этот факт на примере, реализуем класс `UseBC` следующим образом (ради простоты весь код обработки исключений опущен):

```
public class UserBC
{
  public void SaveUser(IUserInfo user)
  {
    user.Validate();
    UserDao daoUser = new UserDao();
    if (user.IsNew)
      daoUser.Insert(user);
    else
      daoUser.Update(user);
  }
}
```

В данном случае класс `UserBC` поручает проверку пользователя бизнес-объекту `IUserInfo`. Если нарушается одно или несколько бизнес-правил, бизнес-объект `IUserInfo` формирует исключение с целым набором ошибок проверки достоверности. Кроме того, класс `UserBC` поручает сохраняемость бизнес-объекта объекту

доступа к данным UserDao, отвечающему за абстрактное представление всех деталей реализации базы данных.

Объект IUserInfo служит в качестве явного параметра метода SaveUser(), и поэтому мы можем утверждать, что в классе UserBC явно объявляется зависимость от интерфейса IUserInfo, реализованного в классе IUserInfo. Но зависимость от объекта UserDao частично скрыта внутри реализации класса UserBC.

Более того, зависимости являются *транзитивными*. Так, если класс UserBC зависит от класса IUserInfo, а в классе IUserInfo требуется установить объекты BusinessRule (Бизнес-правило) для проверки его содержимого, то класс UserBC будет зависеть и от объектов BusinessRule. Программная ошибка в объекте BusinessRule может неожиданно нарушить этот тест и в то же время ряд других тестов.

Сложная логика, которой оперируют в модели предметной области, зачастую реализуется посредством цепочки объектов, передающих часть поведения другим объектам, с которыми они взаимодействуют, до тех пор, пока не будет получен требуемый результат. Следовательно, блочные тесты, предназначенные для проверки правильности поведения объекта со многими зависимостями, могут не пройти, если один из объектов в данной цепочке содержит программную ошибку. В итоге иногда бывает очень трудно выяснить причину ошибки, поскольку тесты на самом деле оказываются тестами мелкомасштабной интеграции, а не исключительно блочными.

Работа с трудными объектами

Со взаимодействующими объектами связана еще одна проблема, весьма распространенная в данной области распределенных приложений: как проверить поведение объекта, если метод, который мы пытаемся выполнить, зависит от трудно воссоздаваемых компонентов или условий? Например, могут быть компоненты, которые:

- еще не реализованы;
- трудно устанавливаются или проверяются (например, компоненты пользовательского интерфейса или каналы сообщений);
- работают слишком медленно (в частности, компоненты уровня доступа к данным или распределенные компоненты);
- имеют поведение, которое трудно воспроизвести (это, например, относится к вопросам периодически прерывающейся связности узлов сети или параллелизма).

Замена взаимодействующих объектов тестовыми заглушками

В качестве эффективного решения перечисленных выше проблем некоторые или же все взаимодействующие объекты можно заменить тестовыми заглушками. *Тестовая заглушка* представляет собой имитацию настоящего объекта, которая используется в целях тестирования. Она предоставляет механизм задания предполагаемых значений для проверяемого кода.

Вернемся к нашему примеру. В данном случае имеются два взаимодействующих объекта в методе `SaveUser()`: `UseInfo` и `UserDao`. Заменить объект `UseInfo` нетрудно, поскольку в данном тесте достаточно лишь предоставить объект, реализующий интерфейс `UseInfo` и не формирующий исключения при вызове метода `Validate()` (если, конечно, поведение метода `SaveUser()` не требуется проверить при появлении исключения в связи с проверкой достоверности).

Намного труднее заменить объект `UserDao`, поскольку его конструкция встроена в класс `UserBC`. Однако этот взаимодействующий объект обязательно нужно заменить, поскольку его доступ к базе данных замедлит выполнение тестов. Кроме того, для проверки значений в базе данных требуется время, и поэтому ее целесообразно выполнять только при тестировании объекта `UserDao` отдельно или же в ходе теста интеграции. Одним из эффективных решений данной задачи является извлечение интерфейса из объекта `UserDao` и ввод в класс `UserBC` конструктора, явно устанавливающего зависимость от объекта `UserDao`.

Код, который требуется для этой цели, достаточно прост, и поэтому вероятность появления в нем программных ошибок очень мала. В некоторых случаях он может быть написан в дополнение к уже существующим конструкторам только в целях тестирования.

```
public class UserBC
{
  private IUserDao _daoUser;
  // Конструктор, используемый по умолчанию
  public UserBC()
  {
    _daoUser = new UserDao();
  }

  // Конструктор, используемый в тестовом коде
  public UserBC(IUserDao daoUser)
  {
    _daoUser = daoUser;
  }

  public void SaveUser(IUserInfo user)
  {
    user.Validate();

      if (user.IsNew)
        _daoUser.Insert(user);
      else
        _daoUser.Update(user);
  }
}
```

Теперь мы можем без труда создать пару заглушек, реализующих объекты IUserInfo и IUserDao. Их реализация достаточно проста, поэтому ниже приведен код только для заглушки UserDaoStub.

```
public class UserDaoStub : IUserDao
{
  private IUserInfo _userResult = null;
  // Примечание: в тесте нужно задать предполагаемое
  // значение, возвращаемое при вызове метода Insert
  public IUserInfo UserResult
  {
    get { return _userResult;}
    set { _userResult = value;}
  }

  public void Insert(IUserInfo user)
  {
    // Примечание: перед вызовом этого метода в тесте
    // нужно задать предполагаемые значения
    // свойства UserResult
    user.ID = UserResult.ID;
    user.Name = UserResult.Name;
    // и т.д.
  }
    . . .
}
```

Теперь рассмотрим ряд важных моментов.

Во-первых, в первоначальном тесте были указаны некоторые предположения относительно состояния объекта IUserInfo в результате вызова метода SaveUser(). Если теперь видоизменить тест, чтобы использовать в нем объект UserDaoStub, нам придется также указать предполагаемый результат, установив свойство UserResult перед выполнением метода Insert().

Во-вторых, зависимость от базы данных полностью исключена, и теперь первоначальный тест выполняется намного быстрее.

Но поскольку метод SaveUser() поручает бо́льшую часть своего поведения другому методу, такой тест не имеет никакой практической ценности. А если поразмыслить трезво, то оказывается, что мы совершаем смертный (и, к сожалению, довольно частый) грех, проверяя возвращаемые значения с помощью утверждений, ибо на самом деле мы проверяем заглушки, а не класс UserBC!

И совсем другое дело, если нужно узнать, каким образом класс UserBC реагирует на формирование на уровне доступа к данным такого исключения, как DalUniqueConstraintException, т.е. на ситуацию, которая может возникнуть, если в базе данных уже имеются регистрационные данные пользователя.

Теперь метод Insert() выглядит в классе заглушки следующим образом:

```
// Класс UserDaoStub
public void Insert(IUserInfo user)
{
  if (ThrowDalUniqueConstraintExceptionOnInsert)
    throw new DalUniqueConstraintException();

  user.ID = UserResult.ID;
  user.Name = UserResult.Name;
// и т.д.
}
```

Допустим, что в классе `UserBC` требуется перехватывать исключение `DalUniqueConstraintException` и формировать соответствующее бизнес-исключение для сохранения абстракции уровней. Данный вариант иллюстрирует следующий код:

```
[Test] // Примечание: база данных уже не требуется!
[ExpectedException(
      typeof(BusinessException),
      "Предоставленные регистрационные данные уже имеются.")]
public void TestUserBCSaveUserThrowsBusinessException()
{
    // Установка заглушек для исключения всех зависимостей
    IUserInfo user = new UserInfoStub (
      "Claudio", "Perrone", "Login", "MyPassword");

    UserDaoStub daoUser = new UserDaoStub();
    daoUser.ThrowDalUniqueConstraintExceptionOnInsert = true;

    // Выполнение теста – он должен формировать бизнес-исключение
    UserBC bcUser = new UserBC(daoUser);
    bcUser.SaveUser(user);
}
```

Как и следовало ожидать, такой тест выполняется довольно быстро, не требует доступа к базе данных или другим взаимодействующим объектам и позволяет нам оперативно проверить функции тестируемого класса в условиях, которые не так-то просто воссоздать. Теперь имитация других условий превращается в очень простое упражнение.

Это приводит нас к важному выводу: применение тестовых заглушек позволяет изолировать проверяемый код и наблюдать за его реакцией на внешние условия, имитируемые поддельными взаимодействующими объектами.

Замена взаимодействующих объектов имитирующими объектами

Имитирующие объекты действуют по совсем иному принципу, чем заглушки. Они имитируют настоящие объекты, заменяя взаимодействующие объекты и представляя предполагаемые значения проверяемому коду. Кроме того, имитирующие объекты предоставляют механизм указания предположений относительно того, как их следует использовать, а также могут выполнять самопроверку на основании этих предположений.

Для того чтобы показать, что все это означает, создадим тест, проверяющий взаимодействие класса `UserBC` с объектами, которые с ним взаимосвязаны. На этот раз воспользуемся распространенной на платформе .NET средой для создания имитирующих объектов под названием NMock [NMock]. К достоинствам этой открытой среды следует отнести применение рефлексии для динамического формирования имитирующих объектов из существующих классов или интерфейсов во время выполнения.

```
[Test]
public void TestUserBCSaveUserInteractsWell()
{
  // (1 - Настройка) Создать имитации динамически на основании
интерфейса
  DynamicMock mockUser =
    new NMock.DynamicMock(typeof(IUserInfo));
  DynamicMock mockUserDao =
  new DynamicMock(typeof(IUserDao));

  // Задать готовые значения (как и в заглушках)
  mockUser.SetupResult("FirstName", "Claudio",
    typeof(string));
  . . .
  mockUser.SetupResult("ID", NEW_OBJECT_ID, typeof(int));
  mockUser.SetupResult("IsNew", true, typeof(bool));

  // Сформировать экземпляры имитирующих объектов
  // (требуется приведение типов)
  IUserInfo user = (IUserInfo) mockUser.MockInstance;
  IUserDao daoUser = (IUserDao) mockUserDao.MockInstance;

  // (2 - Предположения) Каким образом класс UserBC
  // должен обращаться с имитациями
  mockUser.Expect("Validate");
  mockUserDao.Expect("Insert", user);
  mockUserDao.ExpectNoCall("Update", typeof(IUserInfo));

// (3 - Выполнение) Выполнить проверяемый метод
  UserBC bcUser = new UserBC(daoUser);
  // Здесь будут появляться сведения о неудачном
  // завершении непредполагаемых вызовов методов
  // в имитациях (например, о вызове метода Validate() дважды)
  bcUser.SaveUser(user);

  // (4 - Проверка) Проверить, насколько осуществились
  // все предположения
  mockUser.Verify();
  mockUserDao.Verify(); // Примечание: утверждения не требуются
}
```

На заметку

В NMock создается класс, реализующий интерфейс, или же формируется подкласс настоящего класса. В обоих случаях для замены настоящего класса экземпляром объекта сформированного класса применяется полиморфизм. Несмотря на всю эффективность данного подхода, у него все же имеются некоторые существенные ограничения. Например, имитации герметичных классов не допускаются, а имитируемые методы должны быть обозначены атрибутом `virtual`.

Интересной альтернативой NMock является коммерчески доступная среда POCKMock [POCKMock] компании Pretty Objects. В этом случае имитации создаются путем статической замены взаимодействующий объектов.

Как следует из приведенного выше фрагмента кода, в данном тесте утверждения не требуются, поскольку они находятся внутри имитирующих объектов и гарантируют, что имитации будут вызываться так, как и предполагалось в проверяемом коде. Так, при вызове метода `Validate()` дважды сразу же формируется исключение, причем еще до вызова метода `Verify()`.

Данный пример приводит нас к следующему выводу: имитирующие объекты сами проверяют, насколько правильно они используются, и поэтому они позволяют составить более точное представление о том, как проверяемый объект *взаимодействует* с остальными объектами, которые с ним взаимосвязаны.

Последствия для проектирования

Когда я впервые ознакомился с имитирующими объектами, мне показалось, что найти простой механизм замены моих взаимодействующих объектов будет непросто. Основная трудность заключалась в том, что мой код был связан с конкретной реализацией этих взаимодействующих объектов, а не с их интерфейсами. Меня осенило лишь тогда, когда я обнаружил два главных механизма: внесение зависимостей (Dependency Injection) и указатель службы (Service Locator).

В первом механизме (внесении зависимостей) предполагается, что класс явно определяет интерфейсы объектов, с которыми он взаимодействует, например, в конструкторе или в виде параметров метода, но предоставляет контейнеру самому отвечать за их конкретную реализацию. А поскольку класс уже не управляет созданием объектов, с которыми он взаимодействует, такой принцип иногда еще называется *инверсией управления* (Inversion of Control).

Во втором механизме (указателе службы) класс внутренним образом находит конкретные объекты для взаимодействия по зависимости от другого объекта (указателя). Простым примером указателя службы может служить фабричный объект, использующий файл конфигурации для динамической загрузки объектов, которые требуются для взаимодействия.

На заметку

Более подробно о внесении зависимостей, указателей служб и инверсии управления речь пойдет в главе 10.

Выводы

Имитирующие объекты и заглушки позволяют еще больше изолировать код, который требуется проверить. Это одновременно и преимущество и недостаток, поскольку тесты иногда скрывают недостатки интеграции. Чем быстрее тесты выполняются, тем чаще они слишком тесно связаны с тестируемой системой. Следовательно, они быстро устаревают, как только находится более совершенная реализация для тестируемой системы.

Рефакторинг кода, направленный на внедрение имитаций, отделяет объекты от конкретной реализации их зависимостей. И хотя имитации обычно удается создавать из классов, содержащих виртуальные методы, тем не менее, рекомендуется при всякой возможности пользоваться интерфейсами. Уже нередки случаи, когда системы, предназначенные для тестирования с использованием имитирующих объектов, содержат значительное число интерфейсов, внедренных только в целях тестирования.

Дополнительные сведения

Более подробно отличия между заглушками и имитациями (т.е. имитирующими объектами, а также тестированием по состоянию и взаимодействию) обсуждаются в статье *Mocks Aren't Stubs* [Fowler Mocks Aren't Stubs] Мартина Фаулера.

Благодарю Клаудио за еще один инструмент, которым читатели могут пополнить свой арсенал средств разработки! А теперь перейдем к вопросам рефакторинга кода.

Рефакторинг кода

В этой главе рефакторинг кода уже упоминался в качестве третьего этапа общего процесса РПТ, но этот термин все же требует дополнительных пояснений. Рефакторинг кода означает поэтапное внесение мелких, заранее известных изменений с целью улучшить конструкцию кода, т.е. сделать более удобным его сопровождение, не изменяя наблюдаемое поведение. Иными словами, это *качественное*, а не *принципиальное* изменение.

Термин *рефакторинг кода*, или просто *рефакторинг* (от англ. refactoring), стал в последнее время весьма популярным, хотя им зачастую злоупотребляют и пользуются не к месту. Так, вместо того чтобы сказать: мы собираемся внести множество радикальных изменений в наш код, говорят: реорганизуем его, а еще хуже — сделаем его рефакторинг. Но это совсем не отражает саму суть данного термина.

В главе 1 было немало сказано о сущности и назначении рефакторинга кода. А в этом разделе речь пойдет, главным образом, о том, как пользоваться этим методом. Для начала мы продемонстрируем пример простого рефакторинга кода в качестве регулярной его чистки, а затем коснемся вкратце повышения производительности труда в связи с применением данного метода. И наконец, рассмотрим рефакторинг кода как в направлении шаблонов, так и обратно, чтобы исключить осложнения при сопровождении кода.

Чистка недоброкачественного кода

В демонстрационном примере РПТ, приведенном в начале этой главы, уже рассматривались некоторые виды рефакторинга кода. А теперь рассмотрим другой ее пример. Итак, начнем со следующего кода класса `Person`:

```
public class Person
{
  public string Name = string.Empty;
  public DateTime BirthDate = DateTime.MinValue;
  public IList Children = new ArrayList();

  public int HowOld()
{
    if (BirthDate != DateTime.MinValue &&
      BirthDate < DateTime.Now)
    {
      int years = DateTime.Now.Year -
        BirthDate.Year;
      if (DateTime.Now.DayOfYear <
        BirthDate.DayOfYear)
        years--;

      return years;
    }
    else
      return 0;
  }
}
```

Недоброкачественный код

О недоброкачественном коде уже не раз упоминалось в этой книге. Именно такой код будет служить главным объектом рефакторинга на последующих страницах. В книге *Refactoring* [Fowler R] Мартин Фаулер и Кент Бек приводят целый перечень видов недоброкачественного кода, иногда еще называемого гнилым.

Типичным и первым примером такого кода служит дублированный код.

Регулярная чистка кода

В приведенном выше фрагменте кода `Person` представляет собой небольшой класс, предназначенный для решения конкретной задачи. Исходя из характера этого класса, можно сделать вывод, что он служит для отслеживания имен, дат рождения, возраста отдельных людей и наличия у них детей. Допустим, что требования к этому классу выполняются, но за счет некоторых технических излишеств, поскольку код выглядит не совсем упорядоченным. Именно для этого и требуется его рефакторинг. Очистим этот код методом, который я назвал "регулярной чисткой кода", в ряде примеров, основанных на упомянутом выше труде Фаулера [Fowler R].

Первый пример регулярной чистки кода: инкапсуляция поля

Основной целью в этом примере служит поле `Name`. В данный момент оно имеет общедоступный строковый тип, а этого может оказаться слишком много. Преобразуем данное поле из исходной формы

```
public string Name = string.Empty;
```

в следующую форму:

```
// Класс Person
private string _name = string.Empty;
public string Name
{
  get {return _name;}
}

public Person(string name)
{
_name = name;
}
```

Как упоминалось выше, требования к данному классу выполняются. Они, в частности, состоят в указании имени в поле `Name` и его чтении. Но эти требования можно выполнить еще лучше, если учесть свойство "только для чтения" поля `Name`.

А возможны ли другие улучшения или на этом следует остановиться? На мой взгляд, можно было бы использовать общедоступное поле типа `readonly` следующим образом:

```
// Класс Person
public readonly string Name;
public Person(string name)
{
  Name = name;
}
```

Такой способ чистки делает код более ясным, но у него, к сожалению, имеются недостатки. Один из них состоит в том, что указать имя в поле `Name` посредством

рефлексии невозможно, а ведь это очень важно для многих инструментов, включая и объектно-реляционные преобразователи. В данном случае требуется способ для указания имени в поле Name посредством рефлексии, поэтому лучше выбрать вариант с использованием метода выборки значения поля.

А будет ли при этом выполняться весь код потребителя? Ведь наблюдаемое поведение кода немного изменится, поскольку значение поля Name должно выбираться посредством конструктора. Конечно, ошибки компиляции появятся, но их нетрудно устранить. Однако следует ли на этом остановиться?

Пока еще нет. Рассмотрим следующий фрагмент кода:

```
// Некий потребитель
public void SetNameByReflection()
{
  Person p = new Person();

  FieldInfo f = typeof(Person).GetField("Name",
    BindingFlags.Instance | BindingFlags.Public);

  f.SetValue(p, "Jimmy");

  ...
```

В этом фрагменте кода значение в поле Name задается посредством рефлексии (в данный момент причина этого не так важна). Такой код уже неработоспособен, хотя компилятор не обнаружит ошибки. Остается надеяться только на написанные заранее автоматические тесты, которые смогут обнаружить подобную ошибку.

Из этого следует важный вывод: для рефакторинга кода требуются автоматические тесты! На самом деле рефакторинг кода и РПТ представляют собой некий симбиоз, применяемый как своего рода заклинание, упоминавшееся в начале этой главы: красная полоса, зеленая полоса, рефакторинг кода; красная полоса, зеленая полоса, рефакторинг кода...

С одной стороны, для рефакторинга кода требуется не столько РПТ, сколько автоматические тесты. Но с другой стороны, РПТ в значительной мере способствует написанию таких тестов.

Второй пример регулярной чистки кода: инкапсуляция совокупности объектов

Следующий недостаток рассматриваемого кода, который требуется устранить, кроется в списке потомков, который первоначально выглядел следующим образом:

```
public IList Children = new ArrayList();
```

Такое решение означает, что потребитель может добавлять и удалять потомков из списка без ведома родителя. (Откровенно говоря, никакого нарушения требований в этом нет, но мне это все же не нравится.) Кроме того, вместо экземпляров объектов класса Person можно ввести строковые (string) и целые (int) переменные. Потребитель может даже поменять весь список.

В качестве стандартного решения можно создать типизированный класс списка потомков `Children` с методом `Add()`, воспринимающим только экземпляры объектов класса `Person`. Благодаря этому устраняется недостаток типовой безопасности, а родителю становится известно о добавлении и удалении из списка его потомков, хотя это и не позволяет менять весь список. Для этого придется реорганизовать код самого списка, используя инкапсуляцию поля (Encapsulate Field).

Недостаток такого решения состоит в том, что для получения типизированного списка придется написать фиктивный код. Конечно, это можно сделать, но я лично предпочитаю другой стиль программирования. Если целевая платформа поддерживает обобщения (например, .NET 2.0), то устранить недостаток типовой безопасности можно весьма изящным способом.

Независимо от поддержки обобщений, я лично предпочитаю рефакторинг кода в стиле инкапсуляции совокупности объектов (Encapsulate Collection). Сначала код реорганизуется с помощью инкапсуляции поля и приобретает следующий вид:

```
// Класс Person
private IList _children = new ArrayList();
public IList Children
{
 get {return _children;}
}
```

По крайней мере, потребителю уже не удастся поменять весь список, но он все же сможет добавить и удалить из списка любые элементы. Поэтому следующим шагом должна стать передача потребителю свернутого списка в таком виде:

```
get {return ArrayList.ReadOnly(_children);}
```

И наконец, потребителю нужно предоставить способ добавлять элементы в список, поэтому введем метод `AddChild()` в класс `Person` следующим образом:

```
// Класс Person
public void AddChild(Person child)
{
  _children.Add(child);
}
```

Теперь типовая безопасность обеспечена, а родителю известно, кода вводятся новые экземпляры потомков, и поэтому он может прервать, если требуется, такое действие (хотя в данном примере этого не происходит).

К сожалению, такой рефакторинг кода изменяет видимое поведение класса `Person`, поэтому прежде чем оставлять без изменения ее результаты и двигаться дальше, следует тщательно взвесить все аргументы за и против. Для этого можно воспользоваться заранее подготовленными автоматическими тестами, выполняющими роль второго уровня компилятора и обнаруживающими ошибки. Один из таких полезных тестов выглядит следующим образом:

```
[Test]
public void CanCalculateNumberOfKids()
{
  Person p = new Person("Stig");
  p.Children.Add(new Person("Ulla"));
  p.Children.Add(new Person("Inga"));
  Assert.AreEqual(2, p.Children.Count);
}
```

На заметку

Как следует из приведенного выше кода, данный тест на самом деле не указывает на изменения, которые требуются в настоящем коде потребителя. Это лишь некий тестовый код класса `Person`. Поэтому требуются тесты потребителя, указывающие на изменения, которые следует внести в сам код потребителя.

Строки `p.Children.Add()` нетрудно изменить, чтобы в итоге получилось следующее:

```
[Test]
public void CanCalculateNumberOfKids()
{
  Person p = new Person("StigЭ);

  p.AddChild(new Person("Ulla"));
  p.AddChild(new Person("Inga"));
  Assert.AreEqual(2, p.Children.Count);
}
```

Наличие у объекта `IList` метода `Add()` кажется не совсем удачным, поскольку он свернут в список "только для чтения", но этот недостаток без труда выявляется с помощью тестов. Для его устранения достаточно заменить тип `Children` объекта списка потомков на `ICollection`.

Какие еще изменения можно сделать? Нужно что-то изменить в методе `HowOld()`, поскольку его код недостаточно ясен.

Третий пример регулярной чистки кода: метод извлечения

Возможно, у вас имеется лучшее решение данной задачи? Однако прежде чем думать о лучшем решении, попробуем упростить код, чтобы сделать его более удобным для чтения и понятным. Рефакторинг кода, собственно говоря, представляет собой попытку лучше понять код. Это полезный побочный эффект: ведь чем понятнее код, тем легче будет его упростить и найти лучшие решения.

Итак, имеем следующий код:

```
// Класс Person
public int HowOld()
```

```
{
  if (BirthDate != DateTime.MinValue &&
      BirthDate < DateTime.Now);
  {
      int years = DateTime.Now.Year - BirthDate.Year;

      if (DateTime.Now.DayOfYear < BirthDate.DayOfYear)
          years--;

      return years;
  }
  else
      return 0
}
```

Прежде всего, обращают на себя внимание две ситуации, в которых возвращается нулевое значение, что не совсем хорошо. В том случае, если код выполняется неверно, я лично предпочитаю формировать исключение.

Далее, реорганизуя код методом извлечения (Extract Method), следует исключить не вполне понятное логическое выражение, чтобы получить имя, ясно указывающее на используемый метод, и назначение данного логического выражения. Для этого перейдем от кода

```
if (BirthDate != DateTime.MinValue &&
      BirthDate < DateTime.Now)
```

к коду

```
if (_CorrectBirthDate())
```

Каким бы ни было логическое выражение, теперь его назначение в методе `HowOld()` стало намного более ясным.

Здесь уместно упомянуть еще один стиль программирования с использованием операторов защиты. Такой стиль подразумевает, прежде всего, исключение необычной, практически невероятной ситуации. Если воспользоваться для этой цели оператором `else`, то код будет выглядеть следующим образом:

```
if (! _CorrectBirthDate())
    throw new ArgumentException("Неверная дата рождения!");
```

Этот код можно улучшить таким образом:

```
if (_IncorrectBirthDate())
    throw new ArgumentException("Неверная дата рождения!");
```

Теперь этот код более понятен и удобен для чтения.

И наконец, осталось внести поправки в сам расчет с учетом необычной ситуации. Этот расчет выглядит следующим образом:

```
int years = DateTime.Now.Year - BirthDate.Year;
if (DateTime.Now.DayOfYear < BirthDate.DayOfYear)
    years--;

return years;
```

И в этом случае код реорганизуется методом извлечения, для чего расчет перемещается в другой частный метод под названием `_NumberOfYears()`. Теперь весь метод `HowOld()` выглядит следующим образом:

```
// Класс Person
public int HowOld()
{
    if (_IncorrectBirthDate())
      throw new ArgumentException("Неверная дата рождения!");

    return _NumberOfYears();
}
```

Конечно, в данном примере кода можно было бы найти и другие недостатки (например, во вновь созданном методе `_NumberOfYears()`) или вынести некоторую логику за пределы самого класса `Person`, но общая идея должна быть вам уже ясна.

Важно подчеркнуть, что тесты следует повторять после каждого этапа рефакторинга кода, чтобы убедиться в отсутствии внесенных при этом ошибок — по крайней мере, по результатам самих тестов.

Все, что было до сих пор сделано с кодом, требует времени и не исключает ошибки. Если придерживаться строгой дисциплины, то ошибки еще можно как-то исключить, но время на это все равно потребуется. Настал момент воспользоваться вспомогательным инструментом.

Инструменты для чистки кода

Для рефакторинга кода имеются самые разные инструменты. В этом подразделе мы рассмотрим инструменты ReSharper [ReSharper] и Refactor! [Refactor!], являющиеся расширениями Visual Studio .NET. Поддержка рефакторинга кода была встроена в интегрированную среду разработки версии Visual Studio .NET, выпущенной в 2005 году.

Такие инструменты позволяют вносить изменения, подобные тем, что мы сделали в приведенных выше примерах, а это способствует повышению производительности труда. Поэтому их значение для разработки трудно переоценить. На самом деле, привыкнув пользоваться инструментом рефакторинга кода, вы уже вряд ли от него откажетесь!

Инструменты рефакторинга кода не только помогают вносить требуемые изменения, но и сообщают, когда и что следует изменить. Таким образом, они освобождают вас в некоторой степени от необходимости постоянно помнить о рефакторинга кода, ненавязчиво и удобно напоминая о ней.

На заметку

Вспомните, что неудачному завершению теста соответствует красная полоса, а удачному — зеленая. А каким цветом обозначается потребность в рефакторинге кода? В инструменте ReSharper для этой цели выбран, в частности, оранжевый цвет, напоминающий о необходимости рефакторинга кода. Поэтому прежде чем начать чистку кода по новому кругу, следует убедиться в том, что цвет полосы изменился с оранжевого на зеленый. Разумеется, это всего лишь помощь — она не избавляет от необходимости постоянно анализировать сам код на предмет его недоброкачественности.

Если разработчик пользуется в своей работе хорошим инструментом рефакторинга кода, скорость продвижения его работы просто поражает, и при этом неудачный код практически отсутствует. Это совершенно другой подход к повышению качества кода!

До сих пор мы рассматривали простые примеры рефакторинга кода. В этом, конечно, нет ничего дурного, но иногда качество кода все равно не устраивает даже после его мелкой чистки. Кроме того, возможны ситуации, когда постоянно нарастает поток новых проблем, требующих решения.

Предупреждение нарастающих проблем

У вас может сложиться впечатление, что применение шаблонов является естественным решением задачи рефакторинга кода. Но дело в том, что шаблоны по традиции применяются при проектировании наперед, которое зачастую подразумевает немалую долю предположений, тогда как рефакторинг кода, как известно, выполняется на более поздней стадии разработки. Так, что же предпочесть: шаблоны или рефакторинг кода?

Прежде чем дать ответ на этот вопрос, приведем краткий пример применения еще одного шаблона. Речь пойдет о шаблоне Template Method (Шаблонный метод) [GoF Design Patterns]. Его принцип заключается в определении общего алгоритма (или шаблонного метода) в базовом классе следующим образом:

```
public abstract class TotalBase
{
  public void TotalAlgorithm()
  {
    _DoSomeStuff();
    VariationPoint();
    _DoSomeMoreStuff();
  }

  protected abstract void VariationPoint();
}
```

Теперь подклассы могут наследовать от класса `TotalBase`, обеспечивая специальную реализацию метода `VariationPoint()`, в частности:

```
public class TotalSub : TotalBase
{
  protected override void VariationPoint()
  {
    Console.WriteLine("Здравствуй, мир!");
  }
}
```

Коду потребителя ничего неизвестно ни о том, как этот алгоритм создается, ни о базовом классе. Ему известно лишь о подклассе. В данном случае код потребителя может выглядеть следующим образом:

```
// Некий потребитель
TotalSub t = new TotalSub();
t.TotalAlgorithm();
```

Следует заметить, что в базовом классе принимается решение, должен ли метод быть обязательным или необязательным. В приведенном выше примере использован обязательный метод, поскольку для метода `VariationPoint()` указан атрибут `abstract`. Если же указать атрибут `virtual`, метод будет необязательным, например, для подклассов, если требуется определить часть общего шаблонного метода.

Допустим, что заранее решено применить шаблон Template Method, но в дальнейшем это приводит к дилемме о возможных или вынужденных вариациях. Решить эту дилемму не так-то просто.

Тем не менее шаблоны очень полезны, и поэтому вместо выбора между ними и рефакторингом кода можно воспользоваться и тем и другим. Именно таким путем, на мой взгляд, и следует идти, т.е. от рефакторинга кода в направлении к шаблонам, и обратно! Об этом же говорит и Джошуа Кериевски в своей книге *Рефакторинг с использованием шаблонов* [Kerievsky R2P].

Рассмотрим применение шаблона Template Method вместе с рефакторингом кода на двух следующих примерах.

Первый пример предупреждения нарастающих проблем: формирование шаблонного метода

Допустим, что мы написали монитор пакетного режима, в котором можно подключать разные программы. Для работы в пакетном режиме программа должна быть настроена на ряд правил, которые, по существу, сообщают монитору (с помощью ряда специальных процедур регистрации), когда программа начинает и завершает свою работу.

Допустим далее, что мы написали пару пакетных программ и решили повторно использовать базовый класс для каждой пакетной программы, чтобы не писать каждый раз код регистрации. Ниже приведен пример реализации метода `Execute()` в подклассе.

```
// Подкласс для базового класса пакетной программы
public void Execute()
{
  base.LogStart();
  _DoThis();
  _DoThat();
  base.LogEnd();
}
```

Что в этом коде недоброкачественного? Как видите, подкласс отвечает за вызов базового класса в определенные моменты выполнения метода `Execute()`. На мой взгляд, это слишком расплывчатые условия для подкласса. Ведь разработчик подкласса должен не забыть ввести вызовы базового класса в надлежащих местах. Кроме того, в базовом методе присутствует немало кода инфраструктуры, т.е. вызовов регистрации (это обстоятельство становится более очевидным в реальном примере, где имеется много точек регистрации, в том числе после введения, первой стадии, второй стадии и т.д.). Вследствие этого вызовы интересующих нас методов, например, `_DoThis()` и `_DoThat()`, становятся не столь очевидными, как следовало бы.

Данный фрагмент кода требует применения шаблона Template Method. Поэтому воспользуемся рефакторингом кода под названием Form Template Method (Формирование шаблонного метода) [Kerievsky R2P].

Основная идея — исключить общедоступный метод `Execute()` из подклассов и переместить ответственность за его вызов в суперкласс. Для этого переместим сначала метод `Execute()` в суперкласс следующим образом:

```
// Базовый класс пакетной программы
public void Execute()
{
  _LogStart();
  DoTheWork();
  _LogEnd();
}
```

Методы `_LogStart()` и `_LogEnd()` теперь оказываются частными, а не защищенными. Для реализации в подклассах метод `DoTheWork()` определяется следующим образом:

```
protected abstract void DoTheWork();
```

И наконец, реализация метода `DoTheWork()` в подклассах выглядит следующим образом:

```
// Подкласс для базового класса пакетной программы
protected override void DoTheWork()
{
  _DoThis();
  _DoThat();
}
```

Это уже совсем другое дело. Вместо вызова суперкласса в подклассе теперь подкласс вызывается в суперклассе. Такое решение оказывается не только эффективным, ясным и простым, но и допускающим выбор точек вариации! В подклассе можно теперь навсегда забыть об инфраструктуре (например, регистрации) и уделить основное внимание более интересной и конкретной задаче: реализации метода `DoTheWork()`. (Напоминаем, что это упрощенный пример. В реальной ситуации, из которой он взят, в подклассах пришлось реализовать несколько методов.)

Еще одна проблема, возникающая в связи с применением шаблона State [GoF Design Patterns], формулируется следующим образом: трудно с определенностью сказать, следует ли воплощать решение сразу. Так, если решение воплощается с самого начала, это зачастую приводит в избыточному проектированию. Вместо этого лучше немного подождать до тех пор, пока не станет ясно, что данное решение действительно удачное. (Шаблон State достаточно подробно рассматривался в главе 2, поэтому вам уже должно быть хорошо известно, как его применять.)

Итак, мы рассмотрели рефакторинг кода в направлении шаблонов. А теперь перейдем к рефакторингу кода из шаблонов, обратившись к конкретному примеру.

Второй пример предупреждения нарастающих проблем: встраивание синглтона

Применение шаблона Singleton (Синглтон) [GoF Design Patterns] стало все чаще считаться признаком неудачного программирования. Дело в том, что в этом случае создаются глобальные экземпляры объектов, которые зачастую сами вызывают осложнения. Кроме того, синглтон, как правило, усложняют написание тестов (а ведь тестируемость кода имеет немалое значение), причем взаимодействие с синглтонами в коде оказывается не вполне очевидным.

Допустим, что имеется следующий фрагмент кода:

```
public void DoSomething()
{
  DoThis();
  DoThat();
  MySingleton.Instance().Execute();
}
```

Рефакторинг кода, называемая Inline Singleton (Встроенный синглтон) [Kerievsky R2P], не вполне пригодна для данного примера, но может быть выполнена как вариант.

Вместо того чтобы предоставить методу `DoSomething()` возможность вызвать глобальную переменную (т.е. сам синглтон), в данном методе можно запросить экземпляр в качестве параметра.

Еще одно типичное решение состоит в том, чтобы выявить в методе `DoSomething()` экземпляр объекта в виде частного члена, внесенного во время конструирования или же с помощью механизма установки перед вызовом метода `DoSomething()`. В этом случае метод `DoSomething()` принимает следующий вид:

```
public void DoSomething()
{
  DoThis();
  DoThat();
  _nowMyPrivateMember.Execute();
}
```

Таким образом, синглтон исключен, а тестируемость кода стала намного лучше!

На заметку

Синглтон можно применять в приведенном выше примере и без ухудшения тестируемости кода, если синглтон вносится потребителем. В этом случае тестовый объект можно использовать в ходе тестирования, а синглтон — во время выполнения.

Какие же при этом возникают проблемы? Наиболее очевидная из них заключается в следующем: если требуется значение, находящееся на самом дне стека вызовов, параметр передается задолго до его использования. С другой стороны, становится вполне очевидно, где именно используется значение, что само по себе может служить признаком недоброкачественного кода, а значит, он, скорее всего, требует рефакторинга. Следовательно, синглтоны могут фактически скрывать недоброкачественный код.

В таком случае проблема слишком большого числа одиночек решается методом Dependency Injection (Внесение зависимостей), более подробно рассматриваемым вместе с одноименным шаблоном в главе 10.

Резюме

Подводя итоги этой главы, хотелось бы еще раз подчеркнуть, что верный путь состоит в рефакторинге кода и РПТ. Оба метода представляют собой некий симбиоз.

Для надежного рефакторинга кода требуются обширные тесты. В противном случае вносятся программные ошибки или же исключаются всякие изменения ради удобства сопровождения кода из-за слишком большого риска внести подобные ошибки. Но в последнем случае код начинает медленно вырождаться.

В то же время для применения РПТ необходимо тщательно защищать код и постоянно чистить его посредством рефакторинга. Кроме того, тесты следует изолировать с помощью заглушек и имитаций. Если же не защитить тестовый код, он также начинает вырождаться.

РПТ и рефакторинг кода — это совершенно иной способ написания кода по сравнению с большинством традиционных методов разработки. Опробовав его однажды на практике, вы радикально измените свое отношение к разработке и уже не свернете с этого пути.

Итак, вооружившись этими методами, мы можем приступать к рассмотрению вопросов архитектуры.

Часть II

Прикладное ППО

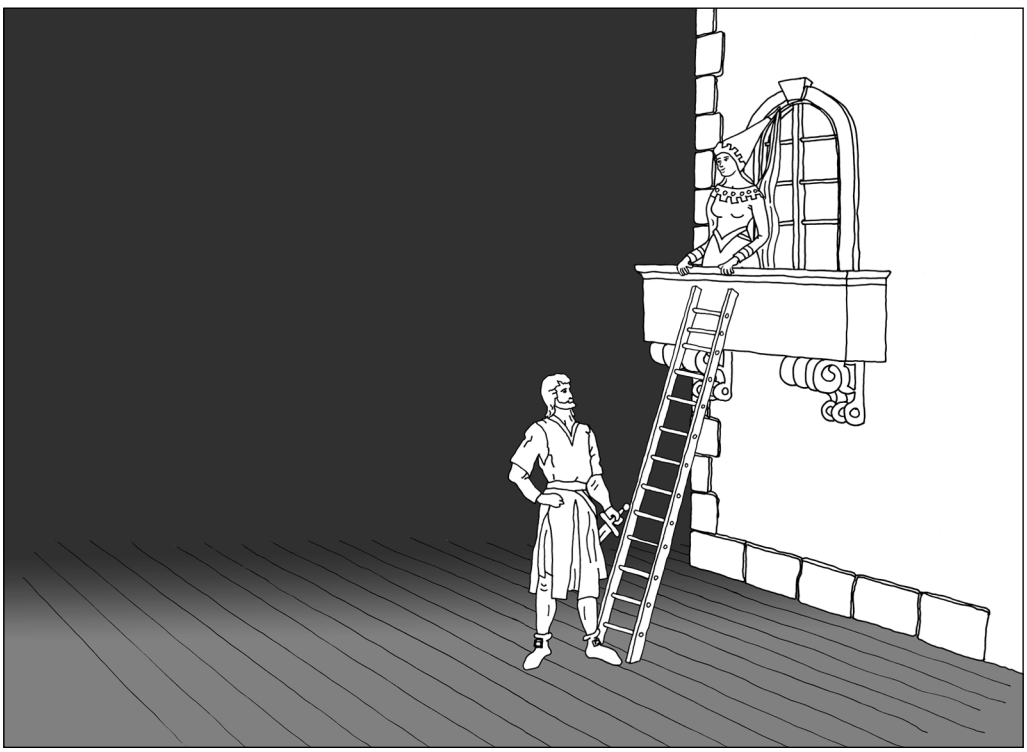
Глава 4. Новая используемая по умолчанию архитектура

Глава 5. Проектирование на основе модели предметной области

Глава 6. Подготовка к инфраструктуре

Глава 7. Порядок устанавливают правила

В этой части речь идет о прикладном ППО, а также о подготовке модели предметной области для инфраструктуры и соответствующих правилах.



Эта часть так же важна, как и подмостки для сцены

Глава 4

Новая используемая по умолчанию архитектура

В прошлом я разрабатывал для своих приложений архитектуры, ориентированные, в основном, на данные. Затем я постепенно изменил свой подход, и теперь используемые мной по умолчанию архитектуры ориентированы на модель предметной области. Отсюда и название этой главы.

Вы можете спросить: "А для кого эта архитектура новая?" Все зависит от обстоятельств. Так, если описать модель предметной области посторонним людям, они спросят: "А это всегда так делается?" В то же время немало разработчиков зададут вопрос: "А это действительно работает?"

Независимо от того, к какой из двух категорий людей вы относитесь, мы пойдем вместе по пути исследования модели предметной области и, в частности, ее построения по принципам *проектирования на основе предметной области* (ППО).

Этот путь мы начнем с краткого изложения моделей предметной области и ППО. На конкретном примере мы рассмотрим перечень требований к приложению и наметим возможные решения, направленные на реализацию разных свойств этого приложения. А для того чтобы получить более ясное представление о требованиях, мы рассмотрим их под новым углом зрения и набросаем черновой вариант исходного пользовательского интерфейса. В конце этой главы мы коснемся стилей исполнения моделей предметной области.

Итак, приступим.

Основа новой используемой по умолчанию архитектуры

Как упоминалось выше, новая используемая по умолчанию архитектура основана на модели предметной области. Эту модель мы уже обсуждали в главе 1, а здесь рассмотрим ее более подробно.

На заметку

Поль Гиленс сделал следующее замечание относительно слова “используемая по умолчанию” в названии этой главы: у читателей может сложиться впечатление, будто такой подход применяется ко всем разрабатываемым системам. Следует, однако, подчеркнуть, что это совсем не так. Данная книга отнюдь не предназначена в качестве образца для подражания. Ведь главный принцип ППО — предоставить прикладной задаче самой определять окончательное решение!

Под “моделью предметной области” в контексте архитектуры приложений обычно подразумевается применение шаблона Domain Model [Fowler PoEAA]. Это сродни попытке, характерной для традиционного направления объектной ориентации, как можно точнее перенести упрощенное (а вернее, абстрактное) представление действительности из предметной области в объектно-ориентированную модель. Получаемый в итоге код очень точно соответствует выбранному уровню абстракции. Это относится к состоянию и поведению, а также к возможным путям перемещения и отношениям между объектами.

На заметку

Кто-то (к сожалению, не помню кто) назвал применение шаблона Domain Model в C# “программированием на C#, как на SmallTalk”. Это было, на мой взгляд, сказано в весьма положительном смысле, хотя для некоторых такой положительный смысл поначалу не совсем очевиден.

Разумеется, модель предметной области — не панацея, но у нее, безусловно, имеется ряд положительных свойств, что особенно справедливо для крупномасштабных и сложных приложений с длительным сроком службы. По мнению Мартина Фаулера [Fowler PoEAA], если попробовать применить модель предметной области в одном проекте, то в дальнейшем, скорее всего, возникнет желание использовать ее даже в небольших, простых приложениях.

Как определить долговечность приложения

Когда речь заходит о приложениях с длительным сроком службы, я припоминаю одно приложение, которое меня попросили создать. Поставщик услуг глобальной связи для передачи данных предложил мне разработать черновой вариант приложения справочной службы для нового бюро в Швеции. Сроки были крайне сжатыми — приложение должно было работать уже на следующий день, чтобы сотрудники не сидели в этом бюро, оснащенном по последнему слову

техники, с листом бумаги и ручкой в руках, когда пресса пожалует к открытию. Это приложение все равно собирались заменить на стандартное для данной компании приложение через несколько недель, и поэтому мне сказали, чтобы я не особенно старался разрабатывать такое приложение основательно.

Да, вы угадали. Это было приложение, сделанное на скорую руку, хотя оно эксплуатировалось в течение пяти лет. Разумеется, я значительно усовершенствовал его со своими коллегами, но в то же время я не смог убедить заказчика перейти на другую платформу, более подходящую для столь ответственного приложения.

Смещение акцента с базы данных на модель предметной области

Я лично сместил свой акцент с базы данных на модель предметной области, главным образом, потому, что стал уделять больше внимания не столько эффективности, сколько сопровождению приложений. Этот новый стиль проектирования совсем не означает, что я принес в жертву эффективность, но я стараюсь избегать преждевременной оптимизации каждой мелкой детали, что нередко обходится дороже, чем оно того заслуживает. При этом я пытаюсь построить как можно более ясную модель предметной области, что в конечном итоге упрощает оптимизацию, когда она действительно требуется. Нельзя сказать, что я совсем забыл о базе данных — просто она не находится в главном фокусе моего внимания. Я стараюсь находить разумные компромиссы и считаю, что с базой данных следует обращаться очень аккуратно.

Проектирование, ориентированное на модель предметной области, предоставляет больше возможностей для сопровождения приложений в долгосрочной перспективе, поскольку оно оказывается более ясным, а реализация точнее соответствует уровню абстракции модели. Еще одна важная причина для такого метода проектирования заключается в том, что эффективная модель предметной области служит удобным инструментом для сокращения дублирования логики. (Разумеется, все эти свойства присущи и объектной ориентации. Я рассматриваю модель предметной области в качестве стиля проектирования, где объектная ориентация применяется в чистом виде, раздвигая границы для написания удобно сопровождаемого и, конечно, проверяемого кода.)

Возможно, шаблон Domain Model [Fowler PoEAA] означает для вас нечто совершенно особенное, но ведь существует множество вариантов его применения. Назовем эти варианты разными стилями проектирования.

Более конкретный акцент на ППО

Стиль построения модели предметной области приводит нас в конечном итоге к ППО [Evans DDD].

Для ППО имеются самые разные средства (см. подробное описание в главе 1). В частности, это набор шаблонов, помогающих структурировать модель предметной области. И вскоре мы начнем пользоваться этими инструментами.

Но прежде чем переходить к более конкретным вопросам, рассмотрим вкратце процесс разделения на слои в соответствии с ППО.

Разделение на слои в соответствии с ППО

В своей предыдущей книге [Nilsson NED] я отвел немало места строгой схеме разделения на слои, но здесь все же нужно сказать несколько слов о том, как я теперь представляю себе разделение на слои.

С одной стороны, я делаю большую ставку на разделение на слои. А поскольку я истово исповедую ППО, то стараюсь всеми силами выделить инфраструктуру, в том числе и сохраняемость, в отдельный слой — подальше он базового слоя предметной области.

С другой стороны, я меньше думаю о разделении на слои и уделяю больше внимания только одному: слою предметной области. При этом я не разделяю данный слой (т.е. модель предметной области) на более мелкие слои, сохраняя его на уровне довольно крупных структурных единиц. (На самом деле некоторые шаблоны ППО вполне подходят для задач, решать которые раньше приходилось с помощью разделения на слои.)

До сих пор мы рассмотрели два слоя: инфраструктуры и предметной области. Поверх этих слоев может находиться слой приложения, обеспечивая определенное согласование, но он очень тонкий и только поручает работу слою предметной области (см. шаблон Service Layer (Уровень обслуживания) [Fowler PoEAA], который перекладывает всю работу на модель предметной области подобно классам сценария.)

И наконец, на самом верху располагается слой пользовательского интерфейса.

Пример упрощенной схемы разделения на слои приведен на рис. 4.1.

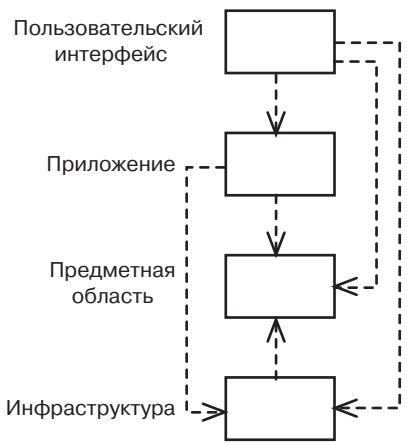


Рис. 4.1. Разделение на слои, типичное для проектов, выполняемых в стиле ППО

На заметку

Опять же, все сказанное выше о разделении на слои нельзя автоматически принимать за эталон только потому, что об этом написано в данной книге! Каждое решение следует подвергать критическому анализу!

А теперь мне бы хотелось сказать несколько слов о том, как я пользуюсь разделением на слои в сравнении со своим старым стилем проектирования. Прежде всего, слой приложения не обязателен — он нужен лишь в том случае, если приносит какую-то пользу. А поскольку слой предметной области получается у меня теперь более обогащенным, чем прежде, то слой приложения меня зачастую мало интересует.

Еще одно отличие заключается в следующем: вместо того чтобы передавать все вызовы вниз по иерархии, слой инфраструктуры может "знать" о слое предметной области, и в нем могут создаваться экземпляры объектов при восстановлении из состояния сохраняемости. Самое главное — чтобы модели предметной области не было ничего известно об инфраструктуре.

На заметку

Следует также подчеркнуть, что в данном случае не принимается во внимание *разбиение*, или рассечение на части в другом измерении по сравнению с разделением на слои. Это очень важно для крупных приложений. Мы еще вернемся к этому вопросу в главе 10.

А теперь пора отправляться в путь, чтобы опробовать некоторые концепции, рассмотренные в этой и предыдущих главах, с акцентом на слое предметной области. И для этого лучше всего подыскать подходящий пример.

Первый набросок

Для большей конкретности составим перечень задач и возможностей приложения, чтобы посмотреть, как в процессе его проектирования решаются некоторые общие задачи упорядочения приложения.

Разумеется, работа над списком возможностей при проектировании на основе предметной области зависит от многих факторов. Как упоминалось выше, рассуждения об "используемой по умолчанию архитектуре" могут показаться странными из-за того, что архитектура в первую очередь зависит от решаемой задачи.

Тем не менее мне бы хотелось рассмотреть здесь одно из возможных решений. Пока что оно будет иметь весьма общий, эскизный характер, а о его деталях речь пойдет в последующих главах книги. Это будет лишь первая попытка сделать грубый набросок проекта, который в дальнейшем обрастет деталями.

Задачи и возможности для примера предметной области

Приведенный ниже список требований будет использован в последующих главах книги для обсуждения некоторых концепций. Итак, рассмотрим подробнее каждую задачу или возможность в произвольном порядке.

1. Список клиентов, формируемый с применением гибкого и сложного фильтра.

   Персоналу, обслуживающему клиентов, необходимо предоставить возможность для удобного поиска клиентов. В частности, они должны пользоваться метасимволами в многочисленных полях, включая имя, адрес, ответственное лицо и т.д. Кроме того, у них должна быть возможность искать клиентов с заказами определенного типа, определенной величины, на определенные виды продукции и т.д. Таким образом, здесь речь идет о полноценной утилите поиска, которая в итоге формирует список клиентов со своими номерами, именами и фамилиями, адресами.

2. Список заказов, формируемый при просмотре отдельного клиента.

   В списке должна отображаться общая стоимость заказа, а также его состояние, тип, дата составления, имя и фамилия ответственного лица.

3. Заказ может состоять из самых разных строк.

   Заказ может быть многострочным, причем в каждой строке описывается заказанная продукция и ее номер.

4. Большое значение имеет выявление конфликтов параллельной обработки.

   В данном случае достаточно выбрать оптимистический вариант управления параллельным выполнением операций, т.е. считать вполне приемлемым, когда пользователя уведомляют о конфликте с предыдущим результатом сохранения после того, как он завершит свою работу и попытается сохранить ее. При этом конфликтами считаются только те, что приводят к настоящим противоречиям. Следовательно, в данном решении следует предусмотреть блок управления версиями для клиентов и заказов. (Это может оказать некоторое влияние на ряд других возможностей.)

5. У клиента может быть долг, не превышающий определенную суму.

   У каждого клиента этот лимит свой. Он устанавливается при первоначальном вводе клиента, а впоследствии может быть изменен. Наличие неоплаченных заказов на общую сумму, превышающую данный лимит, считается противоречием. Это противоречие можно допустить в том случае, если пользователь уменьшит лимит. Пользователь, уменьшивший лимит, получает соответствующее уведомление, но операция сохранения разрешается. Тем не менее заказ не может быть введен или изменен с превышением лимита.

6. Заказ не может быть составлен на сумму свыше 1 млн крон (в данном примере используется шведская валюта).

Этот лимит, в отличие от предыдущего, является правилом, распространяющимся на всю систему.

7. У каждого заказа и клиента должен быть уникальный и удобный для использования номер.

   Допускается следование номеров не по порядку.

8. Прежде чем новый клиент будет признан приемлемым, должен быть проверен его кредит в соответствующем кредитном учреждении.

   Это означает, что лимит, рассматривавшийся в п. 5 данного списка и установленный для клиента, будет проверен на обоснованность.

9. У заказа должен быть клиент, а у строки заказа — заказ.

   Заказов с неопределенными клиентами быть не должно. Это же относится и к строкам заказа — они должны принадлежать определенному заказу.

10. Сохранение заказа и его строк должно быть атомарным.

    Откровенно говоря, я не уверен в том, что такая возможность действительно нужна. Ведь вполне допустимо, чтобы строки заказа вводились после его первоначального составления. Но мне требуется какое-то правило для данного примера, чтобы иметь возможность, связанную с защитой транзакций.

11. У заказов должно быть состояние подтверждения, изменяемое пользователем.

    Это состояние может изменяться пользователями при переходе от подтверждения заказа к неподтверждению и обратно. Что же касается других состояний, то их смена должна происходить косвенно другими методами, доступными в модели предметной области.

Итак, мы составили аккуратный, простой список возможностей, чтобы пользоваться им при рассмотрении разных решений в общей перспективе.

На заметку

Приведенный выше список возможностей может показаться слишком ориентированным на данные. В некоторых случаях такая кажущаяся ориентация на данные воплощается на практике в свойства с некоторым поведением.

Следует, однако, заметить, что мы стремимся применить модель предметной области. И в этом случае главная трудность заключается в следующем: как нам обращаться с данными, если мы пользуемся реляционной базой данных. Поэтому, когда дело доходит до объектной ориентации, имеет смысл уделить данным чуть больше внимания, чем обычно.

Итак, сформулировав возможности и задачи, перейдем к рассмотрению возможного их решения.

Решение задач по очереди

А теперь приступим к решению (как обычно, по очереди) задач из приведенного выше списка задач и возможностей и посмотрим, к чему это приведет. Сосредоточив основное внимание на модели предметной области, попробуем уловить и сформировать универсальный язык, не отвлекаясь на инфраструктуру или другие слои.

Впрочем, не помешает составить некоторое представление и о технической среде. Попробуем не усложнять ее и выберем в качестве контекста приложение с графическим пользовательским интерфейсом, имеющим широкие функциональные возможности (Rich GUI). Это приложение должно быть реализовано в настольном исполнении на стороне клиента и объединено с вместе с моделью предметной области и физическим сервером реляционной базы данных в одну локальную сеть.

> ***На заметку***
>
> Ниже упоминаются многие шаблоны, но мы еще не раз вернемся к ним в последующих главах. Полный перечень шаблонов приведен в Приложении Б.

Итак, рассмотрим поставленные задачи по порядку.

1. Список клиентов, формируемый с применением гибкого и сложного фильтра

Первое требование имеет непосредственное отношение к данным и, отчасти, к поисковому поведению. Прежде всего, наметим структуру класса `Customer` и его окружения (пока что без класса `Order`; рис. 4.2).

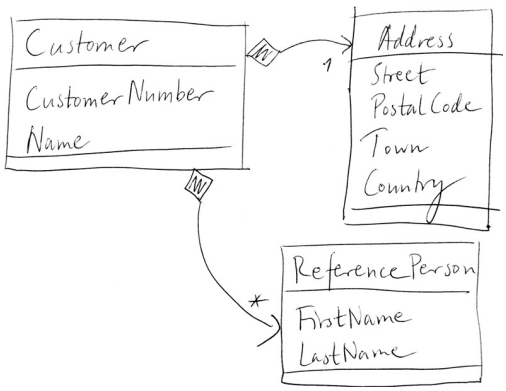


Рис. 4.2. Класс `Customer` и его окружение

На заметку

Как ни странно, но для создания рисунков от руки имеются достаточно веские основания. С их помощью удобно показывать первые наброски, самые общие идеи, с которых, на мой взгляд, можно начинать проект. А код — это лишь исполняемый, отчетливый артефакт, хотя он и является самым важным представлением настоящей модели.

Как видите, я выбрал для клиента достаточно простой (наивный) класс `Customer`, состоящий из классов адреса (`Address`) и ответственных лиц (`ReferencePerson`). Я решил не вносить подобного рода поисковое поведение непосредственно в класс `Customer`, поэтому на данный момент мы располагаем только данными о классах, приведенных на рис. 4.2.

На заметку

Вместо этого можно было бы воспользоваться шаблоном Party Archetype (Групповой архетип) [Arlow/Neustadt Archetype Patterns] или же другими видами реализации ролей, но пока что мы будем пользоваться простыми структурами.

Первое требование можно было бы выполнить, используя шаблон Query Object (Объект запроса) [Fowler PoEAA], о котором речь уже шла в главе 2, но на практике оказывается, что применение таких шаблонов лучше сделать немного инкапсулированным, если имеется такая возможность. Ведь зачастую приходится определять немало из того, что касается результатов запросов, но не имеет никакого отношения к критериям поиска. Речь идет о порядке сортировки, других скрытых критериях, оптимизации, месте, куда следует отправлять объект запроса для последующего выполнения, и прочем. Поэтому я предпочитаю пользоваться шаблоном Repository (Хранилище) [Evans DDD], чтобы немного инкапсулировать выполнение запроса. Это позволяет также сделать код менее многословным для потребителей, а программирование интерфейса API — более ясным.

В главе 2 мы рассматривали шаблон Factory (Фабрика) [Evans DDD] и выяснили, что он определяет начало жизненного цикла экземпляра объекта. А за остальную часть жизненного цикла после создания экземпляра и до его "кончины" отвечает шаблон Repository. В частности, шаблон Repository наводит мост между базой данных и моделью предметной области, когда требуется получить экземпляр объекта, который прежде был сохраняемым. При этом шаблон Repository использует инфраструктуру для выполнения своей задачи.

Методу из шаблона Repository можно было бы передать целый список параметров: по одному на каждое поле фильтра. Но это, без сомнения, привело бы к недоб-

рокачественному коду, который оказался бы не совсем ясным, а значит, и неудобным для сопровождения. Более того, для шаблона Repository не совсем ясно, что делать с параметром `Name` строкового типа, особенно если строка пуста. Означает ли это, что мы должны искать клиентов с пустыми именами или же нас вообще не интересуют их имена? Конечно, можно составить волшебную строку, которая означала бы, что мы ищем клиентов с пустыми именами, а еще лучше — вынудить пользователя вводить метасимвол, если его не интересует имя клиента. Но ни одно из этих решений нельзя считать вполне пригодным. С другой стороны, можно было бы воспользоваться шаблоном Query Object [Fowler PoEAA], в особенности специальным для предметной области, или же шаблоном Specification (Описание) [Evans DDD], но и это решение оказалось бы, на мой взгляд, слишком поспешным и опрометчивым. Поэтому выберем самое простое решение и рассмотрим пока что лишь два возможных критерия. Пример такого решения приведен на рис. 4.3.

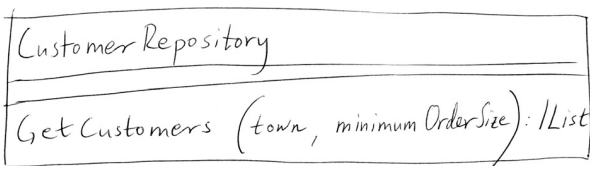


Рис. 4.3. Класс `CustomerRepository` для гибких критериев

На заметку

Разумеется, внутри шаблона Repository можно было бы использовать шаблон Query Object.

ППО прекрасно сочетается с РПТ — вы получаете мгновенную реакцию на свои действия и опробуете модель с помощью тестового кода, что позволяет вам лучше ее понять. Но пока что мы воздержимся от следования по пути РПТ и исследуем набросок модели с помощью мелких тестов только для того, чтобы рассмотреть замысел данного решения под другим углом зрения. В следующей главе мы вернемся к этому вопросу и попробуем разработать модель более тщательно, точнее придерживаясь принципов РПТ.

Итак, для того чтобы выбрать всех клиентов в определенном городе (Роннебю — Ronneby) хотя бы с одним заказом на сумму более 1 тыс. шведских крон, воспользуемся следующим тестом:

```
[Test]
public void
  CanGetCustomersInSpecificTownWithOrdersOfCertainSize()
{
```

```
    int numberOfInstancesBefore = _repository.GetCustomers
      ("Ronneby", 1000).Count;

    _CreateACustomerAndAnOrder("RonnebyЭ, 20000);

    Assert.AreEqual(numberOfInstancesBefore + 1
      , _repository.GetCustomers ("Ronneby", 1000).Count);
}
```

В данном примере метод `GetCustomers()` из шаблона Repository используется для выборки всех клиентов, удовлетворяющих критериям поиска города и минимальной величины заказа.

2. Список заказов, формируемый при просмотре отдельного клиента

Между классами `Customer` и `Order` можно было бы установить двунаправленную связь, когда каждому клиенту соответствует список заказов, а каждому заказу — один клиент. Но такая двунаправленность обходится дорого в смысле сложности, тесного связывания и дополнительных издержек. Поэтому было бы неплохо, на мой взгляд, организовать доступ к заказам клиента через класс хранилища заказов (`OrderRepository`). Это приводит нас к модели, представленной на рис. 4.4.

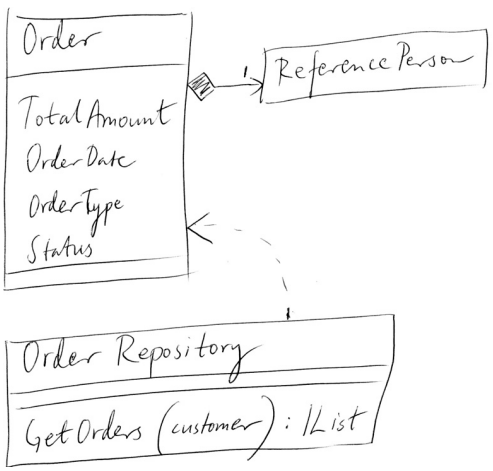


Рис. 4.4. Классы `OrderRepository`, `Order` и их окружение

Так, если потребителю требуется посмотреть заказы определенного клиента, он должен обратиться к классу `OrderRepository` следующим образом:

```
[Test]
public void CanGetOrdersForCustomer()
{
  Customer newCustomer = _CreateACustomerAndAnOrder
```

```
    ("Ronneby", 20000);

  IList ordersForTheNewCustomer =
    _repository.GetOrders(newCustomer);

  Assert.AreEqual(1, ordersForTheNewCustomer.Count);
}
```

Классу `Customer` можно было бы присвоить свойство `OrderList` (Список заказов) и обращаться к хранилищу неявно, но, на мой взгляд, это лучше делать явно. Кроме того, следует избегать связывания классов `Customer` и `OrderRepository`.

Отдельного упоминания требует общая сумма каждого заказа. Это более сложный вопрос, чем кажется на первый взгляд, поскольку для расчета общей суммы требуются все строки заказа. Конечно, это не такая уж непреодолимая трудность, но она все же настораживает. Возможно, мои опасения преждевременны, но я не могу ничего с этим поделать, поскольку загрузка строк заказа может обойтись недешево, когда потребуется, например, показать довольно внушительный список заказов определенного клиента. Поэтому если строки заказа не загружаются, для общей суммы (свойства `TotalAmount`) лучше использовать простое поле, сохраняемое в таблице заказов (`Orders`). Если же строки заказа загружаются, то вместо свойства `TotalAmount` используется полный расчет общей суммы заказа.

Конечно, это не самый актуальный вопрос на данный момент. Но он решается для потребителей очень просто следующим образом:

```
[Test]
public void CanGetTotalAmountForOrder()
{
  Customer newCustomer = _CreateACustomerAndAnOrder
    ("Ronneby", 420);

  Order newOrder = (Order)_repository.GetOrders
    (newCustomer)[0];

  Assert.AreEqual(420, newOrder.TotalAmount);
}
```

Появление этих строк кода объясняется тем, что мы имеем дело с агрегатом данных, а сами строки отражают внутреннее содержание заказа. Хотя в некоторых ситуациях это может привести к снижению производительности. В таком случае решение можно искать с помощью шаблона Lazy Load (Загрузка по требованию) [Fowler PoEAA], используемого, например, для оперативной загрузки списков из базы данных, или же с помощью представления, оптимизированного “только для чтения”.

Однако, как уже отмечалось, данное решение следует рассматривать в том случае, если простого и прямого решения окажется недостаточно.

3. Заказ может состоять из самых разных строк

Эта возможность реализуется достаточно просто. На рис. 4.5 представлена улучшенная модель.

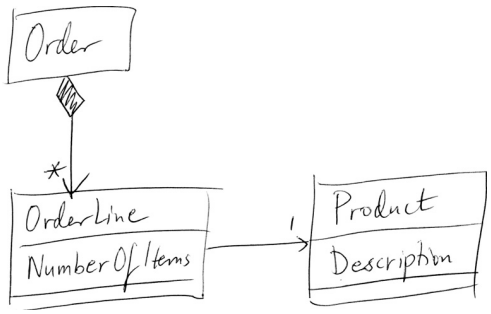


Рис. 4.5. Заказ (класс `Order`) и его строки (класс `OrderLine`)

Как видите, я рассматриваю в этой модели и однонаправленную связь. Конечно, можно было бы обойтись отправкой строки заказа (`OrderLine`) вместе с ее заказом (`Order`), если бы знать, что и то и другое понадобится для определенной части логики. Впрочем, риск отправки заказа вместе со строкой из другого заказа не очень велик.

```
[Test]
public void CanIterateOverOrderLines()
{
  Customer newCustomer = _CreateACustomerAndAnOrder
    ("Ronneby", 420);

  Order newOrder = (Order)_repository.GetOrders
    (newCustomer)[0];

  foreach (OrderLine orderLine in newOrder.OrderLines)
    return;

  Assert.Fail("Этого у меня пока что не должно быть");
}
```

Обратите также внимание на заметные отличия в модели, приведенной на рис. 4.5, по сравнению с реляционной моделью, описывающей то же самое. Для реляционной модели были бы характерны отношения “один ко многим” и “многие к одному” между классами `Product` и `OrderLine`. Но до сих пор нас совершенно не интересовало отношение “один ко многим” в данной конкретной модели предметной области.

4. Большое значение имеет выявление конфликтов параллельной обработки

Эта задача посложнее. На мой взгляд, обоснованное ее решение состоит в отделении класса `Customer` от класса `Order`, который должен составить вместе с классом

OrderLine отдельную структурную единицу для выявления конфликтов параллельной обработки. Это вполне согласуется с применением модели предметной области для логики, необходимой для реализации остальных возможностей приложения. Для этой цели подходит шаблон Aggregate (Агрегат) [Evans DDD], который позволяет определить объекты, принадлежащие определенным группам объектов, с которыми обычно приходится работать как с едиными блоками, загружая их вместе (поначалу хотя бы для написания сценариев), оценивая общие правила для них и т.д.

Следует, однако, заметить, что шаблон Aggregate не относится к самым нужным специальным шаблонам, но чаще применяется для смыслового наполнения модели.

В частности, шаблон Aggregate служит для создания отдельных структурных единиц с целью выявить конфликты параллельной обработки (рис. 4.6).

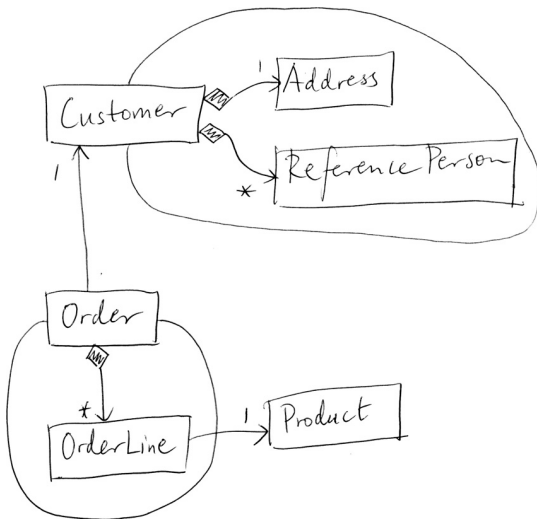


Рис. 4.6. Агрегаты

5. У клиента может быть долг, не превышающий определенную сумму

И эта задача непроста. Первое ее решение, которое приходит на ум, вероятно, состоит в том, чтобы ввести свойство TotalCredit (Общий кредит) в класс Customer. Но такое решение не совсем удачно, поскольку оно настолько прозрачно, что потребитель не увидит расходы при вызове данного свойства. Кроме того, оно не лишено противоречий. Если я даже не вижу, что у агрегата Customer имеются заказы (Order), значит, такую структурную единицу я не могу считать подходящей для обеспечения непротиворечивости в самой модели предметной области. А это служит явным признаком того, что следует искать другое решение.

В модели предметной области, на мой взгляд, лучше применить шаблон Service (Обслуживание) [Evans DDD], который может сообщить текущие сведения об общей сумме кредита для клиента. Соответствующий вид модели приведен на рис. 4.7.

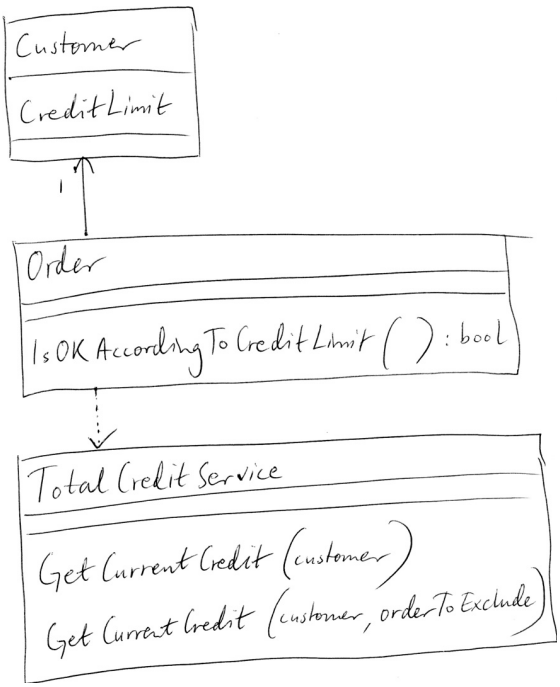


Рис. 4.7. Класс `TotalCreditService`

Основанием для перегрузки метода `GetCurrentCredit()` класса `TotalCreditService` (Обслуживание общего кредита) может, на мой взгляд, служить потребность в проверке величины текущего кредита, не касаясь текущего заказа, который составляется или изменяется. По крайней мере, это неплохая идея. Посмотрим, как такое обслуживание будет выглядеть на деле (не обращая внимания на взаимодействие между отдельными частями модели).

```
[Test]
public void CanGetTotalCreditForCustomerExcludingCurrentOrder()
{
  Customer newCustomer = _CreateACustomerAndAnOrder
    ("Ronneby", 22);

  Order secondOrder = _CreateOrder(newCustomer, 110);

  TotalCreditService service = new TotalCreditService();

  Assert.AreEqual(22+110
    , service.GetCurrentCredit(newCustomer));
  Assert.AreEqual(22,
    service.GetCurrentCredit(newCustomer, secondOrder));
}
```

Общая идея взаимодействия классов `Order`, `Customer` и обслуживания должна быть вам ясна из рис. 4.7 и приведенного выше фрагмента кода.

6. Заказ не может быть составлен на сумму свыше 1 млн крон

Возможность выявления конфликтов параллельной обработки была предусмотрена выше в виде отдельной структурной единицы, а точнее агрегата, составленного из классов `Order` и `OrderLine` с помощью шаблона Aggregate, и поэтому мы вправе воспользоваться данной возможностью в модели предметной области. Инвариант агрегата в данном случае состоит в том, что сумма заказа не должна превышать 1 млн крон. Это правило не должны нарушать другие пользователи, чтобы не создавать осложнения при обработке определенного заказа. Ведь иначе у меня или у другого пользователя возникнет конфликт параллельной обработки. Поэтому в классе `Order` следует создать метод `IsOKAccordingToSize()`, который может быть вызван (при желании) потребителем (рис. 4.8).

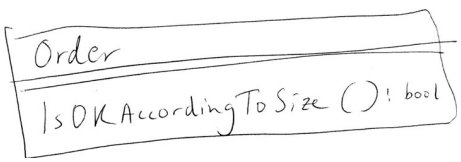


Рис. 4.8. Класс `Order` с методом `IsOKAccordingToSize()`

Ниже приведен простой тест для демонстрации интерфейса API:

```
[Test]
public void CanCheckThatAnOrdersTotalSizeIsOK()
{
  Customer newCustomer = _CreateACustomerAndAnOrder
    ("Ronneby", 2000000);

  Order newOrder = (Order)_repository.GetOrders
    (newCustomer)[0];

  Assert.IsFalse(newOrder.IsOKAccordingToSize());
}
```

> ***На заметку***
>
> В дальнейшем для данного правила следует рассмотреть возможность применения шаблона Specification [Evans DDD].

7. У каждого заказа и клиента должен быть уникальный и удобный для использования номер

Независимо от моего отношения как разработчика к этому требованию, оно очень важно и типично с точки зрения пользователя. В первом приближении его можно вполне удовлетворить средствами базы данных. Так, в SQL Server для этой цели служат ключи IDENTITY. В самой модели предметной области не нужно ничего делать, кроме обновления объекта-сущности (Entity) после его ввода в таблицу базы данных. А до этого в модели предметной области будут, вероятно, отображаться нулевые значения свойств `OrderNumber` (Номер заказа) и `CustomerNumber` (Номер клиента) (рис. 4.9).

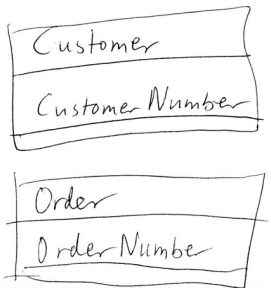


Рис. 4.9. Улучшенные классы `Order` и `Customer`

Я уже жалею, что поторопился с таким решением. Ведь это означает, что номер заказа распределяется при первом же сохранении заказа. Но я не уверен, что это удачная идея.

По существу, я смешал два разных понятия: шаблон Identity Field (Поле идентификации) [Fowler PoEAA], предназначенный для связывания экземпляра объекта со строкой в таблице базы данных (используя первичный ключ из этой строки в виде значения в объекте), и идентификационный номер, имеющий прикладное значение. Иногда эти понятия совпадают, но чаще всего они должны быть представлены двумя разными идентификаторами.

Довольно странным кажется и то, что клиенту присваивается удобный для использования номер в виде маркера, как только он переходит в состояние сохраняемости. Вероятно, этот вариант следовало бы предусмотреть на тот случай, если клиент работает в системе (после проверки его полномочий, а возможно, и подтверждения вручную). К этому вопросу мы еще вернемся в последующих главах книги.

8. Прежде чем новый клиент будет признан приемлемым, должен быть проверен его кредит в соответствующем кредитном учреждении

Для решения этой задачи потребуется еще один шаблон Service [Evans DDD] в модели предметной области, чтобы инкапсулировать обмен информацией с кредитным учреждением (рис. 4.10).

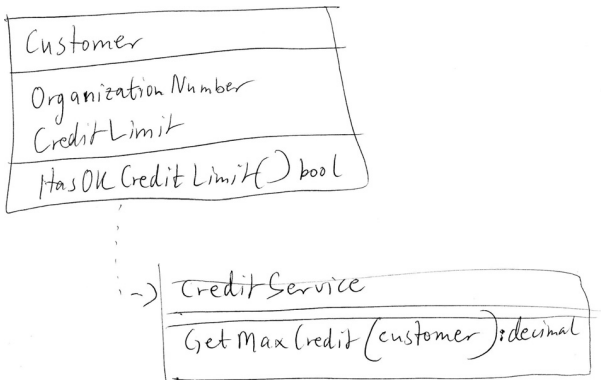


Рис. 4.10. Класс `CreditService`

И в этом случае все сводится к взаимодействию объекта-сущности (в данном случае класса `Customer`) с шаблоном Service, чтобы предоставить этому объекту возможность самому отвечать на поставленный вопрос. Посмотрим, как это взаимодействие происходит на деле в небольшом тесте. (Следует также заметить, что для реального обслуживания клиента, вероятно, потребуется номер его организации (`OrganizationNumber`), который отличается о номера самого клиента (`CustomerNumber`) и является официальным идентификационным номером, предоставляемым органами, регистрирующими организацию.)

```
[Test]
public void CantSetTooHighCreditLimitForCustomer()
{
  Customer newCustomer = _CreateACustomer("Ronneby");

  // Внести упрощенный вариант класса CreditService
  // не допускающий кредит на сумму более 300 крон
  newCustomer.CreditService = new StubCreditService(300);

  newCustomer.CreditLimit = 1000;

  Assert.IsFalse(newCustomer.HasOKCreditLimit);
}
```

На заметку

Грегори Янг обратил внимание на недоброкачественность приведенного выше кода. Это объясняется, скорее всего, неатомарным характером операции. Сначала значение задается, а следовательно, перезаписывается старое значение. Затем это значение проверяется, если оно запоминается. Вместо этого, возможно, лучше было бы использовать выражение `Customer.RequestCreditLimit(1000)`. Мы еще вернемся к этому вопросу в главе 7.

9. У заказа должен быть клиент, а у строки заказа — заказ

Это типичное и вполне обоснованное требование, которое связано с ограничениями в базе данных на целостность ссылочных данных. Теперь нужно проверить также модель предметной области. Нет смысла переводить в состояние сохраняемости изменения, вносимые в модель предметной области, если ясно, что в ней еще немало неточностей, как в данном случае. Но вместо проверки этой модели в качестве первой попытки можно сделать ее обязательной к применению благодаря классу `OrderFactory` (Фабрика заказов) как средству составления заказов, а раз уж приходится иметь дело со строками заказов (`OrderLine`), то аналогичным образом можно сделать для них обязательным наличие заказа (`Order`) и продукции (`Product`) (рис. 4.11).

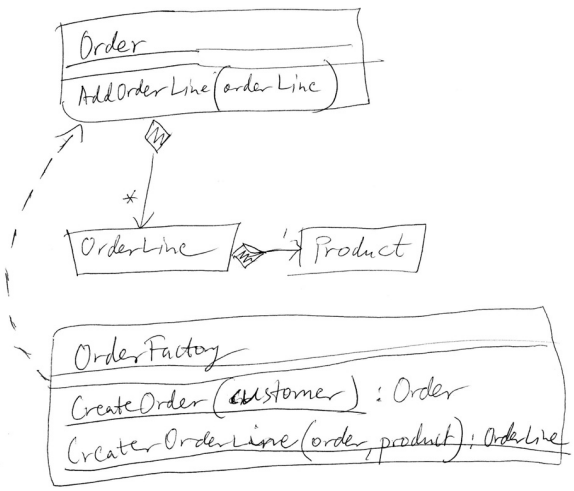


Рис. 4.11. Класс `OrderFactory` и улучшенный класс `Order`

Как обычно, проверим взаимодействие разных частей модели с помощью теста.

```
[Test]
public void CanCreateOrderWithOrderLine()
{
  Customer newCustomer = _CreateACustomer("KarlskronaЭ);

  Order newOrder = OrderFactory.CreateOrder(newCustomer);

  // В классе OrderFactory будет использован метод
  // AddOrderLine() из класса Order.
  OrderFactory.CreateOrderLine(newOrder, new Product());

  Assert.AreEqual(1, newOrder.OrderLines);
}
```

Пожалуй, это похоже на избыточное проектирование, поскольку модель имеет теперь не такой гладкий и аккуратный вид. Впрочем, об этом мы поговорим в следующей главе.

Раз уж мы коснулись класса `OrderFactory`, мне бы хотелось упомянуть об удачной идее использовать шаблоны Null Object (Пустой объект) [Woolf Null Object] для классов `OrderType`, `Status` и `ReferencePerson`. Шаблон Null Object позволяет использовать пустой экземпляр объекта вместо пустого значения (термин "пустой" в данном случае означает устанавливаемые по умолчанию значения членов, например, `string.Empty` для членов типа `string`). Благодаря этому становится более удобной для чтения запись через точку, как например:

```
this.NoNulls.At.All.Here.Description
```

Что же касается базы данных, то внешние соединения можно сократить и чаще использовать внутренние соединения, поскольку внешние ключи будут, по крайней мере, указывать на пустые символы, тогда как столбцы с внешними ключами не будут пустыми. Таким образом, пустые объекты заметно упрощают дело, причем совершенно неожиданным образом.

Как следует из рис. 4.11, класс `Order` был дополнен методом `AddOrderLine()` для ввода строк заказа. Это частичный рефакторинг кода типа Encapsulate Collection (Инкапсуляция совокупности объектов) [Fowler R]. По существу, это означает, что родительский объект будет защищать все изменения в совокупности объектов.

С другой стороны, база данных — это последний сторожевой пост, и поэтому данное правило относится и к ней.

10. Сохранение заказа и его строк должно быть атомарным

В данном случае классы `Order` и `OrderLine` представляются мне как агрегат. Именно на это утверждение и будет ориентировано решение данной задачи.

Для отслеживания экземпляров измененных (новых и удаленных) объектов можно воспользоваться реализацией шаблона Unit of Work (Единица работы) [Fowler PoEAA]. Этот шаблон согласует подобные изменения, используя одну физическую операцию обращения к базе данных в состоянии сохраняемости.

11. У заказов должно быть состояние подтверждения, изменяемое пользователем

Как установлено выше, у заказов имеется состояние подтверждения (рис. 4.12), для чего достаточно ввести метод `Accept()`. А вопрос о том, следует ли использовать внутренним образом шаблон State (Состояние) [GoF Design Patterns], придется отложить до более подходящего момента. Подобные решения лучше всего принимать в ходе рефакторинга кода.

При рассмотрении остальных состояний заказа мы можем ввести дополнительные методы. А до тех пор, пока в этом нет явной необходимости, ограничимся методом `Accept()`. Этот замысел можно представить в данный момент следующим образом:

```
[Test]
public void CanAcceptOrder()
{
  Customer newCustomer = _CreateACustomer("Karlskrona");
  Order newOrder = OrderFactory.CreateOrder(newCustomer);

  Assert.IsFalse(newOrder.Status == OrderStatus.Accepted);

  newOrder.Accept();

Assert.IsTrue(newOrder.Status == OrderStatus.Accepted);
}
```

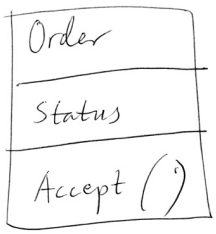


Рис. 4.12. Классы `Order` и `Status`

Модель предметной области на данном этапе

Если подвести итог всему сказанному выше, то модель предметной области можно представить так, как показано на рис. 4.13.

На заметку

Как следует из рис. 4.13, класс `ReferencePerson` оказывается в двух разных агрегатах, чего нельзя сказать об экземпляре его объектов. Это характерный пример того, как статической блок-схеме недостает выразительности. Но в то же время он ясно показывает, насколько простое и лаконичное пояснение может дать блок-схема.

Согласен, что модель предметной области выглядит на данный момент не совсем упорядоченно. В дальнейшем, при обсуждении деталей, мы постараемся привести модель в порядок, разбив ее на более логичные части.

На заметку

Модель, приведенная на рис. 4.13, а также все ее части были созданы наперед, поэтому они потребуют значительного улучшения после перехода к стадии разработки приложения.

Выше была представлена лишь базовая модель предметной области, в которой отсутствуют составляющие, связанные с инфраструктурой, в том числе упоминавшийся выше шаблон Unit of Work [Fowler PoEAA]. Разумеется, это было сделано намеренно. Мы еще вернемся к вопросам инфраструктуры далее в этой книге. А пока уделим основное внимание модели предметной области.

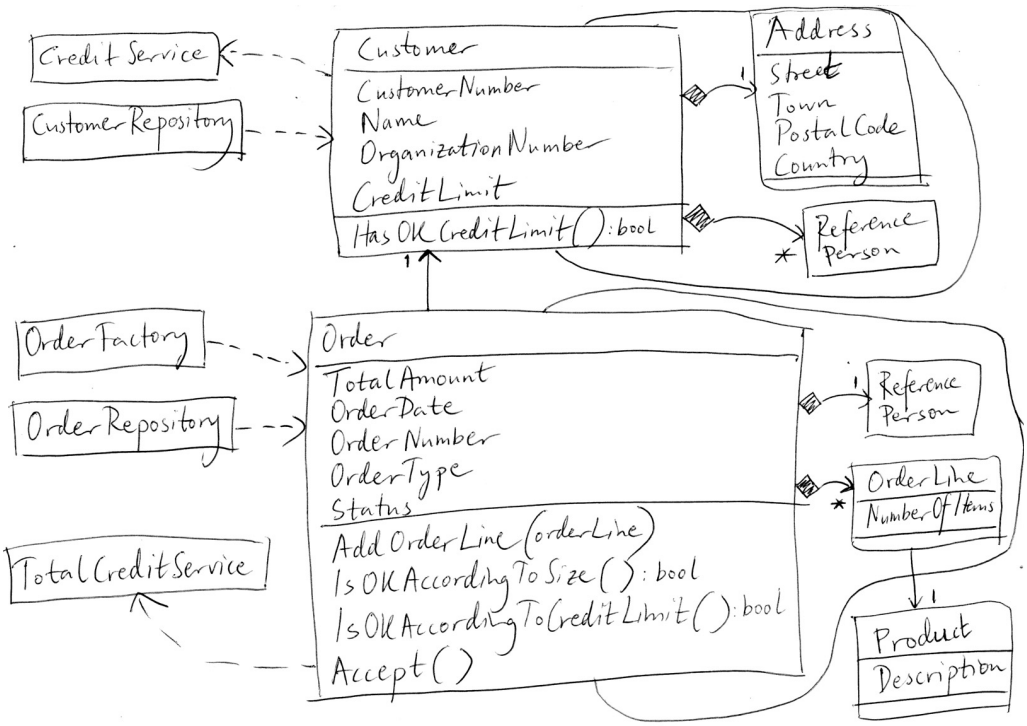


Рис. 4.13. Набросок модели предметной области, отражающий мой подход к выполнению требований по списку возможностей

Как уже отмечалось, существует множество вариантов применения шаблона Domain Model. Для того чтобы показать некоторые из них, я попросил своих коллег описать те способы применения подобных шаблонов, которым они отдают предпочтение. Эти описания вы найдете в Приложении А.

Для того чтобы дать более ясное представление о рассмотренных выше требованиях к приложению, мне бы хотелось взглянуть на них под другим углом зрения и попутно набросать несколько форм для пользовательского интерфейса, который предстоит разработать.

Первая попытка привязать пользовательский интерфейс к модели предметной области

А теперь рассмотрим новую архитектуру снизу вверх, начав с базы данных. Один из рецензентов книги сказал мне совершенно определенно, что рассмотрение архитектуры следует начинать с точки зрения разработчика пользовательского интерфейса. Если она окажется неудачной с этой точки зрения, то продолжать чтение нет смысла. У такого подхода, конечно, есть свои преимущества, поэтому бросим первый взгляд на только что намеченную модель предметной области с точки зрения разработчика пользовательского интерфейса.

Основная цель

Итак, посмотрим, насколько нам удалось добиться одной из основных целей: "упрощение слева" или же "упрощение сверху" — в зависимости от того, как представлять слои в многослойной архитектуре. Под этим подразумевается предоставление простого интерфейса API разработчику пользовательского интерфейса, чтобы ему было легче представить себе модель, а значит, уделить больше внимания вопросам пользовательского интерфейса, не думая о сложных протоколах для модели предметной области.

При первом обсуждении пользовательского интерфейса мы будем касаться привязки данных, хотя это далеко не основная цель данного обсуждения. Несмотря на то что в этой книге не уделяется много внимания привязке данных, мы все же коснемся этого вопроса в последующих главах.

Акцент на простой пользовательский интерфейс

В данный момент модель предметной области кажется слишком абстрактной, но если рассматривать ее с точки зрения пользовательского интерфейса, это положение может немного измениться. Итак, мы должны опробовать два следующих варианта:

- Составление списка заказов клиента.
- Ввод заказа.

Опять же, мы не будем касаться здесь обычной привязки данных, а лишь рассмотрим простой и ясный код для привязки пользовательского интерфейса к модели предметной области.

Составление списка заказов клиента

Итак, требуется сформировать список заказов клиента. При последующем построении модели предметной области останется только ввести новое "представление" выше этой модели и сымитировать некоторые заказы. В данном случае речь идет о такой форме, как на рис. 4.14.

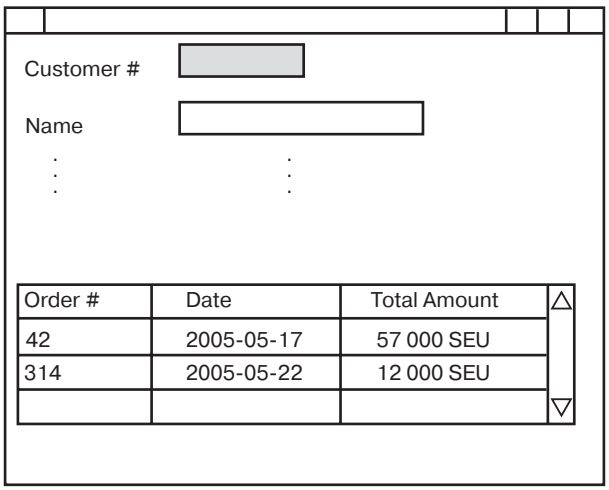


Рис. 4.14. Составление списка заказов клиента

На заметку

У вас может сложиться впечатление, что я избегаю использования таких вспомогательных средств, как редактор VS.NET. Но я решил наглядно представить общий замысел формы с помощью ручки и бумаги потому, что это лишь ранняя стадия эскизного проекта. (Иллюстрации к этой книге визуализированы мной в графической программе.)

Данная форма достаточно проста. Для ее проверки и отображения нужно написать функцию, которую можно было бы вызывать из функции `Main()` для имитации клиента и некоторых его заказов.

Пользователь, скорее всего, будет выбирать клиента из формы со списком клиентов, и поэтому мы предоставим экземпляр объекта `Customer` для конструктора формы, чтобы отобразить подробные сведения о клиенте (дадим этой форме название `CustomerForm`). Экземпляр объекта `Customer` будет храниться в частном поле формы под названием `_customer`.

В таком случае код метода `_PaintCustomer()` для вывода данных о клиенте в этой форме может выглядеть следующим образом:

```
// Форма CustomerForm
private void _PaintCustomer()
{
  // Получить данные:
  IList theOrders = _orderRepository.GetOrders(_customer);

  // Отобразить данные о клиенте:
```

```
  txtCustomerNumber.Text =
    _customer.CustomerNumber.ToString();
  txtName.Text = _customer.Name;

  // Отобразить данные о заказах:
  foreach (Order o in theOrders)
  {
  // ВЫПОЛНИТЬ...
  }
}
```

Следует подчеркнуть, что для отображения данных о заказах в приведенном выше коде можно применить шаблон Null Object [Woolf Null Object]. В этом случае у всех заказов должен быть пустой объект `ReferencePerson` — даже если ответственное лицо еще не указано. Это избавляет от необходимости проверять пустые значения, что, на мой взгляд, служит характерным примером детали, упрощающей задачу разработчика пользовательского интерфейса.

Написание кода для пользовательского интерфейса вручную оказалось несколько утомительным занятием. Но в итоге код все равно получился не очень простым, хотя были предприняты попытки сделать лишь самое элементарное.

Ввод заказа

Рассмотрим еще один пример формы, в которой можно вводить заказ. Предположим, что сначала выбирается клиент, а затем — такая форма, как на рис. 4.15.

Для этого сначала создается экземпляр объекта `Order`, после чего этот объект передается конструктору формы.

В таком случае код метода `_PaintCustomer()` может выглядеть следующим образом:

```
// Метод OrderForm._PaintOrder()
txtOrderNumber.Text = _ShowOrderNumber(_order.OrderNumber);
txtOrderDate.Text = _order.OrderDate.ToString();
// И так далее, код в этой части совершенно такой же,
// как и в приведенном выше примере кода для
// метода _PaintCustomer().
```

Для того чтобы исключить нулевое значение номера заказа (`txtOrderNumber`), в приведенном выше фрагменте кода использована вспомогательная функция (`_ShowOrderNumber()`), которая должна отображать текст типа “Новый заказ” вместо 0.

Что получается в итоге

Такой пользовательский интерфейс можно, в частности, описать с помощью шаблона Separated Presentation (Раздельное представление) [Fowler PoEAA2], чтобы отделить логику от кода, манипулирующего представлением данных. Этот основной

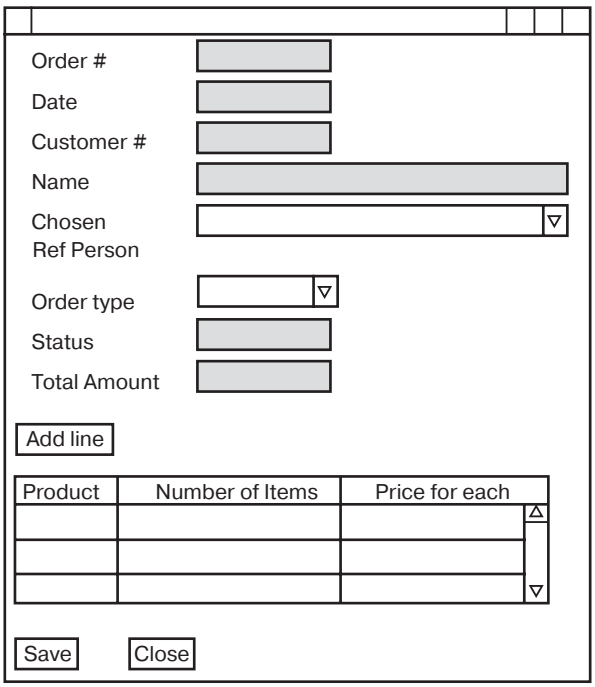


Рис. 4.15. Форма для ввода заказа

принцип рекомендуется уже давно, но зачастую им сильно злоупотребляют. Мы же воспользуемся им в какой-то мере автоматически и без всяких затрат, поскольку выбрали стиль ППО.

И в этом разделе мы попытались лучше представить себе требования к приложению, рассмотрев их с точки зрения пользовательского интерфейса. Впрочем, изучение конкретных вопросов, связанных с пользовательским интерфейсом, придется отложить вплоть до главы 11.

А теперь посмотрим на требования к приложению под еще одним углом зрения.

Еще одно измерение

До сих пор речь в этой главе шла о логической структуре модели предметной области, что, конечно, позволяет представить общую картину лишь в одном измерении. А ведь существуют и другие измерения. Рассмотрим другие стили исполнения данной модели, начав с подготовки среды развертывания по списку возможностей. В этой среде мы создаем, как упоминалось выше, клиентское приложение с расширенными функциональными возможностями (WinForms) без сервера приложений (рис. 4.16).

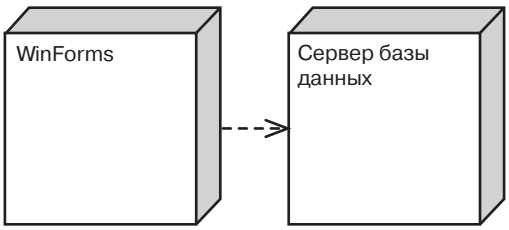


Рис. 4.16. Приложение WinForms без сервера приложений

Ради удобства изложения усложним немного ситуацию, введя требование сервера приложений, как показано на рис. 4.17.

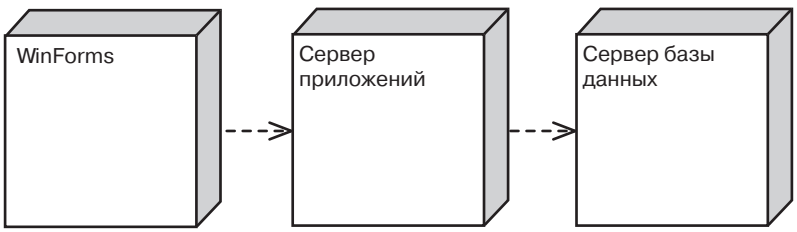


Рис. 4.17. Приложение WinForms и сервер приложений

Прежде всего, возникает вопрос: на каком уровне следует исполнять модель предметной области? Следует ли это сделать на сервере приложений, а затем передать на уровень потребителя структуры данных, используя, например, шаблон Data Transfer Objects (DTO — Объекты для передачи данных) [Fowler PoEAA]? Или же модель предметной области следует исполнять только на стороне потребителя, наполнив ее данными по запросу DTO из сервера приложений? А может быть модель предметной области следует исполнять на обоих уровнях или же должны быть две разные модели?

На заметку

Очень важно подумать о назначении сервера приложений: действительно ли он нужен и нужно ли на самом деле распределять модель предметной области? Напомним первый закон распределенного объектного проектирования: “Не распределять объекты!” [Fowler PoEAA].

Представим на мгновение, что модель предметной области следует исполнять, по крайней мере, на сервере приложений. Нужно ли тогда разделять реализацию этой модели в виде экземпляров таким образом, чтобы один экземпляр служил для представления одного клиента всем пользователям? Или же следует выделять по одному экземпляру модели предметной области на каждого пользователя либо на каждый

сеанс работы в системе, чтобы несколько экземпляров представляли конкретного клиента в определенный момент времени?

Третий вопрос: должна ли реализация модели предметной области сохранять свое состояние в промежутке между вызовами или же она должна исчезать после каждого вызова?

Четвертый вопрос: следует ли стремиться к полной реализации модели предметной области, выбирая как можно больше данных при каждом запросе, и вообще не удалять ее экземпляры?

Мы поставили немало вопросов, и ответы на них, как всегда, зависят от конкретной ситуации. Обсуждение этих вопросов усложняется, в частности, тем, что, действуя в стиле ППО, мы почти не ничего не решили в отношении инфраструктуры.

Тем не менее мне бы хотелось поделиться с вами тем, как я обычно решаю подобные вопросы.

На заметку

Под "реализацией модели предметной области" я подразумеваю создание ряда экземпляров этой модели вместо классов. Разумеется, классы разделяются среди пользователей намного чаще, чем экземпляры. Тем не менее я считаю, что этот термин вносит ясность в обсуждение.

Расположение модели предметной области

Прежде всего, если требуется контролировать уровни потребителя и сервера приложений, то было бы неплохо раскрыть и использовать модель предметной области как на уровне потребителя, так и на уровне сервера приложений. Если же она не используется на обоих уровнях, то существует риск выполнить лишнюю работу и добиться меньшей эффективности, поскольку в этом случае мы можем создать две одинаковых, хотя и не полностью, модели предметной области: одну — для потребителя, а другую — для сервера приложений. Кроме того, нам потребуются средства сопряжения для перехода от одной модели к другой. (При этом очень важно понимать саму ситуацию — с одной или двумя моделями. Если вы думаете, что модель одна, а на самом деле их две, то ситуация может осложниться.)

Следует также заметить, что в сложных ситуациях я предпочитаю использовать шаблон Presentation Model (Модель представления) [Fowler PoEAA2] для представления оптимизированного пользовательского интерфейса или варианта модели предметной области.

Изолирование или совместное использование экземпляров

Следует ли разделять реализацию модели предметной области на сервере приложений среди пользователей или сеансов работы в системе? Мне лично по душе сама идея разделяемой (т.е. совместно используемой) реализации модели предметной области, но на практике я чаще всего держусь от нее подальше. Воплотить эту идею

в жизнь намного сложнее, чем кажется на первый взгляд, поэтому я реализую модель предметной области для каждого пользователя, а зачастую и для каждого сеанса работы в системе. Ведь сделать это намного проще.

Одна из основных трудностей, связанных с разделяемым множеством экземпляров объектов предметной области, возникает в том случае, если требуется исполнение в распределенном виде — возможно, на двух серверах приложений в "ничего не разделяющем" кластере. Это, в свою очередь, приводит к довольно сложной проблеме распределенного кэширования — по крайней мере, в отношении обеспечения согласованности в реальном масштабе времени. Данная проблема настолько сложна, что я пока что отказываюсь ее решать и стараюсь найти более общее решение, которое может оказаться вполне пригодным для вертикального масштабирования системы. Но в особых случаях все же можно прийти к вполне приемлемому решению. Конечно, если все экземпляры модели предметной области находятся в оперативной памяти, то лишь в немногих ситуациях может потребоваться горизонтальное масштабирование системы — по крайней мере, по соображениям эффективности. Тем не менее, если возникает столь сложная проблема, найти ее удачное решение оказывается непросто.

На заметку

Здесь необходимо вкратце пояснить, что означает "ничего не разделяющий" кластер. Этот термин означает, что серверы приложений не используют совместно ЦП, диск или оперативную память. Они действуют совершенно независимо друг от друга, что выгодно с точки зрения масштабируемости.

Подробнее об этом можно прочитать в книге *In Search of Clusters* [Pfister Wolfpack].

Реализация модели предметной области с сохранением или без сохранения состояния

На сервере приложений я не допускаю сохранение состояния реализации модели предметной области и избавляюсь от нее в промежутке между вызовами. А на стороне потребителя я стараюсь сохранить реализацию модели предметной области в промежутке между вызовами, но зачастую делаю это только после прецедента использования.

Полная или частичная реализация модели предметной области

И наконец, я не стремлюсь реализовать модель предметной области полностью. Делать это на стороне потребителя, где базы данных, как правило, достаточно крупные, было бы просто неразумно. Вместо этого я придерживаюсь старой истины: брать только то, что нужно, и ничего лишнего. После того как модель будет реализована, я избавляюсь от нее, освобождая место для других данных, и не храню старые данные слишком долго. Но совсем другое дело — статические данные. Мы должны как можно больше кэшировать данные "только для чтения".

На заметку

Полные, разделяемые, сохраняющие свое состояние реализации модели предметной области поддерживаются в одном открытом проекте под названием Prevayler [Prevayler], а также в других аналогичных проектах. Это означает, что модель предметной области и базу данных можно в какой-то степени рассматривать как одно и то же. А для того чтобы это стало возможным, нам потребуются крупные объемы оперативной памяти, хотя в настоящее время это не проблема, поскольку цены на оперативную память постоянно падают, а 64-разрядные машины стали применяться повсеместно.

Основная идея состоит в том, что при каждом изменении в базе данных осуществляется соответствующая запись в журнале последовательной регистрации. При отсутствии электроэнергии в сети реализацию модели предметной области можно восстановить, прочитав ее копию, сохраненную в определенный момент времени, а также содержимое журнала регистрации. Это довольно простой и весьма эффективный стиль исполнения модели, поскольку все данные находятся в оперативной памяти.

По существу, мы имеем дело с объектной, но простой базой данных, поскольку отпадает необходимость выявлять и устранять неисправности и выполнять прочие типичные процедуры технического обслуживания базы данных. Все экземпляры постоянно находятся в оперативной памяти, и поэтому не требуется объектно-реляционное преобразование как в направлении реляционной (и даже объектной) базы данных, так и обратно. Остается только реализация модели предметной области.

Мы не станем усложнять ситуацию далее в этой книге из-за сервера приложений и будем считать, что мы работаем с клиентским приложением, имеющим расширенные функциональные возможности, включая модель предметной области или Web-приложение, т.е. речь опять-таки пойдет непосредственно о модели предметной области.

Такое решение объясняется, в частности, тем, что классический метод использования серверов приложений постепенно уходит в прошлое или, по крайней мере, меняется. А что же теперь актуально? Возможно, архитектура, ориентированная на службы (AOC), но это уже совсем другая тема. Впрочем, мы слегка затронем ее в главе 10.

Резюме

Эту главу можно было бы продолжать до бесконечности, но, к сожалению, ее придется заканчивать. В ней были рассмотрены вопросы, касающиеся “новой” используемой по умолчанию архитектуры и моделей предметной области, а также было намечено в общих чертах ее применение на одном примере приложения. А теперь настало время для более углубленного исследования модели предметной области и внесения многочисленных изменений в первоначальный набросок проекта, сделанный в этой главе.

Глава 5

Проектирование на основе модели предметной области

Возможно, некоторым читателям уже не терпится задать главный вопрос: “Не пора ли автору продемонстрировать хоть какие-нибудь инструменты и специальные приемы реализации модели предметной области и ее окружения?”

К сожалению, с этим придется повременить. Напротив, я собираюсь создать модель предметной области, даже не думая об инструментах и инфраструктуре, т.е. в старом стиле объектной ориентации, над которым я раньше так потешался, считая его несерьезным подходом к разработке настоящих приложений. Но, как вы уже знаете, теперь мое отношение к этому стилю совершенно изменилось — без него я не представляю себе проектирование.

Все, что до сих пор было изложено и представлено в этой книге, имело довольно общий и абстрактный характер грубых набросков моделей. Но теперь речь пойдет о том, как отдельные задачи проектирования решаются в действительности.

В этой главе мы рассмотрим вопросы разработки и уточнения модели предметной области, намеченной в предыдущей главе. Медленно двигаясь к цели и применяя РПТ, мы будем постепенно развивать эту модель. Поэтому приготовьтесь к неспешному обсуждению и продвижению вперед мелкими шагами. (В следующей главе мы изменим этот стиль обсуждения вопросов разработки приложений.)

Итак, начнем с основной темы этой главы: уточнения модели предметной области.

Уточнение модели предметной области простым экспериментированием с РПТ

Возможно, вы еще не забыли первый набросок модели, сделанный в главе 4? Он представлен вновь на рис. 5.1.

Делая этот набросок, я немного переусердствовал, слишком детализировав модель, чего я обычно не делаю. Так или иначе, сейчас самое время изучить более подробно требования к приложению. Начнем воплощать их в стиле РПТ и попутно выяснять вопросы, которые нам придется решать, развивая по ходу дела саму модель.

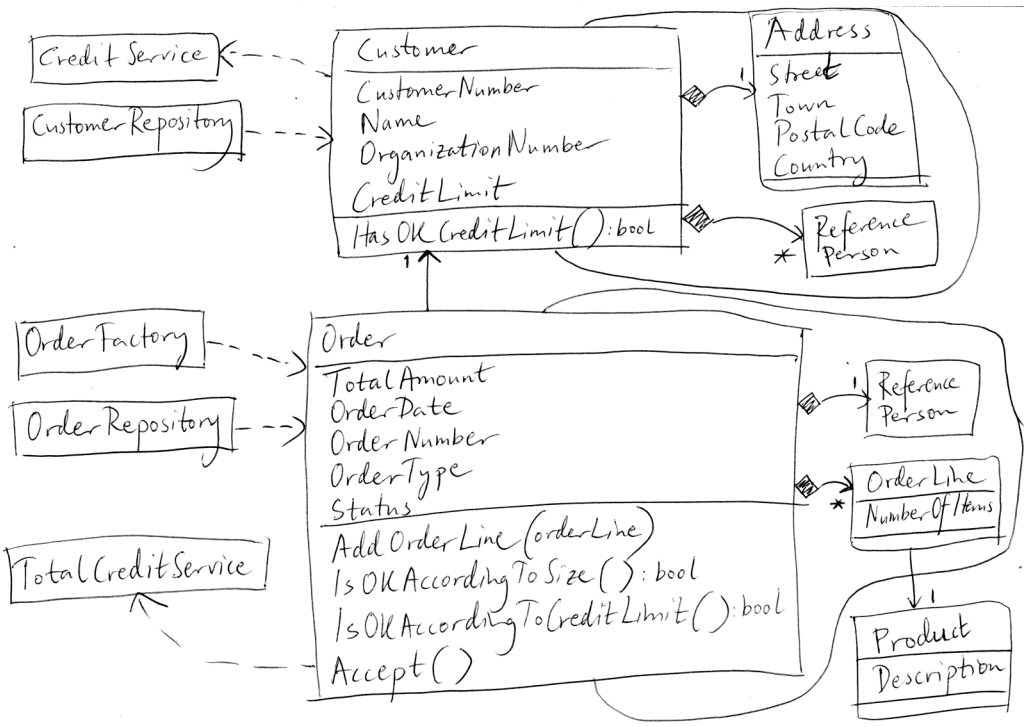


Рис. 5.1. Первый набросок модели предметной области, сделанный в главе 4

Как правило, я рассматриваю возможности приложения по очереди, но учитывая тот факт, что мы уже обсуждали наброски пользовательского интерфейса в предыдущей главе, сначала, на мой взгляд, целесообразно попытаться наполнить эти наброски определенными свойствами, а затем подытожить все рассмотренные свойства.

Первая задача: создание классов `Order` и `OrderFactory`

Прежде всего, создадим классы заказа (`Order`) и фабрики заказов (`OrderFactory`) и будем вести речь лишь о том, что показано на рис. 5.2, опуская остальные детали.

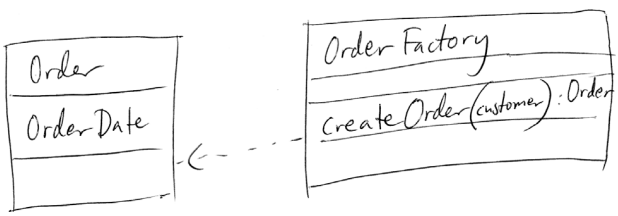


Рис. 5.2. Первая стадия реализации заключается в создании классов `Order` и `OrderFactory`

На заметку

Первые несколько шагов необходимы для правильного создания объектов предметной области. Их следует сделать перед тем, как перейти к самой интересной и важной стадии: тестированию и реализации логики с помощью построенной нами модели.

Итак, напишем первый тест для упомянутых выше классов. В частности, убедимся в том, что при обращении к фабрике заказов по поводу создания нового заказа не возвращается пустое значение. Этот тест может выглядеть следующим образом:

```
[Test]
public void CanCreateOrder()
{
  Order o = OrderFactory.CreateOrder(new Customer());

  Assert.IsNotNull(o);
}
```

На заметку

У вас может возникнуть вопрос: почему я называю свои тесты именно таким образом? Раньше тесты для инструмента xUnit должны были предваряться префиксом “Test”, но, например, для инструмента NUnit этого уже не требуется, поскольку он распознает атрибут `Test`. Это означает, что тесты можно именовать как угодно и сосредоточить основное внимание на самом проектировании и его концепциях, не вдаваясь в технические подробности.

Я лично предпочитаю именовать тесты в соответствии с их назначением. Таким образом, ряд названий тестов превращается в список обязательных и необязательных возможностей.

На первый взгляд, приведенный выше тест кажется очень простым. Для компиляции этого теста нужно сначала создать пустой класс `Order` (чего на данный момент должно быть достаточно), а затем — класс `OrderFactory` с одним методом. Но трудности начинают возникать уже при написании этого метода. Одно из правил, сформулированных в главе 4, гласило: у заказа всегда должен быть определенный клиент, а для этого следует возложить ответственность на класс `OrderFactory` предоставлять заказу (класс `Order`) клиента (класс `Customer`) во время его составления. Следовательно, метод `CreateOrder()` класса `OrderFactory` должен получить экземпляр объекта `Customer` в виде параметра (что и делается в данном тесте). Казалось бы, ничего сложного в этом нет, но наша задача — продолжать работать с агрегатом `Order` и пока что забыть обо всем остальном.

Первое решение, которое приходит на ум (и явствует из ранее написанного нами теста), состоит в создании пустого класса `Customer`. Ведь он все равно вскоре понадобится, а в данном тесте он не используется, и поэтому сильного связывания между стандартными блоками не возникнет — по крайней мере, в данный момент. С другой стороны, в этом первом небольшом тесте мы, похоже, затрагиваем слишком много вопросов, что зачастую чревато серьезными осложнениями. Поэтому постараемся избежать их.

В качестве второго решения, на мой взгляд, можно было бы сымитировать класс `Customer` и тем самым сделать связывание агрегатов слабым — по крайней мере, в данном тесте. Мне, безусловно, по душе идея имитаций, но в данном случае она кажется несколько излишней, поскольку требуется не взаимодействие с классом `Customer`, а всего лишь его заменитель.

В качестве третьего решения можно было бы создать интерфейс под названием `ICustomer` (или аналогичным названием либо сделать `Customer` интерфейсом), а затем реализовать этот интерфейс в виде заглушки. Но и это решение кажется мне в данный момент излишним, поскольку неясно, какой именно цели оно будет служить в дальнейшем. Заменить реализации `ICustomer`? Но это ли нужно нам на самом деле? Вряд ли. Поэтому от данного решения лучше отказаться сразу, пока оно действительно не оказалось неверным.

В качестве четвертого решения можно было бы временно опустить параметр `Customer`, выполнить проверку, а затем изменить тест при вводе данного параметра. Кроме того, общую ответственность за связывание заказа с его клиентом можно было бы не возлагать на фабрику. Но может быть это преждевременное проектное решение, что не замедлит выясниться в ходе РПТ буквально через несколько секунд?

Отказ от связывания заказа с его клиентом в фабрике фактически означает повышение гибкости конструкции пользовательского интерфейса, поэтому начинать с выбора клиента мне бы не хотелось. Вместо этого можно было бы предоставить пользователю возможность ввести другие данные о новом заказе, а затем попросить его указать соответствующего клиента. Но с другой стороны, гибкость в одном зачастую приводит к усложнению в другом. Допустим, что цены зависят от клиента. Какую цену следует выбрать при вводе строк заказа, если его клиент не определен? Что произойдет дальше? Разрешить это затруднение, безусловно, нельзя, поэтому мы не должны усложнять проект, если этого можно избежать.

Еще один недостаток четвертого решения заключается в том, что заказ, получаемый обратно из фабрики, на самом деле не находится в действительном состоянии. Упомянутое выше правило гласит: у каждого заказа должен быть клиент. Конечно, заказ не находится на фабрике в состоянии сохраняемости, но он по-прежнему не находится в действительном состоянии. Ведь потребитель должен что-то предпринять для перевода заказа в действительное состояние после выполнения кода фабрики.

В качестве пятого решения можно было бы указать тип `object` параметра для метода `CreateOrder()`, но это нецелесообразно и только усложняет приведение типов. Это же относится и к созданию некоего обобщенного интерфейса типа `IEntity` для реализации всех классов в модели предметной области, чтобы затем указать его

в виде параметра данного метода. Таким образом, все рассмотренные до сих пор решения сродни раскалыванию ореха кувалдой.

У вас может сложиться впечатление, будто я нарушаю здесь принципы РПТ, слишком много рассуждая, вместо того чтобы тестировать и реорганизовывать код. Но ведь когда вы придумываете название теста, в вашем уме подсознательно проносится немало идей и мыслей, а в целом — решений, что, безусловно, зависит от вашего опыта решения подобных задач.

Так или иначе, не будем затягивать наши рассуждения и выберем, наконец, подходящее решение — в данном случае первое из рассмотренных выше. Его нетрудно и просто проверить, введя пустой класс `Customer` с общедоступным по умолчанию конструктором, что мы, собственно, и сделаем. После компиляции тест не проходит (красная полоса), поскольку мы написали лишь сигнатуру и проверили только возвращение пустого значения из метода `CreateOrder()` в классе `OrderFactory`.

Теперь настало время реализовать метод `CreateOrder()`. Первый его вариант может выглядеть следующим образом:

```
// Класс OrderFactory
public static Order CreateOrder(Customer customer)
{
  return new Order();
}
```

Для компиляции этого кода необходим доступный по умолчанию конструктор класса `Order`. Он мог бы иметь тип `public`, но вместо этого присвоим ему тип `internal`, чтобы потребитель получал экземпляры объектов `Order` только средствами класса `OrderFactory`. Конечно, это далеко не идеальная защита, но, по крайней мере, непосредственное создание экземпляров объектов по ошибке будет происходить только внутри модели предметной области, а это, на мой взгляд, шаг в верном направлении.

Настало ли время для рефакторинга кода? Имеет ли фабрика какую-то ценность? Пока что никакой. Она лишь усложняет проект. На самом деле начинать следовало бы без фабрики, раз уж от нее никакого проку. Тем не менее, она может выполнять немало полезных функций, включая получение моментальной копии объекта `Customer`, ввод пустых объектов, составление разного рода заказов и многое другое.

Но код, написанный до сих пор вместе с фабрикой, выглядит немного нелепо. Фабрика может принести пользу в дальнейшем, но теперь ее следует упростить. Для этого выберем самую простую конструкцию, воспользовавшись непосредственно конструктором.

```
[Test]
public void CanCreateOrder()
{
  Order o = new Order(new Customer());

  Assert.IsNotNull(o);
}
```

На заметку

На первый взгляд, рефакторинг кода фабрики кажется необычной мерой. Но это служит явным признаком того, что мы несколько увлеклись и начали работу со слишком детального проектирования.

Следует также подчеркнуть, что в наброске, сделанном в главе 4, использован метод `CreateOrderLine()`, но пока что можно обойтись и без него, чтобы не усложнять дело.

Нужно ли писать другие тесты? Конечно, нужно. Ведь до сих речь шла о параметре `customer`, поэтому требуется, на мой взгляд, интересный тест для проверки клиента у вновь составленного заказа, чего мы пока еще не делали. Итак, составим подходящий для этой цели тест.

```
[Test]
public void CanCreateOrderWithCustomer()
{
  Order o = new Order(new Customer());

  Assert.IsNotNull(o.Customer);
}
```

Этот тест показывает, что нужно ввести поле или свойство `Customer` в класс `Order`. (Думаю, что оно понадобится не только для теста, но и для выполнения некоторых условий.)

Ради простоты введем в класс `Order` следующее свойство “только для чтения”:

```
// Класс Order
public readonly Customer Customer;
```

Если свойство `Customer` нельзя изменить в классе `Order`, то его можно считать повсеместным (и неизменным), что позволит упростить другие правила. Но о гибкости тогда говорить не приходится.

После компиляции кода тест дает положительный результат (красную полосу). Впрочем, обнаруженную ошибку нетрудно устранить, видоизменив конструктор, чтобы использовать в нем параметр `customer`. Теперь конструктор выглядит следующим образом (а тест дает отрицательный результат — зеленую полосу):

```
// Класс Order
public Order (Customer customer)
{
  Customer = customer;
}
```

Немного логики для предметной области

Как было показано на рис. 5.2, в класс `Order` следует также ввести свойство даты составления заказа `OrderDate`. Какая семантика может быть у этого свойства? Вероятно, заказу должна присваиваться начальная дата, когда он составляется в первый раз, а затем конечная дата, когда заказ переходит в состояние `Ordered` (Заказано) или аналогичное состояние. Выразим все это в тесте:

```
[Test]
public void OrderDateIsCurrentAfterCreation()
{
  DateTime theTimeBefore = DateTime.Now.AddMilliseconds(-1);

  Order o = new Order(new Customer());

  Assert.IsTrue(o.OrderDate > theTimeBefore);
  Assert.IsTrue(o.OrderDate
    < DateTime.Now.AddMilliseconds(1));
}
```

Назначение этого теста — задать определенный интервал моментов времени, а затем проверить значение свойства `OrderDate` в этом интервале.

Как обычно, такой код не компилируется сразу. Для этого придется ввести общедоступное свойство `OrderDate`. На этот раз мы воспользуемся закрытым полем, а также общедоступным методом выборки его значения, поскольку значение `OrderDate` придется впоследствии изменить. И на этот раз установим поле `OrderDate` в конструкторе класса `Order`, не вводя в конструктор еще один параметр. Ради ясности приведем ниже лишь ту часть кода, которая в данный момент интересует нас больше всего.

```
// Класс Order
private DateTime _orderDate;

public Order(Customer customer)
{
  Customer = customer;
  _orderDate = DateTime.Now;
}

public DateTime OrderDate
{
  get {return _orderDate;}
}
```

Теперь тест дает отрицательный результат (зеленую полосу).

Этот раздел я начал с упоминания блок-схемы, описывающей часть модели, приведенную на рис. 5.2. Как следует из схематического представления кода на рис. 5.3, модель получила некоторое развитие.

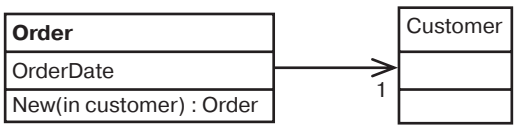


Рис. 5.3. Первая часть реализации модели: вид класса `Order` после правки

На заметку

У вас может возникнуть вопрос: зачем я рисую блок-схемы то от руки, то средствами UML? Дело в том, что общий замысел проще, удобнее и быстрее представить в виде наброска, сделанного от руки. А когда что-то уже реализовано, результат удобнее представить средствами UML.

Если сравнить рис. 5.2 и 5.3, то видимые отличия покажутся незначительными. Это лишь первый вариант реализации класса `Customer` и установления отношения между классами `Order` и `Customer`. Изменения произошли лишь в данной части блок-схемы, но не во всей модели в целом.

На заметку

На рис. 5.3 показан также конструктор, дающий более детальное представление о том, как действуют классы.

Прежде чем двигаться дальше, попробуем реорганизовать код. Сначала можно было бы переместить конструктор класса `Order` в область `[SetUp]`, а затем немного изменить в последнем тесте временной интервал, но толку от этого было бы мало. Более того, три теста, проведенных до сих пор, служат лишь целям создания классов, поэтому вызовы методов лучше оставить внутри самих тестов. Итак, отложим на время рефакторинг кода и перейдем к уточнению модели предметной области.

Вторая задача: создание класса `OrderRepository` и ввод свойства `OrderNumber`

В предыдущем разделе речь шла о том, как мог бы выглядеть класс `OrderFactory`, чтобы служить целям первоначального создания новых экземпляров класса `Order`, но затем было решено оставить до времени реализацию фабрики заказов. Теперь самое время разработать класс хранилища заказов (`OrderRepository`).

В главе 4 мы уже рассматривали метод `GetOrders()` класса `OrderRepository`. Думаю, что теперь больше подойдет типичный метод `GetOrder()`, поэтому начнем именно с него, а также с идентификатора объекта-сущности `Order`.

А что собой представляет свойство номера заказа — `OrderNumber`? Как правило, заказам присваиваются уникальные идентификационные номера для удобства их распознавания. Поэтому одно из распространенных решений заключается в том, чтобы просто присвоить новый номер каждому заказу из постоянно наращиваемой последовательности номеров. На рис. 5.4 вновь показан класс `Order` после модификации, а также вновь введенный метод в классе `OrderRepository`.

Нам бы следовало сначала рассмотреть класс `OrderRepository`, но в данный момент, на мой взгляд, важнее обсудить свойство `OrderNumber`. Так нужно ли присваивать заказу идентификационный номер при его составлении? Правильнее было бы сказать, что свойство `OrderNumber` должно иметь нулевое значение до тех пор, пока заказ не будет сохранен в первый раз (т.е. переведен в состояние `Ordered`). А до тех пор нецелесообразно расходовать понапрасну свойство `OrderNumber` на нечто, что еще только *собирается* стать настоящим заказом.

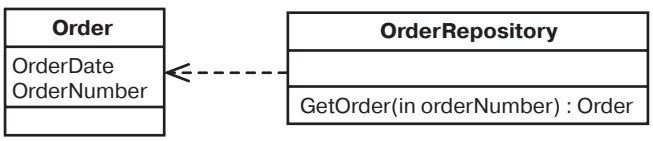


Рис. 5.4. Классы `Order` и `OrderRepository`

На заметку

Все, так или иначе, зависит от реальных требований, но меня, тем не менее, не покидает ощущение, что мы смешиваем здесь два разных понятия — идентификацию предметной области и идентификацию сохраняемости.

Кроме того, я не сторонник связывания сохраняемости с бизнес-правилами. Я имею в виду, что в тот момент, когда заказ переходит в состояние сохраняемости, он оказывается также в состоянии `Ordered`. Мы еще вернемся к данному вопросу далее в главе 7, когда настанет черед рассматривать бизнес-правила.

Итак, напишем, как обычно, следующий очень простой тест:

```
[Test]
public void OrderNumberIsZeroAfterCreation()
{
  Order o = new Order(new Customer());

  Assert.AreEqual(0, o.OrderNumber);
}
```

Возможно, вы уже догадались, что этот код не компилируется, поэтому нам придется ввести новое свойство "только для чтения" под названием `OrderNumber` в класс `Order` и предоставить возможность использовать в нем закрытое поле `_orderNumber`.

Это даст нам возможность обращаться к свойству `OrderNumber` из конструктора. Но если свойство `OrderNumber` — "только для чтения", как же задать его значение, используя класс `OrderRepository` для поиска и восстановления старого заказа? Для прояснения этого вопроса вернемся немного назад и напишем тест, а также метод `Setup()` (с целью объявить имя `_repository` хранилища заказов).

```
[SetUp]
public void SetUp()
{
  _repository = new OrderRepository();
}

[Test]
public void OrderNumberCantBeZeroAfterReconstitution()
{
  int theOrderNumber = 42;
  _FakeAnOrder(theOrderNumber);

  Order o =
  _repository.GetOrder(theOrderNumber));

  Assert.AreEqual(theOrderNumber, o.OrderNumber);
}
```

Как обычно, этот код не компилируется полностью. Ведь у нас пока еще нет класса `OrderRepository`, поэтому создадим его с сигнатурой для метода `GetOrder()`, но без соответствующего кода. Тестовый код по-прежнему компилируется, так что в тестовый класс придется вести метод-заглушку `_FakeAnOrder()`. В итоге тест дает положительный результат (красную полосу).

Похоже, что мы уходим все дальше от задачи, на которой нам бы следовало сосредоточиться. Ведь для того чтобы написать метод `_FakeAnOrder()`, нужно, чтобы в хранилище было известно о заказе. С этой целью можно было бы реализовать новый метод в классе `OrderRepository` только ради проведения других тестов, но в то же время это означает, что нам придется написать еще один тест. Ведь нужно проверить факт сохранения заказов или, по крайней мере, уведомить хранилище о заказах.

На заметку

Должен признать, что у меня имеется немало идей о том, как должно происходить сохранение заказов, но я все же собираюсь продвигаться вперед постепенно, создавая простой код, чтобы затем реорганизовать его. Это обстоятельство не должно вас разочаровывать, поскольку речь пока еще не идет об окончательном решении.

Итак, введем атрибут `Ignore` в тест `OrderNumberCantBeZeroAfterReconstitution()` — номер заказа не может быть нулевым после восстановления, а затем напишем еще один тест следующего содержания:

```
[Test]
public void CanAddOrder()
{
  _repository.AddOrder(new Order(new Customer()));
}
```

Как всегда, этот код не компилируется. Нам еще нужно ввести метод `AddOrder()` в класс `OrderRepository`, как обычно, в виде сигнатуры, хотя этого явно недостаточно, чтобы получить положительный результат данного теста (красную полосу). На самом деле в методе `CanAddOrder()` вообще отсутствует тестовый код. Дело в том, что у меня не вызывает особого энтузиазма идея предоставить хранилищу возможность публиковать любые методы только ради того, чтобы внешний тестовый класс мог проверить внутреннее состояние хранилища. Конечно, можно было бы воспользоваться методом `GetOrder()`, но тогда нам пришлось решать проблему, где трудно отделить причину от следствия. Ведь настоящая реализация данного метода еще и не начиналась.

Сделаем небольшое отступление и введем в хранилище закрытый список `IList` для хранения заказов, ничего не публикуя о списке `IList` вовне. Это всего лишь деталь реализации, хотя я уже подумываю о том, как бы поскорее от нее избавиться. Для проверки того, что действительно подлежит проверке, можно также воспользоваться еще одним утверждением, но не типа xUnit, а из пространства имен `Diagnostics`. В частности, с помощью метода `Assert()` проверяется истинность утверждения. Если оно не истинно, следует уведомить разработчика. Метод `Assert()` может иметь следующий вид:

```
// Класс OrderRepository
public void AddOrder(Order order)
{
  int theNumberOfOrdersBefore = _theOrders.Count;

  //  Выполнить ввод здесь...

  Trace.Assert(theNumberOfOrdersBefore
    == _theOrders.Count - 1);
}
```

Нужна ли очередная библиотека утверждений

Прежде чем пользоваться обыкновенными утверждениями пространства имен `Diagnostics`, следует подумать дважды. Во-первых, они не очень хорошо интегрируются в среду инструмента NUnit. И во-вторых, их нельзя настроить на

реагирование на разные ситуации, включая разработку, непрерывную интеграцию, бета-тестирование и эксплуатацию программ. Более подробно этот вопрос обсуждается в моей предыдущей книге [Nilsson NED].

Кроме того, у вас может возникнуть вопрос: почему я не проверил, может ли тест быть каким-то образом выражен в виде теста NUnit. Я, конечно, мог бы раскрыть то состояние, которое требуется тесту, но предпочитаю не делать этого, если в этом нет особой необходимости. В противном случае мне было бы труднее реорганизовывать класс хранилища, что, скорее всего, придется сделать.

Метод, использованный мной выше, навеян в какой-то степени книгой *Design By Contract* Бертрана Мейера [Meyer OOSC]. Несмотря на то что утверждение совсем не формализовано, оно выражает именно то, что гарантирует метод `AddOrder()`, т.е. выходные условия.

А какие же входные условия требуются для метода `AddOrder()`? Возможно, то, что заказ заранее неизвестен хранилищу? Нет, не очень приятно столкнуться с такой ошибкой. Тогда может быть непустое свойство `_theOrders`? Но ведь в этом случае данный метод сформирует исключение. Кроме того, использование свойства `_theOrders` может оказаться временным решением, в чем мы вскоре убедимся. Поэтому отложим на время рассмотрение данного вопроса.

Итак, создадим метод `MyTrace.Assert()`, формирующий исключение, если он воспринимает в качестве параметра ложное значение. Это, по крайней мере, позволит интегрировать тест в среду инструмента NUnit или Testdriven.net.

Скомпилируем код и проведем тест. Он дает положительный результат (красную полосу), что уже хорошо. А теперь заменим комментарий `Выполнить ввод здесь` в приведенном выше фрагменте кода на следующий вызов:

```
// Класс OrderRepository
public void AddOrder(Order order)
{
  int theNumberOfOrdersBefore +1 = _theOrders.Count;

  _theOrders.Add(order);

   MyTrace.Assert(theNumberOfOrdersBefore
    == _theOrders.Count);
}
```

Теперь тест дает отрицательный результат (зеленую полосу). Приведенный выше код может показаться несколько странным, но мы все же продвигаемся понемногу вперед, создавая по ходу дела тесты, которые сами по себе достаточно просты и понятны. Итак, продолжим.

На заметку

Для проверки работы тестируемой системы в предполагаемом режиме можно было бы также воспользоваться имитацией, но вы теперь и сами можете представить, как это будет происходить (см. главу 3).

Вернемся к тесту `OrderNumberCantBeZeroAfterReconstitution()` и методу-заглушке `_FakeAnOrder()`, использованному для тестирования. Соответствующий код будет выглядеть следующим образом:

```
// Тестовый класс
public void _FakeAnOrder(int orderNumber)
{
  Order o = new Order(new Customer());

  _repository.AddOrder(o);
}
```

При этом возникает следующее затруднение: как превратить имитируемый номер заказа (`OrderNumber`) в экземпляр заказа? Ведь свойство `OrderNumber` — "только для чтения", что ранее было вполне обоснованно, но теперь этим свойством нельзя воспользоваться.

На самом деле это довольно типичное затруднение. Его можно выразить следующим образом: как задать извне (например, в хранилище) значения в экземплярах объектов, восстанавливаемых из состояния сохраняемости?

Восстановление объекта-сущности из состояния сохраняемости: задание значений извне

Итак, выше было сформулировано типичное затруднение, связанное с заданием значений в экземпляре объекта, восстанавливаемом при чтении из базы данных. Что касается свойства `OrderNumber`, то это затруднение вполне очевидно, поскольку данное свойство вообще не изменяется, и потребителю приходится взаимодействовать с ним напрямую, но приблизительно то же самое происходит и с другими атрибутами. Допустим, что свойство `OrderDate` — для "чтения и записи". Если потребитель задает новую дату заказа, для этой цели могут потребоваться определенные проверки. Выполнять такие проверки, вероятно, нецелесообразно, когда свойству `OrderDate` присваивается значение в тот момент, когда объект `Order` считывается из базы данных и восстанавливается в виде экземпляра.

С этим затруднением приходится сталкиваться в самых разных ситуациях. Рассмотрим возможные способы разрешения данного затруднения.

Применение отдельного конструктора

Для упомянутого выше восстановления экземпляра объекта можно воспользоваться отдельным конструктором следующим образом:

```
// Класс Order
public Order(int orderNumber, DateTime orderDate,
  Customer customer)
```

Такой способ вполне работоспособен, но тогда потребителям объектов класса `Order` будет не совсем понятно, что они не должны пользоваться данным конструктором. В таком случае конструктор можно сделать внутренним, чтобы в значительной степени устранить затруднение!

Назначение конструктора можно сделать более понятным, если использовать статический именованный метод как фабрику следующим образом:

```
// Класс Order
public static Order ReconstituteFromPersistence(int orderNumber
  , DateTime orderDate, Customer customer)
```

Теперь назначение метода становится более очевидным, а кроме того, потребителю не придется иметь с ним дело. Тем не менее, потребитель может воспользоваться данным методом по ошибке, но теперь это уже будет внутренний метод. Когда класс становится более открытым для внешнего мира, возникает другое затруднение, если он, например, насчитывает около 50 свойств, которые приходится задавать при восстановлении. В этом случае список параметров данного метода становится слишком длинным и неудобным для обработки.

Применение отдельного метода в специальном интерфейсе

Еще один способ состоит в применении метода, который должен быть реализован в классе `Order`, причем этот класс должен быть доступен лишь посредством специального интерфейса. Таким образом, значительно снижается риск использования данного метода потребителем по ошибке. Конечно, потребитель может сделать это из злого умысла, но вероятность таких действий и соответствующая проблема существует всегда. Если же значения допускается задавать в экземпляре объекта извне, то же самое может сделать и потребитель. На самом деле это не так уж и плохо. Нужно лишь принять активные меры и разрешить такое использование, которое не вызвало бы никаких осложнений.

А как это будет выглядеть в коде? Ниже приведен один из вариантов реализации данного метода в классе `Order`:

```
//Класс Order
public void SetFieldWhenReconstitutingFromPersistence
  (int fieldKey, object value)
```

Такой код оказывается не очень удобным для сопровождения. Ведь теперь придется устанавливать множество ключей полей как таковые и как параметры метода `SetFieldWhenReconstitutingFromPersistence()`.

Сходное (но столь же неудобное для сопровождения) решение состоит в том, чтобы заменить параметр `fieldKey` параметром `fieldName`.

Применение рефлексии к закрытым полям

У применения рефлексии, как обычно, имеются свои достоинства и недостатки. К недостаткам следует отнести замедленность рефлексии (по сравнению хотя бы с обычным доступом), но насколько она медленнее? При этом невозможно закрыть доступ к привилегиям для пользователя, необходимо знать извне внутреннюю структуру, включая имена полей. (Последнее затруднение можно отчасти разрешить с помощью дополнительных закрытых свойств, предназначенных специально для рефлексии, или же правил именования, устанавливаемых таким образом, чтобы свойству под названием `Name` соответствовало закрытое поле под названием `_name`. Такой способ может оказаться надежным, хотя он и не устраняет данное затруднение полностью.)

Самое главное, что рефлексия не навязчива. Вы можете строить свою модель предметной области, не вводя в нее инфраструктуру и не скрывая конструкции.

> ***На заметку***
>
> Впрочем, у такого подхода имеются свои нюансы. Например, свойство `Customer` не может быть `readonly` (только для чтения). Мы еще вернемся к этому вопросу в последующих главах книги.

Иное решение

А теперь рассмотрим совершенно иное решение: постоянное сохранение значений вне экземпляра объекта в "надежном месте". В этом случае экземпляр объекта служит в качестве шаблона Proxy (Заменитель) [GoF Design Patterns]. Такое решение действительно заметно отличается от рассмотренных выше, хотя и у него имеются свои достоинства и недостатки. Но принимать столь серьезное, специфическое решение в данный момент не стоит. Оно привело бы к смешению классов модели предметной области с теми компонентами, которые не имеют никакого отношения к предметной области. Поэтому будем и дальше действовать как можно проще.

Решение задавать значения извне тесно связано с инфраструктурой, поэтому лучше будет отложить его до более подходящего момента, когда настанет пора подумать об инфраструктуре для поддержки построенной модели предметной области. С другой стороны, нужно принять какое-то решение, чтобы продолжить далее работу с тестами. Какой же самый простой механизм мог бы подойти теперь? К сожалению, найти действительно простое решение в данный момент не удается, а это явно указывает на необходимость вернуться назад и хорошенько обдумать все сначала.

На заметку

Начав разрабатывать семантику вокруг свойства `OrderNumber`, мы, по иронии судьбы, пришли к столь неутешительному выводу. Но раз уж мы к нему пришли, то пора заканчивать затянувшиеся рассуждения.

Итак, остановимся пока что на применении рефлексии к закрытым полям. Напишем следующий вспомогательный метод в классе `OrderRepository` (впоследствии его, скорее всего, придется вынести за пределы данного класса):

```
// Класс OrderRepository
public static void SetFieldWhenReconstitutingFromPersistence
  (object instance, string fieldName, object newValue)
{
  Type t = instance.GetType();
  System.Reflection.FieldInfo f = t.GetField(fieldName
    , BindingFlags.Instance | BindingFlags.Public
    | BindingFlags.NonPublic);
  f.SetValue(instance, newValue);
}
```

Теперь можно двигаться дальше. Данное решение потребовало длительных рассуждений из-за того, что свойство `OrderNumber` оказалось "только для чтения". На следует, однако, забывать, что это лишь самое общее решение о том, как предоставить хранилищам возможность задавать значения при восстановлении экземпляров из состояния сохраняемости, избегая осложнений, связанных со свойствами "только для чтения" и кодом их установки.

Вы еще помните тест `OrderNumberCantBeZeroAfterReconstitution()`? Теперь он повторяется в следующем виде:

```
[Test]
public void OrderNumberCantBeZeroAfterReconstitution()
{
  int theOrderNumber = 42;
  _FakeAnOrder(theOrderNumber);

  Order o =
  _repository.GetOrder(theOrderNumber);

  Assert.IsTrue(o.OrderNumber != 0);
}
```

Раньше в этом тесте был введен атрибут `Ignore`, чтобы он не выполнялся. А теперь этот атрибут удален, и тест дает положительный результат (красную полосу).

Далее внесем изменения в метод `_FakeAnOrder()`, чтобы новый заказ получил номер 42 следующим образом:

```
// Тестовый класс
public void _FakeAnOrder(int orderNumber)
{
  Order o = new Order(new Customer());

  OrderRepository.SetFieldWhenReconstitutingFromPersistence
    (o, "_orderNumber", orderNumber);

  _repository.AddOrder(o);
}
```

Тест по-прежнему не проходит, но на этот раз по иной причине. Боюсь, что мы создали ситуацию, когда для появления красных полос после выполнения данного теста имеется несколько причин, чего обычно рекомендуется избегать. Похоже, что мы совершили слишком большой и поспешный шаг, применяя РПТ.

Как бы там ни было, но дело в том, что мы не реализовали метод `OrderRepository.GetOrder()`. Это всего лишь пустой метод, возвращающий пустое значение. В данный момент все заказы находятся в закрытом списке `IList`, поэтому можно просмотреть его элементы циклически, проверив их по очереди следующим образом:

```
// Класс OrderRepository
public Order GetOrder(int orderNumber)
{
  foreach (Order o in _theOrders)
  {
    if (o.OrderNumber == orderNumber)
      return o;
  }
  return null;
}
```

Конечно, такая реализация данного метода может показаться наивной, но именно это теперь и требуется для того, чтобы тест прошел.

Итак, мы вновь обрели твердую почву под ногами, но все мое естество буквально взывает к рефакторингу кода в методе `SetFieldWhenReconstitutingFromPersistence()` — установка поля при воссоздании из состояния сохраняемости. Ведь этот метод не принадлежит классу `OrderRepository`. Поэтому создадим класс вспомогательного хранилища (`RepositoryHelper`) и перенесем в него метод `SetFieldWhenReconstitutingFromPersistence()`. Тест по-прежнему проходит — даже после того, как был изменен способ вызова данного метода.

А теперь посмотрим, как выглядит обновленная модель предметной области. Ее исправленная блок-схема приведена на рис. 5.5.

Меня по-прежнему не покидает ощущение, что модель в целом выглядит не так, как хотелось бы, но я не могу понять причину. Уверен, что она прояснится в дальнейшем, поэтому продолжим наше движение к поставленной цели. Какова же очередная задача? Вероятно, выборка списка заказов для определенного клиента.

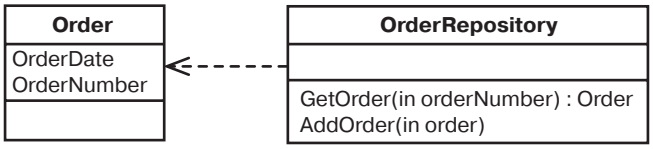


Рис. 5.5. Классы `Order` и `OrderRepository` после правки

Выборка списка заказов

Итак, рассмотрим в классе `OrderRepository` еще один метод под названием `GetOrders()`. Для начала он должен воспринимать объект клиента в качестве параметра. После обнаружения ряда заказов вторая часть данной задачи состоит в отображении значений некоторых свойств, включая `OrderNumber` и `OrderDate`. На рис. 5.6 приведена блок-схема, показывающая, что именно следует обсудить и создать в этом разделе.

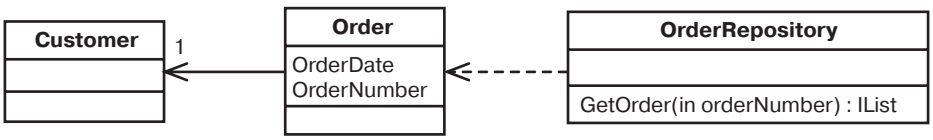


Рис. 5.6. Классы `Order`, `Customer` и `OrderRepository` для перечисления заказов по списку

На заметку

Обратите внимание на иной способ представления множественности в UML-схеме, приведенной на рис. 5.6, по сравнению с ее выражением в блок-схеме базы данных. Если вы привыкли пользоваться блок-схемами баз данных, то для вас могут оказаться не совсем привычными такие отличия в UML-схеме, как, например, отсутствие отношения, прослеживаемого от класса `Customer` к классу `Order`. В данном случае прослеживается отношение от класса `Order` к классу `Customer` с множественностью "один к одному".

А теперь перейдем к очередному тесту. И в этом случае нам потребуется сымитировать заказы в классе `OrderRepository` для проведения следующего теста:

```
[Test]
public void CanFindOrdersViaCustomer()
{
  Customer c = new Customer();

  _FakeAnOrder(42, c, _repository);
  _FakeAnOrder(12, new Customer(), _repository);
  _FakeAnOrder(3, c, _repository);
```

```
  _FakeAnOrder(21, c, _repository);
  _FakeAnOrder(1, new Customer(), _repository);

  Assert.AreEqual(3, _repository.GetOrders(c).Count);
}
```

Как видите, в этом тесте внесены изменения в вызов метода `_FakeAnOrder()`, и теперь он воспринимает объекты классов `Customer` и `OrderRepository` в виде параметров. Это, конечно, простые изменения. Для компиляции данного теста требуется еще одно простое изменение — добавить метод `GetOrders()`. Это изменение будет выглядеть следующим образом (разумеется, после обязательного выполнения теста с получением положительного результата — красной полосы):

```
// Класс OrderRepository
public IList GetOrders(Customer customer)
{
  IList theResult = new ArrayList();

  foreach (Order o in _theOrders)
  {
    if (o.Customer.Equals(customer))
      theResult.Add(o);
  }
  return theResult;
}
```

Не уверен, что мне по душе приведенный выше фрагмент кода. Я имею в виду проверку равенства. Допустим, что имеются два экземпляра объектов класса `Customer`, т.е. не две переменные, указывающие на один и тот же экземпляр, а два отдельных экземпляра объекта, причем с одним и тем же именем `CustomerNumber`. Что же будет возвращено при вызове метода `Equals()`: ложное или истинное логическое значение? Это зависит от следующего: был ли переопределен метод `Equals()` и используется ли объект `CustomerNumber` в переопределенной реализации этого метода при проверке равенства.

На заметку

Читатели, знакомые с шаблоном Identity Map (Карта идентичности) [Fowler PoEAA] и считающие его применение обязательным, могут спросить: зачем вообще нужна вся эта дискуссия? Ведь назначение шаблона Identity Map — автоматически исключить появление двух отдельных экземпляров одного объекта клиента.

На уровне языка программирования, например, C#, это не делается автоматически. Данный вопрос будет затронут при рассмотрении решений сохраняемости в главах 8 и 9. А до тех пор будем двигаться своим путем.

Пора обсудить объекты-сущности

Данная тема уже не раз освещалась в этой главе, поскольку объекты классов `Order` и `Customer` служат характерными примерами объектов-сущностей, тем не менее, уделим этому понятию немного внимания.

Очень важно периодически отслеживать некоторые объекты в предметной области. Ведь независимо от изменения имени, фамилии и адреса клиент по-прежнему остается клиентом, и мы должны его отслеживать. С другой стороны, если меняется ответственное лицо клиента, то отслеживать его, вероятно, не обязательно. В данном случае клиент является типичным примером объекта-сущности и соответствующего шаблона Entity [Evans DDD]. Опять же, объект-сущность — это объект, отслеживаемый по тождественности, а не по всем его значениям.

В качестве характерного примера объекта, не являющегося сущностью, рассмотрим целое число 42. Обращаю ваше внимание не на тождественность числа 42, а на его значение. Когда это значение меняется, мы не думаем от том, что оно изменилось и уже не равно 42, поскольку это совершенно новое значение, никак не связанное со старым. Число 42 относится, скорее, к объекту-значению и соответствующему шаблону Value Object [Evans DDD], чем к объекту-сущности.

Если же применить этот характерный пример к нашей предметной области, следует отслеживать ответственное лицо (`ReferencePerson`) клиента (`Customer`) не по тождественности, а по значению. К объектам-значениям мы еще вернемся далее в этой главе.

На заметку

Разумеется, отслеживание одних объектов по тождественности, а других — по значению зависит от предметной области. Обратимся к примеру. Как уже отмечалось, с ответственным лицом (`ReferencePerson`) следует обращаться в рассматриваемом здесь приложении как с объектом-значением (Value Object), но если приложение предназначено, скажем, для оказания помощи агентам по продажам, они, скорее всего, будут рассматривать ответственное лицо как объект-сущность (Entity).

Назад к основному ходу рассуждений

К сожалению, тест `CanFindOrdersViaCustomer()` — поиск заказов по клиенту — не выполняется успешно после переопределения метода `Equals()`. Дело в том, что у всех клиентов, используемых в данном тесте, имеется один и тот же номер (`CustomerNumber`), равный нулю, и поэтому по искомому клиенту в итоге обнаруживаются пять, а не три заказа. В связи с этим нужно немного изменить тест. Для этого создадим небольшой вспомогательный метод под названием `_FakeACustomer()` следующим образом:

```
// Тестовый класс
private Customer _FakeACustomer(int customerNumber)
{
  Customer c = new Customer();

  RepositoryHelper.SetFieldWhenReconstitutingFromPersistence
    (c, "_customerNumber", customerNumber);

  return c;
}
```

На заметку

Метод `_FakeACustomer()` должен связывать клиента с классом `CustomerRepository` (а, скорее, — с шаблоном Unit of Work, когда он вступает в действие). Однако не будем углубляться в этот вопрос, чтобы не отвлекаться от основного хода рассуждений, для которых он пока что не так важен.

Далее внесем изменения в тест `CanFindOrdersViaCustomer()`, чтобы воспользоваться методом `_FakeACustomer()` вместо непосредственного обращения к конструктору класса `Customer`. После этих изменений код данного теста примет следующий вид:

```
[Test]
public void CanFindOrdersViaCustomer()
{
  Customer c = _FakeACustomer(7);

  _FakeAnOrder(42, c, _repository);
  _FakeAnOrder(12, _FakeACustomer(1), _repository);
  _FakeAnOrder(3, c, _repository);
  _FakeAnOrder(21, c, _repository);
  _FakeAnOrder(1, _FakeACustomer(2), _repository);

  Assert.AreEqual(3, _repository.GetOrders(c).Count);
}
```

Теперь тест дает отрицательный результат (зеленую полосу).

Итак, подведем итог всем приведенным выше рассуждениям, представив полученные результаты в виде блок-схемы. По сравнению со схемой на рис. 5.6 было введено свойство `CustomerNumber` в класс `Customer`, как показано на рис. 5.7.

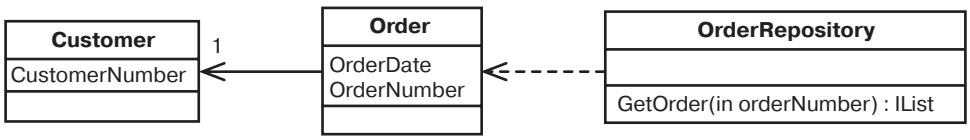


Рис. 5.7. Классы `Order`, `Customer` и `OrderRepository` для перечисления заказов по списку после правки

Общая перспектива

Чего же мы достигли до сих пор? Для разработчика, много лет проработавшего, подобно мне, с базами данных и ориентированными на них приложениями, полученный до сих пор код выглядит не совсем привычно, если попытаться его проанализировать. В частности, в нем вообще отсутствуют обращения к базе данных. Конечно, было бы неплохо ослабить основное внимание на модели предметной области и продолжить создание тестов, но в то же время хотелось бы писать тесты и далее, не касаясь базы данных.

Что же дали нам проведенные до сих пор тесты? Прежде всего, мы пользовались тестами как средствами обнаружения и выявления определенного поведения и описания для реализации модели предметной области. Начав с поддельной и наивной реализации, мы сумели сосредоточить свои усилия на разработке интерфейса API. Кроме того, мы попытались выработать и опробовать универсальный язык. И наконец, наши тесты заложили прочное основание для переноса кода из наивной реализации в нечто более реальное.

Следует также отметить, что при выборе проектных решений мы постоянно учитывали конечную цель проектирования. Кроме того, мы создали уровень абстракции в форме хранилищ, чтобы отложить до более удобного момента решение вопросов, связанных с базой данных (что, безусловно, стоило сделать).

Таким образом, мы создали первый, поддельный вариант класса `OrderRepository` и наделили его несколькими различными полномочиями. Возможно, именно этим объясняется некоторое неудобство такого стиля проектирования. Ведь мы до сих пор старались вообще не уделять внимания инфраструктуре, отложив ее рассмотрение до более удобного момента. Но поскольку работа над инфраструктурой в модели предметной области уже начата, отложим на время список возможностей и перейдем к рефакторингу кода, чтобы получить более изящное решение с поддельным классом `OrderRepository`.

На заметку

В этой книге часто употребляются термины “заглушка”, “имитация” и “подделка”. Чтобы разъяснить их, приведу выдержки из книги *Test-Driven Design* [Astels TDD].

“**Заглушка** — класс с методами, которые ничего не делают. Он просто служит для нормальной компиляции и выполнения кода системы.

Подделка — класс с методами, возвращающими фиксированные значения, которые могут быть жестко запрограммированы или заданы программно.

Имитация — класс, в котором могут быть заданы предположения относительно вызываемых методов, передаваемых параметров, частоты вызова методов и прочего. Кроме того, для разных случаев вызова методов могут быть заданы возвращаемые значения. Имитация предоставляет также возможность проверить, насколько оправдались предположения”.

Но это не единственное определение. В статье Месароша [Meszaros XUnit] дается несколько иное определение этих понятий:

"**Тестовая заглушка** — объект, используемый в тесте для замены настоящего компонента, от которого зависит работа тестируемой системы, благодаря чему тест позволяет управлять косвенным вводом данных в тестируемой системе, вынуждая ее работать в нестандартных режимах.

Поддельный объект — объект, заменяющий в тесте функции настоящего компонента, от которого зависит работа тестируемой системы, для проверки других режимов ее работы, кроме косвенного ввода-вывода. Как правило, он реализует те же самые функции или их часть, что и у настоящего компонента, от которого зависит работа тестируемой системы, но в намного более упрощенном виде.

Имитирующий объект — объект, используемый в тесте для замены настоящего компонента, от которого зависит работа тестируемой системы, благодаря чему тест позволяет наблюдать косвенный вывод. Как правило, имитирующий объект подделывает реализацию, возвращая жестко закодированные результаты, предварительно загруженные в ходе теста".

Подделка класса `OrderRepository`

Первой моей мыслью было подделать некоторые внутренние составляющие класса `OrderRepository`, чтобы использовать один и тот же класс `OrderRepository` в разных ситуациях и получать экземпляры таких объектов, как `DataFetcher` (Средство выборки данных), в конструкторе, обеспечивающем правильную последовательность действий. Но по зрелом размышлении я пришел к выводу, что это было бы чрезмерно. Ведь если позволить классу `OrderRepository` взаимодействовать с реляционной базой данных вместо его подделки, то пришлось бы заменить едва ли не весь класс `OrderRepository`.

Вероятно, для решения данной задачи потребуется интерфейс, схематически показанный на рис. 5.8.

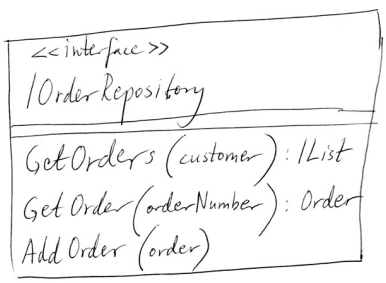


Рис. 5.8. Интерфейс `IOrderRepository`, первый предложенный вариант

Подсознательно этот вариант мне очень нравится, возможно, потому что я немало унаследовал от среды COM или потому что многие авторы, в том числе [Szyperski Component Software], [Johnson J2EE Development without EJB] и [Löwy Programming .NET Components], ясно заявляют, что программирование по интерфейсам более предпочтительно, чем программирование по конкретным классам.

Тем не менее, в сигнатурах интерфейса по-прежнему присутствуют конкретные классы (`Customer` и `Order`), о чем уже говорилось выше. Впрочем, оставим пока что все как есть, поскольку я в данный момент не вижу никакой пользы в создании интерфейсов для этих классов.

На ум приходит еще одна мысль — создать более общий интерфейс хранилища. Он мог бы выглядеть так, как показано на рис. 5.9.

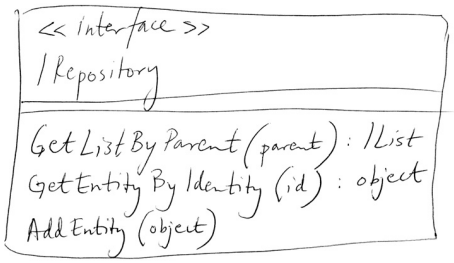


Рис. 5.9. Интерфейс `IOrderRepository` (или, скорее, `IRepository`), второй предложенный вариант

Этот вариант мог бы оказаться весьма привлекательным, но в данный момент я отдаю предпочтение более конкретному интерфейсу, приведенному на рис. 5.8, поскольку он ясно показывает, что нам нужно. А при использовании общего интерфейса `IRepository` требуется уделять больше внимания приведению типов. Кроме того, данный вариант может превратиться в неудачную попытку свести к единой форме все последующие хранилища.

Еще одно решение может состоять в использовании обобщений в интерфейсе `IRepository` и базового класса, чтобы таким образом добиться хотя бы типовой безопасности (и исключить дублирование кода), а также избежать дополнительного приведения типов. Но тогда в класс хранилища можно было бы ввести метод `Delete()`, хотя это требуется не всем хранилищам. Впрочем, даже при наличии обобщений метод `GetListByParent(parent)` вызывает трудности из-за того, что у объекта-сущности может быть несколько родителей. Конечно, это лишь первые наметки, и найти подходящее решение все же можно. Но я не могу отделаться от мысли, что было бы лучше сделать код как можно более ясным и понятным и не прибегать в подобным обобщениям. Впрочем, в данный момент важно принять конкретное решение. Ведь это решение не принимается раз и навсегда и может быть откорректировано в дальнейшем, когда станет яснее, какие именно составляющие могут быть обобщены. Поэтому остановимся пока что на интерфейсе `IOrderRepository`.

В связи с этим переименуем класс `OrderRepository` в `OrderRepositoryFake` (Подделка хранилища заказов) и реализуем в нем интерфейс `IOrderRepository`. Кроме того, нам нужно изменить код в области `[SetUp]` тестового класса. В данный момент он имеет следующий вид:

```
// Область[SetUp] тестового класса

_repository = new OrderRepository();
```

Но может принять такой вид:

```
// Область[SetUp] тестового класса

_repository = new OrderRepositoryFake();
```

В объявлении переменной `_repository` необходимо указывать интерфейс `IOrderRepository` в качестве типа. Тем не менее тест выполняется.

Предвижу протесты пуристов экстремального программирования в связи с проведенным выше рефакторингом кода. На самом деле вносить эти изменения теперь не следовало, но они нужны на будущее, когда потребуется настоящее хранилище вместо поддельного. Согласен, что это не совсем корректно, но ведь нужно подумать немного и о будущем, когда модель придется привязывать к инфраструктуре (см. последующие главы).

Будем надеяться, что такая забота о будущем себя оправдает. Это можно, конечно, отнести к моим старым привычкам делать ранние догадки, но с другой стороны, мы вольны выбрать *собственное*, *взвешенное* решение, а не придерживаться строго догмы ЭП!

Несколько слов о сохранении

Ранее уже упоминалось о сохранении данных. В частности, метод `AddOrder()` на самом деле ничего не сохраняет, а только вводит заказ в хранилище. Должно ли это также означать сохранение? Нет, такое поведение, на мой взгляд, нежелательно. От метода `AddOrder()` требуется, чтобы хранилищу (и базовой инфраструктуре) с этого момента было известно об экземпляре объекта заказа и чтобы в нем были предприняты соответствующие действия при вызове метода `PersistAll()`.

Для этой цели инфраструктура должна быть наделена функциями сохранения. Но в данный момент требуется ответить на следующий вопрос: следует ли передать вызов метода `PersistAll()` экземпляру хранилища или же какому-нибудь другому объекту более высокого уровня, контролирующему экземпляры всех хранилищ? И в этом случае мы несколько отклоняемся от модели предметной области в сторону инфраструктуры. Хорошо, конечно, что данный вопрос возник теперь, но его решение придется частично отложить до следующей главы, где он будет рассмотрен более подробно. А до тех пор нас вполне устроит метод `AddOrder()` как средство уведомления хранилища об экземпляре объекта заказа. Что же касается написанных до сих

пор тестов, то для них не важно, что в данный момент экземпляр объекта заказа является временным (и не подготовленным к сохранению), а не постоянным.

Итак, я, как и вы, горю желанием написать дополнительные тесты. Что же у нас на очереди? Нам придется решать задачу номер 2 "Список заказов, формируемый при просмотре отдельного клиента" из списка, составленного в предыдущей главе. Итак, предстоит решить вопросы вывода общей суммы каждого заказа, вида заказа и ответственного лица.

Рассмотрим их по очереди, начиная с общей суммы каждого заказа.

Общая сумма каждого заказа

На первый взгляд, задача тривиальна. Сначала нужно создать класс `OrderLine`, чтобы привязать его к классу `Order` в виде списка, а затем просмотреть циклически всю совокупность объектов строк заказа (`OrderLine`) и рассчитать общую сумму. Это можно было бы сделать в коде потребителя, но, на мой взгляд, — нежелательно и нецелесообразно, поскольку раскрывает потребителю весь алгоритм расчета. В действительности этот алгоритм будет усложняться по мере его составления, например, в результате ввода разного рода скидок. Вместо этого воспользуемся свойством `TotalAmount` "только для чтения" в классе `Order`. И в этом случае можно было бы просмотреть циклически всю совокупность объектов `OrderLine` и рассчитать общую сумму.

Однако это было бы слишком для меня как сторонника преждевременной оптимизации, хотя я и пытаюсь всеми силами побороть в себе данный порок. Ведь получение экземпляров всех объектов `OrderLine` для каждого заказа потребителя только для того, чтобы вывести общую сумму каждого заказа, может стать тревожным звонком с точки зрения производительности системы. Впрочем, все зависит от конкретной среды выполнения и выбранной инфраструктуры. С другой стороны, это серьезная проблема для распределенной системы, где она может отрицательно сказаться на работе сервера базы данных, сети и программы очистки памяти (так называемого "сборщика мусора").

Честно говоря, в данный момент мне бы не хотелось заниматься оптимизацией этой задачи. Ее можно решить непосредственно, и поскольку речь идет, главным образом, о построении модели предметной области, это не так важно. Теперь намного важнее стремиться к простоте и ясности проектных решений, а впоследствии можно вернуться к данной задаче и оценить ее реальное влияние на производительность системы с помощью программы-профилировщика. Итак, начнем с самого простого решения, которое затем можно откорректировать, реорганизовав код, связанный со свойством `TotalAmount`, если характеристики производительности системы окажутся не на высоте. Тем самым, нам удастся избежать оптимизации на данном этапе, что не так уж и плохо.

А теперь, когда мы обсудили свойство `TotalAmount`, перейдем к написанию теста (как говорится: лучше позже, чем никогда). Начнем с самого простого теста следующего содержания:

```
[Test]
public void EmptyOrderHasZeroForTotalAmount()
{
  Order o = new Order(new Customer());

  Assert.AreEqual(0, o.TotalAmount);
}
```

Этот тест компилируется очень просто. В нем лишь вводится свойство `TotalAmount` "только для чтения" в класс `Order` и затем ему предоставляется возможность возвратить –1. После компиляции этот тест дает положительный результат (красную полосу). Если внести в тест изменения, чтобы вместо –1 возвращался 0, он даст отрицательный результат (зеленую полосу).

На этом можно было бы остановиться, но раз уж мы находимся в данном контексте, сделаем еще один небольшой шаг, написав следующий тест:

```
[Test]
public void OrderWithLinesHasTotalAmount()
{
  Order o = new Order(new Customer());

  OrderLine ol = new OrderLine(new Product("Chair", 52.00));
  ol.NoOfUnits = 2;

  o.AddOrderLine(ol);

  Assert.AreEqual(104.00, o.TotalAmount);
}
```

В этом тесте принято немало проектных решений, и теперь для его компиляции нужно создать два класса и новые методы в старых классах. Для начала рассмотрим этот тест, сосредоточив основное внимание на классе `OrderLine`.

Прежде всего, необходимо извлечь по умолчанию цену из выбранной продукции в строку заказа. Для этого напишем следующий тест:

```
[Test]
public void OrderLineGetsDefaultPrice()
{
  Product p = new Product("Chair", 52.00);

  OrderLine ol = new OrderLine(p);

  Assert.AreEqual(52.00, ol.Price);
}
```

Столь значительный шаг требует написания некоторого кода, но поскольку этот шаг сделан уверенно, перейдем к созданию класса `Product`. Вторым аргументом в его конструкторе должна быть цена единицы продукции. Кроме того, у класса

Product должны быть два свойства “только для чтения”: Description (Описание) и UnitPrice (Цена единицы продукции).

Далее создадим класс OrderLine с двумя членами: Product и Price. Он принимает следующий вид:

```
public class OrderLine
{
  public decimal Price = 0;
  private Product _product;

  public OrderLine(Product product)
  {
    _product = product;
  }

  public Product Product
  {
    get {return _product;}
  }
}
```

Тест компилируется, но дает положительный результат (красную полосу). В данном случае нужно изменить конструктор, чтобы извлечь цену из объекта Product и поместить ее в объект OrderLine. Теперь конструктор принимает следующий вид:

```
// Класс OrderLine
public OrderLine(Product product)
{
  _product = product;
  Price = product.UnitPrice;
}
```

На этот раз тест дает отрицательный результат (зеленую полосу). Можно было бы написать тест, проверяющий переопределение цены в объекте OrderLine, но это не очень интересно, поэтому оставим этот вопрос без внимания.

Следующий тест следует написать для проверки расчета общей суммы в одной строке заказа. Этот тест может иметь такой вид:

```
[Test]
public void OrderLineHasTotalAmount()
{
  OrderLine ol = new OrderLine(new Product("Chair", 52.00));
  ol.NumberOfUnits = 2;

  Assert.AreEqual(104.00, ol.TotalAmount);
}
```

Здесь приняты еще два проектных решения. В классе OrderLine требуется поле NumberOfUnits (Количество единиц) и свойство TotalAmount. Поле NumberOfUnits

должно быть общедоступным, свойство TotalAmount — "только для чтения" и с возвращением 0. В итоге тест компилируется, но дает положительный результат (красную полосу). Далее внесем следующие изменения в свойство TotalAmount:

```
// Класс OrderLine
public decimal TotalAmount
{
  get {return Price * NumberOfUnits;}
}
```

На этот раз тест дает отрицательный результат (зеленую полосу).

Итак, посмотрим, чего же мы только что добились. На рис. 5.10 приведена блок-схема модели с классами OrderLine и Product.

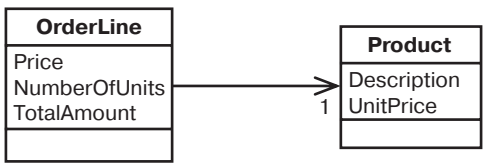


Рис. 5.10. Классы OrderLine и Product

А теперь вернемся к тесту, прокомментированному выше. На этот раз он немного расширен для проверки двух строк заказа.

```
[Test]
public void OrderWithLinesHasTotalAmount()
{
  Order o = new Order(new Customer());

  OrderLine ol = new OrderLine(new Product("Chair", 52.00));
  ol.NoOfUnits = 2;
  o.AddOrderLine(ol);

  OrderLine ol2 = new OrderLine(new Product("Desk", 115.00));
  ol2.NoOfUnits = 3;
  o.AddOrderLine(ol2);

Assert.AreEqual(104.00 + 345.00, o.TotalAmount);
}
```

Для компиляции этого теста необходимо ввести в класс Order метод AddOrderLine() и свойство TotalAmount. В данный момент свойство TotalAmount должно возвращать 0. Кроме того, следует установить атрибут Ignore в тесте OrderWithLinesHasTotalAmount() — заказ со строками имеет общую сумму. Но вместо этого напишем другой тест, в котором основное внимание уделяется методу AddOrderLine(). Этот тест имеет следующий вид:

```
[Test]
public void CanAddOrderLine()
{
  Order o = new Order(new Customer());

  OrderLine ol = new OrderLine(new Product("Chair", 52.00));
  o.AddOrderLine(ol);

  Assert.AreEqual(1, o.OrderLines.Count);
}
```

Для компиляции данного теста требуется метод `AddOrderLine()` и свойство `OrderLines`. А поскольку этот тест не проходит вполне обоснованно, если не действует метод `AddOrderLine()` или свойство `OrderLines`, начнем с написания кода для свойства `OrderLines`, оставив незавершенным метод `AddOrderLine()`. Ниже приведен соответствующий код.

```
// Класс Order
private IList _orderLines = new ArrayList();

public IList OrderLines
{
  get {return ArrayList.ReadOnly(_orderLines);}
}

public void AddOrderLine(OrderLine orderLine)
{
}
```

Тест компилируется, но дает положительный результат (красную полосу). Введем строку кода в метод `AddOrderLine()` следующим образом:

```
// Класс Order
public void AddOrderLine(OrderLine orderLine)
{
  _orderLines.Add(orderLine);
}
```

На этот раз тест компилируется и дает отрицательный результат (зеленую полосу).

А теперь можно удалить атрибут `Ignore` из теста `OrderWithLinesHasTotalAmount()`. Тест дает положительный результат (красную полосу), поскольку свойство `TotalAmount` класса `Order` возвращает 0.

Перепишем код для свойства `TotalAmount` следующим образом:

```
// Класс Order
public decimal TotalAmount
{
  get
  {
```

```
    decimal theSum = 0;
    foreach (OrderLine ol in _orderLines)
      theSum += ol.TotalAmount;

    return theSum;
  }
}
```

Итак, мы получили простой тест, дающий отрицательный результат (зеленую полосу). Возможно, вы обратили внимание, как стойко я противился искушению выполнить оптимизацию? Код получился — проще не придумаешь, причем без каких-либо сверхусилий. По крайней мере, теперь ясно видно, что заказ и его строки составляют агрегат, а значит, *используемую по умолчанию* загружаемую структурную единицу.

Сведения о предыстории клиента

Теперь настало время для рефакторинга кода. Что же в нем недоброкачественного? Мне лично не очень нравится отношение классов `Order` и `Customer`. Ясно, что у заказа должен быть клиент, но если вернуться к заказу через год, то, скорее, всего понадобятся сведения о клиенте на тот момент, когда был составлен заказ, а не на текущий момент.

Кроме того, объект `Customer` передается конструктору в виде параметра. Это означает, что объект `Order` может проникнуть непосредственно в граф постоянного объекта, не вызывая метод `IOrderRepository.AddOrder()`, что не очень хорошо. (Это, конечно, зависит и от конкретной инфраструктуры, хотя не совсем ясно, существует ли вообще какая-то граница.) С помощью метода `AddOrder()` можно указать, что данный заказ должен стать постоянным объектом, когда придет время его сохранить.

Все эти затруднения можно так или иначе разрешить, но как это сделать ясно? Опробуем один предлагаемый вариант, написав следующий тест:

```
[Test]
public void OrderHasSnapshotOfRealCustomer()
{
  Customer c = new Customer();
  c.Name = "Volvo";

  Customer aHistoricCustomer = c.TakeSnapshot();

  Order o = new Order(aHistoricCustomer);

  c.Name = "Saab";

  Assert.AreEqual("Saab", c.Name);
  Assert.AreEqual("Volvo", o.Customer.Name);
}
```

Не уверен, что это удачное решение. В настоящий момент `Customer` представляет собой небольшой объект, но вполне возможно, что он разрастется. Но несмотря на то что он мал, было бы интересно проследить ответственные лица, которые были у клиента в отдельные моменты времени.

На мой взгляд, имеет смысл создать объект другого типа только с теми свойствами, которые могут представлять определенный интерес, потому что в этом случае образуется также явная граница между агрегатами, в особенности потому что именно на это указывает базовая модель. Итак, создадим объект `CustomerSnapshot` (Моментальный снимок клиента), вдохновившись идеей, изложенной в [Fowler Snapshot], — его назначение аналогично, но реализация совсем другая. Пусть это будет объект-значение с минимальным числом свойств. Для этого нужно лишь немного переделать тест (после исправления двух ошибок компиляции: типа для свойства `Customer` и параметра для конструктора).

```
[Test]
public void OrderHasSnapshotOfRealCustomer()
{
  Customer c = new Customer();
  c.Name = "Volvo";

  CustomerSnapshot aHistoricCustomer = c.TakeSnapshot();

  Order o = new Order(aHistoricCustomer);

  c.Name = "Saab";

  Assert.AreEqual("Saab", c.Name);
  Assert.AreEqual("Volvo", o.Customer.Name);
}
```

Рассмотрим далее следующий вопрос: следует ли возлагать ответственность за создание моментального снимка клиента на потребителя или же на конструктор класса `Order`? Раньше эта ответственность возлагалась на потребителя. А теперь возложим ее вновь на конструктор, чтобы он воспринимал экземпляр объекта `Customer` в виде параметра следующим образом:

```
[Test]
public void OrderHasSnapshotOfRealCustomer()
{
  Customer c = new Customer();
  c.Name = "Volvo";

  Order o = new Order(c);

  c.Name = "Saab";

  Assert.AreEqual("Saab", c.Name);
```

```
  Assert.AreEqual("Volvo", o.Customer.Name);
}
```

Конструктор класса `Order` имеет следующий вид:

```
// Класс Order
public Order(Customer customer)
{
  Customer = customer.TakeSnapshot();
}
```

Далее возникают следующие вопросы: является ли создание экземпляра объекта клиента удобным моментом для получения моментального снимка клиента и не слишком ли рано это делать? Что, если клиент изменится? Может быть, лучше сделать моментальный снимок клиента в переходный момент, когда клиент получает заказ? Эти вопросы, конечно, интересны и важны, но в этом разделе мы ограничимся лишь обновлением класса `Customer` так, как показано на рис. 5.11.

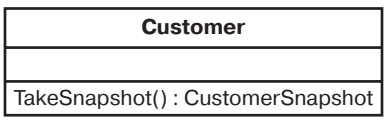


Рис. 5.11. Метод, введенный для получения моментального снимка клиента

И опять возникает мысль о рефакторинге кода. Мне лично не совсем по душе тот факт, что *экземпляр* объекта `Customer` передается методу `GetOrders()` при поиске заказов. На самом деле для нормального функционирования приложения важен лишь идентификационный номер клиента, и поэтому было бы странно использовать текущий экземпляр объекта клиента для выборки заказов с информацией о предыстории клиента. В любом случае этот вопрос требует дополнительного рассмотрения.

Попутно возникает следующий вопрос: как следует называть свойство объекта `Order` для поиска информации о предыстории клиента в данном заказе? В настоящий момент оно называется `Order.Customer`, но это ничего не говорит о временном аспекте информации о клиенте. Вероятно, данное свойство лучше назвать `Order.CustomerSnapshot`. Для этого внесем соответствующие изменения в класс `Order`, как показано на рис. 5.12.

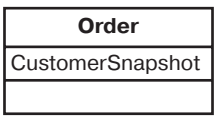


Рис. 5.12. Измененное название свойства `Customer`

Кроме того, экземпляры объектов `Customer` создаются непосредственно, а не с помощью класса `CustomerFactory`. На данный момент этого, на мой взгляд, вполне достаточно. Фабрики мы введем только тогда, когда они понадобятся, но не раньше!

Возможно, стоит вновь вернуться к классу `OrderFactory`, ввести в него метод `CreateOrderLine()` и передать ему в виде параметров объекты `Product` и `Order` (или хотя бы один объект `Order`). С другой стороны, было бы неплохо работать с объектом `OrderLine`, не делая его постоянным, что может произойти при вводе строки заказа в заказ как в постоянный объект. Необходимость таких изменений можно отложить на время, а пока что лучше перейти непосредственно к рассмотрению жизненного цикла экземпляра данного объекта.

Жизненный цикл экземпляра объекта

Как отмечалось в предыдущем разделе, строка заказа превращается в постоянный объект при ее вводе в заказ как в постоянный объект. Это обстоятельство требует пояснения. В данном случае имеется в виду, что экземпляр объекта начинает свой жизненный цикл как временный объект при выполнении, например, следующей операции:

```
Product p = new Product();
```

В этом случае объект `Product` оказывается временным. Он не выбирается из базы данных и поэтому вряд ли станет постоянным. Если же требуется, чтобы он стал постоянным (и сохраняемым в базе данных при очередном вызове метода `PersistAll()`), придется обратиться за помощью к хранилищу, вызвав метод `AddProduct()`. В зависимости от используемой инфраструктуры хранилище уведомляет инфраструктуру о сохраняемом объекте продукции.

Когда же вновь приходится обращаться за помощью к хранилищу, чтобы восстановить экземпляр объекта из базы данных, этот выбранный экземпляр оказывается постоянным объектом. Все изменения в нем сохраняются при последующем вызове метода `PersistAll()`.

А что же объект `OrderLine`? Ведь по поводу ввода строки заказа методом `AddOrderLine()` мы обращались не к хранилищу, а к корневому объекту `Order` агрегата, и поэтому строка заказа также становится постоянным объектом. На мой взгляд, свойство сохраняемости внутри агрегатов должно быть каскадным.

Итак, подытожим все наши рассуждения о семантике, которая требуется для описания жизненного цикла экземпляров объектов в модели предметной области, в виде табл. 5.1.

На рис. 5.13 схематически представлены состояния экземпляра объекта в течение его жизненного цикла.

Таблица 5.1. *Сводка семантики жизненного цикла экземпляров объектов в модели предметной области*

Операция		Результат (временный/постоянный экземпляр объекта)
Вызов нового экземпляра объекта	→	Временный экземпляр объекта
`Repository.Add(экземпляр)` или `persistentInstance.Add(экземпляр)`	→	Постоянный экземпляр объекта в модели предметной области
`x.PersistAll()`	→	Постоянный экземпляр объекта в базе данных
`Repository.Get()`	→	Постоянный экземпляр объекта в модели предметной области (а также в базе данных)
`Repository.Delete(экземпляр)`	→	Временный экземпляр объекта (причем экземпляр удаляется из базы данных при вызове метода `x.PersistAll()`)

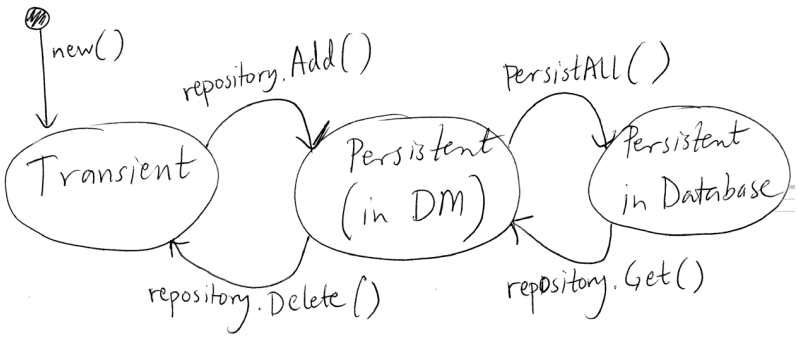


Рис. 5.13. Жизненный цикл экземпляра объекта

> ***На заметку***
>
> Этот вопрос подробно рассматривается в книге Эванса [Evans DDD], где можно найти великолепные пояснения, так что обращайтесь к этому источнику. То же самое можно сказать и о книге Джордана и Расселла [Jordan/Russell JDO], где рассматривается более сложная (и гибкая) семантика, но ведь впоследствии может обнаружиться, что намеченной выше простой семантики явно недостаточно.

Данный вопрос не следует путать с тем, как временные и постоянные объекты воспринимаются инфраструктурой (примером тому служит объектно-реляционный преобразователь NHibernate). В данном случае речь идет о семантике, описывающей жизненный цикл экземпляров объектов для нашей модели предметной области,

а задача инфраструктуры — предоставить нам нужную семантику внутренним образом. Более подробно об этом речь пойдет в последующих главах книги.

Кстати об инфраструктуре, у внимательного читателя, возможно, уже назрел вопрос: что подразумевается под вызовом метода `x.PersistAll()` в табл. 5.1? Это попытка обозначить инфраструктуру символом `x`. Опять же, данный вопрос неактуален, и о нем можно пока что забыть.

Итак, вернемся к задаче номер 2, т .е. к списку заказов клиента. Следующая подзадача состоит в том, чтобы снабдить заказ определенным типом.

Тип заказа

Самое простое решение в данном случае — ввести экземпляр объекта `OrderType` (Тип заказа), как показано на рис. 5.14

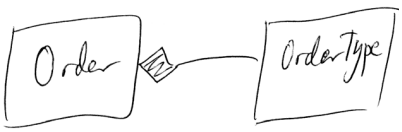


Рис. 5.14. Классы `Order` и `OrderType`

На мой взгляд, `OrderType` следует считать не объектом-сущностью (Entity), а объектом-значением (Value Object) [Evans DDD]. Некоторые его значения являются "глобальными" для всего приложения, а кроме того, они очень малы и статичны.

Конечно, `OrderType` можно реализовать в виде объекта-значения, но все указывает на то, что для этой цели окажется достаточно и перечисления, особенно в настоящий момент. Поэтому определим перечисление `OrderType` и введем аналогичное поле в класс `Order`.

Ответственное лицо для заказа

Итак, вернемся к ответственному лицу (`ReferencePerson`) для заказа (`Order`). Это еще один пример объекта-значения (Value Object) [Evans DDD], хотя он и отличается от объекта `OrderType`, который мы сначала отнесли к категории объектов-значений, поскольку ответственные лица не являются глобальными объектами для всей системы в целом, а специфичны для каждого клиента или заказа. А это явно указывает на то, что `ReferencePerson` не следует делать перечислением.

На рис. 5.15 класс объектов `ReferencePerson` представлен "в действии".

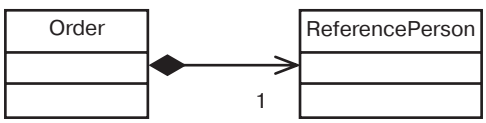


Рис. 5.15. Классы `Order` и `ReferencePerson`

Раз уж мы решили воспользоваться шаблоном Value Object [Evans DDD], то, очевидно, следует выбрать структуру .NET вместо класса. Но у структуры должны быть такие размеры, чтобы ее можно было использовать эффективно. К тому же применение структур вызывает дополнительные трудности, связанные с инфраструктурой, а решение получается более закрытым и менее четким. Я лично склонен выбрать классы, и если вы готовы последовать за мной по этому пути, то нам придется использовать все значения класса при переопределении метода `Equals()`.

Ниже показано, как метод `Equals()` может быть реализован для объекта-значения `ReferencePerson`. (Следует иметь в виду, что в данном случае сравниваются все поля объекта-значения, но не так, как это делалось прежде с объектом-сущностью. Кроме того, переопределение метода `Equals()` для объектов-значений все равно потребуется, независимо от применения шаблона Identity Map.)

```
// Класс ReferencePerson
public override bool Equals(object o)
{
  if (o == null)
    return false;

  if (this.GetType() != o.GetType())
    return false;

  ReferencePerson other = (ReferencePerson) o;

  if (! FirstName.Equals(other.FirstName))
    return false;
  if (! LastName.Equals(other.LastName))
    return false;

  return true;
}
```

На заметку

Ингмар Лундберг так прокомментировал приведенный выше фрагмент кода, предложив более краткий вариант:

```
// Метод ReferencePerson.Equals()
ReferencePerson other = o as ReferencePerson;
return other != null
  && this.GetType() == other.GetType()
  && FirstName.Equals(other.FirstName)
  && LastName.Equals(other.LastName);
```

Итак, мы воплотили в коде некоторые базовые функциональные возможности из модели предметной области. На рис. 15.16 показано, на какой стадии развития теперь находится модель предметной области.

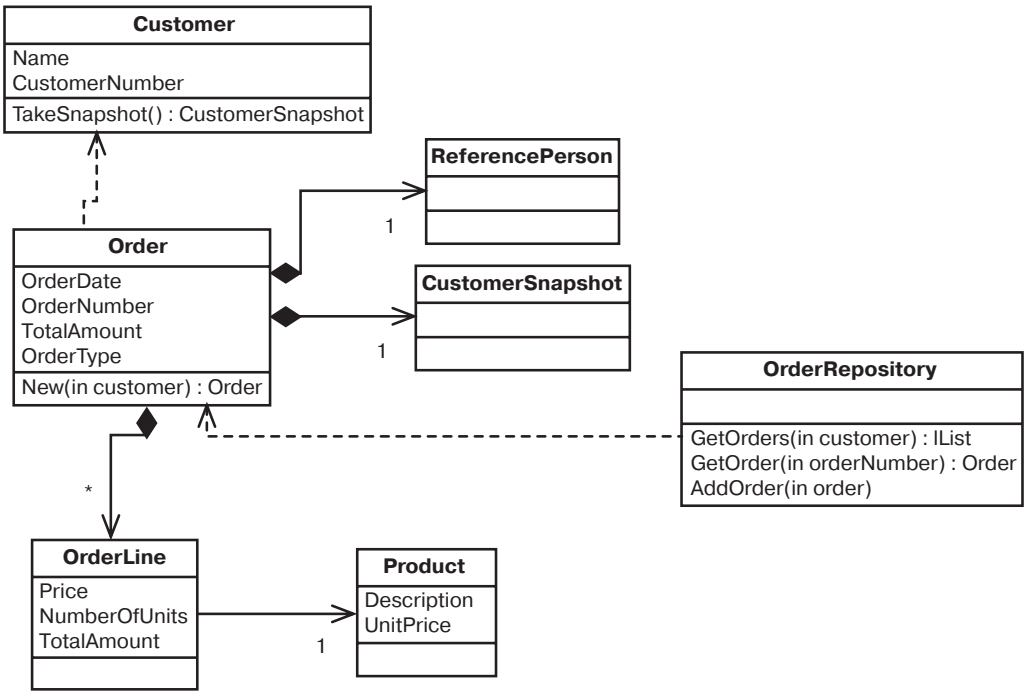


Рис. 15.16. Реализация модели предметной области на данный момент

На этом можно, пожалуй, остановиться и сделать небольшой перерыв. А перерыв означает также смену деятельности. Когда мы вернемся к уточнению модели предметной области в последующих главах книги, то воспользуемся аналогичным подходом в стиле РПТ для описания всех мельчайших проектных решений по мере их рассмотрения, хотя это описание будет более кратким.

А эту главу мне бы хотелось завершить рассмотрением еще одного стиля интерфейса API.

Плавный интерфейс

До сих пор мы наметили простейший, примитивный интерфейс API для нашей модели предметной области. Теперь составление заказа на два типа продукции может выглядеть следующим образом (при условии, что уже имеется конкретный заказчик):

```
Order newOrder = new Order(customer);
OrderLine orderLine;

orderLine = new OrderLine(productRepository.GetProduct("ABC"));
orderLine.NumberOfUnits = 42;
newOrder.AddOrderLine(orderLine);

orderLine = new OrderLine(productRepository.GetProduct("XYZ"));
orderLine.NumberOfUnits = 3;
newOrder.AddOrderLine(orderLine);
```

Это достаточно простая последовательность операций, хотя она несколько многословная и не такая плавная, как хотелось бы. Для ее сокращения можно ввести свойство `numberOfUnits` в конструктор объекта `orderLine`, сведя весь предыдущий код к трем строкам следующим образом:

```
Order newOrder = new Order(customer);

newOrder.AddOrderLine(new OrderLine
  (42, productRepository.GetProduct("ABC")));

newOrder.AddOrderLine(new OrderLine
  (3, productRepository.GetProduct("XYZ")));
```

Однако этому коду по-прежнему недостает плавности. Вдохновившись идеями, почерпнутыми из статьи Мартина Фаулера [Fowler FluentInterface], попробуем наметить интерфейс API, сделав его более плавным, например, следующим образом:

```
Order newOrder = new Order(customer)
  .With(42, "ABC")
  .With(3, "XYZ");
```

Теперь он выглядит более кратким и ясным. В этом наброске интерфейса первая строка оставлена без изменения, поскольку она и так достаточно ясна. А метод `With()` воспринимает в качестве параметров свойство `numberOfUnits` и идентификационный номер продукции, чтобы воспользоваться хранилищем `productRepository` для поиска продукции. (В данный момент предполагается, что метод `With()` может обнаружить хранилище `productRepository` во вполне определенном месте.)

Кроме того, метод `With()` возвращает заказ, а следовательно, мы можем построить несколько операторов в цепочку.

А теперь опробуем другой вариант. Допустим, что мы не можем составить весь заказ за один раз, но хотели бы добавить в него строку в результате действий пользователя. В таком случае для ввода строки заказа можно было бы воспользоваться следующей строкой кода:

```
newOrder.AddOrderLine("ABC").NumberOfUnits = 42;
```

На этот раз метод `AddOrderLine()` возвращает строку заказа, а не сам заказ. Эти изменения можно было бы продолжить, если бы потребовалось задать больше значений в строке заказов, заменив свойство `numberOfUnits` методом, вновь возвращающим строку заказа, но мы не будем этого делать, поскольку общая идея уже ясна.

Главный вывод из этого раздела заключается в следующем: если немного повозиться с интерфейсом API, то можно значительно улучшить его.

Резюме

Эта глава была посвящена достаточно подробному рассмотрению исходных составляющих модели. Подробное изучение таких вопросов лучше всего сопровождать кодом и РПТ вместо UML-схем, что и было воплощено в этой главе.

В завершение этой главы подведем краткий итог того, что было сделано по составленному ранее списку возможностей. До сих пор нам удалось рассмотреть возможности и решить задачи номер 2, 3, 7 и 9. Возможно, это и не так много, но прогресс — налицо, причем этот прогресс подкреплен большим числом тестов, что само по себе уже неплохо.

В следующей главе мы продолжим работу над уточнением модели предметной области, сместив немного акцент. Возможно, вы обратили внимание на то, что мы уже несколько раз примерялись к инфраструктуре в этой главе, обсуждая связанные с ней вопросы, поэтому теперь обратимся непосредственно к вопросам подготовки к вводу инфраструктуры.

Глава 6

Подготовка к инфраструктуре

Итак, продолжим исследование прикладного ППО на примере конкретной предметной области. В следующем шаге, который предстоит сделать на этом пути, нет ничего необычного, хотя вас может удивить, почему мы переходим к вопросам инфраструктуры. В частности, мы еще не добились перехода заказов в состояние сохраняемости или несохраняемости, т.е. их материализации или восстановления из базы данных. Этот вопрос остается открытым, не говоря уже об использовании базы данных.

Но это было сделано намеренно. Подобные вопросы желательно отложить на как можно более отдаленный срок — по крайней мере, до тех пор, пока не будут решены вопросы выбора инфраструктуры.

Причина, по которой привязка к реляционной базе данных откладывается на время, объясняется тем, что это позволяет сосредоточить основное внимание на модели предметной области, отвлекаясь от нее как можно реже. Имея достаточный опыт работы с базами данных, я хорошо знаю, что рассматривать вопросы, связанные с базой данных, — значит, сильно отвлекаться от стратегической линии проекта. Если бы я больше занимался ООП, то вряд ли посмел это утверждать.

А если серьезно, то я совсем не хочу сказать, что взаимодействие с базой данных не имеет особого значения. Напротив, я знаю, что это очень *важно*. Но если отстраниться как можно дальше от базы данных, то опробование и исследование как мелких, так и крупных изменений в модели станет более продуктивным.

Кроме того, это упрощает написание тестов, поскольку изменения, вносимые в тесты, не остаются постоянными при последовательном их выполнении, а сами тесты выполняются намного быстрее, что значительно упрощает задачу разработчиков, когда им требуется тестирование после каждого изменения кода.

Разумеется, у такого подхода к проектированию имеются свои недостатки. Как упоминалось выше, мы исходно допускаем, что взаимодействие с базой данных не вызовет особых трудностей, и поэтому считаем, что проблема в другом. Но с другой стороны, если на ранней стадии проекта потребуется создать ряд прототипов пользовательского интерфейса не только для проверки модели предметной области, но и для организации диалога с потребителем, и если наполнить эти прототипы “живыми данными”, с которыми пользователь мог бы взаимодействовать, вводя, изменяя или удаляя их, такие прототипы оказались бы более полезными, чем те, что чаще всего используются на данной стадии проекта.

В завершение предыдущей главы были предварительно рассмотрены вопросы, связанные с вводом ранних вариантов пользовательского интерфейса. Опробование этих вариантов вызовет у пользователей больший интерес, если после перезапуска для них будут также доступны данные. (Разумеется, это нетрудно сделать, если сразу же начать пользоваться заранее подготовленным решением сохраняемости. Но в то же время можно ожидать, что такой подход приведет к дополнительным издержкам при первых же попытках реорганизовать код, поэтому выгоднее создать недорогую "иллюзию" работоспособной системы.)

На заметку

Будьте бдительны, чтобы после опробования одного или двух прототипов ваш заказчик (или начальник) не подумал, что приложение уже полностью готово. Мы это уже проходили.

Еще одна особенность такого подхода к проектированию состоит в возможности писать тесты для безопасных сценариев, опять же не обращаясь к настоящей базе данных. Вы можете спросить: а какой в этом прок? А такой, что вы можете уделить основное внимание модели и связанной с ней семантике и прочим насущным вопросам.

Конечно, данную задачу можно было бы решать по отдельности в хранилищах, как это было сделано до сих пор, но я лично вижу больше пользы в согласованном применении одного решения ко всем хранилищам. Требуется решение, которое можно наращивать до тестирования ранней интеграции, причем самое главное — мое намерение писать код для реальных хранилищ, а не нелепый временный код, который затем придется все равно выбрасывать!

Итак, я начал эту главу с того, что рассуждать об инфраструктуре еще слишком рано, но несмотря на это мы займемся *подготовкой* к инфраструктуре, чтобы не переделывать всю работу заново, когда придется вводить инфраструктуру. Для этой цели потребуется написать код, независимый от инфраструктуры.

POCO как образ жизни

Выше я дал ясно понять, что на самом деле стараюсь сохранять основные ресурсы своих приложений свободными от уклонений в сторону инфраструктуры. Направление POJO (Plain Old Java Object — Обыкновенный Java-объект) и POCO (Plain Old CLR Object — Обыкновенный CLR-объект) возникло на платформе Java в качестве ответной реакции на появление платформы J2EE и ее широко распространившееся внедрение в приложениях, из-за чего, в частности, усложнился весь процесс разработки приложений, а РПТ стала почти недоступной. Мартин Фаулер, Ребекка Парсонс и Джош Маккензи придумали термин POJO для описания класса, свободного от

"немого" кода, который требовался лишь для среды выполнения. Такие классы должны быть сосредоточены на конкретно решаемой прикладной задаче. В классах из модели предметной области не должно быть ничего лишнего.

На заметку

Причиной зарождения этого направления стали, главным образом, легкие контейнеры, в том числе Spring [Johnson J2EE Development without EJB].

На платформе .NET не сразу обратили внимание на обыкновенные объекты, но теперь они называются POCO.

Термин POCO в какой-то степени уже устоялся, хотя его нельзя применять непосредственно к инфраструктуре, связанной с сохраняемостью. Когда я обсуждал этот термин с Мартином Фаулером, он сказал, что более точным и понятным описанием данного понятия было бы *неведение сохраняемости* (НС — Persistence Ignorance (PI)). Я согласен с этим, и поэтому буду пользоваться далее именно этим термином.

НС для объектов-сущностей и объектов-значений

Допустим, что нам требуется применить НС. А что это вообще означает? НС означает создание ясных, простых классов, сосредоточенных на решаемой прикладной задаче, без ввода дополнительных компонентов, связанных с инфраструктурой. Но это пока еще мало о чем говорит. Поэтому лучше пояснить, что НС *не* означает. Для этого достаточно выяснить, имеются ли в модели предметной области какие-либо ссылки на внешние библиотеки DLL, связанные с инфраструктурой. Так, если используется объектно-реляционный преобразователь NHibernate и имеется ссылка на библиотеку `nhibernate.dll`, это служит явным признаком того, что в модель предметной области введен код, который является не базовым, а, скорее, отклоняющимся от нее.

В чем же эти отклонения? Так, если метод НС применяется к постоянным объектам, то отсутствуют какие-либо требования к выполнению одного из следующих действий:

- Наследование от определенного базового класса (помимо класса `Object`).
- Получение экземпляра объекта только посредством предоставляемой фабрики.
- Использование специально предоставляемых типов данных, в частности, для совокупностей объектов.
- Реализация специального интерфейса.
- Предоставление специальных конструкторов.
- Предоставление обязательных специальных полей.
- Исключение из употребления определенных конструкций.

Этот список далеко не полон. Так, в классах модели предметной области запрещается использовать код вызова хранимых процедур в базе данных. Но это требование столь очевидно, что о нем можно было бы и не упоминать.

А теперь рассмотрим перечисленные выше требования по очереди.

Наследование от определенного базового класса

Для базовых структур требование поддерживать сохраняемость нередко заключается в обязательном наследовании от определенного базового класса.

Так, следующий код может оказаться вполне приемлемым:

```
public class Customer : PersistentObjectBase
{
  public string Name = string.Empty;
…

  public decimal CalculateDepth()
…
}
```

Он *действительно* вполне приемлем, но содержит не совсем оптимальную семантику и механизмы. Например, в этом коде использована только одна возможность наследования для класса `Customer`, поскольку в .NET допускается единичное наследование. Можно, конечно, спорить, насколько эта проблема велика, так как для проектирования всегда найдется обходной прием.

Намного хуже, если разработанную модель предметной области требуется сделать постоянной. Для того чтобы выполнить требование наследования, скорее всего, придется внести некоторые изменения в модель предметной области. Практически то же самое происходит, когда разработка модели предметной области начинается с применением РПТ. При этом с самого начала накладывается следующее ограничение: наследование использовать нельзя, поскольку его нужно приберечь для того, чтобы выполнить требование сохраняемости.

Если наследование привносит в подкласс немало общедоступных функциональных возможностей, потребителю класса придется блуждать среди методов, которые его не интересуют.

В случаях, не столь ясных и простых, как приведенный выше пример, класс `PersistentObjectBase` вынуждает к некоторым реализациям методов в этом классе, как это, например, делается в шаблоне Template Method [GoF Design Patterns]. Конечно, это не смертельно, но довольно хлопотно.

На заметку

Данное требование следует рассматривать не как обязательное, а, скорее, как удобное условие, позволяющее извлечь максимум из реализации интерфейса, которую требует базовая структура конкретного типа. Это общее требование мы рассмотрим далее.

Именно так и было сделано в базовой структуре проекта Valhalla, который я разработал вместе с Кристофером Скьолдборгом. Но, откровенно говоря, в этом случае на базовый класс под названием `EntityBase` было возложено столько функций, что реализация интерфейсов со специальным кодом вместо наследования от класса `EntityBase` превратилась лишь в теоретическую возможность.

Получение экземпляра объекта только посредством предоставляемой фабрики

Прошу понять меня правильно: я совсем не против использования фабрик. Тем не менее я не испытываю особого восторга от того, что вынужден их использовать, если нет другого приемлемого решения. Это, например, означает, что вместо следующей строки кода:

```
Customer c = new Customer();
```

мне *приходится* писать такой фрагмент кода:

```
Customer c = (Customer)PersistentObjectFactory.CreateInstance
  (typeof(Customer));
```

На заметку

Если вы считаете, что я был несправедлив, используя слишком длинные имена, то следующая строка кода вряд ли покажется вам более лаконичной:

```
Customer c = (Customer)POF.CI(typeof(Customer));
```

И это не смертельно, но в большинстве случаев не оптимально. Второй из приведенных выше фрагментов кода кажется более запутанным, чем первый. И как часто бывает, такой код лишь усложняет тестирование.

Нередко одна из причин обязательного использования предоставляемой фабрики состоит в том, что она в конечном итоге помогает выполнять проверку изменения состояния объектов. Таким образом, классы из модели предметной области динамически получают подклассы, а в свойствах подкласса устанавливается один или несколько флажков изменения состояния. Фабрика делает эту операцию прозрачной для потребителя, создавая экземпляр подкласса вместо класса, запрашиваемого потребителем фабрики. К сожалению, для того чтобы это действительно работало, свойства придется сделать виртуальными, тогда как общедоступные поля нельзя использовать (это еще две мелкие детали, преуменьшающие значение НС). Впрочем, общедоступные поля все же *можно* использовать, хотя их нельзя “переопределять” в формируемом подклассе, что вызывает осложнения, если, например, подкласс предназначен для того, чтобы отслеживать изменение состояния объектов.

На заметку

При использовании аспектно-ориентированного программирования (АОП) на платформе .NET доступен ряд других методов, в которых чаще всего применяется упомянутая выше динамическая подклассификация. Я всегда считал, что объявлять члены класса как виртуальные не следует, но Роджер Йоханссон обратил мое внимание на следующее. Допустим, что требуется переопределить член класса, чтобы, таким образом, избежать дополнительных хлопот и ответственности за поддержку подклассификации. Такое решение должно оказать влияние как на обычную подклассификацию, так и на подклассификацию, используемую по причине АОП. Если сделать член класса виртуальным, то тем самым можно подготовить его к переопределению, опять же посредством обычной или АОП-подобной подклассификации. Согласитесь, что в этом есть определенный смысл.

Еще одна типичная проблема, решаемая таким способом, состоит в необходимости применять шаблон Lazy Load, но об этом речь пойдет в следующем подразделе.

Использование специально предоставляемых типов данных, в частности, для совокупностей объектов

В классах из модели предметной области для совокупностей объектов нередко используются специальные типы данных. Эти типы данных называются специальными, потому что они не относятся к тем типам, которые обычно используются, если имеется свобода выбора.

Чаще всего данное требование возникает в связи с поддержкой шаблона Lazy Load, [Fowler PoEAA] или, вернее, *неявной* загрузки по требованию, чтобы не писать для этого код вручную. (Загрузка по требованию означает выборку из базы данных в нужный момент.)

Но помимо этого, специальные типы данных могут предоставить и другие функциональные возможности, в том числе специальный режим удаления: как только экземпляр объекта удаляется из совокупности объектов, он также регистрируется для удаления с помощью шаблона Unit of Work [Fowler PoEAA]. (Шаблон Unit of Work служит для отслеживания действий, которые должны быть предприняты по отношению к базе данных в конце текущей логической единицы работы.)

На заметку

Как я уже отметил, специальные типы данных могут *предоставить* и другие *функциональные возможности*. Этим я хотел подчеркнуть, что не отношусь резко отрицательно к ННС, т.е. к тому, то *не* считается неведением сохраняемости.

И наконец, подобные типы данных избавляют от необходимости писать самостоятельно код для обеспечения двунаправленности. Это еще один пример обязанностей, которые может взять на себя АОП.

Реализация специального интерфейса

Еще одно довольно обычное требование к постоянству классов из модели предметной области состоит в реализации одного или нескольких интерфейсов, связанных с инфраструктурой.

Естественно, что это не вызывает особых затруднений, если для реализации интерфейсов приходится писать очень мало кода, и больше затруднений — в противном случае.

Одним из примеров функциональных возможностей на основе интерфейсов может служить заполнение экземпляра объекта значениями из базы данных, не обращаясь к механизмам установки (имеющим особый код, который нежелательно выполнять во время восстановления экземпляра объекта).

Другим типичным примером служит предоставление интерфейсов для оптимизированного доступа к состоянию экземпляров объектов.

Предоставление специальных конструкторов

Еще один способ предоставления значений, восстанавливающих экземпляры объектов из базы данных, состоит в том, чтобы потребовать специальные конструкторы, которые не имеют никакого отношения к решаемой прикладной задаче.

Вполне возможно, что используемый по умолчанию конструктор потребуется также для того, чтобы упростить получение в базовой структуре экземпляров объектов классов из модели предметной области в результате операции `Get` над базой данных. И сделать это нетрудно, хотя такая работа и отвлекает от самой модели.

Предоставление обязательных специальных полей

Для некоторых инфраструктурных решений требуется, чтобы в классах из модели предметной области были предоставлены специальные поля, например, поля `Id` типа `Guid` или поля `Version` типа `int`. (Имеется в виду, что в полях `Id` в качестве типа данных используются глобально уникальные идентификаторы `Guid`.) Благодаря этому упрощается инфраструктура, хотя задача разработчика модели предметной области немного усложняется — по крайней мере, в том случае, если данное требование оказывает нежелательное влияние на классы модели.

Исключение из употребления или вынужденное использование определенных конструкций

Как упоминалось выше, виртуальные свойства иногда приходится использовать помимо собственного желания, а также избегать употребления определенных конструкций, типичным примером которых могут служить поля “только для чтения”. Такие поля, обозначаемые ключевым словом `readonly`, не могут устанавливаться извне

(разве что с помощью конструкторов). В таком случае в модели предметной области приходится создавать классы, полностью соответствующие требованиям НС.

Частное поле, используемое вместе со свойством "только для выборки", очень близко напоминает поле "только для чтения", хотя не соответствует ему точно. Можно, конечно, спорить, что поле "только для чтения" является именно тем решением, которое лучше всего раскрывает намерения разработчика.

На заметку

По поводу того, насколько атрибуты .NET пригодны для целей предоставления модели предметной области данных о том, как сделать ее сохраняемой, ведется горячая полемика.

Я считаю, что такие атрибуты годятся для этих целей и никоим образом не понижают уровень НС, если они доступны в виде используемой по умолчанию информации, которая может быть переопределена. Другое дело, что они становятся слишком многословными, отвлекая читающего от самого кода.

Соблюдать НС или не соблюдать

Соблюдать НС или не соблюдать — вопрос совсем не двузначный. У этого метода имеются и темные стороны, но пока что нас может вполне устроить лишь намерение соблюдать НС, а не полностью выполнить его требования. Ведь любая крайность обходится дорого. (Мы еще вернемся к этому вопросу в главе 9, кода будем рассматривать инфраструктурное решение.)

На заметку

А можно ли найти в реальной жизни какой-нибудь пример совершенной двузначности? Конечно, можно. Я, например, часто напоминаю о двузначности своей жене, когда она говорит, что женщина была *очень* беременна.

Кроме того, мы еще не учли тот факт, что многое зависит от выбора момента, когда следует оценивать целесообразность применения НС.

НС во время выполнения или компиляции

До сих мы рассматривали НС вне времени, но НС, вероятно, приобретает большее значение во время компиляции, чем во время выполнения. Что же это означает? Допустим, что код, связанный с инфраструктурой, создается автоматически, о чем вы даже не догадываетесь, а значит, и вообще не имеете с ним дело. Такое решение, вероятно, лучше, чем сопровождение подобного кода вручную.

Впрочем, такой подход вызывает противоречивые чувства из-за того, что приходится выполнять некий код, который был написан кем-то другим. Ведь его отладка может превратиться в сущий кошмар!

На заметку

Марк Берхоп так прокомментировал изложенное выше:

"Это был, пожалуй, первый аргумент, выдвинутый в начале 1990-х годов против C++ программирующими на C. Они заявляли, что C++ вставляет новый код, который они не писали, скрывая то, что при этом происходит. Я не уверен, что этот аргумент теперь убедителен".

Кроме того, вносить код для классов на уровне отдельных байтов в .NET намного труднее, чем в Java. Ведь он не поддерживается базовой структурой, и поэтому вы предоставлены самому себе, что, как правило, замедляет процесс разработки приложений.

В таком случае чаще всего применяют альтернативные методы, в том числе и те, что были упомянуты выше при рассмотрении динамической подклассификации в сочетании с предоставляемой фабрикой, но такой подход мало чем отличается от внесения кода. Поэтому назовем данный подход генерированием кода.

Во что обходится НС для объектов-сущностей и объектов-значений

Одна из возможных реакций на все сказанное выше может быть следующей: "Если НС так замечательно, то почему бы не пользоваться им постоянно?" Существует закон природы (или, по крайней мере, программирования), который гласит: если все кажется ясным, понятным и безупречным, значит, весьма вероятны и недостатки. В данном случае таким недостатком являются дополнительные издержки.

Для достижения высокого уровня НС приходится жертвовать быстродействием — по крайней мере, во время выполнения, поскольку в этом случае нужно использовать рефлексию, которая обходится недешево. (Если во время компиляции НС дает вполне приемлемые результаты, то рефлексия не нужна, а вместо нее можно выбрать решение АОП, чтобы добиться повышения производительности.)

С помощью операции, выполняемой в замкнутом цикле, можно легко убедиться в том, что операции чтения и записи в поля или свойства происходят при рефлексии медленнее, чем при вызове этих же операций обычным образом. Но насколько велики подобные издержки? Очевидно, что все зависит от конкретной ситуации. Для того, чтобы выяснить это в конкретном случае, придется написать соответствующие тесты. Не следует, однако, забывать, что обращение к базе данных обходится очень дорого по сравнению с большей частью того, что делается в модели предметной области, но в то же время это трудно сравнимые операции. Так, обычное чтение нельзя сравнивать с чтением на основе рефлексии.

Типичный пример, касающийся быстродействия

Обратимся к примеру, чтобы лучше прояснить весь рассматриваемый вопрос в целом. Для базовой структуры сохраняемости характерна операция, в ходе которой принимается решение, следует ли сохранять экземпляр объекта в базе данных в конце определенного сценария. Обычно для этой цели экземпляру объекта поручается сигнализировать об изменении своего состояния (`IsDirty`), если он должен сохраняться. А еще лучше, если экземпляр объекта будет сигнализировать единице работы (Unit of Work) об изменении своего состояния, чтобы единица работы помнила об этом, когда настанет время сохранить изменения.

Но ведь для этого придется нарушить принцип НС, если только не понадобится расплачиваться средствами АОП.

> **На заметку**
>
> У такого решения имеются и другие недостатки, в том числе следующий: если операция выполняется посредством рефлексии, изменение состояния экземпляра объекта остается незамеченным, а следовательно, изменения, произошедшие в экземпляре объекта, не сохраняются, хотя этот недостаток не вполне очевиден.

Другое решение состоит в том, чтобы вообще ни о чем не сигнализировать, но предоставить базовой структуре возможность запомнить состояние экземпляров объектов при выборке из базы данных. А во время сохранения текущее состояние экземпляров объектов должно сравниваться с тем состоянием, в котором они находились при чтении из базы данных.

Очевидно, что это не только сравнение одной простой операции чтения с другой операцией чтения на основе рефлексии, а совершенно разные подходы к решаемой задаче, дающие в итоге абсолютно разные показатели производительности. Для того чтобы убедиться в этом, вы можете организовать такое сравнение самостоятельно, выбрав один миллион экземпляров объектов из базы данных, видоизменив один такой экземпляр, а затем определив разницу во времени на выполнение операции сохранения в обоих случаях. Понимаю, что и данное решение представляется непростым, но о нем все же следует подумать.

Другие примеры

Итак, мы рассмотрели на конкретном примере возможные издержки НС на быстродействие, но это не единственные издержки. Ранее указывалось также на автоматическое сокращение функциональных возможностей при попытке добиться высокого уровня НС. Мы уже рассмотрели многие возможности, которые сразу же дает несоблюдение требований НС, включая автоматическую поддержку двунаправленности и неявную загрузку по требованию.

Следует также иметь в виду, что отслеживание изменения состояния объектов имеет отношение не только к производительности. Ведь потребитель может быть очень заинтересован и в использовании подобной информации при составлении форм, чтобы, например, знать, какие кнопки следует активизировать.

И в том случае, когда нужно выбирать между соблюдением и несоблюдением НС, приходится, как всегда, искать компромисс. Этот компромисс состоит в нахождении золотой середины между издержками и сокращенными функциональными возможностями с одной стороны и отклонением кода от базовой модели с другой стороны, причем такой код не только привязывает приложение к определенной инфраструктуре, но и затрудняет РПТ. Следовательно, у такого компромиссного решения имеются свои достоинства и недостатки, что вполне оправданно.

Выводы об издержках НС

Из всего изложенного выше можно сделать следующий вывод: компромиссы неизбежны, и поэтому их следует находить очень тщательно. Так, если вам удается добиться желаемого результата, хотя и не без недостатка, с которым можно мириться, из такого решения не следует делать культ!

Должен признаться, что в настоящее время я причисляю себя к сторонникам НС, главным образом, потому что этот метод отлично сочетается с РПТ, а объекты-сущности и объекты-значения получаются достаточно ясными и понятными.

Но я также понимаю, что данный подход может сильно отличаться от того, который предпочитаете вы. Если вам по душе начинать работу над проектом с кода, значит, вам понравится и НС. А если вы привыкли работать в интегрированной среде, где проект приходится начинать с детальной проработки в UML, а затем строить модель предметной области, то НС, скорее всего, не будет иметь для вас вообще никакого значения.

А теперь пора перевести обсуждение с модели предметной области на объекты-сущности и объекты-значения. Я имею в виду хранилища. Ведь, как ни странно, пока еще почти ничего не было сказано о применении НС для хранилищ.

НС для хранилищ

Допускаю, что применение НС для хранилищ кажется несколько чрезмерным, поскольку назначение хранилищ состоит, главным образом, в том, чтобы создать у потребителя такое впечатление, будто в его распоряжении имеется вся совокупность экземпляров объектов из модели предметной области, при условии что для получения этих экземпляров из хранилища соблюдается соответствующий протокол. Это впечатление достигается благодаря тому, что в отдельных ситуациях хранилища связываются с инфраструктурой, а такая связь устанавливается не вполне в духе НС.

Допустим, что хранилищам нужно извлечь что-нибудь, чтобы заставить инфраструктуру работать. Это означает, что для компоновки вместе с хранилищами требуется ссылка на библиотеку DLL инфраструктуры. А это, в свою очередь, означает, что

приходится выбирать между размещением хранилищ в библиотеке DLL отдельно от модели предметной области и предоставлением ссылки из модели предметной области на библиотеку DLL инфраструктуры (ниже мы рассмотрим гибкое решение этой дилеммы).

Трудности тестирования хранилищ

При тестировании хранилищ следует иметь в виду, что они связаны объектно-реляционным преобразователем и базой данных.

> **На заметку**
>
> Допустим на мгновение, что мы собираемся использовать объектно-реляционный преобразователь. Различные варианты, которые при этом возникают, будут подробнее рассмотрены в последующих главах книги.

Это неожиданно вызывает серьезные затруднения при тестировании хранилищ по сравнению с тестированием объектов-сущностей и объектов-значений по отдельности.

Конечно, можно было бы сымитировать используемый объектно-реляционный преобразователь. И хотя я не делал этого сам, думаю, что это не самый лучший выход из положения, поскольку большие затраты труда не соизмеримы с получаемыми результатами.

Трудности тестирования мелкомасштабной интеграции

В предыдущих главах не было продемонстрировано ни одного фрагмента тестового кода непосредственно для хранилищ. В большинстве интересных тестов использовалась модель предметной области. Это может явно указывать на недостаточно развитые функциональные возможности модели предметной области, из которой хотелось бы извлечь максимальную пользу.

В некоторых тестах хранилища все же использовались, но это были, главным образом, мелкие тесты интеграции, позволявшие проверить, насколько взаимодействие потребителя, объектов-сущностей и хранилищ соответствует тому, что было заранее запланировано. На самом деле одно из преимуществ хранилищ над другими подходами заключается в том, что они наделяют модели предметной области свойствами сохраняемости, и поэтому нетрудно было создать поддельные варианты хранилищ. Но вместе с тем пришлось написать немало немого кода, который в дальнейшем придется выбросить или, по крайней мере, переписать перед очередной компоновкой, где хранилища уже будут не поддельными, а настоящими.

Кроме того, семантика в поддельных вариантах хранилищ была не вполне “правильной”. Например, следующий фрагмент кода может показаться довольно странным:

```
[Test]
public void FakeRepositoryHaveIncorrectSemantics()
{
  OrderRepository r1 = new OrderRepository();
  OrderRepository r2 = new OrderRepository();

  Order o = new Order();

  r1.Add(o);
  x.PersistAll();

  // Это верно:
  Assert.IsNotNull(r1.GetOrder(o.Id));

  // А это уже неожиданно:
  Assert.IsNull(r2.GetOrder(o.Id));
}
```

На заметку

Внимательный читатель, возможно, заметил, что в приведенном выше фрагменте кода метод `AddOrder()` был заменен на метод `Add()` по сравнению с тем, что было в предыдущей главе.

Продемонстрировав приведенный выше код, я немного забежал вперед, поскольку мы только собираемся рассматривать сценарии сохранения. Как бы там ни было, но мне хотелось показать, что использовавшиеся до сих пор поддельные варианты хранилищ действуют не так, как предполагалось. И хотя на первый взгляд кажется, что при вызове метода `PersistAll()` становятся сохраняемыми все внесенные изменения, заказ может найти только первый, но не второй экземпляр хранилища. Вы можете спросить: зачем же я пишу такой код? Хороший вопрос, но должен признаться, что для меня это совсем не типично.

Вместо этого мы могли бы сымитировать каждое хранилище, чтобы проверить взаимодействие объектов-сущностей, хранилищ и потребителя. Такой поход не требует больших затрат, но в то же время позволяет проверить взаимодействие потребителя и объектов-сущностей. Конечно, ценность самих хранилищ для тестирования не особенно велика. И похоже, что мы возвращаемся к тому, с чего начали, поскольку теперь придется сделать еще один шаг и сымитировать объектно-реляционный преобразователь (если он применяется для поддержки сохраняемости), а этот вопрос мы уже обсуждали.

Первоначальный подход

Для начала было бы неплохо иметь хранилища, что особенно полезно с точки зрения тестирования. Поэтому раньше с целью решения этой задачи мне приходилось

создавать интерфейс для каждого хранилища, а затем реализовывать два класса: один — для подделки, а другой — для настоящей инфраструктуры. Это может быть сделано следующим образом. Сначала создается интерфейс в компоновке модели предметной области:

```
public interface ICustomerRepository
{
  Customer GetById(int id);
  IList GetByNamePattern(string namePattern);
  void Add(Customer c);
}
```

Затем два класса (например, `FakeCustomerRepository` — поддельное хранилище клиента и `MyInfrastructureCustomerRepository` — хранилище клиента для моей инфраструктуры) помещаются в двух разных компоновках (но в одном пространстве имен модели предметной области, если, конечно, модель не разбита на несколько разделов; см. рис. 6.1).

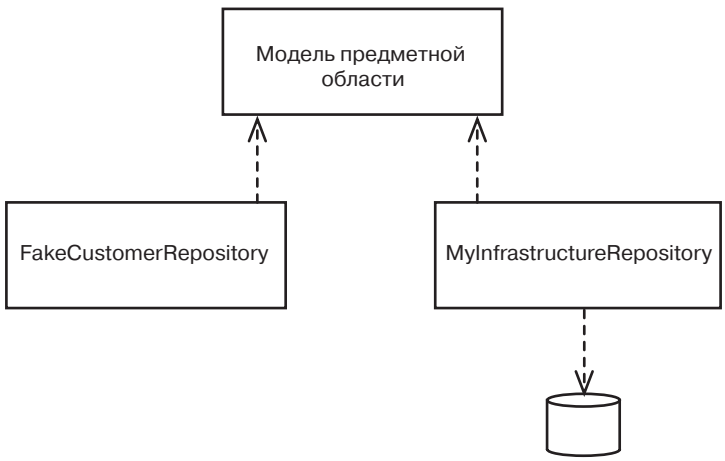


Рис. 6.1. Две компоновки хранилищ

Это означает, что выбранная инфраструктура не оказывает никакого влияния на саму модель предметной области, когда дело доходит до хранилищ, что совсем неплохо, если, конечно, такой подход обходится недорого.

Но на самом деле такой подход требует затрат. Он также подразумевает, что для каждого корня агрегата придется создавать два хранилища, причем с совершенно разным кодом.

Кроме того, окончательный вариант хранилищ должен находиться в другой компоновке (как, впрочем, и поддельные хранилища), несмотря на то что они считаются частью самой модели предметной области. Казалось бы, две дополнительные компоновки не такое уж и большое дело, но для крупного приложения, где модель предметной области разделена на несколько разных компоновок, это, как правило, озна-

чает не две лишних компоновки для хранилищ, а утроение числа компоновок модели предметной области, поскольку у каждой такой компоновки будут свои собственные компоновки хранилищ.

Несмотря на этот отрицательный аспект, данный подход ничем не хуже нелепого кода в поддельных вариантах хранилищ. Впрочем, он все равно не годится.

Лучшее решение

Затем я решил опробовать еще одно решение, создав уровень абстракции, названный мной NWorkspace (Рабочее пространство Нильссона) [Nilsson NWorkspace]. Он состоит из ряда сопрягающих интерфейсов, написанных мной в виде подделки. Такая подделка представляет собой лишь два уровня хэш-таблиц: один ряд хэш-таблиц — для постоянных объектов-сущностей (имитирующих базу данных), а другой — для единицы работы (Unit of Work) и карты идентичности (Identity Map), причем карта идентичности отслеживает идентичность загружаемых в текущий момент объектов (главным образом, первичные ключи).

Еще одна реализация, которую я осуществил, предназначалась для конкретного объектно-реляционного преобразователя.

> **На заметку**
>
> Используемое здесь и далее имя NWorkspace означает “уровень абстракции сохраняемости”. Имя NWorkspace — всего лишь пример и ничего более.

Благодаря такому уровню абстракции я могу переместить хранилища обратно в модель предметной области, а на каждый корень агрегата (Aggregate) мне потребуется только одна реализация хранилища. Одно и то же хранилище может служить как для объектно-реляционного преобразователя, так и для подделки, причем в последнем случае оно ничего не сохраняет в базе данных, а только хранит в оперативной памяти хэш-таблицы экземпляров объектов, но с той же семантикой, что и в первом случае (для объектно-реляционного преобразователя; см. рис. 6.2).

Кроме того, подделка может быть преобразована в последовательную форму или же из последовательной формы непосредственно из файла, что очень удобно для получения вполне реальных ранних версий приложений, достаточно точно отвечающих требованиям и в то же время как нельзя лучше приспособленных для рефакторинга кода.

Еще одна, совершенно неожиданно открывающаяся возможность (по крайней мере, для интерфейса API небольшого уровня абстракции) состоит в имитации инфраструктуры вместо каждого из хранилищ в отдельности. Но на самом деле речь может идти об имитации инфраструктуры не одного, а всех программных продуктов, у которых в то же время должны быть сопрягающие реализации уровня абстракции (в таком случае помимо упомянутых выше двух написанных мной реализаций будут и другие, хотя это маловероятно). А если говорить более конкретно, то имитации подлежит уровень абстракции.

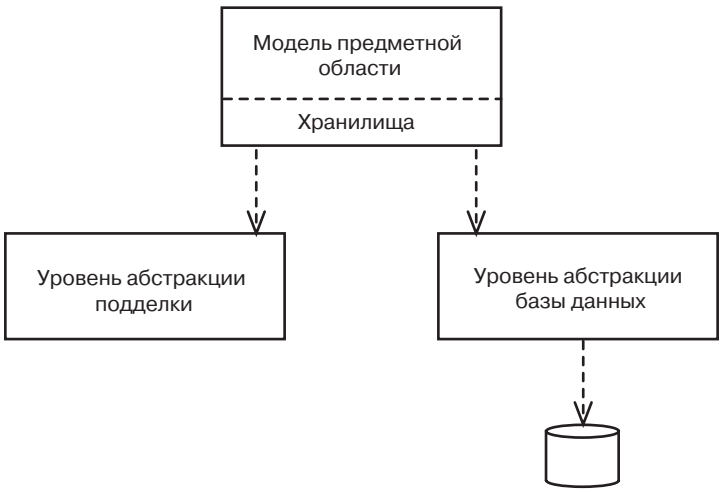


Рис. 6.2. Единственный ряд хранилищ благодаря уровню абстракции

И в этом случае можно говорить о хранилищах, отвечающих требованиям НС, но в данном решении можно избежать обращения к инфраструктуре в модели предметной области. Хотя в реальных приложениях я стараюсь в любом случае оставлять хранилища в отдельной компоновке. Благодаря этому, на мой взгляд, связывание становится более понятным, отдельные части проекта выполняются проще, а в некоторых методах хранилищ могут непосредственно использоваться операторы SQL, если это действительно нужно (в этом случае связывающие строки служат в качестве маркеров, указывающих, следует ли использовать оптимизированный код).

Но вместо того чтобы обращаться к базовой структуре сохраняемости, приходится обращаться к библиотеке DLL уровня NWorkspace с сопрягающими интерфейсами, и это можно считать значительным шагом в правильном направлении. Кроме того, в хранилищах присутствует очень мало каких-либо отклонений или же они вообще отсутствуют. Хранилища достаточно "непосредственны" с точки зрения подобающего интерфейса API уровня абстракции NWorkspace.

Таким образом, вместо того чтобы создавать один ряд хранилищ с кодом для сопряжения с интерфейсом API поставщика инфраструктуры, а другой ряд — с фиктивным кодом, достаточно создать один ряд хранилищ для попытки (хотя бы и наивной) сопряжения со стандартным интерфейсом API.

На заметку

Прошу прощения за надоедливость, но вынужден повториться: это всего лишь концепция, которую я пытаюсь найти! А моя собственная ее реализация вообще не имеет никакого значения.

Попробуем найти другой термин для обозначения хранилищ, отвечающих требованиям НС. Возможно, их лучше называть *хранилищами единственного ряда*? Пожалуй, этот термин годится для обозначения единственного ряда хранилищ, которые можно использовать как для подделки, так и для базы данных. Но важнее другое: проверить эти хранилища на деле.

Код для хранилищ единственного ряда

Для того чтобы напомнить код из поддельного варианта хранилища, ниже приведен метод, упоминавшийся в главе 5.

```
// Класс OrderRepository, поддельный вариант
public Order GetOrder(int orderNumber)
{
  foreach (Order o in _theOrders)
  {
    if (o.OrderNumber == orderNumber)
    return o;
  }
  return null;
}
```

Это не очень сложный, хотя и нелепый код.

Если предположить, что `OrderNumber` — это поле идентификации (Identity Field) [Fowler PoEAA] заказа (т.е. поле, привязывающее строку из базы данных к экземпляру объекта в модели предметной области), то код может иметь следующий вид при использовании уровня абстракции (в этом коде `_ws` обозначает экземпляр `IWorkspace`, который, в свою очередь, является основным интерфейсом уровня абстракции):

```
// Класс OrderRepository, вариант единственного ряда
public Order GetOrder(int orderNumber)
{
  return (Order)_ws.GetById(typeof(Order), orderNumber);
}
```

Такой код выглядит очень просто и непосредственно. И в этом случае указанный в данном коде метод может служить как для подделки, так и для настоящей инфраструктуры!

Во что обходятся хранилища единственного ряда

Итак, я добился еще одной абстракции. Похоже, что их становится все больше. Но с другой стороны, каждая из них, на мой взгляд, приносит определенную пользу.

Разумеется, это обходится не бесплатно. Наиболее очевидной платой за дополнительный уровень абстракции, вероятно, будет преобразование, которое потребуется во время выполнения для используемого объектно-реляционного преобразователя. Теоретически у объектно-реляционного преобразователя может быть собственная

реализация уровня абстракции, но для этого потребуется создать какой-нибудь весьма распространенный уровень абстракции.

Кроме того, приходится расплачиваться за построение уровня абстракции и сопряжения с конкретным объектно-реляционным преобразователем. Эта задача, как правило, решается на уровне базовой структуры. Затраты на построение такой структуры могут быть весьма значительными, но если она окажется *действительно* полезной, то может быть использована неоднократно.

Если повезет, то реализация сопряжения с используемой инфраструктурой окажется удачной, и тогда затраты не будут отнесены на счет разработки — по крайней мере, не на счет построения базовой структуры. Впрочем, предстоят и другие затраты. Ведь придется осваивать не только выбранную инфраструктуру, но и уровень абстракции, а этим пренебрегать нельзя.

На заметку

Раньше в этом отношении было проще, поскольку требовалось лишь знать немного Cobol и соответствующие файлы. А теперь требуется опыт программирования на C# или Java, а также SQL, организации реляционных баз данных, применения объектно-реляционных преобразователей и т.д. Поэтому попытка упростить дело введением еще одного уровня абстракции дает определенные преимущества, особенно для начинающих.

Еще одни затраты связаны с тем, что уровню абстракции отводится роль своего рода общего знаменателя. Он не обладает всеми теми возможностями, которые можно найти в выбранной инфраструктуре. Конечно, уровень абстракции можно всегда обойти за счет сложности кода внешнего хранилища и прочего. Поэтому очень важно оценить, насколько потребности проекта могут быть удовлетворены с помощью уровня абстракции: на 30, 60 или 90%. Если этот процент невысок, то вряд ли уровень абстракции может вообще представлять какой-то интерес.

А теперь вернемся на время к требованиям потребителя и уделим внимание функциональным возможностям сохранения внесенных изменений.

Сценарии сохранения

Как упоминалось в конце предыдущей главы, мы будем двигаться далее быстрее, не обсуждая подробно все стадии процесса обдумывания проектных решений. Вместо этого мы будем переходить от одного принятого решения к другому, хотя эти решения будут считаться принятыми не окончательно, а лишь на текущий момент. При этом мы будем пользоваться тестами как средством для прояснения ситуации.

Итак, рассмотрим ряд сценариев сохранения с точки зрения потребителя. Ниже приведен тест, демонстрирующий сохранение двух новых клиентов.

```
[Test]
Public void CanSaveTwoCustomers()
{
  int noOfCustomersBefore =
    _GetNumberOfStoredCustomers();
  Customer c = new Customer();
  c.Name = "Volvo";
  _customerRepository.Add(c);

  Customer c2 = new Customer();
  c2.Name = "Saab";
  _customerRepository.Add(c2);

  Assert.AreEqual(noOfCustomersBefore,
    _GetNumberOfStoredCustomers());

  _ws.PersistAll();

  Assert.AreEqual(noOfCustomersBefore + 2,
    _GetNumberOfStoredCustomers());
}
```

На первый взгляд, приведенный выше фрагмент кода достаточно прост, тем не менее, в нем "скрывается" немало вопросов, которые мы еще не обсуждали. Прежде всего, обнаруживается следующий вопрос: какой код потребителя требуется написать? Сначала требуется выполнить целый ряд операций над самыми разными экземплярами объектов, а затем сохранить всю проделанную работу с помощью единственного вызова такого метода, как, например, `PersistAll()`. (При вызове метода `_GetNumberOfStoredCustomers()` происходит обращение к механизму сохранения для проверки постоянно сохраняемых клиентов. Число таких клиентов увеличивается только после вызова метода `PersistAll()`.)

Но дело в том, что в приведенном выше коде ссылка `_ws` на хранилище предоставляется во время выполнения. Это дает возможность управлять теми хранилищами, которые должны принимать участие в одной единице работы, а также теми хранилищами, которые должны быть изолированы в другой единице работы.

Кроме того, в данном коде происходит обращение к хранилищам за помощью (посредством вызова метода `Add()`), чтобы уведомить единицу работы о новом экземпляре объекта, который требуется сохранить при последующем вызове метода `PersistAll()`. По существу, это обозначение жизненного цикла, который требуется для постоянного экземпляра объекта (этот вопрос уже обсуждался в главе 5).

Следует, однако, заметить, что связывания корня агрегата с единицей работы может быть достаточно для сохранения при вызове метода `PersistAll()` экземпляров, составляющих агрегат и достигающих его корня.

И в этом случае на помощь приходят агрегаты, предоставляя средство, позволяющее выяснить размеры сохраняемого графа, поскольку корень агрегата обозначается как сохраняемый. В этом отношении агрегаты упрощают дело.

На заметку

Объектно-реляционные преобразователи нередко можно настроить на определенную степень достижимости сохраняемости. Но даже если их и можно настроить подобным образом, агрегаты, на мой взгляд, могут оказать в этом большую помощь. Это означает, что агрегаты служат для определения степени достижимости при выполнении настройки.

А теперь рассмотрим более подробно основания для принятых до сих пор решений.

Основания для принятых решений

Почему я поступил именно так, а не иначе? Прежде всего, мне хотелось воспользоваться шаблоном Unit of Work, чтобы создать логическую единицу работы.

Это дает возможность внести многочисленные изменения в модель предметной области, собрать необходимую информацию в единице работы, а затем обратиться к единице работы, чтобы сохранить накопившиеся изменения в базе данных.

Единицу работы можно использовать по-разному. Я лично предпочитаю сделать ее как можно более прозрачной для потребителя, а следовательно, обращаться к хранилищу только с помощью метода `Add()` (а хранилище, в свою очередь, будет обращаться к единице работы).

Если восстановление осуществляется с помощью хранилища, реализация единицы работы позволяет внести некоторый объект для сбора информации об изменениях. В противном случае, при чтении может быть получен моментальный снимок для последующего управления изменениями со стороны единицы работы во время сохранения.

Кроме того, принимать решение по поводу сохранения или несохранения (методом `PersistAll()`) лучше всего вне хранилищ. В данном случае можно было бы вызвать метод `PersistAll()` непосредственно в классе `CustomerRepository`, но я решил этого не делать. Почему? Почему бы не предоставить хранилищам возможность полностью скрывать единицу работы? Это, пожалуй, можно было бы сделать, но часто возникает необходимость синхронизировать изменения, сделанные в нескольких агрегатах, а следовательно, и в нескольких хранилищах, в одной логической единице работы. Именно поэтому приведенный ниже код не только может быть написан, но и весьма типичен.

```
Customer c = new Customer();
_customerRepository.Add(c);

Order o = new Order();
_orderRepository.Add(o);

_ws.PersistAll();
```

Альтернативой ему может служить следующий код:

```
Customer c = new Customer();
_customerRepository.Add(c);
_customerRepository.PersistAll();

Order o = new Order();
_orderRepository.Add(o);
_orderRepository.PersistAll();
```

Но тогда получаются два разных экземпляра единицы работы и две карты идентичности, чего не должно быть, но в целях обсуждения мы можем допустить такое. В этом случае вполне возможны необычные результаты, если не соблюсти особую осторожность. Ведь все пять строк в первом варианте кода относятся к одному сценарию, и поэтому их, на мой взгляд, намного удобнее и понятнее трактовать как один сценарий, касающийся обращения как с единицей работы, так и с картой идентичности. Это означает, что сценарий должен быть рассчитан на обращение с одной единицей работы и с одной картой идентичности.

Еще одно затруднение может быть связано с тем, что сокрытие единицы работы в хранилище, скорее всего, означает две операции обращения к базе данных. А это, в свою очередь, означает, что нужно быть готовым ко вводу компенсирующих операций, когда такой сценарий дает непредвиденный результат. Если в приведенном выше примере сохранить в базе данных клиента, а не заказ, это, возможно, и не приведет к катастрофе, но может вызвать серьезные осложнения, что зависит от выбранной конструкции агрегата.

Иными словами, конструкция агрегатов должна быть такой, чтобы они находились в согласованном состоянии во время вызова метода `PersistAll()`. Но слабая связь между агрегатами, как правило, не удовлетворяет столь строгим требованиям. В таком случае можно выбрать второй вариант сценария. Но с другой стороны, по второму сценарию предполагается, что при вызове одного и того же метода `PersistAll()` сохраняются два совершенно не связанных друг с другом клиента, если оба клиента связаны с хранилищем, хотя это и не так важно, как группировать клиента вместе с его заказами. Ведь агрегаты связаны с объектами, а не с классами.

В пользу второго варианта говорит тот факт, что если один агрегат выбирается из одной базы данных, то второй агрегат сохраняется во второй базе данных на другом сервере. В таком случае проще всего иметь два экземпляра единицы работы: по одному на каждую базу данных. Таким образом, второе решение получается чуть менее связанным.

Если одна операция сохранения проходит успешно, а другая — нет, возникает еще одно затруднение. (На это можно, конечно, возразить, что физическая операция обращения к базе данных и единица работы — это не одно и то же, но в таком случае задача усложняется. Поэтому я считаю, что делать следует только то, что действительно нужно делать.)

На заметку

Можно было бы даже предоставить методу `Add()` возможность выполнить операцию обращения к базе данных, но это означало бы две разные семантики по сравнению с тем, что было рассмотрено выше. Ведь метод `Add()` пришлось бы непременно вызывать в нужный момент времени. Впрочем, для выбранного мной решения это не так важно, поскольку данный метод вызывается до метода `PersistAll()`.

Выполнение операции обращения к базе данных в методе `Add()` означает также, что речь не идет о накоплении всех изменений и последующем их сохранении при вызове метода `PersistAll()`. И этим оно отличается от выбранного мной решения.

И раз уж мы коснулись этой темы, то почему бы не инкапсулировать все эти действия в объекте-сущности, написав следующий код?

```
Customer c = new Customer()
c.Name = "Saab";
c.Save();
```

Это несколько отличается от выбранного мной стиля проектирования и нарушает принцип единственной ответственности (ПЕО) [Martin PPP], а также соответствует низкому уровню НС. Впрочем, это дело личных предпочтений.

Но ведь нам ничего не мешает добиться избранным методом тех же самых результатов поочередного сохранения агрегатов, если предоставить хранилищам возможность скрывать единицу работы и карту идентичности. Это можно реализовать в коде следующим образом:

```
Customer c = new Customer();
_customerRepository.Add(c);
_ws1.PersistAll();

Order o = new Order();
_orderRepository.Add(o);
_ws2.PersistAll();
```

На заметку

Результат выполнения приведенного выше сценария будет таким же, как и при использовании одной ссылки `_ws`, хотя это зависит от конкретного примера.

В данном случае имеется в виду, что при желании можно иметь одну или же несколько единиц работы и карт идентичности. Мне кажется, что это более гибкий

подход, именно поэтому я и выбрал его. В частности, я предпочитаю рассматривать единицу работы как нечто принадлежащее потребителю модели предметной области (т.е. уровню приложения или же уровню представления), чем самой модели.

Ели допустить, что у каждого хранилища имеется своя карта идентичности, то дело усложнится при обращении к одному и тому же заказу из двух разных хранилищ — по крайней мере, если вносить изменения в одном и том же логическом порядке (но в два разных экземпляра объектов) в обоих хранилищах.

Итак, вся проблема сводится к допустимой степени риска зафиксировать экземпляры объектов не полностью из-за того, что при вызове метода `PersistAll()` приходится иметь дело с бо́льшим числом экземпляров, чем предполагалось, или же риска забыть зафиксировать внесенные изменения из-за того, что приходится помнить, к каким именно хранилищам следует обращаться для выполнения операций по методу `PersistAll()`.

Нельзя сказать, что рассматриваемое здесь решение не лишено недостатков, но опять же я лично предпочитаю, чтобы карта идентичности и единица работы принадлежали одному сценарию.

На заметку

Как всегда, выбор используемой модели программирования остается за вами. У каждой из них имеются свои достоинства и недостатки.

Итак, я все чаще стал упоминать карту идентичности вместе с единицей работы. В таком сочетании они очень часто встречаются не только в этой книге, но в программных продуктах, например, Persistence Manager в среде JDO [Jordan/Russell JDO] и Session в среде Hibernate [Bauer/King HiA].

Я даже подумывал написать о том, что для этой цели было бы неплохо создать шаблон. Но когда я обсуждал этот вопрос с Мартином Фаулером, он заметил, что уже рассматривал его в своей книге [Fowler PoEAA]. Поэтому я решил, что этого будет достаточно, и не стал повторяться.

До сих пор было много рассуждений и мало действий. Поэтому перейдем непосредственно к построению поддельного механизма и посмотрим, к чему это нас приведет.

Построение механизма подделки

Итак, приступим (хотя бы для начала) к проектированию в интерфейсном стиле. Мы уже рассматривали выше некоторые методы интерфейса, но теперь нам придется начать все с самого начала. Усеченный вариант интерфейса для уровня абстракции (получивший ранее в этой главе название `IWorkspace`) может иметь следующий вид:

```
public interface IWorkspace
{
  object GetById(Type typeToGet, object idValue);
  void MakePersistent(object o);
  void PersistAll();
}
```

Такой интерфейс достаточно прост. Первый вызываемый в нем метод `GetById()` служит для восстановления объекта из базы данных. Для этого достаточно указать предполагаемый тип объекта и значение его идентичности.

Второй вызываемый метод `MakePersistent()` служит для связывания новых экземпляров с экземпляром интерфейса `IWorkspace` таким образом, чтобы они сохранялись при последующем вызове метода `PersistAll()`. И наконец, метод `PersistAll()` служит для сохранения в базе данных того, что находится в единице работы.

Метод `MakePersistent()` не требуется, если экземпляр объекта считан из базы данных с помощью метода `GetById()`, поскольку в этом случае экземпляр объекта уже связан с единицей работы.

Согласитесь, что интерфейс API пока что достаточно прост, а это, как мне кажется, очень важно для того, чтобы исключить чрезмерную сложность на данном уровне абстракции. В то же время интерфейс не вполне отвечает предъявляемым к нему требованиям, и поэтому должен быть дополнен.

Дополнительные свойства механизма подделки

Прежде всего, необходимо правильно трактовать операции обращения к базе данных (или транзакции) как очень важное понятие. Но с другой стороны, это понятие не так важно для разработчиков пользовательского интерфейса. Имеется в виду, что ответственность за управление транзакциями не стоит возлагать на разработчика пользовательского интерфейса, поскольку это слишком отвлекает его от главной задачи, что очень важно. Тем не менее область действия транзакций не должна ограничиваться только предварительно заданным набором возможных транзакций.

Наша цель очень близка к декларативным транзакциям в COM+, но в то же время мы выбрали совсем другой интерфейс API. Вместо того чтобы задавать атрибуты в классах для описания требуемых транзакций, предположим, что всю необходимую работу, связанную с явной транзакцией, должен выполнить метод `PersistAll()` — даже если его не просят об этом явно.

На первый взгляд, это может показаться опытным разработчикам слишком простым решением. И я так думаю, поскольку считаю, что обработка транзакций настолько важна, что ее лучше организовать вручную. Но если это нужно делать в 90% всех случаев, то в методе `PersistAll()` достаточно организовать обработку явных транзакций, что делается так же просто.

Несмотря на кажущуюся простоту, у такого решения, безусловно, имеются трудности. К самым характерным из них относится регистрация. Допустим, что требу-

ется зарегистрироваться на сервере базы данных. Операция регистрации не должна завершаться неудачно, если неудачно завершается обычная работа. Но это затруднение нетрудно устранить, воспользовавшись для регистрации отдельным экземпляром рабочего пространства. Если же для этой цели вообще не требуется абстрактный уровень, то регистрацию следует реализовать в виде отдельной службы, которая, скорее всего, не будет иметь ничего общего с уровнем абстракции. На самом деле для регистрации чаще всего используются программные продукты независимых поставщиков или нечто, вроде log4net [Log4Net]. В этом случае отсутствует какая-либо связь или влияние со стороны интерфейса API для транзакций на уровне абстракции.

В качестве еще одной трудности можно указать на необходимость использовать метод `GetById()`, чтобы работать с той же самой транзакцией, что и при последующем вызове метода `PersistAll()`. Это не происходит по умолчанию, но если требуется, то может быть организовано с помощью следующего вызова метода `GetById()`:

```
void BeginReadTransaction(TransactionIsolationLevel til)
```

Более того, для запроса исключающей блокировки может быть осуществлено переопределение метода `GetById()`, хотя это делается с использованием дескриптора предупреждения. При этом вы должны быть уверены в своих действиях! В частности, после вызова метода `BeginReadTransaction()` должно отсутствовать всякое взаимодействие с пользователем, а перед вызовом метода `PersistAll()` — чтение с исключающей блокировкой.

Но какое отношение все это имеет к подделке? Дело в том, что поддержка предназначена только для однопользовательских сценариев, и поэтому семантика транзакций для нее не так важна, а ради простоты ею можно вообще пренебречь. Тем не менее, код потребителя может быть написан с учетом обработки транзакций при использовании подделки, чтобы не менять его, когда настанет момент переходить от подделки к настоящей инфраструктуре.

Предвижу озабоченный вопрос опытного разработчика: “А куда же девался метод `Rollback()`?”

На мой взгляд, откат транзакции в интерфейсе API не так важен. Если ответственность за фиксацию или откат транзакции возлагается на метод `PersistAll()`, то изменения либо сохраняются полностью либо вообще не сохраняются. (Об откате транзакции потребитель уведомляется посредством исключения.)

Исключением из этого правила может быть отмена транзакции после ее начала по методу `BeginReadTransaction()`. Для этой цели служит следующий метод:

```
void Clean()
```

В этом случае произойдет откат текущей транзакции, а также очистка единицы работы и карты идентичности. Метод `Clean()` удобно использовать после неудачной транзакции, поскольку на уровне абстракции NWorkspace вообще не предпринимается попытка отмены изменений в экземплярах объектов из модели предметной

области. Конечно, это зависит от причины неудачи транзакции, но самый простой выход из этого положения — начать все сначала.

Некоторые осложнения могут привести к повторной попытке выполнения транзакции внутри метода `PersistAll()`. В тупиковой ситуации такая попытка может быть совершена несколько раз, прежде чем транзакция будет признана неудачной. Это еще одна особенность, упрощающая задачу программиста, работающего на уровне потребителя, что дает ему возможность сосредоточиться на том, что для него важнее, а именно: на организации удобного пользовательского интерфейса, не отвлекаясь на множество протоколов.

До сих пор речь шла, в основном, о важных функциональных возможностях уровня абстракции NWorkspace, но не его поддельного варианта. Поэтому перейдем непосредственно к подделке и ее реализации.

Реализация подделки

Мне бы не хотелось слишком вдаваться в детали реализации подделки, рассмотрев ее лишь на концептуальном уровне, чтобы дать общее преставление о ее построении и применении.

В реализации подделки используются два слоя карт идентичности. Первый слой аналогичен обычным картам идентичности в базовых структурах сохраняемости. В этом слое отслеживаются все объекты-сущности, считанные в текущем сценарии. А во втором слое карт идентичности имитируется механизм сохранения, поэтому здесь экземпляры сохраняются не на уровне сценария, а на глобальном уровне (т.е. для всех сценариев используется один и тот же ряд карт идентичности).

Так, если вызывается метод `GetById()`, а в карте идентичности осуществляется поиск идентификационного номера по запрашиваемому типу объекта, циклического обращения к базе данных не последует (а в подделке не произойдет переход во второй слой карт идентичности). С другой стороны, если идентификационный номер не найден в первом слое карт идентичности, он выбирается из второго слоя, копируется в первый слой и затем возвращается потребителю.

Метод `MakePersistent()` достаточно прост. В нем экземпляр объекта связывается с первым слоем карт идентичности. А когда приходит время для вызова метода `PersistAll()`, все экземпляры объектов копируются из первого слоя во второй. Все очень ясно и просто.

Но это лишь самые простые функциональные возможности. Было бы интересно рассмотреть и некоторые затруднения. В частности, хотелось бы вообще исключить влияние подделки на модель предметной области. Ведь если такое влияние имеется, то мы возвращаемся к тому, с чего начали, отвлекаясь от модели предметной области в сторону инфраструктуры, а еще хуже — в сторону подделки.

Например, мне может быть неизвестно, к какому классу принадлежат поля идентификации. В настоящей инфраструктуре, скорее всего, будут известны некоторые метаданные. Такие метаданные можно было бы обнаружить и в подделке, но тогда

подделке должно быть известно, как обращаться (хотя бы теоретически) с метаданными разного формата, что мне совсем не подходит.

В качестве самого простого решения в этом случае можно допустить, чтобы свойство (или поле) называлось `Id`. Если разработчик модели предметной области использовал другое условное обозначение, оно может быть описано для подделки при создании экземпляра интерфейса `FakeWorkspace`.

Опять же, этой информации может оказаться больше, чем требовалось, но она позволяет нам выяснить дополнительные вопросы, которые можно и нужно решать на стадии создания экземпляра подделки по сравнению с реализациями уровня абстракции NWorkspace.

В качестве еще одного примера можно прочитать из файла или же сохранить в нем данные следующим образом:

```
// Некий первоначальный потребитель
IWorkspace ws = new
  NWorkspaceFake.FakeWorkspace("c:/temp/x.nworkspace");

// Выполнить полезные действия...

((NWorkspaceFake.FakeWorkspace)ws).
  PersistToFile("c:/temp/x.nworkspace");
```

При подробном рассмотрении НС и механизма подделки у вас могло сложиться следующее впечатление: если сначала выбрать механизм подделки, то затем придется выбрать инфраструктуру, поддерживающую НС. Но это совсем не так. Более того, отсутствие поддержки НС в инфраструктуре совсем не означает, что применять РПТ станет труднее. Так считается по традиции, хотя это вовсе не обязательно.

Раз уж мы коснулись РПТ, то выясним, какое влияние подделка оказывает на блочные тесты.

Воздействие подделки на блочные тесты

Этому воздействию, безусловно, подлежат не все тесты. Большинство тестов следует писать с как можно более изолированными классами из модели предметной области и без единого обращения к хранилищам. В частности, тесты должны быть написаны в процессе разработки всей логики, обычно сопровождающей классы из модели предметной области. Такие классы вообще не подлежат воздействию.

Блочные тесты, в которых предусмотрено взаимодействие с хранилищами, подлежат воздействию, причем в положительном смысле. Можно, конечно, спорить, что такие тесты имеют большее отношение к тестированию интеграции, но это совсем не обязательно. Ведь хранилища также являются блоками, и поэтому их тесты считаются блочными.

Даже если тестирование интеграции осуществляется с использованием хранилищ, было бы неплохо написать тесты как можно раньше, причем написать (вместе

с хранилищами) таким образом, чтобы использовать их и после выбора подходящей инфраструктуры.

Мне представляется вполне обоснованным получение таких тестов, которые можно было бы выполнять при наличии как механизма подделки, так и настоящей инфраструктуры. Таким образом, тесты можно выполнять в повседневной работе с механизмом подделки (из соображений времени выполнения) и два раза в день с инфраструктурой для проверки автоматических конструкций.

Такой подход позволяет также довольно долго работать без выбранной инфраструктуры. Разумеется, нужно подумать немного и о сохраняемости, особенно в первых проектах, выполняемых в стиле ППО, где очень важно с самого начала работать по принципу неоднократных повторений. Но с опытом задержки, связанные с вводом функций сохраняемости в процессе разработки, станут более короткими, а код — совершенно ясным.

Кроме того, такой подход позволяет изменять периодичность рефакторинга кода и получать мгновенную реакцию, какими бы ни были результаты. Сначала нужно, чтобы все работало с подделкой (сделать это намного легче и быстрее, чем добиться работоспособности всей системы в целом, включая и базу данных). Достигнув желаемого результата, можно переходить к приведению системы в рабочее состояние вместе с инфраструктурой. Преимущество данного подхода заключается в том, что он поощряет к более разумному рефакторингу кода, который обычно вызывает трудности, особенно если нет уверенности в конечном результате. В данном случае намного проще осуществить попытки реорганизовать код.

Разумеется, в блочных тестах и в "настоящем" коде очень важно стремиться исключить дублирование кода, хотя не менее важно правильно структурировать код. Поэтому я стараюсь писать тесты один раз, но выполнять их как для подделки, так и для настоящей инфраструктуры. Следует, однако, иметь в виду, что это касается только некоторых тестов. Ведь большинство тестов вообще не связаны с хранилищами, и поэтому их важно разделять по данному признаку.

Структура тестов, связанных с хранилищами

Один из возможных подходов состоит в написании базового класса для каждого набора тестов, связанных с хранилищами. А для нужной настройки интерфейса `IWorkspace` обычно используется шаблон Template Method (Шаблонный метод). Итак, создадим сначала базовый класс.

```
[TestFixture]
public abstract class CustomerRepositoryTestsBase
{
  private IWorkspace _ws;
  private CustomerRepository _repository;

  protected abstract IWorkspace _CreateWorkspace();

  [SetUp]
```

```
  public void SetUp()
  {
    _ws = _CreateWorkspace();
    _repository = new CustomerRepository(_ws);
  }

  [TearDown]
  public void TearDown()
  {
    _ws.Clean();
  }
}
```

Затем создадим подкласс.

```
public class CustomerRepositoryTestsFake :
    CustomerRepositoryTestsBase
{
  protected override IWorkspace _CreateWorkspace()
  {
    return new FakeWorkspace("");
  }
}
```

Итак, мы получили структуру для тестов, связанных с хранилищами, а что же сами тесты? Их можно создавать в самых разных стилях, но я лично предпочитаю определять максимум функций и свойств в базовом классе, в то же время оставляя в подклассе возможность для принятия решения относительно реализации определенного теста в данный момент. При таких требованиях тест, написанный в выбранном мной стиле, выглядит (сначала в упрощенном виде в базовом классе) следующим образом:

```
[Test]
public virtual void CanAddCustomer()
{
  Customer c = new Customer();
  c.Name = "Volvo";
  c.Id = 42;
  _repository.Add(c);
  _ws.PersistAll();
  _ws.Clean();

  // Проверка
  Customer c2 = _repository.GetById(c.Id);
  Assert.AreEqual(c.Name, c2.Name);

  // Очистка
  _repository.Delete(c2);
  _ws.PersistAll();
}
```

Следует заметить, что второй уровень карт идентичности не очищается при выполнении операции `new FakeWorkspace("")`, поскольку карты идентичности на этом уровне подделки не относятся к типу `static`, а следовательно, не подлежат воздействию при воссоздании *экземпляра* подделки. Именно так все и происходит при наличии базы данных. Одного только установления соединения с ней недостаточно, например, для очистки таблицы `Customers`.

Совсем неплохо, если подделка действует именно таким образом. Ведь тогда очистку после тестов с подделкой нужно будет выполнить точно так же, как и после тестов с настоящей базой данных, если выбрать именно такой способ проверки базы данных.

Разумеется, в интерфейсе `IWorkspace` должен присутствовать метод `Delete()`, функции которого мы еще не рассматривали, иначе очистка окажется невозможной. На самом деле функции метода `Delete()` очень интересны, несмотря на кажущуюся его простоту. Ведь ему требуется отдельная карта идентичности для подделки в единице работы. Экземпляры объектов, зарегистрированные для удаления, будут храниться вплоть до вызова метода `PersistAll()`, когда удаление выполняется окончательно.

Для поддержки этих функций в интерфейсе `IWorkspace` вводится следующий метод:

```
void Delete(object o);
```

К сожалению, вместе с вводом этого метода возникает следующий вопрос: что должно произойти с отношениями, имеющимися у объекта, который удаляется? Ответить на этот вопрос непросто. Опять же, для того чтобы выяснить степень каскадирования операции удаления, требуются дополнительные метаданные. Но такие метаданные имеют отношение к настоящей инфраструктуре и в данном случае бесполезны. (В настоящий момент для подделки принято условие, не предусматривающее каскадирование.)

Но вернемся к самим тестам. Когда дело доходит до теста `CanAddCustomer()`, то в подклассе обычно имеются три варианта выбора. Первый вариант состоит в том, чтобы ничего не предпринимать. В этом случае тест выполняется так, как он определен в базовом классе. Остается только надеяться, что это именно то, что нужно.

Второй вариант выбирается в том случае, если в данный момент для конкретного подкласса отсутствует отдельный тест. В подклассе это выглядит следующим образом:

```
[Test, Ignore("Пока еще не поддерживается...")]
public override void CanAddCustomer() {}
```

Таким образом, при выполнении теста данный код будет явно сигнализировать о временном игнорировании самого теста.

И наконец, если для подкласса отдельный тест не предусматривается, то в этом подклассе можно написать следующий код:

```
[Test]
public override void CanAddCustomer()
{
```

```
  Console.WriteLine
    ("Тест CanAddCustomer() не поддерживается подделкой.");
}
```

Впрочем, вы можете и забыть о тесте, написав следующую строку кода и опустив атрибут `[Test]`:

```
public override void CanAddCustomer() {}
```

Это можно сделать и по-другому, но данный стиль представляет собой некую золотую середину между количеством кода и риском "забыть" о тестах.

Конечно, такой стиль не вполне соответствует принципу YAGNI (это вам не понадобится), поскольку в данный момент у нас пока еще нет другой реализации, кроме подделки, но его следует рассматривать просто как краткое указание на то, что может произойти впоследствии.

На заметку

Такой стиль написания тестов может быть также использован и для нескольких реализаций каждого хранилища.

Для многих приложений данный подход к тестированию всей системы в целом едва ли вообще возможен, особенно на последних стадиях проектирования. Он, скорее всего, годится для тестирования на ранней стадии проектирования и демонстрации приложений, но и это уже хорошо, если действительно помогает, а еще лучше, если помогает и дальше.

Однако в реальной жизни, где всегда находится место для исключений из общих правил, мне лично приходится уделять основное внимание написанию некоей основы тестов для реализации подделки. Кроме того, я пишу тесты операций создания, чтения, обновления и удаления (CRUD) для реализации подделки, но в этом случае я стараюсь также передать их по наследству тестам базы данных, чтобы проверить особенности преобразования.

Впрочем, независимо от того, используется принцип организации уровня абстракции или нет, рано или поздно возникают вопросы, связанные с тестированием базы данных. Я попросил своего коллегу Фила Нельсона написать об этом отдельный раздел, который и приводится ниже.

Тестирование базы данных

Фил Нельсон

На определенной стадии работы в стиле ППО с разного рода автоматическим тестированием возникает потребность в проверке того, что имеет непосредственное

отношение к базе данных. На первый взгляд, в этом нет ничего сложного. Нужно лишь создать тестовую базу данных, связать с ней разрабатываемое приложение и начать тестирование.

Итак, рассмотрим автоматическое тестирование более подробно, предположив, что первый тест написан с помощью инструмента JUnit или NUnit. Этот тест загружает из базы данных объект предметной области `User` и проверяет правильность установки всех его свойств. Такой тест можно выполнять неоднократно, и при этом он постоянно работает. Допустим далее, что написан тест, проверяющий возможность обновления полей, а поскольку известно, что в базе данных имеется конкретный объект, то происходит их обновление.

Этот тест работает, но возникает следующее затруднение. Тест чтения из базы данных уже не действует, поскольку только что были изменены данные, составляющие его основу, и теперь они не соответствуют предположениям в этом первом тесте. В качестве выхода из этого затруднения нужно сделать так, чтобы в обоих тестах не использовались общие данные, создав другой объект `User` для теста обновления полей. Теперь оба теста действуют независимо друг от друга, но самое главное — добиться того, чтобы при переходе от одного теста к другому отсутствовали остаточные эффекты. Для тестов базы данных всегда существуют предварительные условия, которые необходимо выполнять. При написании таких тестов следует принимать во внимание тот факт, что базу данных приходится устанавливать в исходное состояние, чтобы удовлетворить предварительным условиям теста. Подробнее об этом речь пойдет далее.

В определенный момент возникает потребность в написании тестов для всех состояний жизненного цикла базового объекта `User`. Это состояния *создания*, *чтения*, *обновления* и *удаления* (CRUD). В разрабатываемом приложении возможны и другие операции обращения к базе данных. Например, объект `User` может оказывать помощь в соблюдении правила, определяющего максимальное число неудачных попыток регистрации. А при проверке обновления неудачных попыток регистрации и полей последней регистрации выясняется, что такое обновление не действует. Тест обновления не выявил ошибку, поскольку в поле базы данных уже установлено предполагаемое значение. Для выяснения этого недостатка теста вам, возможно, потребуется немного времени, но у молодого специалиста на это может уйти не один час. Для этого ему потребуется не столько умение программировать, сколько ясное представление о том, как код связывается с базой данных и как полученные из нее данные изолируются в тесте от других имеющихся данных. Ведь он может просто не обратить внимания на то, что поле `LastUpdateDate` (Дата последнего обновления) не устанавливается в исходное состояние в промежутке между тестами. В конце концов код служит для сокрытия базы данных как отдельной составляющей реализации системы, не так ли?

В такой момент приходит осознание того, что изолирование проверяемых данных для каждого теста превращается в более сложную задачу, чем хотелось бы. Если вы раньше не до конца понимали, насколько важно сбрасывать данные состояния в промежутке между тестами, то теперь осознали это в полной мере. Правда, среда тес-

тирования xUnit делает многое за вас. Для этой цели в ней имеется блок кода настройки и очистки. Но, как и многое другое в программировании, эту задачу можно решить разными методами, у каждого из которых имеются свои достоинства и недостатки. Нужно лишь найти компромиссное решение, в котором недостатки оптимально уравновешиваются достоинствами выбранного метода.

Доступные методы можно разделить на четыре категории.

- Установка базы данных в исходное состояние перед каждым тестом.
- Сохранение состояния базы данных во время выполнения тестов.
- Сброс данных только для теста или настройка выполняемых тестов перед выполнением.
- Тестирование блока кода отдельно от проверки обращения к базе данных.

Установка базы данных в исходное состояние перед каждым тестом

На первый взгляд, данный метод кажется наиболее приемлемым, хотя он и может отнять больше всего времени. К его преимуществам следует отнести тот факт, что в начале теста известно состояние всей системы. При этом не нужно беспокоиться о непредсказуемых взаимодействиях при выполнении тестов вследствие того, что все они начинаются одновременно. А к его недостаткам следует отнести затрачиваемое время. После того, как будут пройдены начальные стадии проекта, задержки на выполнение набора тестов становятся все более заметными. Так, если тесты при блочном тестировании выполняются после каждого изменения, данный процесс может очень быстро стать слишком продолжительным. Впрочем, из этого положения можно найти не один выход, хотя все зависит от многих факторов, начиная от выбранного типа архитектуры и кончая типом используемой СУБД.

Один простой, хотя и небыстрый способ состоит в восстановлении базы данных из резервного состояния перед каждым тестом. Для многих СУБД такой способ оказывается непрактичным из-за времени, которое для этого требуется. Но для некоторых СУБД он вполне пригоден. Так, если программируется обращение к файловой базе данных, то для целей тестирования достаточно разорвать соединение с базой данных и скопировать данные в файл, чтобы затем вернуться в исходное состояние. Еще одну возможность предоставляет база данных в оперативной памяти, например HSQL для Java, которая очень быстро восстанавливается. Даже если используется такая стандартная СУБД, как Oracle или SQL Server, при выборе достаточно гибкого способа обращения к базе данных можно добиться оптимального переключения между базой данных в оперативной памяти и файловой базой данных для выполнения тестов. Это особенно справедливо в том случае, если используется объектно-реляционный преобразователь, отвечающий за организацию вызовов SQL и различающий разные диалекты этого языка структурированных запросов.

В проекте DbUnit предлагается другой способ установки базы данных в исходное состояние перед выполнением тестов. По существу, это базовая структура для среды

JUnit, где можно определить "наборы данных", загружаемые в очищенные таблицы перед выполнением каждого теста. Аналогичная система имеется и в среде Ruby on Rails. Она позволяет описывать данные тестируемого класса в открытом формате YAML (YAML Ain't Markup Language — YAML не язык разметки документов) и применять их при настройке теста. Эти инструменты вполне работоспособны, но при вводе такой информации в базу данных могут возникнуть затруднения в связи со слишком медленной регистрацией операций. Еще один подходящий метод заключается в применении утилит массовой загрузки данных, которая может не так тщательно регистрироваться. В SQL Server этот метод реализуется следующим образом. Сначала вариант восстановления базы данных опробуется на тестовой базе данных в режиме простого восстановления (Simple Recovery). Благодаря этому исключается бо́льшая часть регистрации и повышается производительность. А затем в процессе проектирования с использованием тестов выполняется следующее.

- Ввод информации в тестовую базу данных, которая будет служить лишь источником очищенных данных для тестов.
- Вызов команд `transact sql` для экспорта тестовых данных в файлы.
- Запись команд `transact sql` для массового ввода данных в эти файлы.

После этого при настройке теста выполняется следующее.

- Усечение всех таблиц. Эта операция не регистрируется и поэтому выполняется очень быстро.
- Выдача сформированных ранее команд массового копирования для загрузки тестовой информации в тестовую базу данных.

В качестве разновидности этого метода настраивается тестовая база данных и в нее загружается исходная тестовая информация обычными средствами управления базой данных. А затем файлы данных, составляющие основу базы данных, отделяются от ее сервера, при условии что эта операция поддерживается в базе данных. Эти файлы следует скопировать, поскольку они будут служить источником для очищенных тестовых данных. После этого создается код, выполняющий следующее.

- Отсоединение сервера от его файлов данных.
- Копирование очищенных тестовых данных в конкретные файлы на сервере.
- Присоединение базы данных к полученной копии.

Как правило, эти операции выполняются довольно быстро, и поэтому вполне доступны для выполнения в среде тестовой конфигурации, а если объем данных невелик, то и в самой тестовой среде.

Еще одна разновидность данного метода поддерживается некоторыми инструментами объектно-реляционного преобразования для составления схемы базы данных на основании информации преобразования. В этом случае модель предметной области может быть использована в тестовой среде для наполнения данными, которые требуются для каждого набора тестов, а возможно, и для каждой тестовой конфигурации.

Сохранение состояния базы данных во время выполнения тестов

На первый взгляд, эта задача имеет большее отношение к управлению базой данных. Но ее можно решить более простым методом. Его суть заключается в выполнении каждого теста во время транзакции, а по завершении теста следует откат транзакции. Реализовать такой метод будет непросто, поскольку транзакции не всегда открыты для общедоступных интерфейсов, но опять же, все зависит от выбранной архитектуры. Так, для сред .NET Рой Оширови нашел очень простое решение, основанное на поддержке в ADO.NET связи с COM+-транзакциями. Данное решение позволяет помещать атрибут `[Rollback]` в отдельные тестовые методы и выполнять эти методы в их собственной COM+-транзакции. По завершении тестового метода происходит автоматический откат транзакции, который не требует написания дополнительного кода и не зависит от функциональных возможностей класса в отношении очистки данных.

Главное преимущество такого метода заключается в том, что тесты могут пребывать в блаженном неведении относительно того, что происходит на более низком по сравнению с ними уровне. Функциональные возможности данного метода были реализованы в проекте XtUnit [XtUnit] и в его расширении среды тестирования NUnit. По существу, это самый простой способ поддержания базы данных в подобающем состоянии во время тестирования, хотя он обходится не бесплатно. Сам характер транзакций предполагает их регистрацию, что приводит к увеличению времени выполнения. В COM+-транзакциях применяется координатор распределенных транзакций (Distributed Transaction Coordinator), и поэтому распределенные транзакции выполняются медленнее, чем локальные. А сочетание и тех и других транзакций может сильно сказаться на времени выполнения набора тестов.

Если же проверяется семантика самих транзакций, то использование данного метода может вызвать дополнительные трудности. Таким образом, данное решение может оказаться идеальным или, по крайней мере, более приемлемым, чем остальные, когда число тестов начнет нарастать — все зависит от особенностей выполняемого проекта. Правда, при увеличении времени выполнения тестов переписывать большую часть кода не придется, поскольку данное решение прозрачно для теста. Остается либо приноровиться к нему, либо выбрать другие методы, как только масштабы проекта начнут увеличиваться.

Сброс данных для теста перед его выполнением

Этот метод освобождает разработчика и его систему от необходимости устанавливать в исходное состояние всю базу данных перед началом теста. Вместо этого базе данных выдаются соответствующие команды до и после теста, опять же, в методах настройки теста или очистки данных. Но при этом приходится меньше заниматься одним тестируемым классом. Рассмотренные выше методы могут по-прежнему применяться, но теперь имеется дополнительная возможность выдачи простых команд ввода, обновления или удаления данных. Выполнение таких команд теперь

в меньшей степени сказывается на производительности — даже с учетом регистрации. Но если система поддерживает настройку базы данных, то целесообразно сократить действия, связанные с регистрацией.

Впрочем, у данного метода имеются и недостатки. Стоит вам допустить, что вы точно знаете, на какие именно данные оказывает влияние ваш тест, вы сразу же начинаете путаться не только о том, что именно проверяемый тест делает с данными, но и в том, какой код требуется в дополнение к проверяемому коду, причем это может происходить совершенно незаметно. В итоге тесты могут прерываться, не выявляя нарушенный код, а зарегистрированные программные ошибки могут и не оказаться таковыми, хотя эксплуатационные характеристики системы остаются без изменения.

Данный метод можно применять успешно, но мой многолетний опыт создания тестов показывает, что он, скорее всего, окажется неработоспособным без каких-либо веских на то оснований, хотя вероятность такого исхода можно все же свести к минимуму. Если сымитировать классы, которые требуются для проверяемого кода, такой код не будет воздействовать на базу данных. Это, в принципе, очень похоже на блочный тест. Но имитация оказывается малопригодной для автоматизации тестов, где требуется проверять взаимодействие между настоящими классами. Во всяком случае для этого требуется такое архитектурное решение, которое допускает простую замену внешних классов средой тестирования. Если вы собирается сделать это, то естественным образом окажетесь на пути к последнему из рассматриваемых методов тестирования базы данных.

Не забывайте о развитии схемы базы данных

Со временем вы, без сомнения, обнаруживаете, что сами вносите изменения в базу данных. Если на начальных стадиях проекта возникает потребность в непосредственных изменениях базы данных, а код правится после сдачи системы в эксплуатацию, значит, пора подумать о том, чтобы сделать все это частью процесса проектирования. Эту задачу значительно упрощает работа с тестовой системой.

Я лично предпочитаю создавать и выполнять изменяемые сценарии применительно к базе данных. Такие сценарии можно выполнять сразу же после установки базы данных в исходное состояние, но поскольку все состояние базы данных сохраняется в среде разработки с контролем исходного кода, то сценарии целесообразно создавать и использовать для модификации среды тестирования. Если тесты проходят, далее производится полная проверка кода, а сценарии автоматически настраиваются на выполнение в среде контроля качества, если в этом возникает потребность.

Еще лучше делать это на сервере сборки, поскольку проверка выполнения изменяемых сценариев зачастую осуществляется до их переноса в работающие системы. Разумеется, это подразумевает регулярную установку в исходное состояние среды контроля качества, чтобы привести ее в соответствие с требованиями рабочей среды. У такого процесса может быть немало разновидностей, но самое главное — планировать все заранее, чтобы вовремя вносить изменения как в код, так и в базу данных.

Тестирование блока кода отдельно от проверки обращения к базе данных

Тестирование кода, взаимодействующего с базами данных, является едва ли не самым животрепещущим вопросом, активно обсуждаемым в списках рассылки и форумах по гибкой разработке программного обеспечения. Среди тех, кто применяет на практике (и по-прежнему определяет) методы РПТ, весьма ценится код, который может быть проверен независимыми блоками по мере его написания. В соответствии с принципами РПТ нецелесообразно писать тест для класса и начинать его реализацию с прямого обращения к базе данных. Вместо этого обращение к базе данных поручается другому объекту, отвечающему за абстрактное представление базы данных, которая заменяется заглушкой или имитацией при написании теста. После этого пишется код, фактически реализующий абстрактное представление базы данных, и затем проверяется правильность вызова в нем объектов доступа к данным, составляющим основу базы данных. И наконец, создается ряд тестов, проверяющих правильность функционирования инфраструктуры, связывающей обращения к настоящей базе данных с написанным кодом. Как всегда, большинство задач программирования решается на очередном уровне косвенности. Но вся суть в деталях.

Прежде всего уточним ряд определений. Заглушка — это фрагмент заменяющего кода, который фактически ничего не делает, но и не осложняет выполнение вызывающего кода. Так, заглушка для обращения к базе данных (например, с помощью класса `Statement` интерфейса JDBC) организуется посредством пустого класса, реализующего интерфейс для класса `Statement` и возвращающего при вызове результирующее множество `ResultSet`, игнорируя, скорее всего, передаваемые ему параметры и, конечно, не выполняя обращение к базе данных.

То же самое выполняет и имитирующий объект `Statement`. Кроме того, он допускает установку определенных предположений. Аналогично заглушке, имитирующая команда используется в тесте вместо настоящей команды, но по завершении теста имитирующая команда проверяет, насколько правильно она выполнялась. В качестве *предположений* для объекта `Statement` могут служить значения любых параметров, которые требуются при вызове, количество вызовов метода `executeQuery` — выполнить запрос, правильный оператор SQL, передаваемый этому методу, и прочее. Иными словами, имитации можно сначала указать, чего следует ожидать, а затем попросить ее проверить, насколько эти ожидания оправдались по завершении теста.

Вы, наверное, согласитесь, что блочный тест для класса `User` вообще не должен быть связан с тем, что происходит в базе данных. Если этот класс правильно взаимодействует с абстрактным представлением базы данных, то можно предположить, что база данных правильно выполняет свои функции или, по крайней мере, этот факт может быть проверен с помощью кода доступа к данным. Остается лишь убедиться в том, что проверяемый код правильно передает все поля коду доступа к базе данных. Но если бы это было так просто всегда!

Вполне логичным следствием из всего сказанного выше была бы имитация в коде реализаций многих классов. В этом случае члены и свойства такой имитации пришлось бы наполнять данными, удовлетворяющими требованиям тестов. Для этой цели потребовалось бы написать немало кода, что затруднило бы его сопровождение. Рассмотрим другой вариант. Во многих языках программирования появились новые базовые структуры, позволяющие создавать имитирующие объекты из определений настоящих классов или интерфейсов. Такие динамически создаваемые имитации дают возможность устанавливать предположения, задавать предполагаемые возвращаемые значения и выполнять проверку, не создавая много кода для имитации самого класса. Данный метод носит общее название динамических имитаций и состоит в том, чтобы просто передать базовой структуре класс для имитации и получить обратно объект, реализующий его интерфейс.

Имеется немало источников информации об имитации кода, но намного меньше источников информации об эффективной имитации кода доступа к базе данных. Код доступа к данным обычно связан со следующими составляющими механизма такого доступа: соединениями, командными объектами, транзакциями, результатами и параметрами. Библиотеки доступа к данным послужили толчком для создания самых разных вспомогательных методов доступа к данным, призванных упростить такой код. Но по-видимому, все эти методы и библиотеки, положенные в их основу, в том числе JDBC и ADO.NET, были написаны без учета потребностей тестирования. Несмотря на то что многие из этих инструментов обеспечивают абстрактное представление доступа к данным, сымитировать все составляющие механизма доступа к данным оказывается очень трудно. Нерешенным остается также вопрос проверки преобразования всех тех полей, которые оказываются в промежутке между кодом доступа к данным и остальными классами. В связи с этим ниже приведен ряд рекомендаций, помогающих преодолеть упомянутые трудности.

Проверяйте все, что только можно, не касаясь самого кода доступа к базе данных. Такой код должен выполнять только то, без чего нельзя обойтись. Как правило, для этого достаточно операций создания, чтения, обновления и удаления (CRUD). Если имеется возможность проверить все функции классов, не обращаясь к базе данных, напишите тесты базы данных, которые сводятся к простым операциям CRUD. С помощью вспомогательных методов вы можете проверить свои предположения до и после установки полей, используя рефлексию для сравнения объектов вместо ручного кодирования тестов всех свойств. Это может оказаться особенно полезно, если вы пользуетесь таким объектно-реляционным преобразователем, как Hibernate, где доступ к данным тщательно скрыт, но требуется проверка содержимого файла преобразования. Если все остальные функции класса проверяются без обращения в базе данных, то зачастую остается лишь проверить правильность выполнения методов CRUD этого класса и преобразования.

Проверьте правильность вызова кода доступа к данным отдельно от тестирования самой базы данных. Так, если у вас имеется класс `User`, сохраняемый с помощью хранилища `UserRepository`, или уровня доступа к данным с точки зрения n-уров-

невой архитектуры, вам нужно лишь проверить правильность вызова хранилища `UserRepository` из одного из классов верхнего уровня. Такая проверка подразумевает также проверку операций CRUD обращения к базе данных.

Для проверки остальных видов доступа к данным столь простые тесты операций CRUD могут оказаться непригодными. К этой категории относятся, например, вызовы хранимых процедур, не связанные напрямую с сохраняемостью класса. В какой-то момент вам может потребоваться проверка правильности вызова хранимых процедур или же данные, возвращаемые остальной части теста после вызова таких процедур. Ниже приведен ряд общих методов, которые могут пригодиться на тот случай, если потребуется непосредственная имитация JDBC, ADO.NET или других библиотек доступа к данным.

Пользуйтесь фабриками для создания объектов доступа и программирования интерфейсов вместо конкретных типов данных. Базовые структуры могут предоставить вам необходимые фабрики, как это делается, например, в последних версиях функционального блока Microsoft Data Access Application Block. Но вам придется пользоваться фабриками и для имитируемых реализаций классов доступа к объектам — такая возможность отсутствует в большинстве базовых структур фабрик. Если у вас имеется фабрика, которую можно настроить на имитацию с помощью тестового кода, подставьте вариант, имитирующий классы доступа к данным. После этого вы можете виртуально проверить любой вид обращения к базе данных, хотя вам по-прежнему понадобятся имитирующие реализации этих классов. Что касается ADO.NET, то такие реализации доступны из проекта под названием .NET Mock Objects [MockObjects]. В других средах также могут быть их версии. Мне известна лишь одна базовая структура, ориентированная на фабрики и способная непосредственно работать с имитациями, — это базовая структура SnapDAL [SnapDAL], построенная на основе проекта .NET Mock Objects для поддержки имитирующих реализаций родовых классов ADO.NET и объектов, имитирующих обобщенные интерфейсы ADO.NET.

Применение этих базовых структур зависит от многих факторов, но так или иначе в вашем приложении потребуется поддержка фабрики, которая может возвращать настоящие или имитируемые классы доступа к данным. Когда вы дойдете до того, что сможете сами создавать экземпляры имитирующих объектов доступа к данным, воспользуйтесь для тестов некоторыми или всеми перечисленными ниже рекомендациями:

- Имитация возвращает результирующие множества, действуя подобно заглушки в коде. Данные для результирующего множества получаются из тестового кода стандартным для имитирующих объектов образом.
- Имитация может возвращать тестовые данные из файлов, в том числе из файлов формата XML или электронных таблиц с приемочными тестовыми данными, предоставляемыми организационными единицами.
- Имитация может возвращать тестовые данные из другой базы данных или по более уточненному запросу, чем тестируемый запрос.

- Если можно заменить команду доступа к данным, значит, в выполняемых тестах данные могут быть “записаны” и затем “воспроизведены” из имитирующего объекта, поскольку в этом случае имеется доступ ко всем параметрам и возвращаемым результатам.
- Имитации способны проверять предположения о таких важных операциях, как установление или разрыв соединения, фиксация или отказ транзакций, формирование и перехват исключений, чтение данных устройствами считывания информации и т.д.

Надеюсь, что приведенные выше рекомендации помогут вам начать разработку приложений со многими возможностями для тестирования кода связывания с базами данных. В этом разделе были рассмотрены методы тестирования базы данных по порядку возрастания их сложности, начиная с установки базы данных в исходное состояние или отката транзакций как самых простых для реализации методов. Полная изоляция кода доступа к данным от остального кода приложения всегда приветствуется и способствует повышению быстродействия и успешному выполнению тестов. На еще более высоком уровне находятся имитации фактических обращений к базе данных в качестве гибкого метода сбора, предоставления и организации тестовых данных. Если вы работаете с устаревшим кодом, в котором используются библиотеки на избранном языке программирования для доступа к данным на низком уровне, вам, скорее всего, придется выбрать именно такой подход. Если же вы начинаете работу над новым приложением, то для начала можете попробовать просто устанавливать базу данных в исходное состояние в промежутке между тестами, а затем усовершенствовать структуру своих тестов, как только их быстродействие начнет сказываться на производительности вашего труда.

Благодарю Фила за то, что он вооружил читателей полезными методами и рекомендациями относительно тестирования базы данных!

На заметку

Нирая Гупта так прокомментировал изложенное выше: “Для возврата базы данных Oracle в предыдущее состояние с целью тестирования можно воспользоваться свойством Flashback (Возврат к прошлому состоянию)”.

Итак, мы продвинулись достаточно далеко, но нам еще предстоит рассмотреть один вопрос, который связан с подготовкой к инфраструктуре и которого мы пока еще не касались: как получить число сохраненных клиентов? Речь, конечно, идет о формировании запросов!

Формирование запросов

Формирование запросов сильно отличается в зависимости от выбранной инфраструктуры. Кроме того, существует большая доля риска, что оно будет оказывать воздействие на потребителя. Поначалу это не так заметно, когда составляются простые запросы, но в дальнейшем обнаруживается немало требований к формированию запросов, и, выполняя их, вы обычно привязываетесь к выбранной инфраструктуре.

Итак, сделаем небольшое отступление и рассмотрим другое решение, не основанное на запросах. Ранее я уже демонстрировал, каким образом объекты выбираются по их идентичности из хранилища с помощью следующего кода:

```
// Класс OrderRepository
public Order GetOrder(int orderNumber)
{
 return (Order)_ws.GetById(typeof(Order), orderNumber);
}
```

Но в отсутствие идентичности объекта `orderNumber` интерфейс метода обращения к хранилищу должен явно измениться, чтобы на этот раз возвратить список, поскольку нулевой номер может оказаться у нескольких заказов, прежде чем они могут достигнуть определенного состояния. Что же из этого следует? Метод `GetById()` теперь бесполезен из-за отсутствия идентичности объекта `orderNumber` (в данном случае предполагается, что он не единственный, и поэтому может быть возвращен список). Следовательно, нужно найти способ достичь в подделке второго слоя карт идентичности для объектов заказов. Допустим, что это можно сделать с помощью метода `GetAll(Type typeOfResult)` следующим образом:

```
// Класс OrderRepository
public IList GetOrders(int orderNumber)
{
  IList result = new ArrayList();
  IList allOrders = _ws.GetAll(typeof(Order));

  foreach (Order o in allOrders)
  {
    if (o.OrderNumber == orderNumber)
      result.Add(o);
  }
  return result;
}
```

Но это довольно нелепый код и явно не то, что нам требуется, когда инфраструктура уже выбрана — по крайней мере, не для реальных приложений.

Объекты запросов единственного ряда

Аналогично хранилищам, было бы неплохо начать однажды с "правильного" формирования запросов в единственной реализации, которую можно было бы использовать как с подделкой, так и с настоящей инфраструктурой. Как же это сделать?

Допустим, что нам требуется работать с шаблонами Query Object [Fowler PoEAA] (т.е. инкапсулировать запросы в виде объектов, используя объектно-ориентированный синтаксис для работы с запросами) и, кроме того, изменить сигнатуру метода с `GetAll()` на `GetByQuery()`, чтобы данный метод воспринимал на этот раз интерфейс `IQuery` (определенный на уровне абстракции NWorkspace). Соответствующий код теперь может выглядеть следующим образом:

```
// Класс OrderRepository
public IList GetOrders(int orderNumber)
{
  IQuery q = new Query(typeof(Order));
  q.AddCriterion("OrderNumber", orderNumber);
  return _ws.GetByQuery(q);
}
```

Пока что все складывается достаточно просто. Сначала мы создаем экземпляр `IQuery`, указав тип объекта, который предполагаем получить в итоге, а затем задаем подходящие критерии, чтобы максимально ограничить размеры результирующего множества, для чего запрос, как правило, обрабатывается в базе данных (или же в коде подделки в том случае, если используется реализация подделки).

> ***На заметку***
>
> Конечно, интерфейс запроса можно было бы сделать более плавным, но постараемся пока что придерживаться самого элементарного его варианта, какой только можно придумать.

Вот, собственно, и все, что следовало бы сделать для реализации части модели предметной области. А теперь вернемся к методу `_GetNumberOfStoredCustomers()`. Как его код мог бы выглядеть после ввода нового инструмента формирования запросов? Допустим, что в этом коде метод обращения к хранилищу вызывается следующим образом:

```
// Класс CustomerRepository
public int GetNumberOfStoredCustomers()
{
  return _ws.GetByQuery(new Query(typeof(Customer))).Count;
}
```

Такое решение вполне работоспособно, но в условиях эксплуатации оно может довести всякого администратора базы данных до отчаяния, особенно если в результа-

те выполняется команда SELECT, выбирающая из таблицы все строки, чтобы можно было подсчитать их количество. Но в большинстве случаев это просто неприемлемо.

Следовательно, нам нужно наделить бо́льшими возможностями интерфейс API для формирования запросов. Ниже приведен пример, в котором возврат таких простых типов, как int, сочетается с агрегирующим запросом (на этот раз агрегат относится не к шаблону ППО Aggregate, а, например, к запросу SUM или AVG в SQL).

```
// Класс CustomerRepository
public int GetNumberOfStoredCustomers()
{
  IQuery q = new Query(typeof(Customer),
    new ResultField("CustomerNumber", Aggregate.Count));
  return (int)_ws.GetByQuery(q)[0];
}
```

Этот код немного сыроват и недостаточно зрел, но он дает ясное представление о том, как элементарный интерфейс API для формирования запросов конструируется на уровне абстракции NWorkspace.

Конечно, было бы неплохо иметь стандартный язык формирования запросов. Вероятно, отсутствие такого языка и послужило причиной того, что объектные базы данных так и не нашли широкого применения. Безусловно, был создан объектно-ориентированный язык запросов (Object Query Language — OQL), но, на мой взгляд, он появился слишком поздно, и сделать его стандартным было бы очень трудно. Он вполне отвечает своим требованиям, но слишком сложен. Как это часто бывает, широкому распространению объектных баз данных, скорее всего, помешали обстоятельства, сложившиеся именно так, а не иначе. Подробнее об объектных базах данных речь пойдет в главе 8.

В настоящий момент хотелось бы иметь некий стандарт на формирование запросов для базовых структур сохраняемости, который был бы столь же распространенным, как и SQL, но для моделей предметной области. А до тех пор, пока он отсутствует, можно воспользоваться вариантом реализации на уровне абстракции NWorkspace, чтобы восполнить пробел. Нужно лишь выяснить, во что это обойдется.

Во что обходятся объекты запросов единственного ряда

Разумеется, преобразование обходится не бесплатно, что имеет место и в данном случае. Прежде всего, переход от реализации интерфейса IWorkspace к конкретному решению инфраструктуры сказывается на производительности. Но такие издержки на производительность могут оказаться незначительными в контексте сквозных сценариев.

Следует также учитывать снижение эффективности из-за упрощенного характера интерфейса API на уровне NWorkspace по сравнению с теми возможностями, которые должны обеспечивать грамотные инфраструктурные решения, а это намного хуже. Кроме того, сам интерфейс API на уровне NWorkspace достаточно сырой и, вероятно,

не вполне пригоден для формирования запросов применительно к конкретному инфраструктурному решению. Но если значительные выгоды, которые можно извлечь из такого подхода, перевешивают все эти издержки, то с ними можно как-то мириться.

А теперь рассмотрим один очень важный вопрос, оставшийся ранее без внимания и касающийся формирования запросов и карты идентичности: следует ли обходиться без кэша при формировании запросов?

Кэш и формирование запросов

Как отмечалось выше, если при вызове метода `GetById()` данная операция должна выполняться с переходом в состояние сохраняемости, выбранный экземпляр объекта будет введен в карту идентичности перед тем, как возвратить его потребителю. Это же относится и к методу `GetByQuery()`. И в этом случае экземпляры объекта будут вводиться в карту идентичности.

Но главное отличие метода `GetByQuery()` заключается в том, что в нем не исследуется карта идентичности перед переходом в состояние сохраняемости. Это объясняется отчасти тем, что нам нужно использовать возможности машины базы данных, но самое главное, что нам неизвестно, имеется ли в карте идентичности (или в кэше, если угодно) вся информация, необходимая для выполнения запроса. В поисках такой информации мы и обращаемся к базе данных, что приводит к очередному затруднению. Если метод `GetById()` начинается с обращения к карте идентичности, то в методе `GetByQuery()` этого не делается. Это совершенно другое поведение, и оно фактически означает, что метод `GetByQuery()` обходится без кэша, что, собственно, и вызывает затруднение. Так, если запросить нового клиента, введенного в карту идентичности и единицу работы, но не сохраненного, он будет найден только по идентификационному номеру, но не по имени, что не совсем удобно.

Откровенно говоря, данное решение весьма проблематично, но я все же решил предоставить методу `GetByQuery()` возможность вызывать метод `PersistAll()` неявно *по умолчанию* перед обращением к базе данных. (Во избежание этого можно было бы переопределить метод `GetByQuery()`, но и в этом случае метод `PersistAll()` вызывался бы неявно по умолчанию.) Я пришел к выводу, что такой стиль проектирования по умолчанию лучше всего согласуется с остальной частью интерфейса на уровне абстракции NWorkspace, цель которого — максимальная простота и снижение вероятности появления ошибок. Именно поэтому я принес отчасти в жертву семантику транзакций. На это можно возразить, что тем самым нарушается принцип минимального числа неожиданностей. Однако неожиданности в данном случае зависят от уровня подготовки разработчика.

Безусловно, самый главный недостаток такого решения состоит в том, что при вызове метода `GetByQuery()` могут появиться исключения, связанные с операцией сохранения. Данное решение действительно небезгрешно, но ведь нужно же на что-то решиться, чтобы двигаться дальше.

Вспомним простой и неоптимизированный вариант метода `_GetNumberOfStoredCustomers()`. Он не такой медленный, хотя и может работать не так, как предпола-

галось, если используется в следующем виде (который, впрочем, годится и для оптимизированного варианта):

```
public int GetNumberOfStoredCustomers()
{
  return _ws.GetByQuery(new Query(typeof(Customer))).Count;
}
```

Этот вариант не подходит для нашей цели, поскольку в методе `GetByQuery()` неявно вызывается метод `PersistAll()`. Поэтому придется прибегнуть к перегрузке, как показано ниже, где `false` — параметр метода `implicitPersistAll()`:

```
public int GetNumberOfStoredCustomers()
{
  return _ws.GetByQuery(new Query(typeof(Customer)
, false)).Count;
}
```

На заметку

Разумеется, мы можем и должны вместо этого выбрать вариант с использованием агрегата. А здесь мы просто уделили должное внимание проблеме неявного вызова метода `PersistAll()`.

Все это оказывает в какой-то степени влияние на выбранную модель программирования. Прежде всего, следует определенно принять стиль работы блоками, когда дело доходит до формирования запросов и изменений в том же самом экземпляре рабочего пространства. Так, если вы внесли изменения, они должны вас вполне устраивать до формирования запроса, поскольку после этого изменения становятся постоянными.

На заметку

Разумеется, это возлагает на вас ответственность за небрежный стиль программирования на уровне потребителя, где основная цель состоит в том, чтобы код работал, а о таких вещах, как явное сохранение, можно и забыть.

При этом возникает весьма неожиданный и нежелательный эффект: вызов метода `GetByQuery()` может привести к формированию совершенно других исключений, чем предполагалось, поскольку эти исключения на самом деле будут происходить в результате вызова метода `PersistAll()`. Поэтому совершенно ясно, что в коде потребителя целесообразно в любом случае вызывать метод `PersistAll()` явно.

Но опять же, если вас не устраивает такое поведение, то вам ничего не мешает использовать перегрузку. (В действительности с помощью метода `GetById()` это

можно сделать совсем иначе, т.е. выполнить перегрузку с обращением к базе данных, но не проверяя карту идентичности.) В данном случае вас, вероятно, будут больше интересовать не промежуточные результаты вашей работы, а то, что находится в базе данных.

Итак, рассмотрев формирование запросов по отношению к карте идентичности, перейдем к вопросу размещения самих запросов.

Размещение запросов

Для размещения экземпляров запросов имеются по крайней мере три следующих места:

- хранилища;
- потребители хранилищ;
- модель предметной области.

Рассмотрим их по очереди.

Хранилища

Вероятно, хранилища являются наиболее часто используемым местом для составления запросов. В этом случае запросы превращаются в средство вызова таких методов, как `GetCustomersByName()` и `GetUndeliveredOrders()`. Это означает, что потребитель может передать параметры для методов, но не в виде объектов запроса, а в виде обычных типов данных, после чего объекты запроса устанавливаются в соответствии с методом и возможными значениями его параметров.

Потребители хранилищ

В этом случае запросы составляются у потребителей хранилищ и передаются хранилищам в виде параметров. Такой способ обычно используется для очень гибких запросов, когда, например, пользователь может выбрать заполнение любых полей в крупной фильтруемой форме. Для этой цели в хранилище может использоваться метод `GetCustomersByFilter()` — получение клиентов посредством фильтрации, который воспринимает интерфейс `IQuery` в виде параметра.

Модель предметной области

И наконец, было бы интересно устанавливать типизированные объекты запроса в модели предметной области, где для запросов по-прежнему реализуется интерфейс `IQuery` на уровне NWorkspace. Потребитель может по-прежнему направлять запросы хранилищам, пользуясь при этом достаточно интуитивным и типизированным интерфейсом API. Внешний вид такого интерфейса, конечно, полностью определяется разработчиком модели предметной области.

Вместо следующего простого бестипового запроса:

```
// Код потребителя
IQuery q = new Query(typeof(Customer));
q.AddCriterion("Name","Volvo");
```

можно составить тот же самый запрос с помощью следующего кода:

```
// Код потребителя
CustomerQuery q = new CustomerQuery();
q.Name.Eq("Volvo");
```

Помимо упрощения кода потребителя, такой подход позволяет еще больше инкапсулировать модель предметной области, но в то же время он менее гибок, хотя это и не так уж плохо. Ведь нельзя же предусмотреть все сразу.

Допустим, что нам пришлась по душе идея размещать запросы в модели предметной области. Какие же запросы могут в этом случае нам потребоваться и в каком количестве?

Агрегаты как инструмент формирования запросов

Последний вопрос мог бы так и остаться открытым, но у меня имеется свой ответ на него. Возможно, вы уже догадались, что я имею в виду использование агрегатов.

Каждый агрегат вполне пригоден для установки объектов запроса в модели предметной области. Конечно, для их создания можно было бы сначала пойти по пути ЭП, что, вероятно, еще лучше.

Что же касается агрегатов, то следует еще раз подчеркнуть, что агрегаты считаются механизмом, используемым *по умолчанию* для определения необходимой величины графов загрузки.

На заметку

Под графом загрузки подразумевается глубина загрузки экземпляров объектов относительно целевого экземпляра. Так, при запросе определенного заказа мы должны также решить, следует ли загружать его клиента, а возможно, и строки заказа.

Если же используемый по умолчанию механизм не обеспечивает достаточную производительность, имеется немало возможностей для ее оптимизации. Типичным тому примером служит неполная загрузка графов, когда требуется список экземпляров объектов (в частности, заказов). Вместо этого можно создать усеченный тип объекта (например, `OrderSnapshot` — моментальный снимок заказа) только с теми полями, которые требуются для составления списка заказов. Такие списки не интегрируются с единицей работы и картой идентичности (что, возможно, и требуется) по тем же соображениям производительности или по причине других осложнений — все, конечно, зависит от конкретной ситуации.

Составление таких списков может поддерживаться на уровне абстракции, чтобы создаваемый код вообще не зависел от конкретной реализации уровня абстракции. Это может выглядеть следующим образом:

```
// Пример метода OrderRepository.GetSnapshots()
IQuery q = new Query(typeof(Order), typeof(OrderSnapshot)
    , new string[]{"Id", "OrderNumber", "Customer.Id"
    , "Customer.Name"});
q.AddCriterion("Year",year);
result _ws.GetByQuery(q);
```

Для того чтобы этот код оказался работоспособным, необходимо иметь подходящий конструктор класса `OrderSnapshot`, как в следующем примере, где идентификационные номера реализованы как глобально уникальные идентификаторы `Guid`.

```
// Класс OrderSnapshot
public OrderSnapshot(Guid id, int orderNumber
    , Guid customerId, string customerName)
```

На заметку

Бурное размножение типов обычно считается ахиллесовой пятой объектно-реляционных преобразователей (пример тому приведен выше). Мой опыт показывает, что это действительно проблема, хотя она решаема. Во-первых, это делается только по мере надобности, поэтому не во всех корнях агрегатов из модели будет присутствовать класс моментального снимка. И во-вторых, при составлении других списков можно всегда избавиться от класса моментального снимка. Но прежде чем решаться на такой шаг, нужно подумать о последствиях для оптимизации, чтобы не делать ничего лишнего.

Еще один распространенный подход заключается в использовании шаблона Lazy Load для настройки загрузки данных в нужный момент. (Этот вопрос будет обсуждаться в последующих главах данной книги.)

Но если и этого окажется недостаточно, то можно написать код SQL вручную и самостоятельно реализовать тип объекта моментального снимка. Следует лишь ясно осознавать, что с этого момента начинает активно использоваться модель базы данных.

Одна или две компоновки хранилища

Итак, у нас в распоряжении все рассмотренные выше компоненты, и значит перед нами открываются самые разные возможности для структуризации хранилищ. Но как это делается в реальных проектах?

В одном из своих недавних реальных проектов я использовал хранилища единственного ряда как для подделки, так и для базы данных. Это потребовало незначительной оптимизации с использованием собственного кода SQL, но мне пришлось прибегнуть к мелкому приему, чтобы вызывать специально написанный мной метод, если оптимизированный метод обнаруживает внесенную связывающую строку.

В противном случае выполняется неоптимизированный код. Таким образом, для подделки используется неоптимизированный вариант, а для кода обращения к базе данных — оптимизированный вариант.

Опять же, данный прием использовался лишь в некоторых случаях. И хотя он оказался далеко не самым изящным, тем не менее, вполне работоспособным.

Описания в качестве запросов

Еще один подход к формированию запросов состоит в использовании шаблона Specification [Evans DDD], инкапсулирующего концептуальные описания чего-то, например, характеристик клиента, которому больше не разрешается ничего покупать. Каждая концепция получает описательное имя и может использоваться неоднократно.

Такой подход сродни созданию типизированных запросов специально для выбранной модели предметной области, но он применяется на совсем другом уровне, поскольку это не типичное, а особенное формирование запросов на основе концепций, характерных для модели предметной области. Это более высокий уровень, чем у определений запросов, например, клиента, в модели предметной области. Вместо раскрытия типичных свойств для отфильтровывания, описание обозначает концепцию с особенным значением предметной области. Типичным тому примером служит концепция покупателя золота. В описании указывается, что именно требуется для того, чтобы клиент стал покупателем золота.

В итоге получается очень ясный и содержательный код, поскольку концепции понимаются и описываются в виде классов описания.

Но даже такие классы могут явно выдавать интерфейс `IQuery`, что дает возможность использовать их вместе с методом `GetByQuery()` интерфейса `IWorkspace` (или его эквивалента), если вам пришлась по душе идея уровня абстракции.

Другие методы формирования запросов

До сих пор мы рассматривали объекты запроса как своего рода язык запросов, хотя иногда такие объекты встраиваются в другие конструкции. На самом деле, когда запросы становятся сложными, то их лучше, как правило, составлять на таком языке, основанном на операциях со строками, как SQL.

В настоящее время еще рано говорить о каких-то других языках запросов для уровня абстракции NWorkspace. Но когда-нибудь поддержка запросов на уровне

NWorkspace в качестве результатов запросов станет актуальной. Таким образом, запросы (как в SQL, но для модели предметной области) станут полезными и для инфраструктурных решений, предполагающих реализацию сопряжения с уровнем абстракции NWorkspace.

Мне трудно отделаться от мысли, что требуемый язык запросов, как, впрочем, и язык программирования, во многом определяется выбранным стилем проектирования. Разумеется, немало зависит и от конкретной ситуации, для которой требуется язык запросов, если, конечно, есть из чего выбирать. На этом можно, пожалуй, оставить данную тему (мы продолжим ее в главах 8 и 9) и перейти к следующей.

Резюме

В этой главе был рассмотрен принцип неведения сохраняемости и некоторые методы, позволяющие отделить инфраструктуру от классов из модели предметной области и одновременно подготовиться к инфраструктуре. Кроме того, были затронуты вопросы тестирования базы данных и представления уровня абстракции.

В следующей главе мы вновь перенесем акцент на саму модель предметной области, уделив на этот раз внимание правилам. При этом обнаружится, что многие требования из списка, составленного в главе 4, имеют непосредственное отношение к правилам, а значит, это уже знакомая тема моделей предметной области.

Глава 7

Порядок устанавливают правила

В предыдущей главе мы сделали неплохой перерыв, рассмотрев некоторые вопросы подготовки к инфраструктуре. А теперь вернемся к базовой модели. В этой главе речь пойдет о правилах.

Тема правил весьма обширна, поэтому мы коснемся ее лишь частично, уделив основное внимание правилам проверки достоверности с учетом требований, сформулированных в главе 4 и с соответствующими комментариями там, где это уместно.

Возвращаясь к списку требований, составленному в главе 4, мы ясно видим, что большая часть требований так или иначе связана с правилами. На мой взгляд, это одна из тех областей, где в полной мере проявляются достоинства модели предметной области, поскольку можно очень долго обходиться без механизма соблюдения правил. Как показано в книге [Evans DDD], одновременное использование нескольких систем понятий затруднительно, и поэтому мы тщательно избегаем смешения моделей предметной области с механизмами соблюдения правил. (Именно поэтому так трудно сочетать объектно-ориентированную и реляционную модели.)

В своей предыдущей книге [Nilsson NED] я посвятил этому вопросу целую главу "Бизнес-правила", где привел немало примеров затруднений, связанных с выбором места для оценки правил. По ряду причин почти все эти примеры завершались проверкой в базе данных. Но в то время я был приверженцем баз данных, а теперь являюсь сторонником модели предметной области, в чем вы сможете убедиться, прочтя эту главу. Думаю, что эти перемены произошли к лучшему, и надеюсь, что вы со мной согласитесь.

Итак, с чего же начать? Может быть, с разделения правил на категории?

Разделение правил на категории

Если попытаться разделить правила на категории, то сразу же обнаруживается множество аспектов, которые следует при этом учесть, в том числе и следующие.

- Где выполнять правила (т.е. в каком слое или на каком уровне).
- Когда выполнять правила.

- Степень сложности правил (простые ограничения, охват нескольких полей, охват нескольких экземпляров объектов и т.д.).
- Начальные или другие действия по соблюдению правил.
- И т.д. и т.п.

Мне бы не хотелось углубляться в этот вопрос, поэтому отсылаю читателя к следующим источникам, где он излагается достаточно подробно: [von Halle BRA], [Ross BRB] и [Halpin IMRD]. Несмотря на то что в этих книгах уделяется больше внимания данным, они все же представляют немалый интерес.

Итак, рассмотрим данную тему под другим углом зрения, определив сначала некоторые принципы, а затем перейдя к деталям.

Принципы для правил и их применения

Термин *принцип* в данном случае вполне уместен, поскольку он не строгий. Ведь принципы намного проще сформулировать, чем придумать категории из-за присущей им неопределенности. Тем не менее, принципы достаточно хорошо описывают намерения, а именно это нам теперь и нужно.

Двухсторонняя проверка правил: необязательная (профилактическая) и обязательная (задним числом)

Проверка возможных осложнений (например, в связи с попытками сохранить текущие изменения) должна выполняться с упреждением (профилактически), чтобы принять меры к их устранению. Благодаря этому упрощается модель программирования.

Если же проверка не выполнена с упреждением, то пощады ждать не приходится, а о возникших осложнениях (задним числом) сообщают исключения.

Сохранению подлежат все состояния — даже ошибочные

Пользователям очень не нравится, когда в приложениях нельзя сохранять текущие изменения из-за ошибок. Мой коллега сравнил это с невозможностью сохранить в текстовом редакторе документ, если в нем имеются грамматические ошибки.

Возможно, сохранение всех состояний — это идеал, но к нему все же нужно стремиться. Для этого следует избегать состояний, которые нельзя сохранить.

При этом важно также выяснить, что означают ошибки. Так, заказ может считаться ошибочным, если он крупный. Но до тех пор, пока пользователь не попытается передать заказ на утверждение, заказ не может считаться ошибочным. Это типичная ситуация, в которой пользователю приходится принимать решение, прежде чем передавать заказ на утверждение. У него должна быть возможность сохранить такой заказ перед тем, как передать его на утверждение, чтобы продолжить работу с ним в дальнейшем и тогда принять окончательное решение.

Правила должны применяться продуктивно

Работа с правилами должна быть продуктивной как при определении, так и при применении правил.

Продуктивность применения правил означает возможность выборки метаданных для настройки пользовательского интерфейса. Благодаря этому сокращается время на поиск ошибок во время проверки, поскольку правильно настроенный пользовательский интерфейс способствует тому, чтобы некоторые ошибки не появлялись. Впрочем, при проверке правил нельзя полагаться только на пользовательский интерфейс.

А продуктивность определения правил может быть достигнута с помощью небольшой базовой структуры, которая позволяет сосредоточить основное внимание на наиболее интересных положениях правил и автоматически сформировать заготовки кода.

Правила должны допускать поправки для ввода специальных правил

Одни правила зависят от сценариев, а другие от них не зависят, поэтому важно не только иметь возможность объявлять все правила непосредственно в определениях классов, но и вводить специальные правила динамически как для некоторых прецедентов использования, так и во время развертывания в виде информации о конфигурации системы.

На мой взгляд, информация о конфигурации приобретает особое значение при создании программных продуктов, рассчитанных на самых разных заказчиков. Если же система создается для одного заказчика, такая информация оказывается менее полезной.

Правила должны размещаться вместе с состоянием

Правила должны быть, по крайней мере, *определены* максимально близко к соответствующим классам из модели предметной области. Под этим подразумевается не реализация самих правил, а их объявления. Применение правил следует инкапсулировать таким образом, чтобы без труда получать “полную” картину класса из модели предметной области при проверке этого класса.

Этот принцип, на мой взгляд, можно было бы назвать так: “Правила должны быть согласованы.” Когда база кода достигает определенного размера, существует риск, что правила могут мешать или даже противоречить друг другу. Если же правила размещаются рядом и вместе с состоянием, это упрощает в определенной степени управление базой правил.

Правила должны быть очень удобными для тестирования

Нам придется потратить немало рабочего времени на тестирование правил. Поэтому очень важно, чтобы правила были составлены с учетом удобства их тестирования.

В действительности ввод правил только усложняет тестирование. Ведь нужно устанавливать экземпляры объектов в правильное состояние. Для этой цели можно написать вспомогательные тестовые методы (например, закрытые методы в тестовой конфигурации), хотя это и не устраняет всех препятствий для тестирования.

Система должна препятствовать переходу в недействительное состояние

Для того чтобы упростить работу с правилами, достаточно обращаться к ним как можно реже. Для этого можно, например, работать с методами создания, устанавливающими объекты из модели предметной области в действительное состояние, которое вообще не может стать недействительным.

Начало создания интерфейса API

Мы еще вернемся к перечисленным выше принципам и оценим их соблюдение далее в этой главе, а до тех пор вернемся на грешную землю и обратимся к задачам, сформулированным в главе 4. Начнем с внешнего интерфейса API и посмотрим, к чему это приведет — это даст ясное и точное представление обо всем, что мы делаем.

Итак, задачи и требования, сформулированные в главе 4, таковы.

1. Список клиентов, формируемый с применением гибкого и сложного фильтра.
2. Список заказов, формируемый при просмотре отдельного клиента.
3. Заказ может состоять из самых разных строк.
4. Большое значение имеет выявление конфликтов параллельной обработки.
5. У клиента может быть долг, не превышающий определенную сумму.
6. Заказ не может быть составлен на сумму свыше 1 млн крон.
7. У каждого заказа и клиента должен быть уникальный и удобный для использования номер.
8. Прежде чем новый клиент будет признан приемлемым, должен быть проверен его кредит в соответствующем кредитном учреждении.
9. У заказа должен быть клиент, а у строки заказа — заказ.
10. Сохранение заказа и его строк должно быть атомарным.
11. У заказов должно быть состояние подтверждения, изменяемое пользователем.

Рассмотрим задачу номер 6 “Заказ не может быть составлен на сумму свыше 1 млн крон”. Как подойти к ее решению? Это можно сделать несколькими способами.

В частности, можно создать нечто вроде механизма проверки правил, где экземпляры объектов передаются на проверку достоверности или же ответственность за эту

проверку возлагается на сами экземпляры. Мне лично по душе идея возложить на экземпляры максимум такой ответственности (или хотя бы столько, сколько уместно). Возможно, такая идея и кажется наивной, но в то же время она достаточно проста.

Для начала напишем следующий тест:

```
[Test]
public void CantExceedMaxAmountForOrder()
{
  Order o = new Order(new Customer());

  OrderLine ol = new OrderLine(new Product());
  ol.NumberOfUnits = 2000000;
  ol.Price = 1;
  o.AddOrderLine(ol);

  Assert.IsFalse(o.IsValid);
}
```

В этом тесте в класс `Order` вводится свойство `IsValid`, которое является действительным. Это довольно простой подход, хотя с ним связаны следующие вполне очевидные проблемы.

- *Какая* именно проблема возникла? Известно, что проблема возникла, но мы не знаем, в чем она состояла.
- Мы *допустили* неправильный переход.
- А что, если мы *забыли* выполнить проверку?

В данном тесте мы лишь проверили, все ли было правильно. Мы еще вернемся к перечисленным выше проблемам, а до тех пор нам нужно справиться со следующей более простой и важной проблемой:

Для чего именно действителен (или недействителен) заказ?

Контекст, контекст и еще раз контекст!

Контекст, как всегда, очень важен, тем не менее, он часто забывается при том подходе к соблюдению правил, с которого мы начали. Нам нужно обратиться к контексту, прежде чем двигаться дальше, поскольку это имеет *решающее* значение для всего, что будет рассмотрено далее.

Столь общий подход к использованию свойства `IsValid` в качестве своего рода универсального, всеобъемлющего теста объясняется слишком большим вниманием к сохраняемости. Если же отделить сохраняемость от правил и попытаться придерживаться принципа сохранения всех состояний, то в конечном итоге это будет означать уже совсем другой подход.

Что касается правил, то особый интерес также представляют *переходы*. Так, если заказ находится в состоянии `NewOrder` (Новый заказ), в таком состоянии, пожалуй,

возможно все, что угодно. Но если заказ должен перейти в состояние `Ordered` (Заказано), этот переход должен быть разрешен только при соблюдении определенных правил.

Таким образом, *сохранение* заказа, находящегося в состоянии `NewOrder`, должно быть возможным, даже если заказу не разрешен переход в состояние `Ordered`. Такой подход к соблюдению правил позволяет акцентировать их на значении модели и не допустить воздействие на нее таких технических аспектов, как сохраняемость или несохраняемость.

Однако действительность ставит перед нами немало задач практического свойства, и мы вынуждены их как-то решать. Ведь даже если мы намерены сосредоточить основное внимание на модели, все равно придется считаться и с инфраструктурой. Посмотрим, что нам даст рассуждение о сохраняемости или несохраняемости как об особом состоянии, но прежде следует выяснить, что может дать решение проблем, связанных с правилами, если мы не будем в достаточной степени учитывать инфраструктуру.

Ограничения, связанные с базой данных

Даже если уделить основное внимание переносу всех правил в модель предметной области, некоторые правила все равно останутся в базе данных. Объясняется это, главным образом, эффективностью. Ведь некоторые правила наиболее эффективно проверяются именно в базе данных. Проектирование, скорее всего, будет сосредоточено вокруг большинства этих правил, но вполне возможны ситуации, в которых данное обстоятельство становится реальным фактом.

Дело еще и в том, что проектирование базы данных заново или же так, как хотелось бы, не всегда оказывается возможным, но вместо этого приходится вносить поправки в структуру текущей базы данных. Кроме того, нередко считается положительным фактором, если базы данных сами принимают меры в отношении соблюдаемых правил, особенно если одна и та же база данных используется в ряде различных систем.

При этом проявляется также потеря соответствия объектной ориентации и реляционной базы данных (см. главу 1). Строки в базе данных чаще всего имеют статическую максимальную длину, например, `VARCHAR(50)`. То, что конкретная строка не может быть длиннее этого предела, и есть своего рода общее правило, хотя оно связано, главным образом, с сохраняемостью. Переход в состояние `Persisted` (Сохранено) не разрешается, если строка оказывается слишком длинной.

Следовательно, вполне возможно, что придется иметь дело с правилами, недостоверность которых нельзя обнаружить до тех пор, пока состояние не будет сохранено в базе данных. А это, в свою очередь, представляет две следующие трудности.

Во-первых, синтаксический анализ информации об ошибках затруднителен, особенно если принять во внимание то обстоятельство, что типичное решение объектно-реляционного преобразования обычно пытаются сделать переносимым с одной базы данных в другую. Более того, суть данного затруднения состоит в том, что объектно-

реляционный преобразователь, скорее всего, не сможет выполнять преобразование автоматически при появлении ошибок и укажет на те поля, которые стали причиной затруднения, возникшего в модели предметной области; например, в связи с ошибкой дублирования ключей.

Во-вторых, ошибки, как правило, возникают в самый неподходящий момент. При этом модель предметной области, скорее всего, уже изменена и восстановить ее после ошибки без того, чтобы начать все сначала, вряд ли удастся. И как правило, приходится начинать все сначала вместо того, чтобы сделать что-то более разумное.

Тем не менее следует взять на вооружение идею сделать возможным сохранение экземпляров объектов в любой момент. (Во всяком случае, нужно выбрать некоторые *предпочтения* — даже если мы и не сможем следовать им в полной мере. Ведь крайности всегда обходятся чрезвычайно дорого.)

Старайтесь проектировать с максимальным (или же уместным) упреждением, чтобы свести к минимуму число исключений, формируемых в базе данных, и быть готовым к тому, чтобы по очереди обрабатывать в коде те исключения, которые могут все же появиться, или же выполнить откат транзакции, удалить изменения из модели предметной области и начать все сначала. Это не самый лучший, но все же какой-то выход из положения. Главный вывод из всего сказанного выше заключается в следующем: необходимо уделять максимум внимания проектированию, чтобы не создавать больших проблем.

Итак, мы рассмотрели чисто техническую сторону дела, связанную с некоторыми неудобствами, которые может доставить база данных. Но дело в том, что правила связаны не *только* с инфраструктурой и не *только* с базой данных. Они на самом деле связаны и с тем и с другим.

Привязка правил к переходам, связанным с моделью предметной области или же с инфраструктурой

Итак, мы выяснили, что нужно сделать все возможное, чтобы не допустить переход в недействительное состояние. Следуя этому принципу, можно всегда сохранить состояние (если отвлечься на мгновение от ограничений, связанных с инфраструктурой).

Данный принцип — своего рода идеал. Однако в действительности трудно не нарушить некоторые правила во время выполнения многоэтапного действия. Например, следует ли запрещать замену имени клиента именем, которое по правилам считается слишком коротким? Вероятно, следует, но в то же время это может не дать пользователю возможность делать то, что он хочет, а именно: удалить старое имя и ввести вместо него новое.

Это затруднение можно разрешить с помощью метода `Rename(string newName)`, берущего на себя всю заботу о переименовании клиента. При выполнении данного метода объект оказывается в неверном состоянии, но это затруднение можно рассматривать только до или после выполнения метода.

На заметку

Это вполне согласуется с инвариантами, рассматриваемыми в книге [Meyer OOSC]. Такие инварианты подобны утверждениям о том, что экземпляр объекта должен оставаться неизменным в любой момент времени: как до, так и после выполнения метода. Единственным исключением из этого правила может быть само *время выполнения* метода.

Еще один подход заключается в том, чтобы просто не устанавливать свойство до тех пор, пока внесение изменений не будет завершено. Покажем на следующем примере, что в этом смысле можно предложить. Допустим, что имеются два поля, которые устанавливаются вместе либо вообще не устанавливаются. Эта задача решается с помощью специально написанного метода, но если усложнить ее или предпочесть раскрытие свойств в пользовательском интерфейсе с помощью механизмов установки, то, очевидно, будет нелегко избежать перехода в несогласованное состояние время от времени.

Попутно возникает еще один вопрос: как убедиться в том, что недействительное состояние *не* достигнуто? И если, несмотря ни на что, недействительное состояние достигнуто (например, ошибочное изменение, когда заказ находится в состоянии `Ordered`), то следует ли разрешать сохранение такого состояния?

Вполне возможно, что база данных не позволит нам организовать такую защиту в некоторых ошибочных ситуациях. Но даже если сделать все возможное для того, чтобы база данных сама принимала меры в отношении соблюдаемых правил, то все равно что-то да будет упущено из виду. Ведь при переходе к стилю разработки приложений со смещением акцента на модель предметной области меры самозащиты в базе данных вряд ли станут более жесткими. В конце концов правила, сосредоточенные на предметной области, в какой-то степени более развиты и чаще всего *намного* проще выражаются в модели предметной области, чем в реляционной базе данных, а кроме того, хотелось бы иметь их именно в модели предметной области.

Но можем ли мы доверять “среде”, т.е. можно ли доверять тому, что нам не следует сохранять состояние, которого нельзя достичь? В частности, можем ли мы предположить, что у нас вообще не будет программных ошибок? И можем ли мы предположить, что потребитель не проявит творческую инициативу и не воспользуется рефлексией, чтобы задать интересующие его значения в некоторых полях?

Уточнение принципа “Сохранению подлежат все состояния — даже ошибочные”

Попробуем воплотить в конкретном примере упомянутый выше принцип “Сохранению подлежат все состояния — даже ошибочные”.

Вновь допустим, что заказ может находиться в двух состояниях: `NewOrder` и `Ordered`. При этом у нас имеется только одно правило перехода, связанное с предметной облас-

тью: заказ не может быть слишком крупным для перехода в состояние `Ordered`. В итоге получаются действительные комбинации состояний, сведенные в табл. 7.1.

Таблица 7.1. *Действительные комбинации состояний заказа и следствия из правил*

Состояния заказа	Следствия из правил
`NewOrder`	Заказ не слишком крупный
`NewOrder`	Заказ слишком крупный
`Ordered`	Заказ не слишком крупный

Пока что все складывается достаточно просто и так, как нам требуется. А теперь введем условие сохраняемости или несохраняемости, как показано в табл. 7.2.

Таблица 7.2. *Действительные комбинации состояний заказа и следствия из правил*

Состояния заказа	Следствия из правил	Сохраняется или не сохраняется
`NewOrder`	Заказ не слишком крупный	Не сохраняется
`NewOrder`	Заказ не слишком крупный	Сохраняется
`NewOrder`	Заказ слишком крупный	Не сохраняется
`NewOrder`	Заказ слишком крупный	Сохраняется
`Ordered`	Заказ не слишком крупный	Не сохраняется
`Ordered`	Заказ не слишком крупный	Сохраняется

Прежде всего следует подчеркнуть, что переходы в состояние `Ordered` в сочетании со слишком крупным заказом считаются невозможными.

Еще важнее, что если по какой-то причине это все же происходит, то может служить примером состояния, которое связано с реальной исключительной ошибкой и поэтому *не* должно сохраняться.

> ***На заметку***
>
> У вас могут получиться совсем иные результаты, если вы проверяете правила перехода, связанные с предметной областью, и во время сохранения. Хотя такая проверка зачастую считается крайностью.

Следует также иметь в виду, что, придерживаясь уточненного принципа “Сохранению подлежат все состояния — даже ошибочные”, мы уже не возвращаемся к идее использовать свойство `IsValid`, с которого мы начинали рассмотрение темы правил. Теперь это свойство называется `IsValidRegardingPersistence` — является действительным с учетом сохраняемости, что предполагает сохранение многих “ошибочных” состояний.

Требования к интерфейсу API для основных правил, связанных с сохраняемостью

А теперь, когда мы привязали правила к конкретному контексту (т.е. к переходу в состояние сохраняемости), перейдем к интерфейсу API, который с этим связан. После этого мы вернемся к другим вопросам перехода в состояние, рассмотрев их с точки зрения модели, которая представляется более интересной.

> ***На заметку***
>
> Как обычно, мы будем рассматривать данный вопрос лишь в самых *общих* чертах. Я не буду вести речь о том, как создаются интерфейсы (и API, в частности), а просто изложу свои мысли по данному вопросу, которые, надеюсь, помогут вам найти собственное решение.

Рассмотрим новый пример кода, воплощающего идеи упреждать ошибки, возникающие в связи с несохраняемостью исключительных состояний, или же осложнения, приводящие к формированию исключений в механизме сохранения.

Я лично предпочитаю вторую идею: показать ситуацию, в которой механизм сохранения формирует исключения, поскольку первую идею воплотить труднее. Ведь если все работает, как намечено, то достичь исключительного состояния практически невозможно.

Допустим, что в классе `Order` имеется свойство `Note` (Примечание), определенное в базе данных в виде строки длиной `VARCHAR(30)`. В зависимости от решения, выбранного для сохраняемости, может быть получено исключение при попытке сохранить длинную строку, иначе информация окажется утерянной без всякого предупреждения. В данном случае важно принять упреждающие меры к тому, чтобы выявить подобное осложнение. В коде это можно выразить следующим образом:

```
[Test]
public void CantExceedStringLengthWhenPersisting()
{
  Order o = new Order(new Customer());
  o.Note = "01234567890123456789012345678901";

  Assert.IsFalse(o.IsValidRegardingPersistence);
}
```

Итак, вместо свойства `IsValid` заказа в данном случае используется свойство `IsValidRegardingPersistence`, чтобы было совершенно ясно, является ли заказ действительным в контексте сохраняемости.

Далее организуем простой интерфейс, чтобы предоставить потребителю возможность запрашивать корень агрегата о следующем: находится ли он в действительном

состоянии или же не подлежит сохранению. В коде это можно выразить следующим образом:

```
public interface IValidatableRegardingPersistence
{
  bool IsValidRegardingPersistence {get;}
}
```

Следовательно, данный интерфейс реализуется в классах, наделяемых правилами, связанными с сохраняемостью.

О внешнем выражении правил сохраняемости

Как следует из приведенного выше фрагмента кода, на классы из модели предметной области возложена ответственность за проверку правил. Это может показаться странным, если подходить к сохраняемости "извне".

Об этом, конечно, стоит подумать, но пока что мне бы не хотелось отказываться от своего первоначального замысла, отчасти потому что, на мой взгляд, правила находятся в самой модели предметной области и не являются только инфраструктурным аспектом, как сохраняемость.

Возврат к проблемам интерфейса API

Ранее в этой главе были обнаружены некоторые проблемы интерфейса API, который только намечался. Я отложил их рассмотрение, поскольку сначала нужно было решить вопрос отсутствия контекста. Теперь же мы можем вернуться к этим проблемам, рассмотрев их на примере приведенного выше кода теста `CantExceedStringLengthWhenPersisting()` — при сохранении длина строки не может быть превышена. Эти проблемы оказались следующими.

- *Какая* именно проблема возникла? Нам только известно, что проблема возникла, но мы не знаем, в чем она состояла.
- Мы *допустили* неправильный переход.
- А что, если мы *забыли* выполнить проверку?

Рассмотрим эти три проблемы в контексте правил, связанных с сохраняемостью.

Какая именно проблема возникла

Первая проблема по-прежнему актуальна. Так, если экземпляр объекта недействителен, было бы интересно выяснить, какие правила при этом нарушены. Нарушенные правила будут агрегироваться в корне агрегата как от его потомков, так и от него самого и затем возвращаться в виде списка `IList`.

Введем для этой цели интерфейс следующим образом:

```
public interface IValidatableRegardingPersistence
{
  bool IsValidRegardingPersistence {get;}
  IList BrokenRulesRegardingPersistence {get;}
}
```

На заметку

Список нарушенных правил должен представлять собой реализацию шаблона Notification (Уведомление) [Fowler PoEAA2].

Важно отметить, что этот шаблон предназначен не только для уведомления об ошибках, но и для предупреждения и сообщения иных сведений. Составление списка нарушенных правил связано с шаблоном Aggregate и агрегатами, хотя несколько таких списков можно объединять в один список в едином слое Application (Приложение) [Evans DDD], в том числе из нескольких экземпляров корня агрегата.

Мы допустили неправильный переход

Ясно, что строковое значение свойства `Note` получилось слишком длинным, что можно расценивать как переход в недействительное состояние. Но это вряд ли это может представлять особый интерес. На какое-то время строковое значение свойства `Note` можно оставить слишком длинным и дать пользователю возможность удалить из него несколько символов.

К сожалению, сохранить слишком длинное строковое значение свойства `Note` в этом случае не удастся. С другой стороны, такой переход из состояния несохраняемости в состояние сохраняемости может представлять определенный интерес, хотя он и *не* будет разрешен. (Как упоминалось выше, при этом формируется исключение или же строка укорачивается без предупреждения.)

Переход в состояние сохраняемости имеет в какой-то степени "особый" характер. Мы еще вернемся к вопросу о разрешении неправильных переходов в контексте правил, связанных с предметной областью.

Если мы забыли выполнить проверку

Если потребитель не придерживается протокола первоочередной проверки того, что переход вообще возможен, прежде чем пытаться сделать его, возникает исключительная ситуация, а следовательно, формируется исключение. То же самое происходит и в данном случае. Так, если бы мы попытались осуществить переход в состояние сохраняемости, нас бы вполне устроило формирование исключения с подклассификацией `ApplicationException` и возвратом списка `IList` нарушенных правил.

Опять же, если потребитель не придерживается протокола, это приводит к исключительной ситуации.

Итак, мы остановились на конкретной проблеме перехода в состояние сохраняемости. А теперь пора сменить акцент с переходов, связанных с инфраструктурой, на переходы, связанные с предметной областью, попутно рассмотрев ряд идей, в том числе и формирование исключения, если потребитель не придерживается протокола.

Акцент на правилах, связанных с предметной областью

Обратимся вновь к требованиям, сформулированным в главе 4, рассмотрев их по очереди. Мы начали с требования номер 6 "Заказ не может быть составлен на сумму свыше 1 млн крон", но вынуждены были прерваться, как только осознали, что у намеченного интерфейса API отсутствует контекст. Попробуем немного переделать тест, который ранее имел следующий вид:

```
[Test]
public void CantExceedMaxAmountForOrder()
{
  Order o = new Order(new Customer());

  OrderLine ol;

  ol = new OrderLine(new Product());
  ol.NumberOfUnits = 2000000;
  ol.Price = 1;
  o.AddOrderLine(ol);

  Assert.IsFalse(o.IsValid);
}
```

Как мы уже выяснили, у этого фрагмента кода имеется ряд недостатков. Следуя рассмотренным до сих пор идеям, мы могли бы ввести метод для перехода в состояние `Ordered`, например, метод под названием `OrderNow()`. Но меня это не вполне устраивает, поскольку метод `OrderNow()`, скорее всего, сформирует исключение, прежде чем у нас появится возможность для проверки.

Опять же, мне бы не хотелось теперь заострять внимание на том, что подлежит и что не подлежит сохранению. На мой взгляд, важнее другое: когда мы пытаемся осуществить переход в состояние `Ordered`, нам не удается этого сделать. Отразим этот факт, внеся ряд изменений в тест.

```
[Test]
public void Cant GoToOrderedStateWithExceededMaxAmount()
{
  Order o = new Order(new Customer());

  OrderLine ol = new OrderLine(new Product());
  ol.NumberOfUnits = 2000000;
  ol.Price = 1;
```

```
  o.AddOrderLine(ol);

  try
  {
    o.OrderNow();
    Assert.Fail();
  }
  catch (ApplicationException ex) {}
}
```

Теперь важно выяснить следующее: нарушим ли мы правило, выполняя проверку в методе `AddOrderLine()` и находясь в состоянии `Ordered`. В методе `AddOrderLine()` можно было бы допустить переход заказа обратно в состояние `NewOrder`, если бы это вообще было разрешено, а так удается избежать осложнения.

Еще одно осложнение связано с тем, что если в состоянии `Ordered` изменяется одна строка заказа, это может привести к нарушению правила. В качестве типичного выхода из этого положения можно было бы разрешить работать со строками заказа, не изменяя их. Отразим этот факт, внеся соответствующие изменения в тест.

```
[Test]
public void CantGoToOrderedStateWithExceededMaxAmount()
{
  Order o = new Order(new Customer());

  o.AddOrderLine(new OrderLine(new Product(), 2000000, 1));
  try
  {
    o.OrderNow();
    Assert.Fail();
  }
  catch (ApplicationException ex) {}
}
```

> ***На заметку***
>
> Вместо конструкции `try catch` и метода `Assert.Fail()` можно было бы воспользоваться атрибутом `ExpectedException` инструмента NUnit, чтобы сделать тест покороче. Единственное преимущество данной конструкции заключается в том, что она позволяет проверить исключение, появление которого предполагается при вызове метода `OrderNow()`, но, как правило, это преимущество весьма незначительно.

Как только строка заказа введена, ее уже нельзя изменить, и в этом простом примере данное правило должно проверяться в методах `OrderNow()` и `AddOrderLine()`.

Перейдем к очередному требованию по списку.

Правила, требующие взаимодействия

Рассмотрим требование номер 5 “У клиента может быть долг, не превышающий определенную суму”. Это правило кажется, на первый взгляд, довольно простым, но вся суть кроется в деталях. Итак, начнем проверку данного правила и посмотрим, к чему это нас приведет. Первоначальный тест может иметь следующий вид:

```
[Test]
public void CanHaveCustomerDependentMaxDebt()
{
  Customer c = new Customer();
  c.MaxAmountOfDebt = 10;

  Order o = new Order(c);
  o.AddOrderLine(new OrderLine(new Product(), 11, 1));
try

  {
    o.OrderNow();
    Assert.Fail();
  }
  catch (ApplicationException ex) {}
}
```

Но он вряд ли вообще будет работать, поскольку для текущего заказа должна быть предусмотрена возможность поиска других заказов выбранного клиента в определенном состоянии, для чего, как правило, требуется хранилище заказов (класс `OrderRepository`).

> ***На заметку***
>
> Если вы помните, мы выбрали одностороннее отношение между классами `Order` и `Customer`. Именно поэтому требуется поддержка со стороны класса `OrderRepository` в качестве одной из нескольких возможных методик имитации двунаправленности для перехода от клиента к его заказам.

Для этой цели можно было бы внести в конструктор ссылку `_orderRepository` на заказ следующим образом:

```
Order o = new Order(c, _orderRepository);
```

Но насколько такое решение удачно? Это обычно зависит от конкретного приложения. Проверка в заказе текущего долга клиента, когда наступает момент для вызова метода `OrderNow()`, может оказаться несколько запоздалой с точки зрения удобства для пользователя. Вероятно, лучшим решением был бы запрос максимально возможной суммы для нового заказа перед его составлением. В этом случае пользователь

составляет новый заказ, получая соответствующее уведомление посредством пользовательского интерфейса при достижении установленного лимита. Это не требует изменения рассмотренного выше интерфейса API и в то же время более удобно.

В качестве разновидности распределения ответственности можно было бы запросить у класса `Customer` метод `IsOrderOK(orderToBeChecked)` — составлен ли правильно проверяемый заказ. Во всяком случае возложение на класс `Customer` ответственности за принятие такого решения выглядит вполне естественно, поскольку у класса `Customer` имеются для этого все необходимые данные. Но если писать код с учетом предметной области, то стоит ли возлагать на клиента ответственность за проверку заказов? Вероятно, стоит, если сформулировать вопрос немного по-другому: следует ли просить клиента *принять* заказ? На самом деле такой подход мало чем отличается от того, что было ранее воплощено в коде, кроме размещения метода `OrderNow()` в классе `Order`, а это уже может о чем-то говорить.

Еще один недостаток решения, представленного в приведенном выше фрагменте кода, заключается в том, что ссылка `_orderRepository` вносится в экземпляр объекта `Order`. Если заказ восстанавливается из состояния сохраняемости, причем это делается с помощью объектно-реляционного преобразователя, мы уже не отвечаем за создание экземпляра объекта `Order`. Ссылку `_orderRepository` можно было бы вставить в хранилище во время восстановления, но это, по-видимому, не самый лучший способ, если, например, сначала приходится считывать список заказов, а затем вносить хранилище в каждый экземпляр этого списка. Итак, принимая во внимание все приведенные выше рассуждения, попробуем оставить на заказе ответственность за самопроверку, но в то же время снабдить хранилище механизмом установки значений или же параметром для метода `OrderNow()` в ходе проверки, хотя такая методика, по-видимому, не имеет достаточных возможностей для расширения.

На заметку

В ходе приведенной выше дискуссии проявился, по крайней мере, один недостаток, присущий некоторым объектно-реляционным преобразователям: мы не можем контролировать создание экземпляра объекта во время восстановления так, как нам бы хотелось, особенно если учесть растущую популярность базовых структур на основе шаблона Dependency Injection (Внесение зависимостей), о которых речь пойдет в главе 10.

Поиск методов обработки на основе множеств

Независимо от того, на каком из рассмотренных выше стилей проектирования вы решите остановить свой выбор, нужно подумать немного и о текущем долге клиента. Для этого можно было бы, конечно, написать следующий код (при условии, что для определения текущего долга клиента используются заказы, а не счета-фактуры, что, вероятно, сделать труднее):

```
// Код для проверки в заказе текущего долга клиента
// для метода OrderNow()
decimal orderSum = 0;
IList orders = _orderRepository.GetOrders(c);

foreach (Order o in orders)
{
  if (_HasInterestingStateRegardingDebt(o))
    orderSum += o.TotalValue;
}
```

На мой взгляд, такой код недопустим! Лучшим решением было бы раскрытие соответствующего метода в классе `OrderRepository` с помощью метода под названием `CurrentDebtForCustomer()` — текущий долг клиента, воспринимающего экземпляр объекта `Customer` в качестве параметра.

На заметку

При создании интерфейса для хранилища зачастую целесообразно предоставлять такие средства перегрузки, как `X(Customer)` и `X(Guid)`. Ведь было бы весьма досадно получить экземпляр объекта `Customer` только для того, чтобы вызвать метод, когда уже имеется идентификационный номер клиента.

Предвижу следующие вопросы специалистов по базам данных: “А как же согласованность в реальном масштабе времени? Будет ли метод хранилища выполняться в явной транзакции на упорядочиваемом уровне изоляции транзакций хотя бы во время сохранения?” Это вполне осуществимо, но не без трудностей.

Одна из таких трудностей заключается в том, что объектно-реляционный преобразователь (при условии, что он используется, — подробнее об этом в последующих главах данной книги) может не позволить нам полностью контролировать процесс сохранения с точки зрения проверки достоверности, блокировки и прочего. Другая трудность заключается в самом характере процесса сохранения. Это означает, что для сохранения множества еще несохраненных экземпляров объектов требуется некоторое время, а следовательно, время блокировки всех заказов может оказаться довольно продолжительным.

В этом случае можно рассчитывать не на полную, а лишь на частичную согласованность в реальном масштабе времени — рассчитывать можно лишь на проверку суммы других заказов непосредственно перед сохранением текущего заказа, но не на охват операции чтения высоким уровнем изоляции и даже не на проверку в пределах той же самой транзакции.

Это вполне согласуется с тем, что говорится об агрегатах и шаблоне Aggregate в [Evans DDD], где согласованности в реальном масштабе времени рекомендуется добиваться в самом агрегате, а не между агрегатами. Вопросы согласованности между агрегатами будут рассмотрены далее.

Следовательно, если вы считаете рассматриваемый подход вполне приемлемым (опять же, речь может идти о минимальном риске — по крайней мере, для типичной системы), но хотели бы выявить проблему, то можете создать пакетную программу, которая запускается в любой момент, проверяет наличие подобных проблем и возвращает соответствующий отчет. С другой стороны, вы можете поставить запрос в очередь на время сохранения, чтобы затем проверить наличие аналогичных проблем, связанных с конкретным клиентом. Ведь если появление такой проблемы маловероятно, то пользователь не должен ждать, пока завершится проверка ее наличия.

Тот факт, что искомый метод должен быть атомарным, свидетельствует о том, что следует использовать шаблон Service (Обслуживание) [Evans DDD] вместо того, чтобы стремиться к более явному решению в целом. Шаблон Service позволяет проверить текущий долг клиента, а следовательно, может использоваться в классе `Order` с упреждением в удобные моменты времени. Впрочем, его можно использовать и задним числом во время сохранения, поставив в очередь запрос, чтобы обработать его после сохранения. В шаблоне Service, скорее всего, будет использоваться тот же самый метод хранилищ, о котором речь шла выше, но вызывающей части кода знать об этом, конечно, необязательно.

Что же следует сделать задним числом после сохранения? Вероятно, перевести заказ в одно из двух состояний, дающих пользователю возможность сохранить заказ, после чего он перейдет в состояние `NewOrder`. После этого начинается обслуживание, в ходе которого заказ переводится в состояние `Accepted` (Принят) или `Denied` (Отклонен), как показано на рис. 7.1.

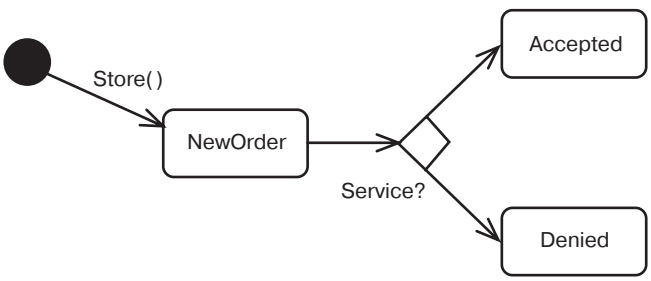


Рис. 7.1. Граф состояний

Итак, мы пришли к трем разным решениям рассматриваемой проблемы: одно — с довольно тесным связыванием (когда все делается в реальном масштабе времени — желательно с высоким уровнем изоляции транзакций), другое — с менее тесным связыванием (хотя оно кажется более сложным из-за большего числа “переменных составляющих”) и, наконец, третье — не предполагающее никаких лишних действий, кроме того, что было показано в тесте, хотя и с определенным риском потери согласованности, которая может быть обнаружена с некоторым запозданием. Вы вольны выбрать то решение, которое подходит вам больше всего.

Раз уж мы коснулись применения шаблона Service для решения задачи обработки, требующей определенных издержек, следует упомянуть и о том, что потребность в шаблонах Service возникает в модели предметной области и по другим причинам.

Обслуживание проверки достоверности по шаблону Service

По совершенно случайному стечению обстоятельств нам требуется такое правило, которое по своему характеру основано на обслуживании. Речь идет о требовании номер 8 "Прежде чем новый клиент будет признан приемлемым, должен быть проверен его кредит в соответствующем кредитном учреждении".

Конечно, можно было бы обратиться к кредитному учреждению в методе `Grant()` или же выбрать решение, аналогичное одному из упомянутых выше, организовав обслуживание после сохранения, обратившись к внешней службе и переведя клиента в одно из двух состояний. Один недостаток такого решения заключается в том, что мы слишком привязываем правила предметной области к сохраняемости.

А другой недостаток выноса функциональных возможностей за пределы текущей единицы работы или карты идентичности заключается в том, что содержимое локального кэша может оказаться не синхронизированным с базой данных, поскольку обработка выполняется в автономной службе. Разумеется, при кэшировании такой риск *всегда* имеется.

Если заранее известно, что содержимое кэша окажется недействительным из-за явного вызова службы в реальном масштабе времени, в таком случае достаточно уничтожить это содержимое. Но его восстановление обойдется недешево, если точно неизвестно, что именно следует восстанавливать. С другой стороны, если неизвестно, когда именно произойдет операция, делающая недействительным содержимое кэша, то придется принять дополнительные меры для восстановления содержимого кэша как можно чаще. Все это подразумевает очень строгие меры обращения с кэшем, не допускающие слишком большой объем и продолжительность хранения содержимого кэша, — по крайней мере, не данные, которые изменяются достаточно часто.

На заметку

Когда я веду речь о кэше, то имею в виду, главным образом, шаблон Identity Map [Fowler PoEAA].

А теперь уделим больше внимания переходам.

Попытка сделать непредусмотренный переход

Для подкрепления принципа "постоянной сохраняемости" мы использовали три состояния. Но не менее важно учесть и такой момент, когда пользователю требуется сделать переход в графе состояний. Конечно, для этого можно было бы поступить

таким же образом, как было показано выше, т.е. перейти в промежуточное состояние, а затем передать бразды правления шаблону Service для перехода в действительное или сбойное состояние. В некоторых случаях это действительно целесообразно, но для выполнения требования номер 11 "У заказов должно быть состояние подтверждения, изменяемое пользователем" такое решение может оказаться нелепым, что опять же зависит от контекста.

На заметку

Возможно, вы заметили, как все сошлось. Прежде мы рассмотрели метод `OrderNow()`, который мог бы стать подходящим техническим решением задачи "постоянной сохраняемости", но на самом деле он послужил лишь основой для выполнения требований. Что же касается соответствия требованиям, то более целесообразным мог бы оказаться выбор метода `Accept()`.

Разумеется, назвать такое решение "нелепым" — это, пожалуй, слишком, но если предположить, что правила проверки представляют собой не более чем простые ограничения, не принимающие во внимание осложнения, связанные с нарушением границ агрегатов, то я бы предпочел применить рассмотренный выше принцип: предоставить пользователю возможность проверки с упреждением, и если она даст отрицательный результат, то считать, что метод изменения состояния будет выполнен успешно. Этот принцип воплощается в коде следующим образом:

```
[Test]
public void CanMakeStateTransitionSafely()
{
  Order o = new Order(new Customer());

  Assert.IsTrue(o.IsOKToAccept);

  // Теперь мы можем благополучно выполнить эту
  // операцию, не доводя до исключения...
  o.Accept();
}
```

Проверка свойства `IsOKToAccept` (Пригоден для принятия) необязательна, но если не сделать ее и затем вызвать метод `Accept()` в тот момент, когда это не предусмотрено, возникнет исключительная ситуация, а следовательно, появится исключение. Для разрешения исключительных затруднений служат исключения, а для разрешения "предполагаемых" затруднений — логические операторы типа `bool`.

Обычно нам нужно знать не только о самом затруднении, но и о причинах его возникновения. В таких случаях с помощью упреждающего метода можно получить список нарушенных правил следующим образом:

```
[Test]
public void CanMakeStateTransitionSafely()
{
  Order o = new Order(new Customer());

  Assert.AreEqual(0, o.BrokenRulesIfAccept.Count);
  // Теперь мы можем благополучно выполнить эту
  // операцию, не доводя до исключения...
  o.Accept();
}
```

Я лично предпочитаю делать это от случая к случаю, руководствуясь лишь общим принципом построения интерфейса, поскольку для реализации в моем распоряжении больше ничего нет — даже стандартного интерфейса. По существу, это протокол для выявления с упреждением факта нарушения правил.

Абстрактное представление транзакций

Абстрактное представление транзакций — весьма эффективный принцип, который часто недооценивается. Альтернативным решением проблемы перехода могло бы стать абстрактное представление транзакций в модели предметной области наряду с проверками достоверности. В этом случае обращать внимание на правильное состояние можно было бы не при выполнении транзакции, а только до или после нее. Например, интерфейс мог бы иметь следующий вид:

```
// Некий потребитель...
DMTransaction dMTx = Something.StartTransaction();
c.Name = string.Empty;
c.Name = "Volvo";
dMTx.Commit();
```

В этом фрагменте кода правила не проверяются до вызова метода `Commit()`. Как вы понимаете, это приводит к очень сложной проблеме, которую придется решать, но в перспективе такое решение представляется интересным.

Оно становится еще более интересным в свете нового пространства имен `System.Transaction` на платформе .NET 2.0. Ведь в будущем могут появиться соответствующие классы со списками транзакций, хэш-таблицы и прочие компоненты. Интерес к такому решению подогревается еще и тем, что транзакции способны охватывать модель предметной области и базу данных. Конечно, и в данном случае появляется немало проблем, но все равно это очень интересное решение.

Очевидно, что у проблемы переходов в разные состояния немало разновидностей. К одной из них, в частности, относится принятие решения о том, когда именно следует предпринять действие. Поэтому нужно быть постоянно начеку.

Идентификационный номер клиента

Примером перехода в состояние наряду с предпринимаемым действием может служить выполнение требования номер 7 "У каждого заказа и клиента должен быть уникальный и удобный для использования номер".

В качестве спонтанного решения этой задачи можно было бы воспользоваться таким встроенным в базу данных механизмом, как `IDENTITY` в MS SQL Server. Но у такого подхода имеется ряд недостатков. Прежде всего, обработку придется выполнять вне модели предметной области, что обойдется недешево. Ведь нужно будет принудительно обновлять экземпляр объекта, на который воздействует данный механизм, и хотя это нетрудно сделать, отслеживать операции будет не так-то просто.

Другой недостаток заключается в том, что значение может стать доступным слишком рано, т.е. задолго до того, как пользователь решит перевести в действительное состояние заказ, который, по его мнению, все еще находится в "предполагаемом" состоянии. (Очевидно, этот недостаток можно преодолеть.)

Еще один недостаток заключается в том, что объектно-реляционные преобразовали нередко вызывают осложнения, если в качестве первичного ключа используется свойство `IDENTITY`. В то же время семантика для использования идентификатора `Guid`, требующего меньшего связывания с базой данных, в качестве первичного ключа зачастую оказывается несколько иной.

> ***На заметку***
>
> Сказанное выше совсем не означает, что столбец `IDENTITY` (как свойство автоматического увеличения в MS SQL Server) следует использовать в качестве первичного ключа только потому, что имеется такой столбец. Его можно сделать только альтернативным ключом, но время от времени все же возникает искушение поступить иначе.

Каковы же альтернативные подходы? Их несколько. Вероятно, самый простой и непосредственный подход заключается в том, чтобы снабдить хранилище методом для выборки (тем или иным способом) следующего свободного идентификационного номера. В коде это может выглядеть следующим образом (при условии, что в заказе отсутствуют строки до тех пор, пока он не будет принят):

```
// Класс Order
public void Accept()
{
  if (_status != OrderStatus.NewOrder)
    throw new ApplicationException
    ("Для перевода заказов в состояние NewOrder можно вызвать только
метод Accept().");

  _status = OrderStatus.Accepted;
  _orderNumber = _orderRepository.GetNextOrderNumber();
}
```

На заметку

Вместо исключения `ApplicationException` (или подкласса) целесообразно использовать исключение `InvalidOperationException` или его производную.

Тем не менее такой подход только усложняет дело — по крайней мере, в тех системах, которые время от времени оказываются в напряженном состоянии, или же в том случае, если номера заказов следуют не по порядку. Это означает, что метод `GetNextOrderNumber()` не сумеет занести изменения в базу данных или, по крайней мере, не зафиксирует их. А это, в свою очередь, означает, что вызов метода `Accept()` и последующего метода сохранения придется заключить в блокирующую оболочку, что обойдется недешево.

Еще один подход состоит в том, чтобы использовать следующее *возможное* значение и быть готовым к появлению его дубликатов в том случае, если данное значение уже использовано кем-то.

Если вы готовы перехватывать исключения в связи с появлением дубликата ключа во время выполнения метода сохранения, то можете вновь вызвать метод `Accept()`, но тогда исключение будет сформировано в методе `Accept()` именно так, как это в нем предусмотрено.

В таком случае лучше просто запросить новый номер заказа в коде, пытающемся восстановить нормальное выполнение после исключения в связи с появлением дубликата ключа. Это, в свою очередь, создает дополнительные трудности, в том числе обращение к базе данных во время выполнения метода сохранения или выявление, прежде всего, экземпляра объекта, испытывающего подобное затруднение (а также столбца, для которого имеется несколько возможных ключей, приходящихся на один экземпляр объекта).

Всех этих затруднений можно избежать, если допустить следование номеров заказов не по порядку и если метод выборки следующего номера заказа фиксирует текущее максимальное значение аналогично тому, как это делает свойство `IDENTITY` в SQL Server. Это означает, что свойство `IDENTITY` допускает пробелы в значениях, которые расходуются напрасно при выполнении отката транзакции (`ROLLBACK`).

А если быть более точным, то метод `GetNextOrderNumber()` фактически вносит мгновенные изменения в базу данных и не ждет следующего вызова метода сохранения, поэтому лучше воспользоваться шаблоном Service. Ведь хранилища сами не принимают решение, когда следует выполнять фиксацию или аналогичную операцию, а службы делают это, поскольку они должны быть автономными и атомарными.

С учетом всех этих изменений метод `Accept()` может теперь принять следующий вид (при условии, что интерфейс `IOrderNumberService` внесен в ссылку `_orderNumberService` на заказ перед вызовом данного метода):

```
// Класс Order
public void Accept()
{
  if (_status != OrderStatus.NewOrder)
    throw new ApplicationException
      ("Для перевода заказов в состояние NewOrder можно вызвать только
метод Accept().");

  _status = OrderStatus.Accepted;
  _orderNumber = _orderNumberService.GetNextOrderNumber();
}
```

Теперь риск возникновения упомянутых выше недостатков сокращается до такой степени, что это можно расценивать лишь как исключительную ситуацию.

Вы будете правы, если посчитаете, что я нашел выход из положения, допустив следование номеров заказов не по порядку (что на самом деле вполне согласуется с установленными требованиями), но иногда это оказывается невозможно. Мне, конечно, не очень нравится такой порядок следования номеров заказов, но в некоторых случаях этого просто не избежать. Что же тогда делать? Пожалуй, нужно будет вернуться к тому, с чего мы начали. Если всего сделанного до сих пор оказывается недостаточно, нам придется заменить весь метод перехода в состояние шаблоном Service, перевести заказ в состояние `Accepted` и задать номер заказа. Конечно, для этого придется выйти за рамки модели предметной области, но что поделаешь, если такова действительность.

До сих пор мы уделили немало внимания разным проблемам, которые приходится решать, когда дело доходит до соблюдения правил в модели предметной области. Рассмотрим еще одну проблему, прежде чем двигаться дальше.

Как избежать осложнений

Требование номер 9 “У заказа должен быть клиент, а у строки заказа — заказ” может служить характерным примером правила, которое я бы предпочел не интерпретировать как правило, если это вообще возможно, а вместо этого включить в конструктор, чтобы избежать его проверки.

Это очень простой, ясный и эффективный прием, хотя и он не лишен недостатков и затруднений.

Например, следующее затруднение возникает в том случае, если требуется получить экземпляр объекта заказа, не связывая его с клиентом ради простоты конструкции пользовательского интерфейса.

На заметку

Это затруднение можно, конечно, устранить непосредственно в самом пользовательском интерфейсе самыми разными способами. Тем не менее не стоит создавать трудности, когда их можно избежать.

Еще одно затруднение заключается в том, что код создания может оказаться настолько сложным, что сам будет вызывать появление исключений, а это возвращает нас к проблемам раскрытия нарушенных правил и т.д. Кроме того, мы уже установили, что исключения должны появляться лишь в исключительных ситуациях, поэтому было бы неверно формировать их в коде создания и считать это нарушением правил.

Впрочем, из этого положения есть выход. Мы можем составить запрос так, чтобы иметь дело только с постоянными, но не временными экземплярами объектов, например, строк заказа. Такая строка сама по себе может не являться агрегатом, но лишь частью агрегата заказа. Следовательно, единственный способ сделать строку заказа постоянной — связать ее с заказом с помощью метода `AddOrderLine()`. А до тех пор можно вообще не думать о том, что у строки заказа нет соответствующего заказа. Строка заказа не может быть сохранена, а значит, метод сохранения вообще не будет формировать исключение, поскольку ему ничего не известно об этой висячей строке.

Конечно, вполне возможно, что для обработки в строке заказа могут потребоваться некоторые сведения о самом заказе, но в противном случае данное решение оказывается вполне приемлемым.

На заметку

Можно, конечно, спорить, нужны ли вообще сведения о самом заказе для строки заказа. Но чаще всего мы можем избежать такой двунаправленности, чтобы не иметь с ней дело.

Возможно, вы заметили, что я все чаще возвращаюсь к одному из шаблонов ППО — Aggregate. Поэтому остановимся на нем, прежде чем двигаться дальше.

Агрегаты как инструмент проверки правил

Итак, рассмотрим агрегаты в качестве блока проверки правил. Конструкция шаблона Aggregate очень важна для проектирования, и поэтому я уделяю ей немало внимания в своей повседневной работе над проектами.

Об агрегировании правил уже упоминалось вкратце ранее в этой главе, теперь рассмотрим этот вопрос немного подробнее.

Корни агрегатов для агрегирования правил

Когда потребитель корня агрегата делает запрос о нарушении правил сохраняемости, он должен получить нарушенные правила из полного агрегата, а не только из экземпляра его корня.

При агрегировании совсем не обязательно использовать тот же самый интерфейс `IValidatableRegardingPersistence` для порожденных классов. Я лично предпочитаю не пользоваться им, поскольку это может нарушить выбранную

мной методику проверки достоверности задним числом в состоянии сохраняемости. (Такое нарушение означает не появление серьезных осложнений, а дублирование исполнительных циклов без веских на то оснований.)

Кроме того, в корне агрегата должно приниматься решение, следует ли использовать правила для потомков, что зависит от контекста.

Управление версиями — очевидное правило агрегатного уровня

Мне бы хотелось завершить этот короткий раздел, посвященный агрегатам, несколькими словами о требовании номер 4 “Большое значение имеет выявление конфликтов параллельной обработки”. Данное требование можно рассматривать как своего рода бизнес-правило, но, на мой взгляд, это техническая, а не прикладная задача. Это еще и удобный случай предоставить хоть какую-то поддержку выявления конфликтов параллельной обработки для большинства объектно-реляционных преобразователей, но мы отложим рассмотрение этого вопроса до глав 8 и 9.

А теперь сместим немного акцент. Прежде чем перейти к реализации интерфейса API, рассмотрим возможности его расширения.

Расширение интерфейса API

Мы уже упоминали об интерфейсе API для основных правил, связанных с сохраняемостью, а также о том, как получить информацию о возникающих в связи с этим проблемах. Что же еще нам требуется в этом интерфейсе?

В зависимости от характера создаваемого приложения нам может потребоваться немало других свойств, связанных с правилами. Основываясь на принципах, сформулированных в начале этой главы, можно прийти к мысли, что было бы неплохо иметь возможность проверять правила со значительным упреждением, особенно в пользовательском интерфейсе. Не менее важно иметь возможность вводить специальные правила не только во время компиляции, но и во время развертывания. Итак, начнем с применения правил в пользовательском интерфейсе.

Запрос правил для настройки пользовательского интерфейса

В своей первой книге [Nilsson NED] я упоминал о необходимости повторного применения (но не дублирования) правил из логических уровней пользовательского интерфейса. Теперь для этого настало время.

Для этой цели предлагается следующий подход: предоставить классам возможность раскрыть для пользовательского интерфейса метаданные, если они потребуются.

Аналогичный подход принят в Web-службах, раскрывающих описание схемы XSD Schema, которое предоставляет требования к данным. Но мы начнем с несколько иного подхода, предоставив списки экземпляров объектов с аналогичной информацией.

На заметку

Более подробно этот вопрос рассматривается в статье Мартина Фаулера [Fowler Fixed-LengthString]. Он предлагает несколько иной подход, чтобы, например, объекту для строки было самому известно о ее максимальной длине.

Возможность для внесения правил

Было бы неплохо иметь возможность определять правила лишь от случая к случаю, поскольку некоторые правила, очевидно, сильно зависят от контекста.

Очень часто требуется, чтобы потребитель имел возможность вводить свои специальные правила, и предоставить ему такую возможность нетрудно, если создать и раскрыть простой язык, пользуясь которым, потребитель мог бы вводить свои правила.

Уточнение реализации

До сих пор мы рассматривали интерфейс API для основных правил, главным образом, с точки зрения потребителя. Однако было бы интересно также обсудить ряд идей относительно реализации этих правил. Итак, опробуем некоторые из этих идей внутри класса `Order`.

Прежде чем приступать непосредственно к уточнению реализации, нам нужно найти в качестве отправной точки самое простое решение данной задачи.

Наивная реализация

Итак, нам требуется, прежде всего, тест. Для этого можно воспользоваться одним из уже созданных ранее тестов и немного переделать его — например, тест `CantExceedStringLengthWhenPersisting()`. Он имел следующий вид:

```
[Test]
public void CantExceedStringLengthWhenPersisting()
{
  Order o = new Order(new Customer());
  o.Note = "01234567890123456789012345678901";

  Assert.IsFalse(o.IsValidRegardingPersistence);
}
```

Для компиляции теста введем в класс `Order` свойство `IsValidRegardingPersistence` следующим образом:

```
// Класс Order
public bool IsValidRegardingPersistence
{
```

```
  get
  {
    if (Note.Length > 30)
      return false;

    return true;
  }
}
```

На заметку

Разумеется, я еще не забыл заклинание РПТ: красная полоса, зеленая полоса, рефакторинг кода; красная полоса, зеленая полоса, рефакторинг кода. Просто мне не хотелось замедлять ритм вашего чтения.

Итак, тест действует. Сделаем его копию, воспользовавшись буфером обмена, изменим имя и немного расширим его:

```
[Test]
public void TryingIdeasWithTheRulesAPI()
{
  Order o = new Order(new Customer());
  o.Note = "01234567890123456789012345678 9";
  Assert.IsTrue(o.IsValidRegardingPersistence);

  o.OrderDate = DateTime.Today.AddDays(1);
  Assert.IsFalse(o.IsValidRegardingPersistence);

  o.OrderDate = DateTime.Today;
  Assert.IsTrue(o.IsValidRegardingPersistence);

  o.Note += "012345";
  Assert.IsFalse(o.IsValidRegardingPersistence);
}
```

Как видите, в данном тесте введено еще одно правило, которое устанавливает, что дата заказа (свойство `OrderDate`) не может быть будущей датой, и по ряду причин (не связанных с каким-то промежутком времени) данное правило стало обязательным в отношении сохраняемости.

На заметку

Для того чтобы связь между правилом проверки даты заказа и инфраструктурой стала очевидной и обязательной, эта дата не должна быть больше или меньше значения, которое может быть сохранено в поле `DATETIME` базы данных, если такое отличие характерно для используемой базы данных.

Для того чтобы данный тест дал отрицательный результат (зеленую полосу), мы должны немного изменить свойство `IsValidRegardingPersistence` следующим образом:

```
// Класс Order
public bool IsValidRegardingPersistence
{
  get
  {
    if (Note.Length > 30)
      return false;

    if (OrderDate > DateTime.Today)
      return false;

    return true;
  }
}
```

С точки зрения сложности правил этого пока что достаточно, хотя и не так уж много. Далее необходимо сообщить пользователю о результатах проверки данного правила, для чего нам потребуется свойство `BrokenRulesRegardingPersistence` — нарушенные правила сохраняемости. И в этом случае мы будем действовать максимально просто, но сначала придется внести в тест ряд следующих изменений:

```
[Test]
public void TryingIdeasWithTheRulesAPI()
{
  Order o = new Order(new Customer());
  o.Note = "01234567890123456789012345678 9";
  Assert.IsTrue(o.IsValidRegardingPersistence);
  Assert.AreEqual(0, o.BrokenRulesRegardingPersistence.Count);

  o.OrderDate = DateTime.Today.AddDays(1);
  Assert.IsFalse(o.IsValidRegardingPersistence);
  Assert.AreEqual(1, o.BrokenRulesRegardingPersistence.Count);

  o.OrderDate = DateTime.Today;
  Assert.IsTrue(o.IsValidRegardingPersistence);
  Assert.AreEqual(0, o.BrokenRulesRegardingPersistence.Count);

  o.Note += "012345";
  Assert.IsFalse(o.IsValidRegardingPersistence);
  Assert.AreEqual(1, o.BrokenRulesRegardingPersistence.Count);
}
```

А затем осуществим первую реализацию:

```
// Класс Order
public IList BrokenRulesRegardingPersistence
```

```
{
  get
  {
    IList brokenRules = new ArrayList();

    if (Note.Length > 30)
      brokenRules.Add("Слишком длинное примечание.");

    if (OrderDate > DateTime.Today)
      brokenRules.Add("Дата заказа относится к будущему.");

    return brokenRules;
  }
}
```

Зачастую такой информации о нарушенных правилах оказывается явно недостаточно, но, по крайней мере, мы начали двигаться в нужном направлении. Другое дело, что нежелательно выражать одни и те же правила в свойствах `IsValidRegarding-Persistence` и `BrokenRulesRegardingPersistence`. Поэтому заменим пока что свойство `IsValidRegardingPersistence` решением, которое исключает дублирование, хотя и менее эффективно с точки зрения исполнения.

На заметку

Неэффективностью решения с точки зрения исполнения можно зачастую пренебречь, если сравнить ее, например, с некоторыми обращениями к базе данных.

```
// Класс Order
public bool IsValidRegardingPersistence
{
  get {return BrokenRulesRegardingPersistence.Count == 0;}
}
```

Итак, мы реализовали возможность запрашивать следующее: находится ли экземпляр объекта в действительном состоянии для сохранения в любой момент и имеются ли в связи с этим какие-то затруднения (конечно, до настоящей реализации пока еще далеко, тем не менее, мы кое-что сделали в этом отношении).

Впрочем, имеется и другой протокол, связанный с переходами. Итак, сделаем переход и попробуем получить информацию о связанных с этим затруднениях. Кроме того, мы будем рассматривать соответствующие ситуации задним числом, т.е. реагировать на попытки перехода в определенное состояние, когда уже известны связанные с этим затруднения.

Допустим, что правило принятия заказа (`Order` по методу `Accept()`) состоит в том, чтобы клиент (`Customer`) заказа оказался в состоянии `Accepted`. В коде это может выглядеть следующим образом:

```
[Test]
public void TryingTheAcceptTransitionWithTheRulesAPI()
{
  Order o = new Order(new Customer());

  Assert.IsFalse(o.IsOKToAccept);
  o.Customer.Accept();
  Assert.IsTrue(o.IsOKToAccept);
}
```

Следует особо подчеркнуть, что для нового заказа, находящегося в состоянии `NewOrder`, вполне допустимо, чтобы клиент не был принят.

Далее, если перевести клиента в состояние `Accepted`, то вполне допустимо принять заказ (по методу `Accept()`).

Но мне бы хотелось также показать, что общие правила для заказа распространяются и на переходы, а для этого потребуются следующие дополнения в тесте:

```
[Test]
public void TryingTheAcceptTransitionWithTheRulesAPI()
{
  Order o = new Order(new Customer());

  Assert.IsFalse(o.IsOKToAccept);
  o.Customer.Accept();
  Assert.IsTrue(o.IsOKToAccept);

  o.OrderDate = DateTime.Today.AddDays(1);
  Assert.IsFalse(o.IsOKToAccept);

  try
  {
    o.Accept();
    Assert.Fail();
  }
  catch (ApplicationException ex) {}
}
```

На заметку

По умолчанию в конструкторе задается значение `DateTime.Today` свойства `OrderDate` нового заказа, и поэтому заказ проверяется на готовность быть принятым непосредственно (свойство `IsOKToAccept`).

Попутно мне хотелось бы также показать, что произойдет, если сделать непредусмотренный переход.

Для этого нам придется сделать кое-что в классе `Order` — для начала в свойстве `IsOKToAccept`.

```
// Класс Order
public bool IsOKToAccept
{
  get
  {
    if (Customer.Status != CustomerStatus.Accepted)
      return false;

    return IsValidRegardingPersistence;
  }
}
```

Как видите, сначала проверяются особые правила (в данном случае одно правило), а затем правила, связанные с сохраняемостью, поскольку нежелательно иметь далее дело с тем, что не подлежит сохранению. Следовательно, нужно как можно раньше обратить внимание на возможные проблемы (в том числе и те, что связаны с сохраняемостью).

Дополнительные изменения могут потребоваться для того, чтобы сообщить о нарушении правил при вызове метода `Accept()`, но это нетрудно сделать и поэтому необязательно показывать.

Это же относится и самому методу `Accept()`. Он достаточно прост и требует лишь проверки свойства `IsOKToAccept` для выполнения перехода.

Более каверзным и не столь очевидным оказывается тот факт, что правила, связанные с сохраняемостью (свойство `IsValidRegardingPersistence`), теперь усложнились, поскольку после перехода заказа в состояние `Accepted` нам придется проверять те же самые правила даже в свойстве `IsValidRegardingPersistence`. Как было показано выше, настоящая обработка была перенесена в свойство `BrokenRulesRegardingPersistence`, поэтому именно в нем мы и сделаем следующие изменения:

```
// Класс Order
public IList BrokenRulesRegardingPersistence
{
  get
  {
    IList brokenRules = new ArrayList();

    if (Note.Length > 30)
      brokenRules.Add("Слишком длинное примечание.");

    if (_orderDate > DateTime.Today)
      brokenRules.Add("Дата заказа относится к будущему.");

    if (_OrderIsInThisStateOrBeyond(OrderStatus.Accepted))
```

```
      _CollectBrokenRulesRegardingAccepted(brokenRules);

    return brokenRules;
  }
}
```

Таким образом, в методе `_CollectBrokenRulesRegardingAccepted()` проверяется особое правило перехода в состояние `Accepted` (в данном случае пребывания в этом состоянии).

На заметку

Возможно, вы обратили внимание на то, что в методе `_CollectBrokenRules-RegardingAccepted()` использован шаблон Collecting Parameter (Накопление в параметре) [Beck SBPP]. Этому методу передан список нарушенных правил и запрошен метод ввода в список дополнительных нарушенных правил, если таковые имеются.

Подводя итог, следует признать, что полученный выше код не вполне доброкачественен и недостаточно функционален. В частности, необходимо подчеркнуть следующее.

- Разрастание кода в свойстве `BrokenRulesRegardingPersistence` из-за того, что проверка правила кодируется вручную.
- Информация о нарушенных правилах возвращается лишь в виде строк, чего явно недостаточно.
- Не решена задача специальной настройки.
- Не решена задача предоставления сведений о самих правилах, применяемых в пользовательском интерфейсе.

Итак, настало время немного уточнить эту наивную реализацию, не повторяя многократно пройденный путь к данной цели. Воспользуемся уже имеющимися тестами и немного обобщим реализацию.

Переход от самой наивной стадии к созданию классов правил

Потребность в создании классов правил объясняется тем, что благодаря их декларативному использованию можно сократить разрастание кода, решить задачу специальной настройки и предоставить вовне сведения о правилах.

Сложные правила приходится всегда кодировать вручную в каждом отдельном случае, и поэтому они с трудом поддаются специальной настройке с помощью информации, вводимой в файл конфигурации. Более того, для общих форм сведения о сложных правилах вообще не имеют значения. Но в данном случае наша цель не правила. Нельзя сказать, что они нас вообще не интересуют — напротив, мы можем даже

закодировать их вручную, если они вписываются в общую картину. Я просто хотел бы подчеркнуть, что объявление этих правил не имеет значения.

Вместо это нам нужно объявить простые правила, т.е. те правила, которые неинтересно кодировать всякий раз вручную. Итак, попробуем это сделать.

На заметку

Будьте начеку, чтобы не впасть в заблуждение, строя базовые структуры наперед. Как правило, лучше дать им самим проявиться в коде и затем начинать их построение [Fowler Harvested Framework].

В рассматриваемых здесь классах правил должен быть реализован интерфейс. Назовем его `Irule`.

```
public interface IRule
{
  bool IsValid {get;}
  int RuleId {get;}
  string[] ParticipatingLogicalFields {get;}
}
```

Свойство `IsValid` служит для проверки достоверности правила, свойство `RuleId` — для преобразования информации о правилах и т.п. Для этого требуется дополнительная информация (как правило, в виде длинных постоянных списков, но это остается на ваше усмотрение).

Назначение свойства `ParticipatingLogicalFields` — участвующие логические поля, состоит в том, чтобы предоставить потребителю возможность выделить те поля, которые оказывают влияние на определенное нарушенное правило.

Ниже приведен пример класса правил. В нем выполняется проверка даты в определенных пределах.

```
public class DateIsInRangeRule : RuleBase
{
  public readonly DateTime MinDate;
  public readonly DateTime MaxDate;

  // Примечание: здесь приведен только "самый крупный" конструктор.
  public DateIsInRangeRule(DateTime minDate, DateTime maxDate,
    int ruleId, string[] participatingLogicalFields,
    string fieldName, object holder)
    : base(ruleId, participatingLogicalFields, fieldname
    , holder)
  {
    MinDate = minDate;
    MaxDate = maxDate;
```

```
  }

  public override bool IsValid
  {
    get
    {
      DateTime value = (DateTime)base.GetValue();

      return (value >= MinDate && value <= MaxDate);
    }
  }
}
```

Кроме того, для общего функционирования, включая чтение значений посредством рефлексии, а также для абстрагирования подклассов от инфраструктуры требуется базовый класс (`RuleBase`). Таким образом, для подклассов остается лишь "самый интересный" код и некоторые конструкторы (один из них представлен в приведенном выше фрагменте кода).

На заметку

Кстати, `fieldName` может быть либо полем либо свойством. А `holder` — это экземпляр модели предметной области, например, экземпляр объекта заказа, в котором используется объект правила.

Для удобства пояснения на рис. 7.2 приведено схематическое представление разных компонентов приложения.

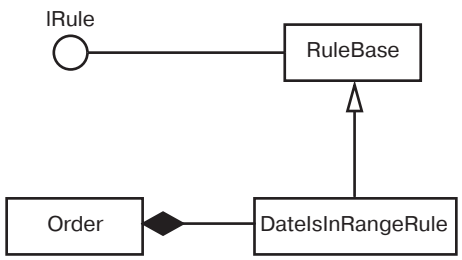


Рис. 7.2. Схематическое представление компонентов `IRule`, `RuleBase`, `DateIsInRangeRule` и класса из модели предметной области

Теперь остается "только" написать такие классы, как `DateIsInRangeRule`, для разных нужд. И хотя такой код будет использоваться довольно часто, его написание требует некоторых усилий.

Возникает резонный вопрос: а для чего требуется рефлексия? Требуется, скорее, не сама рефлексия, а ее *положительный* результат. Это означает, что правило

желательно определить один раз, а затем выполнять его, не предоставляя *значение* для явной проверки данного правила. Вместо этого лучше предоставить поле или свойство, *имеющее* такое значение. Посмотрим, как это можно сделать.

> ***На заметку***
>
> Один из возможных вариантов оптимизации состоит в том, чтобы использовать рефлексию только по мере надобности. Так, если используются ссылочные типы данных, для получения того же результата не нужно прибегать к рефлексии. С другой стороны, это несколько повышает сложность, поэтому нужно быть начеку.
>
> Грегори Янг добавил к этому следующее замечание: "В качестве еще одного варианта можно было бы предложить генерацию кода для амортизации издержек на рефлексию".

Составление списка правил

До сих пор мы создали лишь один класс правил. Допустим, имеется еще один такой класс `MaxStringLengthRule` — правило соблюдения максимальной длины строки. Применим их в классе `Order` следующим образом:

```
// Класс Order
private IList _persistenceRelatedRules = new ArrayList();
private DateTime _orderDate;
public string Note = string.Empty;

private void _SetUpPersistenceRelatedRules()
{
  _persistenceRelatedRules.Add(new DateIsInRangeRule
    (MyDateTime.MinValue, DateTime.Today, "OrderDate", this));

  _persistenceRelatedRules.Add(new MaxStringLengthRule
    (30, "Note", this));

}
```

Остается лишь вызвать метод `_SetUpPersistenceRelatedRules()` в конструкторах класса `Order`.

> ***На заметку***
>
> В данном случае ничего не делается для того, чтобы назначить подходящие идентификаторы для экземпляров объектов правил. Вместо этого используется конструктор, в котором вообще не предоставляется никаких значений.

Использование списка правил

Итак, правила, связанные с сохраняемостью, обрели достойное место. Посмотрим, как это повлияет на реализацию метода BrokenRulesRegardingPersistence.

```
// Класс Order
public IList BrokenRulesRegardingPersistence
{
   get
   {
      IList brokenRules = new ArrayList();

      RuleBase.CollectBrokenRules(brokenRules
        , _persistenceRelatedRules);

    if (_OrderIsInThisStateOrBeyond(OrderStatus.Accepted)
      _CollectBrokenRulesRegardingAccepted(brokenRules);

    return brokenRules;
  }
}
```

На заметку

Еще раз подчеркну, что список нарушенных правил должен представлять собой реализацию шаблона Notification [Fowler PoEAA2]. Я по-прежнему так считаю, хотя и немного отклонился в сторону.

Грегори Янг дал по этому поводу следующие пояснения: "Главное отличие заключается в том, что конкретное правило (IRule) раскрывается по отношению к суррогатному объекту, который просто сообщает, что оно нарушено, как это и делается в шаблоне Notification".

Преимущество такого решения состоит в том, что мы можем без труда написать метод RuleBase.IsValid(), передав ему в виде параметра список правил, связанных с сохраняемостью (_persistenceRelatedRules). Благодаря этому мы можем повысить эффективность выполнения для свойства IsValid, не увеличивая объем дублированного кода. Имеется в виду, что в такой реализации должно быть предусмотрено выявление любого недействительного правила, а следовательно, нет необходимости накапливать список нарушенных правил, просматривать все правила (даже если нарушено первое из них) и т.д.

Но вернемся к основному ходу рассуждений. Посмотрим, как метод CollectBrokenRules() может выглядеть в классе RuleBase:

```
// Класс RuleBase
public static void CollectBrokenRules
```

```
  (IList brokenRules, IList rulesToCheck)
{
  foreach (IRule r in rulesToCheck)
    if (! r.IsValid)
      brokenRules.Add(r);
}
```

Вот, собственно, и все, что касается реализации правил, связанных с сохраняемостью. Она оказалась достаточно простой. А что же правила, приходящиеся на каждый переход в состояние?

Составление подсписков

В качестве одного из решений данной задачи можно было бы также составить списки для конкретных переходов, например, `_acceptedSpecificRules`. В этом случае метод `BrokenRulesRegardingPersistence` может быть написан следующим образом:

```
// Класс Order
public IList BrokenRulesRegardingPersistence
{
  get
  {
    IList brokenRules = new ArrayList();

    RuleBase.CollectBrokenRules(brokenRules
      , _persistenceRelatedRules);

    if (_OrderIsInThisStateOrBeyond(OrderStatus.Accepted)
      RuleBase.CollectBrokenRules(brokenRules,
        _acceptedSpecificRules);
    return brokenRules;
  }
}
```

Таким образом, *б*ольшая часть работы теперь связана с определением списка правил (опять же, мы не будем развивать дальше данный подход, поскольку это на самом деле очень длинный путь).

Теперь нужно разместить все правила в одном месте (фактически в нескольких местах, если имеются списки правил для переходов), т.е. в экземплярном списке. Если приложить немного усилий, то оба приведенных до сих примера классов правил (`DateIsInRangeRule` и `MaxStringLengthRule`) можно заменить на список статических правил. Необходимо также изменить правило соблюдения даты, чтобы проверять по нему значение свойства `DateTime.Today` отдельно, а не помещать его в конструктор. Кроме того, конструктор можно было бы снабдить относительными датами, где значения –1 и 1 означали бы предыдущий и следующий день соответственно, а также функциональными возможностями, которые позволяли бы только одному статическому списку существовать до следующего дня. Разумеется, правила

следует стремиться переносить в статические (static) списки, чтобы сократить издержки. Но все эти меры никак не нарушают общий принцип.

Вместо того чтобы рассматривать все эти детали, касающиеся области действия правил, лучше перейдем к следующим насущным вопросам.

Усовершенствование интерфейса API

Мой коллега Ингмар Лундберг заметил, что использование свойства для метода BrokenRulesRegardingPersistence ему кажется несколько странным, а Кристиан Кроухэрст посчитал, что данному методу было бы лучше передать накапливающий параметр и возвратить логическое значение bool, например, следующим образом:

```
public bool CheckOKToAccept(IList brokenRules)
```

Это повлияет немного на код потребителя. Так, вместо следующего кода:

```
// Некий потребитель...
if (! o.IsOKToAccept)
_Show(o.BrokenRulesIfAccept);
```

Как правило, следует написать следующий код:

```
// Некий потребитель...
IList brokenRules = new ArrayList();
if (! o.CheckOKToAccept(brokenRules))
_Show(brokenRules);
```

Что же касается накапливающего параметра brokenRules, то если он не инициализирован (т.е. имеет пустое значение null) при вызове метода CheckOKToAccept(), последний может проигнорировать его и возвратить только логическое значение bool. Но это лишь один подход, тогда как перегрузка метода CheckOKToAccept() без параметра, по-видимому, больше отвечает данной цели. (Это означает не разделение на два разных метода, а только переход от одного вида перегрузки к другому. Но даже если логическое значение bool оказывается истинным (true), в возвращаемом списке *могут* присутствовать, в частности, предупреждения).

Еще одним следствием такого подхода является избавление потребителя от необходимости дважды выполнять код проверки правил: один раз — для свойства IsOKToAccept, а другой раз — для свойства BrokenRulesIfAccept. Например, маловероятно, чтобы потребитель написал следующий код:

```
// Некий потребитель...
if (! o.CheckOKToAccept())
{
  IList brokenRules = new ArrayList();
  o.CheckOKToAccept(brokenRules);
  _Show(brokenRules);
}
```

Мне такой подход по душе.

Специальная настройка

Для специальной настройки модели предметной области имеются самые разные способы. В частности, двум разным клиентам можно предоставить возможность самостоятельно ввести некоторое поведение в приложение, которым они пользуются. Одно из привлекательных решений этой задачи состоит в использовании шаблона Decorator (Декоратор) [GoF Design Patterns], а другое — в предоставлении перехватчиков наследования и настраиваемого кода создания.

Еще один достаточно простой метод состоит в том, чтобы дать возможность определить правила или, по крайней мере, ввести экземпляры конкретных правил, описав их в виде файлов конфигурации.

Большая часть этих функциональных возможностей уже реализована в текущем черновом варианте решения. Нам нужно лишь найти способ ввода правил, поэтому наметим следующий интерфейс под названием `ICustomRules`:

```
interface ICustomRules
{
  void AddCustomRule(IRule r);
}
```

В этом первом черновом варианте предполагается, что правила, связанные с сохраняемостью, могут быть введены, причем в экземплярный список правил. Впрочем, если требуется, это положение можно исправить.

На заметку

Необходимость вводить экземпляры правил в статические списки напомнила мне о давнем желании. Мне бы хотелось определить также статические интерфейсы. Почему бы и нет?

Теперь мы можем ввести дополнительные правила в ходе получения экземпляров классов из модели предметной области, но, как уже отмечалось, получение экземпляров может оказаться затруднительным, и поэтому лучше воспользоваться шаблоном Factory или иной разновидностью фабрики.

Правда, специальная настройка, о которой здесь шла речь, действует слаженно и дает возможность перетаскивать информацию о правилах, что может оказаться очень удобно для создания форм.

Предоставление пользователю метаданных

Правила, связанные с сохраняемостью, следует интерпретировать как нечто, извлекаемое из модели предметной области с упреждением, поскольку ими может легко воспользоваться потребитель. С этой целью введем еще один метод в интерфейс `IValidatableRegardingPersistence`.

```
interface IValidatableRegardingPersistence
{
  bool IsValidRegardingPersistence {get;}
  IList BrokenRulesRegardingPersistence {get;}
  IList PersistenceRelatedRules {get;}
}
```

А далее потребитель сам решает, что ему делать с обнаруженными правилами. Вполне возможно, что некоторые правила подойдут для пользовательского интерфейса при отображении новых форм, в том числе заполняемых полей, заданных пределов значений, длины строк и прочего.

Концептуально мне такое решение нравится. Нечто подобное может даже оказаться пригодным для большого числа проектов. Но я не могу отделаться от ощущения, что у этого решения имеется один изъян, связанный, в частности, с правилами для отдельных переходов, особенно если масштабы осложнений нарастают.

Проблема подстать шаблону

А можно ли в данном случае воспользоваться шаблоном State (Состояние) [GoF Design Patterns], который подробно был рассмотрен в главе 2? Выглядит заманчиво, поскольку можно было бы возложить ответственность за составление списка правил для конкретных переходов на определенные классы состояний. Однако еще неясно, насколько хорошо шаблон State подходит для решения такой проблемы.

Что же касается разных состояний, то с ними можно также работать на нескольких уровнях иерархии наследования (например, в том случае, если экземплярное определение правила оказывается одинаковым для нескольких состояний).

Как обращаться со сложными правилами

Как уже отмечалось, сложные правила приходится писать вручную, но я не намерен этого делать. На мой взгляд, для сложных правил вполне подойдет шаблон Specification [Evans DDD].

Используя описания, мы пытаемся выразить важные концепции предметной области в коде. Для ППО вообще характерен акцент на концепциях предметной области, а не на таких технических концепциях, как числовые интервалы и длины строк.

Так, в шаблоне Specification может быть класс под названием `ReadyToInvoiceSpecification` (Описание готового счета-фактуры) со следующим методом:

```
public bool Test(Order o);
```

Рассмотрим краткий и простой пример. Допустим, что требуется сформировать единый счет-фактуру для клиента, несмотря на то что у клиента в настоящий момент имеются несколько заказов, по которым может быть выписан счет-фактура. Вероятно, в этом случае следует запросить клиента, имеется ли у него что-нибудь для выписывания счета-фактуры. Для этой цели в класс `Customer` можно ввести следующий метод (при условии, что интерфейс `IOrderRepository` уже внесен):

```
// Класс Customer
public IList OrdersToInvoice(ReadyToInvoiceSpecification
  specification)
{
  IList orders = _orderRepository.GetOrders(this);
  IList result = new ArrayList();
  foreach (Order o in orders)
  {
    if (specification.Test(o))
      result.Add(o);
  }

  return result;
}
```

Как всегда, конкретные рекомендации относительно проектирования, если они вообще возможны, зависят от контекста, но на этом простом примере удобно будет показать, как следует применять концепцию с помощью очень простого и достаточно гибкого кода. (Параметр шаблона Specification настраивается до передачи соответствующему методу.)

Опять же, правила, определяющие готовность заказа для выписывания счета-фактуры, инкапсулированы в шаблоне Specification и не распространяются на несколько методов.

На заметку

Для реализации шаблона Specification характерна следующая замечательная особенность: ее можно определить и в базе данных, чтобы использовать для формирования запросов с акцентом на концепцию.

Главная трудность применения шаблона Specification к правилам заключается в том, чтобы упростить автоматическое выполнение упреждающих действий в пользовательском интерфейсе вместо того, чтобы допускать нарушение правил. С другой стороны, с таким недостатком можно мириться, особенно если описания используются для разрешения более сложных затруднений, которые все равно невозможно предвидеть заранее. Так, если в описании применяется концепция, аналогичная интерфейсу `IRule`, отдельные части описания могут использоваться, как обычно, для автоматической настройки пользовательского интерфейса с упреждением.

Следует также отметить, насколько удобно выполняется блочное тестирование сложных правил, инкапсулированных в шаблоне Specification. Кроме того, описания очень легко тестируются по отдельности.

Привязка к абстрактному представлению сохраняемости

В предыдущей главе речь шла о подготовке к инфраструктуре сохраняемости на уровне абстракции под названием NWorkspace (который служил лишь для примера такой абстракции). Нетрудно догадаться, что на этом уровне предусматривается также некоторая поддержка проверки достоверности, поскольку характер вызовов метода `PersistAll()` требует реагирования задним числом.

Разумеется, было бы нежелательно, чтобы для абстрактного представления сохраняемости предполагалась реализация интерфейса в корнях агрегатов.

Внедряемый интерфейс проверки достоверности

Следовательно, интерфейс `IValidator` целесообразно внедрить в интерфейс `IWorkspace`. Он может иметь следующий вид:

```
public interface IValidator
{
  bool IsValidatable(object entity);
  bool IsValid(object entity);
  IList BrokenRules(object entity);
}
```

Таким образом, реализация интерфейса `IWorkspace` связывается с реализацией интерфейса `IValidator`, а в последней принимается решение, какие интерфейсы следует искать для проверки объектов-сущностей. В данном случае в реализации интерфейса `IValidator` осуществляется поиск реализаций интерфейса `IValidatableRegardingPersistence` и принимается решение, что именно следует делать с реализацией интерфейса `IValidatableRegardingPersistence` для формирования, если требуется, исключения во время вызова метода `PersistAll()`. Для большей наглядности это взаимодействие отдельных компонентов представлено схематически на рис. 7.3.

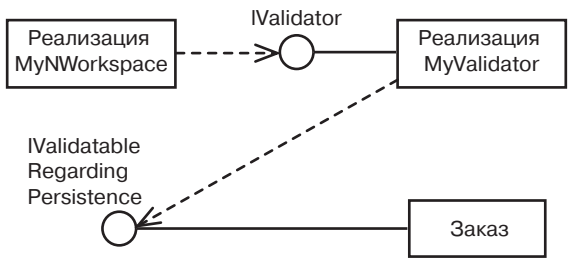


Рис. 7.3. Схематическое представление взаимодействия отдельных компонентов

Интерфейс `IValidator`, как и база данных, допускает проверку задним числом, но он реагирует чуть раньше, чем база данных, причем его реакция немного сложнее и в большей степени сосредоточена на правилах предметной области.

Для того чтобы упростить тестирование, требуется также реализация интерфейса `IValidator`, называемая `ValidatorThatDoesNothing` (Средство проверки достоверности, которое ничего не делает). Она предназначена для того, чтобы не мешать выполнению блочных тестов. С этой целью реализация интерфейса `ValidatorThatDoesNothing` внедряется в реализацию интерфейса `IWorkspace` вместо поиска экземпляров интерфейса `IValidatableRegardingPersistence`. Для написания ее кода требуется не больше минуты, что не так уж и накладно по сравнению с извлекаемой из этого выгодой.

Альтернативные решения для проверки достоверности задним числом при сохранении

Как уже отмечалось, в связи с выполнением проверки достоверности при сохранении возникают определенные трудности для некоторых объектно-реляционных преобразователей. Поэтому было бы интересно поразмыслить над тем, как преодолеть эти затруднения.

Я лично предпочитаю проверять правила в реальном масштабе времени, т.е. именно тогда, когда это требуется. И для решения этой задачи вполне подходят шаблоны Service.

Одно из альтернативных решений состоит в том, чтобы обновлять список нарушенных правил после каждого изменения свойства. Таким образом, свойство предварительно рассчитывается по мере надобности. Такое решение может оказаться более или менее эффективным в зависимости от того, как пользоваться шаблонами.

На заметку

Грегори Янг по этому поводу заметил, что, применяя упоминавшиеся ранее метаданные по правилам (т.е. выясняя влияние отдельных полей на правила), можно весьма эффективно перепроверять правила и всегда иметь текущие результаты проверки достоверности.

Но в этом случае приходится обращаться к отслеживанию изменений состояния объектов, иначе придется писать вручную много кода в механизмах установки, если, конечно, для решения задачи взаимно пересекающихся отношений не применяется АОП. Кроме того, вместо общедоступных полей придется воспользоваться свойствами — даже если можно было бы вполне обойтись и общедоступными полями.

На заметку

Такой стиль проектирования применяется в среде CSLA.NET [Lhotka BO], где также имеется поддержка многоступенчатой отмены.

Еще одно решение состоит в том, чтобы проверять логическое значение свойства `IsValidated`, которое становится истинным всякий раз, когда воспроизводится список нарушенных правил, и ложным после каждого изменения. При проверке достоверности в процессе сохранения предполагается, что свойство `IsValidated` принимает истинное логическое значение, а иначе формируется исключение. Такое решение оказывает некоторое влияние на выбранную модель программирования, поскольку требует обязательного обращения к свойству `IsValidRegardingPersistence` или `BrokenRulesRegardingPersistence`. Еще один недостаток такого решения (если считать недостатком обязательное обращение к свойству, а я именно так и считаю) заключается в тех же самых трудностях, которые возникают в связи с упоминавшимся выше отслеживанием изменений состояния объектов.

Так или иначе, всегда полезно иметь два разных решения на тот случай, если одно из них окажется более проблематичным, чем другое.

Повторное использование метаданных преобразования

Прежде чем двигаться дальше, рассмотрим еще одну идею. Она тесно связана с соблюдением упоминавшегося ранее полезного принципа определять что-нибудь один раз, но применять везде, где это уместно, в том числе в пользовательском интерфейсе, модели предметной области или базе данных.

Если приходится тратить немало сил на определение вручную многих простых правил, например, соблюдения максимальной длины строк, то следует рассмотреть возможность для динамического установления этих правил вместо чтения метаданных из какого-либо источника. Так, если в базе данных имеется некоторая информация, то в зависимости от выбранного решения задачи преобразования воспользоваться этой информацией будет нелегко. В качестве альтернативы можно было бы выполнить преобразование самих метаданных, чтобы обнаружить в них источник информации для нескольких правил.

Как бы там ни было, самое главное в рассматриваемой идее — прочитать из какого-либо источника данные при запуске приложения и затем динамически установить правила.

Разумеется, как и все остальное в этой главе, данная идея не является панацеей, а лишь одним из многих подходов к решению сложной задачи.

Обобщения и анонимные методы как спасение

Если используется среда, поддерживающая обобщения (например, .NET 2.0), то, применяя обобщения во всех предложенных ранее решениях, можно добиться преимущества в отношении ясности, типовой безопасности и производительности. Это же относится и к большинству других глав данной книги.

В качестве небольшого примера вместо следующего кода:

```
IList BrokenRulesRegardingPersistence {get;}
```

Можно было бы применить обобщения следующим образом:

```
IList<IRule> BrokenRulesRegardingPersistence {get;}
```

На первый взгляд, это не такое уж и значительное улучшение или нечто, без чего нельзя обойтись (как показано на протяжении всей этой книги), но оно окажет положительное влияние на многие места в коде.

Еще один способ, которым я не пользовался в средах, в которых работал чаще всего, состоит в применении анонимных методов, аналогичных блокам или замыканиям в других языках программирования.

Принцип, положенный в основу таких методов, состоит в том, чтобы передавать методу блок кода в виде параметра, но использовать в этом блоке кода переменные, определенные в области действия метода, получающего данный блок.

Некоторые усматривают в этом способ реализации описаний без создания новых классов, но тогда теряется ясность и четкость при *именовании* описаний, которое считается весьма эффективным свойством шаблонов Specification. Я же рассматриваю анонимные методы как удобный способ выбора разных вариантов многократного использования ресурсов. Например, при желании в код можно внести лишь часть реализации метода. По существу, это та же настраиваемость, только рассматриваемая под другим углом зрения.

Для того чтобы дать общее представление о данном способе тем, кому он еще малоизвестен, объединим обобщения с анонимными методами в рассмотренном выше фрагменте кода и посмотрим, насколько он улучшится. Напомним, что циклическая проверка готовности заказов клиента к выписыванию счета-фактуры была реализована в классе `Customer` следующим образом:

```
// Класс Customer
public IList OrdersToInvoice(ReadyToInvoiceSpecification
  specification)
{
  IList orders = _orderRepository.GetOrders(this);
  IList result = new ArrayList();
  foreach (Order o in orders)
  {
    if (specification.Test(o))
      result.Add(o);
  }

  return result;
}
```

Ниже приведен немного видоизмененный вариант.

```
// Класс Customer
public List<Order>OrdersToInvoice(Predicate<Order>
readyToInvoicePredicate)
```

```
{
  List<Order> orders = _orderRepository.GetOrders(this);
  return orders.FindAll(readyToInvoicePredicate);
}
```

На этот раз параметр является предикатом для готовности заказа к выписыванию счета-фактуры. При вызове в качестве параметра предоставляется анонимный метод. Этот вызов может быть организован следующим образом (при условии, что в данный момент используется очень простой алгоритм):

```
c.OrdersToInvoice(delegate(Order o) {return o.Status == OrderStatus.
ToBeInvoiced;});
```

Данное "описание" или предикат передается методу `FindAll()` обработки общего списка для отфильтровывания заказов, готовых к выписыванию счета-фактуры.

На момент написания этой книги еще рано было говорить о практическом применении обобщений и анонимных методов, но я уверен, что в ближайшем будущем они существенно изменят мой подход к проектированию. Я имею в виду не такие мелочи, как приведенные выше примеры, а значительный прогресс в мышлении.

Но в то же время это не более чем полезные инструменты в арсенале разработчика, которые вряд ли приведут к радикальным переменам.

Другие подходы

Выше уже упоминалось вкратце, как рассматриваемые здесь вопросы решаются другими разработчиками, и прежде чем завершить эту главу, мне бы хотелось привести еще два источника на данную тему.

Во-первых, это книга *Streamlined Object Modeling* [Nicola et al. SOM], в которой подробно рассматриваются различные правила и методики проектирования, а также их реализация в объектно-ориентированном контексте.

И во-вторых, это книга *Enterprise Patterns and MDA* [Arlow/Neustadt Archetype Patterns], в которой исходные шаблоны рассматриваются применительно к организации правил. На мой взгляд, в обеих книгах можно почерпнуть немало полезных идей, которые послужат в качестве отправной точки.

Резюме

Уже немало лет данная тема остается все еще неразработанной и не имеет очевидных решений. Поэтому мы рассмотрели лишь несколько идей в отношении правил: как типичных, так и нетипичных.

В этой главе было уделено немало внимания объектам правил как методу их реализации, но мы даже не коснулись других вариантов. Как всегда, отсутствие всякого

решения оказалось самым лучшим из решений. В частности, намеченные нами объекты правил подошли бы лучше всего для очень простых из всех имеющихся правил.

Главный вывод из этой главы заключается, вероятно, в следующем: нужно пытаться как можно четче разграничивать правила, связанные с сохраняемостью и предметной областью, хотя связь между некоторыми правилами по-прежнему остается. Работа с разными состояниями представляется весьма эффективным способом приблизиться к цели сохранения всех состояний. И наконец, не следует забывать, что правила очень сильно зависят от контекста, поэтому нужно уделять внимание значению правил для модели.

В этой главе мы рассмотрели многие возможности приложения, определенные в главе 4. Теперь перейдем к инфраструктуре, которая для этого требуется, сделав акцент на поддержке сохраняемости.

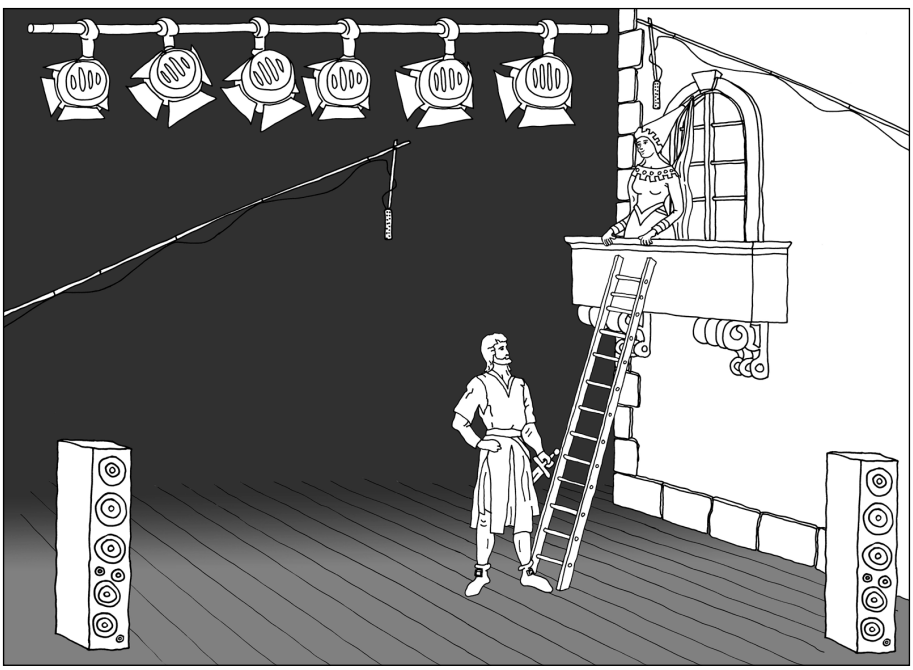
Часть III

Применение шаблонов архитектуры корпоративных программных приложений

Глава 8. Инфраструктура для сохраняемости

Глава 9. Приведение в действие преобразователя NHibernate

В этой части некоторые шаблоны из книги *Архитектура корпоративных программных приложений* Мартина Фаулера [Fowler PoEAA] переносятся в контекст дискуссии о том, что требуется от инфраструктуры для поддержки сохраняемости в модели предметной области. Выполнение подобных требований рассматривается на примере инструмента.



Эта часть посвящена инфраструктуре, которая очень ценна, хотя и отвлекает от предметной области

Глава 8

Инфраструктура для сохраняемости

Итак, чего же мы достигли? Мы построили сызнова модель предметной области в стиле ППО [Evans DDD]. До сих пор мы делали все возможное, чтобы отложить решения относительно инфраструктуры на как можно более дальний срок. И хотя в главе 6 было отведено немало места подготовке к упрощенной процедуре ввода инфраструктуры, тем не менее, теперь настало время ввести механизмы поддержки модели предметной области, т.е. саму инфраструктуру. Попутно мы внесем некоторые изменения в модель предметной области, чтобы выполнить список требований, сформулированных в главе 4.

Это нужно сделать, поскольку необходимо будет приложить максимум усилий для решения задач предметной области.

Но прежде чем переходить к рассмотрению инфраструктуры, нужно решить, *какая* именно инфраструктура требуется. На самом деле инфраструктура требуется немалая, в том числе для следующих целей.

- Проверка полномочий.
- Интеграция (запросы обслуживания и отклики на них).
- Управление данными и доступ к ним (сохраняемость).
- Представление (которое можно лишь с большой натяжкой отнести к инфраструктуре).
- Регистрация.

И так далее и тому подобное. Возможно, этот перечень в какой-то мере отражает мои былые пристрастия к базам данных, но мне кажется, что сохраняемость является наиболее очевидным и интересным свойством инфраструктуры. По крайней мере, именно ей мы и уделим внимание в этой главе.

Определяя инфраструктуру для сохраняемости, мы попутно внесем небольшие изменения в текущую конструкцию модели предметной области, чтобы она лучше соответствовала выбранному варианту инфраструктуры сохраняемости.

На заметку

Стиль проектирования *без* повторений таит в себе немало опасностей. Допустим, что вам поручено только построить модель предметной области и обеспечить сохраняемость. Но даже в этом случае вам придется сначала поработать немного над моделью предметной области, затем выбрать и опробовать инфраструктуру сохраняемости, вновь вернуться к модели предметной области, проверить инфраструктуру сохраняемости и т.д. Именно так и следует выполнять первые проекты в стиле ППО, поскольку в этом есть своя логика.

Следует также подчеркнуть, что я хорошо осознаю особую важность того же пользовательского интерфейса и что было бы не совсем корректно рассматривать его лишь как некий "механизм". Ясно, что я (и пользователи) придаю особое значение удобству использования такого интерфейса. Ведь нередки случаи, когда бо́льшая часть времени работы над проектом уделяется именно пользовательскому интерфейсу. Учитывая все это, мы сделаем все же основной акцент на модели предметной области, поскольку рассмотрение пользовательского интерфейса выходит за рамки этой книги (хотя некоторое внимание ему будет уделено в главе 11).

Итак, мы решили сосредоточиться, главным образом, на инфраструктуре сохраняемости. Каким же требованиям должна отвечать такая инфраструктура, чтобы дать нам возможность не отвлекаться от предметной области?

Требования к инфраструктуре сохраняемости

Следует еще раз подчеркнуть, что требуется отстранить инфраструктуру как можно дальше от модели предметной области, чтобы сосредоточиться на создании эффективной модели для решения конкретной прикладной задачи, отвлекаясь на инфраструктуру лишь по мере надобности. Для этого, на мой взгляд, нужно выполнять проектирование на очень высоком уровне НС, т.е. *неведения сохраняемости*, что и составляет основное требование к инфраструктуре.

В то же время очень важно не возлагать слишком большую ответственность на потребителя модели предметной области. Потребителю следует предоставить самый простой интерфейс API, чтобы программирующий потребитель мог сосредоточиться на том, что важнее всего для него, т.е. на создании удобного интерфейса для тех, кто будет пользоваться разрабатываемым приложением.

Некоторые желательные свойства жизненного цикла постоянных объектов-сущностей, определенные в главе 5, сведены в табл. 8.1.

Требования к инфраструктуре сохраняемости можно рассматривать и как нефункциональные. Типичными тому примерами служат масштабируемость, безопасность и удобство эксплуатации.

Таблица 8.1. *Сводка семантики жизненного цикла экземпляров объектов в модели предметной области*

Операция		Результат (временный/постоянный экземпляр объекта)
Вызов нового экземпляра объекта	→	Временный экземпляр объекта
`Repository.Add(`*`экземпляр`*`)` или `persistentInstance.Add(`*`экземпляр`*`)`	→	Постоянный экземпляр объекта в модели предметной области
`x.PersistAll()`	→	Постоянный экземпляр объекта в базе данных
`Repository.Get()`	→	Постоянный экземпляр объекта в модели предметной области (а также в базе данных)
`Repository.Delete(`*`экземпляр`*`)`	→	Временный экземпляр объекта (причем экземпляр удаляется из базы данных при вызове метода `x.PersistAll()`)

Разумеется, свойства инфраструктуры сильно разнятся не только из-за требований, но и потому что определенная базовая структура сохраняемости в одних случаях может удовлетворять, а в других случаях не удовлетворять одним и тем же нефункциональным требованиям, причем это может зависеть от характера самого приложения, применения шаблонов и прочих факторов. Я предоставляю читателю возможность самому определить нефункциональные и прочие требования к инфраструктуре сохраняемости, а затем проверить, насколько выбранная базовая структура сохраняемости удовлетворяет этим требованиям, тщательно выполнив тесты. Не следует только забывать о том, что чем более непомерными становятся нефункциональные требования, тем дороже это обходится в чем-то другом.

На заметку

Чаще всего от заказчика очень трудно добиться формулирования каких-либо нефункциональных требований. Если же вам удастся это сделать, то существует риск, что он потребует максимум возможного, и тогда самое время вспомнить, во что это может обойтись.

Подробнее о нефункциональных требованиях см. также в [POSA 1], [Fowler PoEAA] или [Nilsson NED].

Возможно, вы заметили, что я не упомянул в качестве требования наличие в базовой структуре сохраняемости сети между потребителем и моделью предметной области. Построить такую сеть вполне возможно, что мы и сделали с коллегой в проекте Valhalla [Valhalla]. Но все же лучше не выходить за пределы явных границ, за которыми начинаются сети, и предусмотреть возможное пересечение этих границ, опираясь на соответствующие службы. Именно поэтому я редко прибегаю к таким кратчайшим путям, как удаленный вызов.

Итак, мы определили требования с точки зрения модели предметной области, но не выяснили, как использовать интерфейс API для базовой структуры сохраняемости.

Теперь, когда известны требования к инфраструктуре, рассмотрим несколько вариантов ее выбора, и прежде всего, место для расположения сохраняемых данных.

Место для хранения данных

Если начинать с чистого листа, то следует выбрать средство и место для хранения данных. Для этого имеются, по крайней мере, четыре варианта выбора:

- оперативная память;
- файловая система;
- объектная база данных;
- реляционная база данных.

> ***На заметку***
>
> Я, конечно, понимаю, что эта задача превращается в дилемму: *что* хранить (объекты, иерархии, например, XML, или же таблицы) и *где* хранить (в оперативной памяти, файловой системе, объектной или реляционной базе данных). Выбрать и то и другое нелегко. Но по зрелом размышлении я пришел к выводу, что выбирать нужно все же место, а не то, что сохраняется, чем мы и займемся далее.

Итак, начнем с оперативной памяти.

Оперативная память

Каким бы нелепым это ни показалось на первый взгляд, но хранить данные в оперативной памяти совсем не обязательно. Я представляю себе модель предметной области как базу данных, но в то же время считаю, что все постоянные изменения следует сохранять в виде журнала регистрации на диске, чтобы восстановить модель предметной области после аварийного отказа системы.

Преимущество такого подхода заключается в том, что он позволяет избежать преобразования из модели предметной области в нечто другое. При этом модель предметной области сама оказывается сохраняемой.

В оперативной памяти можно было бы также хранить иерархии, типичным примером которых служат XML-документы. Но это привело бы к потере соответствия из-за того, что используются две модели и требуется преобразование из одной модели в другую. С другой стороны, автоматически появляются такие функциональные возможности, как формирование запросов, если для этой цели годятся языки типа XPath и XQuery.

Независимо от того, что именно хранится в оперативной памяти, проблема заключается в объеме оперативной памяти. Так, если на машине с оперативной памятью 2 Гбайт установлена система и приложения, объем базы данных не может превышать 2 Гбайт, иначе это непременно скажется на производительности.

С другой стороны, все большее распространение получают 64-разрядные серверы, а цены на оперативную память постоянно падают. С точки зрения возможного объема базы данных аппаратные средства нынешнего и будущего поколений вполне пригодны для большинства современных приложений.

Еще одна проблема состоит в том, что для восстановления модели предметной области из объемистого журнала регистрации после аварийного отказа системы требуется определенное время. Эту проблему можно свести к минимуму, если периодически сохранять на диске моментальные снимки текущей модели предметной области. Но само получение моментальных снимков может превратиться в проблему, поскольку оно приводит к резкому снижению производительности системы и к переводу других запросов в состояние ожидания на период получения таких снимков.

На заметку

Грегори Янг по этому поводу заметил, что в качестве решения проблемы получения моментальных снимков можно было бы разделить предметную область на контекстные границы и делать моментальные снимки по отдельности в пределах этих границ.

Но едва ли не самый большой недостаток данного подхода кроется в изменениях схемы. Для таких изменений приходится преобразовывать объект в последовательную форму при сохранении на диске, а затем обратно из последовательной формы в модель предметной области с новой схемой, а это довольно длительный процесс. Устранению этого недостатка способствует переход от чистых классов к XML.

Еще один крупный недостаток кроется в транзакциях. Соблюсти принцип *атомарности*, *непротиворечивости*, *изоляции* и *долговечности* (АНИД — Atomicity, Consistency, Isolation and Durability (ACID)) не так-то просто, не нарушая в значительной степени масштабируемость. В этом случае вместо того чтобы действовать наудачу, лучше подготовиться к транзакции как можно более тщательно, чтобы выяснить вероятность ее успешного выполнения. Разумеется, это не решает всю проблему в целом, но, по крайней мере, сводит ее к минимуму. С этой целью полезно действовать с упреждением, особенно принимая во внимание высокую эффективность из-за отсутствия заметных границ между процессами или сетями.

На заметку

Это, конечно, зависит от топологии. Если обращение к модели предметной области (которая в данном случае выполняет роль базы данных) происходит из другой машины, границы между процессами и сетями пересекаются.

Опять же, упреждение не предполагает никакой семантики транзакций, а лишь делает менее вероятным их откат. Для нормального выполнения транзакций может, в частности, потребоваться удвоенный объем оперативной памяти по сравнению с размерами модели предметной области, чтобы вносить изменения в копию этой модели. Если изменения внесены успешно, они повторяются в исходной модели предметной области. Между тем модель предметной области по своей сути является однопользовательской. И хотя операции в оперативной памяти обычно выполняются очень быстро, этого, на мой взгляд, явно недостаточно для обеспечения масштабируемости. Я также считаю это признаком незрелости и надеюсь на появление в дальнейшем более зрелых решений. На момент написания этой книги существовали на стадии разработки и внедрения упрощенные администраторы транзакций, которые могли бы стать удачным решением данной проблемы.

Другой проблемой является отсутствие очевидного выбора среди языков запросов. Для этого имеются открытые альтернативные варианты, но опять же на момент написания книги они оказались не вполне зрелыми. Данная проблема усложняется, если принять во внимание составление отчетов и потребность в формировании незапланированных запросов конечными пользователями помимо приложений. В этом случае для конечных пользователей оказываются совершенно бесполезными инструменты составления отчетов.

На заметку

Желательно, чтобы отчеты составлялись на специально выделенных для этой цели сервере и базе данных, и тогда осложнений окажется меньше, чем можно было бы предположить. Но в действительности, при выполнении повседневных транзакций границы между отчетами и списками нередко как бы размываются, в итоге осложнения появляются вновь.

И еще одна проблема связана с тем, что перемещение по модели предметной области, как правило, основывается на обходе списков, а встроенная поддержка индексирования может отсутствовать. Конечно, в этом случае можно использовать хэш-таблицы, но они решают данную проблему лишь частично. При желании можно также воспользоваться индексированием в оперативной памяти. Но с другой стороны, данная проблема возникает не так скоро, как в решениях, основанных на использовании дисков.

И наконец, как не раз уже отмечалось, я считаю такой подход недостаточно зрелым, хотя и очень интересным и многообещающим в перспективе — по крайней мере, он пригоден в некоторых ситуациях.

Файловая система

Другое решение состоит в использовании файловой системы вместо оперативной памяти. В этом случае в файлах, как и в оперативной памяти, сохраняются объекты из модели предметной области или документов XML. На самом деле данное решение может быть очень похожим на сохранение в оперативной памяти, если база данных невелика и выводится на диск, когда оперативная память заполняется до определенных границ.

Такому подходу присущи те же проблемы и недостатки, что и предыдущему, за исключением того, что объем оперативной памяти в данном случае не является ограничивающим фактором. Но с другой стороны, показатели производительности менее впечатляющи.

Конечно, было бы заманчиво разработать отдельное решение для сохранения модели предметной области (а также XML-документов), но, как всегда, при развертывании собственной инфраструктуры нужно быть готовым к большому объему работы. Ничто так не стимулирует разработчика, как задачи, которые другим кажутся слишком сложными и трудными. Если бы эти задачи были решены, каждый мог бы разработать собственное решение на основе файловой системы. Поначалу решение кажется обманчиво простым, но трудность кроется в деталях, причем его сложность повышается по мере продвижения к поставленной цели.

Если вы решите построить модель предметной области, которую можно выводить на диск во время сохранения, то на самом деле вы создаете нечто вроде объектной базы данных. (Возможно, это дает некоторое представление об объеме и сложности предстоящей работы.)

Объектная база данных

Исторически сложилось так, что существовали самые разные виды объектных баз данных, но общей для них была попытка избежать преобразования объектов в какой-нибудь другой формат. Это делалось, в общем, по тем же самым причинам, о которых упоминалось выше в связи с откладыванием ввода инфраструктуры в модель предметной области на дальний строк, а также из соображений производительности.

На заметку

Разновидностей объектных баз данных станет еще больше, если отнести к их числу и гибридные базы данных, в том числе объектно-реляционные, хотя гибридные базы данных имеют в большей степени реляционную, чем объектно-ориентированную природу.

Как оказалось, отвлечения от модели на инфраструктуру все же имелись. В действительности, потеря соответствия по-прежнему имеет место, но по сравнению с наведением мостов между объектами и реляционной базой данных применение объектной базы данных оказалось довольно простым и ясным.

До сих пор объектным базам данных были присущи следующие недостатки.

- отсутствие стандартов;
- отсутствие критической массы;
- недостаточная зрелость;
- наличие большого числа узкоспециализированных продуктов;
- трудности интеграции с другими системами;
- сложности составления отчетов.

На заметку

Мой коллега Мартин Розен-Лидхольм заметил, что многие из перечисленных выше аргументов можно выдвинуть против ППО и объектно-реляционного преобразования по сравнению с информационно-ориентированными решениями в среде .NET, принимая во внимание прикладные блоки, документацию, наставления и прочие вспомогательные средства, предоставляемые компанией Microsoft.

Будем все же надеяться, что в наставлениях компании Microsoft все чаще будет уделяться должное внимание предметной области, с чем Мартин полностью согласен.

Я, конечно, не специалист по объектным базам данных, хотя и пробовал работать с двумя их разновидностями. Подробнее об этом можно прочитать в книгах [Cattell ODM] и [Connolly/Begg DB Systems].

Когда-то, а точнее в 1994 году, я считал, что объектные базы данных станут фактическим стандартом. Но я основывал свои ожидания исключительно на технических характеристиках, тогда как жизнь показала, что не все так просто. Более десяти лет назад объектные базы данных действительно выглядели многообещающе. И во многих случаях они применяются в настоящее время, хотя по-прежнему больше обещают, чем предлагают. На мой взгляд, фактическим стандартом теперь стали реляционные базы данных.

Реляционные базы данных

Как уже отмечалось, фактическим стандартным решением для хранения данных в приложениях является реляционная база данных — и для работы с моделью предметной области в том числе.

Хранение данных в реляционной базе данных, по существу, означает, что они сохраняются в табличном формате, где все является информацией, включая и отношения. Такое решение оказалось простым и достаточно эффективным во многих приложениях. Но, как известно, у любого решения имеются свои недостатки и проблемы, и в данном случае они связаны с потерей соответствия при сохранении модели предметной области в реляционной базе данных. (Этот вопрос мы уже обсуждали достаточно подробно в главе 1, и поэтому не будем повторяться.)

Если пойти по этому пути, то наиболее распространенным решением будет реализация шаблона Data Mapper (Преобразователь данных) [Fowler PoEAA]. Его назначение состоит в наведении мостов между моделью предметной области и ее сохраняемым представлением с целью перемещения данных в обоих направлениях. (Мы еще вернемся к этому шаблону.)

Выбор подходящего решения для хранения данных не всегда очевиден, Тем не менее, его приходится делать. Но прежде чем сделать такой выбор и двигаться дальше, нам предстоит рассмотреть ряд других вопросов.

Один или несколько администраторов ресурсов

Возникает еще один вопрос совершенно другого свойства: сколько администраторов ресурсов использовать: один или несколько? Иногда оказывается, что выбора просто нет.

Для модели предметной области характерно наличие нескольких администраторов ресурсов, поскольку она позволяет скрыть при желании все эти сложности от потребителей. Следует, однако, иметь в виду, что наличие нескольких администраторов ресурсов усложняет преобразование модели предметной области в состояние ее сохраняемости. Поэтому ради простоты допустим, что имеется только один администратор ресурсов.

Другие факторы

В действительности начинать с чистого листа приходится очень редко. На принимаемое решение оказывают также влияние такие факторы, как наши знания и опыт. Для того чтобы работать эффективно, полезно выбрать уже знакомую технологию.

Помимо факторов, определяющих исходную технологию, значение имеют и такие менее очевидные факторы, как системы, приобретенные заказчиком и освоенные его персоналом.

Что касается сохранения данных, то важным фактором является зрелость выбираемого решения. Ведь потеря данных может иметь серьезные последствия для бизнес-процессов и зачастую просто недопустима, поэтому заказчики нередко очень придирчиво рассматривают выбранное решение, если оно кажется им не вполне подходящим.

Выбор решения и переход к его воплощению

Принимая во внимание технические основания и прочие упомянутые выше факторы, вполне естественно выбрать реляционную базу данных. Поэтому остановим свой выбор именно на ней и будем двигаться дальше. Такой выбор кажется подобающим и, вероятно, еще долго будет делаться по умолчанию.

В связи с этим мне бы хотелось внести следующие дополнения в список требований к инфраструктуре сохраняемости.

- Аккуратное обращение с реляционной базой данных. (Это означает, что нужно стремиться в большей степени к программированию обращений к ней вручную.)
- Основательная поддержка формирования запросов. (Это в значительной степени связано с предыдущим требованием.)
- Поддержка выявления конфликтов параллельной обработки.
- Поддержка усовершенствованного преобразования, включая различные методики наследования (даже если "наследование" аккуратно используется в базе данных), а также мелкоструктурных типов данных в модели предметной области.

Следующим шагом должна стать реализация шаблона Data Mapper. Вопрос лишь в том, *как* реализовать этот шаблон.

Подход

Обращаться с шаблоном Data Mapper можно по-разному, наиболее распространенные подходы заключаются в следующем.

- Написание вручную заказного кода.
- Генерация заказного кода.
- Преобразование метаданных (объектно-реляционное преобразование).

Итак, начнем рассмотрение перечисленных выше подходов с написания вручную заказного кода.

Написание вручную заказного кода

В этом случае пишется (обычно самостоятельно) код сохранения. Такой код постоянно находится в хранилищах. Разумеется, для его написания потребуются вспомогательные классы, хотя это и не решит все проблемы. К некоторым техническим проблемам в данном случае относятся следующие.

- Как сохранить изменения в полном графе?
- Требуется ли шаблон Unit of Work?

- Требуется ли шаблон Identity Map?
- Требуется ли шаблон Lazy Load?

На заметку

Следует дать краткие пояснения к каждому из перечисленных выше шаблонов. Так, шаблон Unit of Work [Fowler PoEAA] служит для фиксации информации об изменениях, вносимых в модель предметной области во время выполнения логической единицы работы. Эту информацию можно затем использовать для воздействия на сохраняемое представление.

Шаблон Identity Map [Fowler PoEAA] служит для хранения не более чем одного экземпляра каждого объекта-сущности в течение сеанса работы. Он действует подобно кэшу с идентификацией. Данный шаблон очень важен для устранения потери соответствия. Во время работы с объектами требуется использовать встроенную идентичность объекта, чтобы обращаться к объекту.

И наконец, шаблон Lazy Load [Fowler PoEAA] служит для загрузки подграфов по требованию.

Шаблоны Unit of Work и Identity Map нужны, поскольку это отвечает сформулированным выше требованиям и условиям формирования запросов. А шаблон Lazy Load *может* и не потребоваться. Его функции можно реализовать и самостоятельно, если иногда в них возникает потребность. Впрочем, было бы неплохо иметь такую поддержку на тот случай, если она понадобится.

Если придерживаться определенных выше требований, то окажется, что написание вручную заказного кода, по существу, означает построение отдельного и, как можно надеяться, упрощенного объектно-реляционного преобразователя, т.е. переход от данной задачи непосредственно к третьей задаче, связанной с преобразованием метаданных. Поэтому не будем пока что обращать на это внимания и просто допустим, что мы могли бы вполне обойтись без шаблонов Unit of Work, Identity Map и Lazy Load, динамического и гибкого формирования запросов и прочего.

Прежде всего нам придется подробно выяснить, как все должно работать. А после того, как вся эта работа будет сделана, применить ее результаты ко всей модели предметной области. Это довольно трудоемкая и продолжительная работа, не исключающая ошибки. Более того, ее придется вновь выполнять при каждом последующем изменении модели предметной области.

Генерация заказного кода

Еще один подход заключается в генерации кода для работы с шаблоном Data Mapper (для этой цели может быть специально создан типичный инструмент с написанными на заказ шаблонами или же приобретено готовое решение, если оно окажется

подходящим). Имеется в виду решение, аналогичное написанию заказного кода, но после того, как проект проработан, оно используется неоднократно благодаря генерации кода.

Такому подходу присущи те же недостатки, что и написанию заказного кода вручную, не говоря уже о трудностях самого генерирования кода. Но главное его преимущество состоит в повышении производительности труда, как только сам проект становится зрелым.

Одна из проблем генерации кода заключается управлении исходным кодом. Ведь даже незначительное изменение в модели предметной области обычно вынуждает проверять все классы, связанные с преобразованием данных, а затем повторно генерировать их код, если используется специальный инструмент для управления исходным кодом, требующий такой проверки перед внесением изменений. Конечно, бывают ситуации и похуже, тем не менее, это не очень удобно.

Кроме того, в базу кода приходится вводить много неинтересного кода, который будет читаться не человеком, а только компилятором.

Нередко сгенерированный код тесно связан с конкретным продуктом СУБД. И хотя это затруднение может быть разрешено на уровне абстракции, оно по-прежнему остается, если требуется поддержка разных *схем* баз данных. В таком случае приходится хранить код, отдельно сгенерированный для каждой схемы базы данных. Это очень важно для разных продуктов СУБД, хотя такое решение далеко от идеала.

Еще одна проблема состоит в ограниченных возможностях настройки во время выполнения. Для сравнения оптимизатор реляционной базы данных принимает только следующие решения: что сохранять статически, как это делать во время выполнения, а следовательно, как внести изменения в зависимости от конкретной ситуации. Впрочем, для большинства современных приложений это не такая уж и большая проблема. Более того, если сравнить конкретный сценарий с аналогичными подходами к получению сгенерированного и рефлексивного кода, то добиться лучшей производительности зачастую можно, скорее, от статического, чем от рефлексивного кода.

Вероятно, больше хлопот доставляют новые версии генераторов кода. Ведь не исключено, что новую версию такого генератора нельзя будет ввести в эксплуатацию без повторной генерации кода.

Кроме того, при изменении схемы базы данных внести изменения во время выполнения без перекомпиляции кода не удастся. Впрочем, делать это рекомендуется очень редко, поскольку для подобных изменений потребуются разработчики.

Код может генерироваться циклически или напрямую. Если он генерируется циклически, то изменения в сгенерированном коде сохраняются, а если код генерируется напрямую, такие изменения не сохраняются и могут быть вообще утрачены при последующей генерации.

И в заключение отметим следующее преимущество данного подхода. Генерация кода упрощает его отладку по сравнению преобразованием метаданных, хотя такой код трудно понять, особенно если его много.

Что же касается формирования запросов, то такому подходу недостает динамичности. Если же требуется реализовать динамическое формирование запросов, шаблоны Unit of Work и Identity Map и прочее, то, на мой взгляд, следует рассмотреть следующий подход: объектно-реляционное преобразование.

Преобразование метаданных (объектно-реляционное преобразование)

К особой разновидности шаблона Data Mapper относится шаблон, называемый Metadata Mapping (Преобразование метаданных). С его помощью в метаданных определяется взаимосвязь модели предметной области с реляционной базой данных, а остальное делается автоматически.

Наиболее характерным примером реализации такого шаблона, вероятно, являются объектно-реляционные преобразователи. Здесь и далее под термином "объектно-реляционный преобразователь" понимается семейство программных продуктов, выполняющее преобразование метаданных.

На заметку

Мой коллега Матс Хеландер сравнил объектно-реляционное преобразование с гимнастикой Тайцзи.

Тайцзи делится на две части. В первой части нужно научиться медленно поднимать и опускать руки согласованно с дыханием, а во второй части — научиться всему остальному. И это не шутка. Как бы хорошо вы ни освоили Тайцзи, вам следует и даже рекомендуется и далее уделять первому движению столько же времени, сколько и всем остальным вместе взятым.

Объектно-реляционное преобразование также делится на две части. Первая часть состоит в том, чтобы перемещать данные из объектов, находящихся в оперативной памяти, в строки таблицы в базе данных, и обратно. А все остальное относится ко второй части.

Возможно, вы уже догадались, что в большинство объектно-реляционных преобразователей встроена поддержка шаблонов Unit of Work, Identity Map и Lazy Load, а также формирования запросов.

Но и такое решение не лишено недостатков. Главная претензия к объектно-реляционным преобразователям состоит в том, что они неспособны формировать качественный код SQL. Рассмотрим в связи с этим ряд характерных примеров. В первом примере используется оператор `UPDATE` с предложением `WHERE` следующим образом:

```
UPDATE Inventory
SET Balance = Balance - 1
WHERE Id = 42 AND Balance >= 1
```

Это означает, что если товары находятся на складе, то требуется лишь изменить их остаток. В противном случае изменять остаток не нужно. Такая операция обычно не поддерживается непосредственно в объектно-реляционных преобразователях. Вместо этого в них выполняется чтение строки `Inventory` (Склад) с оптимистической блокировкой, вносится изменение и результат записывается обратно в строку (в надежде на то, что не появится исключение в связи с конфликтом параллельной обработки).

А теперь рассмотрим для полной ясности пример оптимистичного подхода (на этот раз в виде групповой операции SQL, но без каких-либо блокировок, поскольку отсутствует явная транзакция, что упрощает дело):

```
-- Запомнить старый остаток.
SET @oldBalance =
  (SELECT Balance
  FROM Inventory
  WHERE Id = 42)

-- Рассчитать новый остаток...

UPDATE Inventory
SET Balance = @newBalance
WHERE Id = 42 AND Balance = @oldBalance
-- Если значение @@ROWCOUNT теперь равно 0, значит,
-- обновление завершилось неудачно!
```

С другой стороны, в объектно-реляционном преобразователе выполняется чтение строки `Inventory` с пессимистической блокировкой, вносится изменение и результат записывается обратно в строку. Но и при том и другом подходе страдает масштабируемость.

Еще одна претензия к объектно-реляционным преобразователям состоит в том, что они зачастую неэффективно обновляют большое количество строк таблицы. В частности, для обновления всех товаров посредством объектно-реляционного преобразователя, скорее всего, придется считать все товары в список, циклически просмотреть весь этот список и обновить товары по очереди. Если бы это можно было сделать с помощью одного оператора `UPDATE`, производительность значительно улучшилась бы.

И еще одна претензия состоит в том, что очень трудно найти оптимальное количество операций чтения. Их оказывается либо слишком мало, и тогда возрастает число циклических обращений к базе данных при загрузке по требованию, либо больше, чем нужно. Как правило, это приводит к разрастанию типов, при котором появляется несколько разновидностей определений типов данных.

Данный вопрос можно рассматривать и под другим углом зрения. Можно ли написать весь код исключительно хорошо и в то же время согласованно? И все ли разработчики в группе могут делать это так же хорошо, как и самый лучший из них? Но

даже *если* такое возможно, следует ли тратить на это драгоценное рабочее время, если можно добиться приемлемой производительности, подобрав нужный подход к автоматическому формированию кода?

На заметку

Откровенно говоря, такой подход справедлив и для генерации кода.

Итак, мы рассмотрели некоторые достоинства и недостатки трех подходов. Мне это по душе уже потому, что хотя бы отчасти я понимаю, о какой технологии вообще идет речь.

Повторный выбор

Конечно, сделать такой выбор непросто, но мой опыт показывает, что подход, в наибольшей степени отвечающий поставленным нами требованиям, заключается в объектно-реляционном преобразовании. Это подобающий подход для многих ситуаций, но и он не лишен недостатков.

Если проектировать в стиле YAGNI ("Это вам не понадобится"), то целесообразно использовать объектно-реляционные преобразователи, поскольку в этом случае мы можем решить задачу просто (и, будем надеяться, не сильно отвлекаясь от модели предметной области) и тем самым упрочим перспективы на будущее. Если же производительность окажется недостаточной, то можно обратиться к написанию заказного кода. Это очень эффективное решение (хотя бы в отношении производительности), но вряд ли оно потребуется в данном случае.

Итак, допустим, что мы выбрали объектно-реляционное преобразование.

Классификация

Рассмотрим подробнее, что же собой представляет объектно-реляционный преобразователь, с двух сторон: сначала его различные характеристики, а затем его применение с точки зрения реализации некоторых шаблонов архитектуры приложений для предприятия. Такой классификацией объектно-реляционных преобразователей мы воспользуемся в следующей главе, рассматривая примеры того, как все это воплощается в конкретном объектно-реляционном преобразователе.

Откровенно говоря, все, о чем пойдет речь ниже, включая и описание шаблонов, относится и к заказному коду, кроме вида преобразования. Но ради большей конкретности мы не будем отвлекаться от самого понятия объектно-реляционных преобразователей.

Первым в классификации стоит вид поддерживаемой модели предметной области.

Вид модели предметной области

В какой степени и каким образом нужно приспосабливать модель предметной области к объектно-реляционному преобразователю? Этот вопрос обычно распадается на три следующих общих аспекта:

- неведение сохраняемости;
- наследование;
- реализация интерфейса.

Неведение сохраняемости означает, что для сохраняемости модели предметной области в нее вообще не вносятся изменения. Конечно, у этого аспекта имеется немало нюансов в промежутке между полярными решениями. В частности, подходы, основанные на рефлексии, предъявляют одни требования, а подходы АОП — совсем другие требования.

Наследование

Традиционно сложившийся общий подход состоит в том, чтобы классы из модели предметной области наследования непременно наследовали от суперкласса, предоставляемого базовой структурой сохраняемости.

Реализация интерфейса

Другой традиционно сложившийся общий подход состоит в том, чтобы классы из модели предметной области реализовывали один или несколько интерфейсов, предоставляемых базовой структурой сохраняемости.

Один лишь этот аспект показывает, как можно приспособить модель предметной области к базовой структуре сохраняемости. Кроме того, предстоит немало сделать в отношении управления версиями и загрузки по требованию, но мы отложим обсуждение данной темы до следующей главы.

Вид объектно-реляционного преобразователя

Преобразование на основе метаданных зачастую реализуется двумя разными способами.

- Генерация кода.
- Структурно-рефлексивный метод.

Внимательный читатель, возможно, заметил, что генерация кода упоминается здесь вновь — на сей раз в контексте объектно-реляционных преобразователей. Но в отличие от рассмотренного выше, генерация кода в данном случае выполняется, как правило, непосредственно перед компиляцией путем чтения метаданных и выдачи кода преобразования. Конечно, с таким же успехом можно было бы сгенерировать заказной код, но главное *отличие* заключается в поддержке, кото-

рая для этого требуется. Впрочем, это лишь начальное и пока еще не совсем четкое определение.

Структурно-рефлексивный метод означает, что для преобразования генерация исходного кода не требуется. Вместо этого считываются метаданные во время преобразования.

На заметку

Нетрудно догадаться, что дело обстоит не так просто. В частности, к какой категории следует отнести статическое АОП? Я стараюсь придерживаться ясного и простого описания и разделения на категории.

Следует также заметить, что генерация кода и структурно-рефлексивный метод совсем не исключают друг друга.

Отправная точка

Для работы с определенным объектно-реляционным преобразователем зачастую выбирается одна или несколько следующих отправных точек.

- Схема базы данных.
- Модель предметной области.
- Данные преобразования.

Под отправной точкой подразумевается конкретный аспект, на котором сосредоточивается основное внимание, когда начинается работа над приложением. Конечно, было бы неплохо, если бы объектно-реляционный преобразователь допускал несколько отправных точек, поскольку в этом случае появляется больше возможностей для его применения в самых разных ситуациях, возникающих во время работы над проектом.

Я лично предпочитаю начинать работу с модели предметной области, но такая возможность имеется не всегда, и поэтому начинать иногда приходится со схемы базы данных или, по крайней мере, не упускать ее из виду. Бывает и так, что начинать приходится с данных преобразования, описывая взаимосвязи классов с таблицами, например, с помощью интерфейса построения UML-схем.

Если вы предпочитаете начинать с модели предметной области, но базовая структура сохраняемости, которую вам требуется опробовать, не дает такой возможности, то в качестве выхода из данного положения следует рассмотреть редактирование UML-схемы или проектирование базы данных как способ *построения* модели предметной области. Естественно, что в этом случае о применении РПТ уже не может быть и речи.

Преобразование для устаревших баз данных

Даже если в качестве отправной точки выбрана база данных, это совсем не означает, что такой выбор обусловлен базовой структурой сохраняемости или желанием разработчика работать именно так. Иногда такой выбор объясняется необходимостью использовать устаревшую базу данных.

В таком случае нужно быть готовым к более трудному пути, который придется пройти вместе с объектно-реляционным преобразователем (при условии, что он не приспособлен специально для этой цели), особенно если изменения в структуре базы данных не разрешаются. И хотя в устаревшей базе данных зачастую не разрешается изменять уже существующие столбцы, вводить в нее представления, столбцы и таблицы, тем не менее, можно. А это уже какая-то польза! (Изменения, не оказывающие влияния на существующие приложения, вполне допустимы, но следует иметь в виду, что такие приложения могут оказаться далеко не самыми надежными.)

Особые трудности возникают в том случае, если структура устаревшей базы данных находится не в лучшем состоянии, что может произойти после двух и более лет эксплуатации приложения.

Попутно возникает следующий вопрос: в каком направлении двигаться дальше после перехода от первой отправной точки, например, модели предметной области? Позволяет ли выбранный объектно-реляционный преобразователь в какой-то степени упростить переход к двум другим частям проекта? Так, если имеется модель предметной области, то можно ли автоматически получить базу данных и метаданные? Разумеется, отсутствие такой поддержки не может служить препятствием, ведь соответствующие основные инструменты можно без особого труда создать и самостоятельно. Однако было бы лучше, если бы это делалось автоматически.

Итак, допустим, что модель предметной области, база данных и данные преобразования уже находятся на своих местах. Перейдем к рассмотрению интерфейса API.

Акцент на интерфейсе API

Интерфейс API может быть представлен в двух разных вариантах.

- Реляционные таблицы.
- Модель предметной области.

Первый вариант можно лишь с большой натяжкой отнести к рассматриваемой теме, поскольку исходное назначение объектно-реляционных преобразователей — предоставить разработчикам возможность работать с объектами, а не с таблицами. Следовательно, для типичных объектно-реляционных преобразователей акцент на интерфейсе API всегда означает модель предметной области. (Примером интерфейса API, ориентированного в большей степени на реляционные таблицы, может слу-

жить реализация шаблона Recordset (Набор записей) с помощью свойства класса `DataTable` в .NET.)

Еще одним важным аспектом интерфейса API, который совсем не одинаков для всех объектно-реляционных преобразователей, является формирование запросов.

Тип языка запросов

Выделить здесь какие-то категории довольно трудно, но мы все же попытаемся это сделать. С точки зрения разработчика формирование запросов с использованием объектно-реляционного преобразователя, как правило, осуществляется одним из следующих способов.

- Построчно в стиле SQL.
- На основании объектов запроса.

Поясним оба способа более подробно.

Построчное формирование запросов в стиле SQL

Под этим подразумевается использование языков запросов, аналогичных SQL, но работающих с классами из модели предметной области вместо таблиц из базы данных. Некоторые запросы (особенно развитые) очень хорошо выражаются на таком языке. С другой стороны, ему явно недостает типовой безопасности.

Следует, однако, заметить, что формирование запросов построчно имеет значение только на уровне интерфейса API, но необязательно на уровне реализации.

Формирование запросов на основании объектов запроса

Для представления запросов в обычном языке запросов данного типа используются объекты в соответствии с шаблоном Query Object [Fowler PoEAA]. Подобным образом зачастую лучше выражаются простые запросы. При этом вполне возможно наличие типовой безопасности, а разработчику не обязательно знать семантику формирования запросов, чтобы работать эффективно. Но дело в том, что для выражения сложных запросов может потребоваться немало кода, который быстро становится неудобочитаемым.

Чистый SQL

Еще один привлекательный (хотя и пригодный в редких случаях) способ состоит в том, чтобы выразить запрос непосредственно в SQL, но получить результат в виде экземпляров модели предметной области. Конечно, это приводит к осложнениям, связанным с привязкой кода к базе данных, но если другого выхода нет, то можно, по крайней мере, воспользоваться и таким способом. Он очень удобен для оптимизации. Если же в качестве результата требуется получить объекты-сущности, а интеграция SQL позволяет сделать это автоматически, то в конечном итоге сокращается объем кода, который приходится писать вручную.

Разумеется, в этом случае должна быть предусмотрена возможность непосредственного перехода к SQL и без получения результатов в виде экземпляров модели предметной области.

Какой способ лучше

На мой взгляд, идеальной была бы поддержка нескольких способов, поскольку в одних случаях оказываются более подходящими построчные выражения, а в других — выражения на основе объектов запроса.

Сам *способ* формирования запросов — это лишь часть дела. А другая, не менее важная часть — выбрать *наиболее подходящее* решение для формирования запросов. Об этом и пойдет речь далее.

Улучшенная поддержка базы данных

Для того чтобы объектно-реляционный преобразователь действовал достаточно гибко и не вынуждал разработчика выходить за пределы предметной области, он должен поддерживать некоторые операции, непосредственно связанные с базой данных и не вполне очевидные с точки зрения модели предметной области, хотя и вполне подходящие для оптимизации.

Ниже приведены некоторые примеры таких операций.

- **Агрегирование**
  Одним из основных свойств SQL является возможность выполнять такие операции, как `SELECT SUM(x),MIN(y),MAX(z)`. При проектировании с акцентом на модель предметной области подобные операции агрегирования требуются не так часто, тем не менее, они очень полезны, а иногда и просто необходимы.
- **Упорядочение**
  Упорядочение результирующего множества в базе данных без его упорядочения в модели предметной области может оказаться самым эффективным способом.
- **Группирование**
  Во время работы с SQL иногда приходится обращаться к операции `GROUP BY`. Особенно часто такая операция группирования требуется для формирования незапланированных запросов и составления отчетов, хотя редко оказывается полезной в приложениях, созданных на основе модели предметной области.
- **Формирование скалярных запросов**
  Получение полных экземпляров объектов в ответ на запросы требуется далеко не всегда. Иногда оказывается достаточно и значений одних полей из объекта какого-нибудь одного класса и значений других полей из объектов какого-нибудь другого класса.

Было бы неплохо найти и другие способы перенести в базу данных такие функции, которые в большей степени отвечают ее назначению, включая типичные опера-

ции интенсивной обработки данных. Под этим обычно подразумевается возможность вызова хранимых процедур. Но кроме того, в базу данных можно было бы перенести и функции, определяемые пользователем.

Главная проблема переноса функций в базу данных заключается в том, что при этом может быть утеряна переносимость приложения и структуры базы данных. Но если пользоваться этим методом для оптимизации и только тогда, когда это действительно необходимо, данная проблема может быть сведена к минимуму.

Кроме того, выходя за пределы предметной области, мы остаемся предоставленными самим себе, но это еще не все. Мы должны также позаботиться об интеграции с моделью предметной области. Ведь вполне возможно, что нам придется очищать карту идентичности после каждого вызова хранимой процедуры. А этого никто за нас не сделает, кроме нас самих.

На заметку

Характерный тому пример приведен в статье *The Law of Leaky Abstractions* Джоэла Спольски [Spolsky Leaky Abstractions]. Абстракции сами по себе хороши, но ведь нужно еще разбираться и во внутреннем механизме работы отдельных компонентов системы.

Именно с такой позиции и следует подходить к объектно-реляционным преобразователям. Не следует ожидать, что в них удастся скрыть все, что выходит за пределы абстракции. Напротив, нужно знать, что происходит за ее пределами. Объектно-реляционные преобразователи следует рассматривать в качестве инструмента, помогающего решать трудоемкую задачу, которую *можно* было бы теоретически решить вручную, что на практике делать было бы нежелательно.

Другие функциональные возможности

Помимо уже перечисленных, ниже приведены другие функциональные возможности объектно-реляционных преобразователей.

- **Поддержка конкретных СУБД**

  Было бы совсем неплохо, если бы конкретный объектно-реляционный преобразователь поддерживал целый ряд различных СУБД и не был привязан только к одной из них, например, к Oracle или SQL Server.

- **Не только функции объектно-реляционного преобразователя**

  Помимо функций объектно-реляционного преобразователя, многие программные продукты выполняют и другие функции. Например, они могут оказывать помощь в привязке данных (т.е. в привязке элементов управления пользовательского интерфейса к объектам без заказного кода), определении бизнес-правил или же в организации двунаправленности.

- **Усовершенствованное кэширование**

  Усовершенствованное кэширование не имеет особого значения для разработчика, но на тот случай, если оно все же потребуется, было бы неплохо иметь его поддержку в выбранном программном продукте. Под усовершенствованным кэшированием подразумевается не только кэш в виде карты идентичности, но и кэш, используемый для формирования запросов. Это означает, что если, например, запрашиваются определенные клиенты, то такой запрос может направляться на обработку непосредственно в кэш второго уровня вместо базы данных.

- **Улучшенная поддержка контроля транзакций**

  К примерам такой функции можно отнести возможность контролировать транзакции по уровням изоляции, снижать риск взаимных блокировок, исключая нарастание блокировок при чтении и записи, а также поддерживать распределенные транзакции. Конечно, это несопоставимые возможности, но они могут оказаться очень важными.

  Не менее важен и метод, используемый для контроля транзакций. Как правило, такой контроль осуществляется вручную, на основании перехвата или же полностью скрывается в администраторе сохраняемости.

- **Открытость исходного текста**

  Открытость исходного текста имеет свои достоинства и недостатки по сравнению с коммерческими программными продуктами, хотя эта тема выходит за рамки данной книги. Тем не менее, данный фактор следует принимать во внимание при выборе объектно-реляционного преобразователя.

- **Версия (поколение)**

  И последняя важная функция имеет отношение к поколению, к которому принадлежит объектно-реляционный преобразователь. Это может свидетельствовать о степени его зрелости.

Не следует также забывать и том, что лучше меньше, да лучше. Так, чем шире назначение конкретного программного продукта, тем меньше он сосредоточен на деталях, хотя это и не обязательно. Это лишний раз подчеркивает, насколько трудно выбрать подходящий программный продукт, исходя только из перечня его возможностей.

А теперь рассмотрим классификацию под совершенно иным углом зрения.

Классификация по шаблонам инфраструктуры

В данном случае классификация будет произведена по шаблонам архитектуры приложений для предприятия [Fowler PoEAA], причем некоторые из них имеют непосредственное отношение к инфраструктуре.

Опять же, данную классификацию можно использовать и в том случае, если остановить свой выбор на заказном коде. Кто знает, возможно, она поможет кому-нибудь

из читателей избежать заказных решений и вместо этого воспользоваться одним из готовых решений. Обычно это имеет прямой смысл, поскольку разработка собственного полноценного решения требует немалого труда. Мы это уже проходили.

Тип метаданных для преобразования

В метаданных требуется описать взаимосвязь между моделью предметной области и схемой базы данных, а для этого существует шаблон Metadata Mapping [Fowler PoEAA]. Объектно-реляционные преобразователи являются реализациями шаблона Metadata Mapping, хотя в разных преобразователях используются разные типы метаданных. Ниже приведены типичные тому примеры:

- Документы XML или других форматов.
- Атрибуты (точнее говоря, аннотации, например, `[MyAttribute]` в C#).
- Исходный код.

Как обычно, у каждого такого типа метаданных имеются свои каверзные особенности. Например, XML-документы страдают недостатками формата XML. Часто говорят, что XML существует не для людей, а для синтаксических анализаторов. В то же время имеющиеся инструменты не соответствуют возможностям данного формата, и поэтому крупные XML-документы нередко приходится править вручную.

Еще одно затруднение, связанное с XML-документами, заключается в том, что они являются внешними по отношению к исходному коду модели предметной области, а следовательно, оба компонента могут оказаться несинхронизированными друг с другом. Многим интегрированным средам разработки недостает полного представления о семантике XML-документов, и поэтому рефакторинг кода не получается таким гладким.

Атрибуты служат для оформления модели, поэтому риск выйти из синхронизма с моделью предметной области в данном случае оказывается меньше. Другое дело, что модель предметной области оказывается при этом чуть более тесно связанной с базой данных. Так, если модель предметной области предназначается для использования вместе с двумя разными базами данных, то вполне возможно появление двух разных версий исходного кода для модели предметной области, если выбрать атрибуты вместо какого-то внешнего типа метаданных. Вероятно, это не такая уж и большая проблема, тем не менее она существует. Кроме того, в данном случае намного труднее получить общее представление о преобразовании, хотя на помощь можно призвать соответствующие инструменты.

Можно, конечно, спорить, является ли предоставление информации о преобразовании в исходном коде отдельной категорией. На самом деле это не более чем очередной формат документа. Так или иначе, наличие только двух категорий было бы некоторым упрощением. Дело в том, что в исходном коде информация о преобразовании описывается процедурным, а не декларативным способом. В этом состоит отличие между XML-документами и атрибутами. Метаданные находятся в исходном коде и

поэтому компилируются, но они записаны так, чтобы давать при их выборке общее представление о преобразовании. Для кого-то это может означать "изящный код C# вместо скверного кода XML". С этим можно отчасти согласиться. Тем не менее, этот вариант используется крайне редко.

И как обычно, это не единственно возможный вариант. Например, информация может предоставляться в виде атрибутов, но заменяться информацией в формате XML.

Что же следует описывать в метаданных? Взаимосвязь модели предметной области с базой данных, но для большей конкретности рассмотрим пример, воспользовавшись полями идентификации.

Поля идентификации

В поле идентификации объекта-сущности, представленном шаблоном Identity Field [Fowler PoEAA], содержится значение первичного ключа базовой таблицы из базы данных. Именно так интерпретируется отношение между экземпляром объекта-сущности и строкой таблицы.

Когда дело доходит до новых объектов-сущностей, то значения в полях идентификации могут формироваться, по крайней мере, на четырех разных уровнях:

- потребителя;
- модели предметной области;
- базы данных;
- объектно-реляционного преобразователя.

Два первых уровня аналогичны с точки зрения объектно-реляционного преобразователя, который не контролирует предоставляемое значение и может рассчитывать лишь на то, что протокол будет соблюдаться на уровне потребителя или модели предметной области. Если бы у меня был выбор, я бы предпочел уровень модели предметной области уровню потребителя. Но дело в том, что в объектно-реляционных преобразователях значение из поля идентификации используется для того, чтобы выяснить, является ли экземпляр объекта новым и следует ли его ввести или же обновить.

На заметку

Это характерный пример того, как на один компонент возлагаются две ответственности. На первый взгляд, все ясно и просто, но осложнения поджидают за углом.

На уровне базы данных все происходит достаточно просто, хотя и не без некоторых семантических затруднений. Это означает, что для установки значения первичного

ключа при сохранении объекта-сущности в базе данных используется свойство, аналогичное `IDENTITY` (в SQL Server) или Sequence (в Oracle). Однако это делается на слишком поздней стадии жизненного цикла, и поэтому возможны осложнения в тех множествах, в которые объект-сущность вводится при изменении значения идентичности.

И наконец, объектно-реляционный преобразователь может самостоятельно формировать значения в полях идентификации, что является самым удобным решением — по крайней мере, для объектно-реляционного преобразователя.

Итак, все сказанное выше сводится к тому, что для каждого объекта-сущности целесообразно иметь два вида идентичности. Одна из них является естественной идентичностью (или так называемым идентификатором предметной области) и получает свое значение в разные моменты жизненного цикла — совсем не обязательно с самого начала, хотя такая идентичность присуща не всем объектам-сущностям. Для классов также характерна не единственная идентификация такого рода.

И другая идентичность связана с полем идентификации, а точнее с идентификатором сохраняемостии или идентификатором базы данных. Если сравнить эти виды идентичности с терминами, принятыми в реляционных базах данных, то получим результаты, сведенные в табл. 8.2.

Таблица 8.2. *Термины, обозначающие идентичность в модели предметной области и в реляционной базе данных*

Модель предметной области	Реляционная база данных
Идентификатор предметной области	Альтернативный ключ
Поле идентификации	Первичный ключ

Подробнее этот вопрос рассматривается в [Bauer/King HiA].

Итак, уже имеются некоторые детали для внедрения в модель предметной области. В частности, номер заказа является идентификатором предметной области, а не значением поля идентификации. Это совершенно очевидно, например, для реализаций методов `Equals()/HashCode()`, что, в частности, создает осложнения для множеств. Поэтому необходимо также ввести поля `Id`, значения которых формируются объектно-реляционным преобразователем, когда вновь созданный экземпляр объекта-сущности связывается с его хранилищем. После этого значения в полях `Id` уже больше не изменяются. Для полей `Id` обычно используется идентификатор `Guid`, как показано на рис. 8.1.

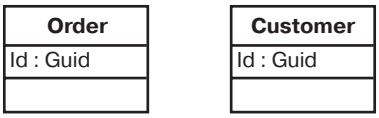


Рис. 8.1. Поля идентификации, введенные в объекты-сущности, имевшие идентификаторы предметной области

На заметку

Номера заказа (`OrderNumber`) и клиента (`CustomerNumber`) по-прежнему существуют. Из-за этого немного изменятся хранилища, а также произойдут новые перегрузки методов `GetOrder()` и `GetCustomer()`.

Такое изменение было крайне необходимым для инфраструктуры, а следовательно, его можно считать типичным примером отвлечения от предметной области. Это было сделано не столько ради одного конкретного объектно-реляционного преобразователя, сколько ради упрощения в той или иной степени всех объектно-реляционных преобразователей.

На заметку

По иронии судьбы, когда мы с коллегой работали над проектом Valhalla, то посчитали обязательным использовать идентификаторы `Guid`. Нам такое решение не очень нравилось, поскольку оно было "принудительным" для разработчика, в связи с чем мы включили его в список изменений на будущее. Теперь, когда я пользуюсь другой базовой структурой сохраняемости, у меня больше свободы для выбора. Тем не менее, я по-прежнему считаю, что идентификаторы `Guid` целесообразно использовать в качестве значения поля идентификации.

Итак, вернемся к метаданным и рассмотрим еще один пример их содержимого.

Преобразование внешних ключей

В метаданных нередко можно также найти применение шаблону Foreign Key Mapping (Преобразование внешнего ключа) [Fowler PoEAA]. Он служит для описания внешних ключей и соответствующих связей в модели предметной области.

В отличие от шаблона Identity Field, этот шаблон имеет отношение не к скопированным значениям в модели предметной области, а непосредственно к метаданным.

Использовать можно самые разные отношения, причем объектно-реляционные преобразователи поддерживают такое разнообразие отношений. Мой опыт показывает, что чаще всего приходится использовать относительно небольшое число отношений, но если потребуются менее очевидные и более сложные отношения, то было бы очень приятно обнаружить их поддержку.

Внедренное значение

Один из способов устранения потери соответствия состоит в том, чтобы иметь крупноструктурные таблицы и мелкоструктурные классы в модели предметной области. Для этой цели служит шаблон Embedded Value (Внедренное значение) [Fowler PoEAA].

Это означает возможность хранить данные клиента в одной таблице `Customers` базы данных, но работать с ними как с объектами `Customer` и `Address` в модели предметной области. (Данный пример очень упрощен; в действительности дело обстоит совершенно иначе.)

В простейшем виде (назовем это первым уровнем) мы можем лишь описать отношение между внедренным значением и столбцами в таблице базы данных. А на втором уровне мы можем написать дополнительный код, помогающий выполнять преобразование в более сложных ситуациях.

Решения проблемы наследования

Иерархии наследования в модели предметной области не имеют полного соответствия в реляционной базе данных, поскольку реляционные базы данных никогда не строились по принципу наследования (по крайней мере, до появления SQL:1999 — одной из немногих СУБД, поддерживавших наследование на момент написания этой книги). Более того, наследование, по-видимому, используется в модели предметной области не так часто, как можно было бы предположить. Если же требуется использовать наследование, следует обеспечить его поддержку в объектно-реляционном преобразователе.

Существуют три решения проблемы наследования с использованием шаблонов Single Table Inheritance (Наследование с таблицей), Class Table Inheritance (Наследование с таблицей для каждого класса) и Concrete Table Inheritance (Наследование с таблицей для конкретного класса) [Fowler PoEAA]. Эти шаблоны сгруппированы вместе, просто потому что они представляют собой разные решения данной проблемы.

Главное отличие между ними заключается в количестве таблиц, используемых для хранения иерархии наследования. Допустим, что `Person` (Лицо) — это базовый класс, а `Student` (Учащийся) и `Teacher` (Преподаватель) — его подклассы. В этом случае разные шаблоны обусловливают появление типичных таблиц, сведенных в табл. 8.3. Из этих таблиц нетрудно выяснить местоположение отдельных столбцов.

Таблица 8.3. *Шаблоны для сохранения иерархий наследования и необходимые для этой цели таблицы*

Шаблон	Таблицы в базе данных
Single Table Inheritance	`People`
Class Table Inheritance	`People`, `Students`, `Teachers`
Concrete Table Inheritance	`Students`, `Teachers`

Если объектно-реляционный преобразователь поддерживает только одну из приведенных выше иерархий наследования, структура базы данных оказывается менее гибкой по сравнению с поддержкой всех трех иерархий, из которых можно выбрать наиболее подходящую. Насколько это важно — решать вам.

Карта идентичности

Когда я делал первые попытки создать объектно-реляционный преобразователь, то сначала думал, что смогу обойтись без шаблона Identity Map, но затем я пришел к выводу, что такой преобразователь получается слишком сложным, что накладывает на программирующего потребителя слишком большую ответственность.

С другой стороны, многое зависит от структуры модели предметной области. Так, если отношения между объектами-сущностями в разных агрегатах вообще отсутствуют, то потребность в шаблоне Identity Map и соответственно в карте идентичности не столь велика. Такие принципы ППО, как упрощение и развязывание внутри модели предметной области, позволяют обходиться и без карты идентичности. Но я все же считаю, что наличие активной карты идентичности приносит определенную пользу.

На заметку

Если рассматривать объектно-реляционный преобразователь как некое связующее звено между базой данных и объектом, то роль карты идентичности может оказаться не столь значительной. Но в этом случае пришлось бы выполнить больше работы, что неинтересно. Намного интереснее обеспечить поддержку модели предметной области, выбрав как можно более простое (и в то же время удачное) решение.

Карта идентичности может оказаться пригодной не только для контроля графа объекта, известного потребителю, но и для других целей. В частности, с ее помощью можно образовать отношения M:N между объектами в модели предметной области при чтении из базы данных.

На заметку

Обозначение M:N описывает отношение типа “многие ко многим” между объектами касательно количества элементов или множественности. Например, в доме проживает много людей и в то же время каждый человек может владеть несколькими домами.

Нередко из соображений производительности приходится также рассматривать возможность кэширования, но, как вам уже известно, я не являюсь сторонником кэширования и поэтому пользуюсь шаблоном Identity Map как удобным средством для программирования, а не для улучшения производительности.

Объектно-реляционные преобразователи отличаются по уровню “сеанса работы”, для которого может или должна использоваться карта идентичности. Это может быть отдельная машина, процесс или сеанс работы в системе.

Вместе с шаблоном Identity Map зачастую используется другой шаблон. Речь идет о шаблоне Unit of Work.

Единица работы

В большинстве объектно-реляционных преобразователей используется или, по крайней мере, поддерживается шаблон Unit of Work. Главное отличие между ними заключается в том, насколько прозрачной для потребителя оказывается единица работы. Так, если объектно-реляционный преобразователь создан в стиле "неведения сохраняемости во время выполнения", потребитель может явно обратиться к единице работы, чтобы зарегистрировать новый экземпляр объекта, вводимого при последующем сохранении. В этом отношении другие виды объектно-реляционных преобразователей действуют более прозрачно, но в таком случае экземпляр объекта, скорее всего, придется получать с помощью фабрики, предоставляемой объектно-реляционным преобразователем. У каждого из этих подходов имеются свои достоинства и недостатки.

Как и для шаблона Identity Map, уровни "сеанса работы" могут быть разными для шаблона Unit of Work — по крайней мере, теоретически.

Загрузка по требованию или энергичная загрузка

Как уже отмечалось, при построении модели предметной области я редко пользуюсь шаблоном Lazy Load [Fowler PoEAA] в агрегатах по шаблону Aggregate [Evans DDD]. Несмотря на это, далеко не всегда бывает ясно, что же требуется для инфраструктуры сохраняемости, и поэтому для целей оптимизации должна быть предусмотрена возможность использовать шаблон Lazy Load.

Но вместо этого я пользуюсь энергичной загрузкой в качестве стандартной методики для своих агрегатов. Такую загрузку можно назвать агрессивной, жадной, всеохватывающей или предварительной, в дальнейшем она будет обозначаться как энергичная.

Энергичная загрузка совершенно противоположна загрузке по требованию, поскольку в этом случае граф загружается полностью, а не по частям и сразу же, а не с задержкой.

Энергичная загрузка хорошо сочетается с агрегатами — по крайней мере, как стандартное решение.

На заметку

Кому-то может показаться, что я слишком настаиваю на агрегатах, когда речь заходит о сценариях чтения. Я пользовался агрегатами в качестве *стандартной* схемы загрузки, выполняя оптимизацию, когда в этом возникала потребность.

В конце концов, агрегаты очень важны для *написания* сценариев. С этой точки зрения было бы целесообразно ввести в протокол метод, аналогичный

`GetForWrite()`, перед внесением изменений в агрегат. Метод `GetForWrite()` мог бы загружать все, что нужно, из агрегата, возможно, даже обеспечивая согласованность чтения.

Если многие экземпляры объектов требуется интерпретировать как единое целое, то для этой цели как нельзя лучше подходят агрегаты со всеми вытекающим отсюда последствиями.

Для активного перехода в режим записи пользователям полезно отличать свойство "только для чтения" от свойства записи. Кроме того, пользователь не должен начинать внесение изменений в уже устаревший объект, если применяются оптимистические методы управления параллельным выполнением операций. Это означает, что риск возникновения конфликтов снижается. Такой подход вполне пригоден и для пессимистических методов управления параллельным выполнением операций.

Энергичную загрузку можно реализовать самыми разными способами. Нередко это делается с помощью операторов `OUTER JOIN` в SQL, после чего результирующее множество разделяется на граф. В качестве другого распространенного способа используется несколько следующих друг за другом операторов `SELECT`.

Грамотно построенный объектно-реляционный преобразователь должен поддерживать как загрузку по требованию, так и разные методики энергичной загрузки списков, а также отдельных экземпляров объектов. Он может даже поддерживать загрузку по требованию групп полей одного экземпляра объекта. Хотя такую группу атрибутов в большинстве ситуаций можно всегда вынести в отдельный шаблон Value Object. И тогда мы вновь возвращаемся к вопросу загрузки экземпляров объектов по требованию или же энергичной загрузки.

Управление параллельным выполнением операций

Как уже неоднократно отмечалось, шаблон Aggregate отлично подходит для управления параллельным выполнением операций. Он позволяет выбирать требуемую структурную единицу и работать с ней как с единым целым. Тем не менее, такое решение не является полным. Ведь нужно еще избежать конфликтов или хотя бы обнаружить их, чтобы не получить непротиворечивые данные (в особенности без всякого уведомления об этом).

В книге [Fowler PoEAA] предлагается несколько решений этой проблемы.

- **Крупноструктурная блокировка (Coarse-Grained Lock)**
  Вместо блокировки на уровне экземпляра объекта в этом шаблоне предполагается блокировка более крупной структурной единицы. Например, мы можем использовать его на уровне корня агрегата, тем самым неявно блокируя все составляющие агрегата.

- **Оптимистическая автономная блокировка (Optimistic Offline Lock)**
  Не предполагает конфликты, но *проверяет* их наличие перед фиксацией.
- **Пессимистическая автономная блокировка (Pessimistic Offline Lock)**
  Предупреждает конфликты с помощью механизма проверки, исключающего их появление.

Это наводит на мысль о необходимости ввести информацию о контроле версий в модель предметной области, чтобы установить, где именно может потребоваться управление параллельным выполнением операций в свете требования номер 4 "Большое значение имеет выявление конфликтов параллельной обработки", сформулированного в главе 4. В частности, для поддержки оптимистической автономной блокировки в корни агрегатов следует ввести поле `Version`. Это делается не автоматически, а только по мере надобности. Соответствующие изменения приведены на рис. 8.2.

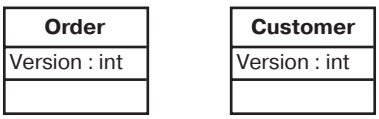


Рис. 8.2. Поля `Version`, введенные в корни некоторых агрегатов

Нет ничего страшного в том, что объектно-реляционный преобразователь не поддерживает шаблон Pessimistic Offline Lock. Во-первых, нужно научиться аккуратно пользоваться им, поскольку это требует определенных издержек. И во-вторых, если пессимистическая автономная блокировка действительно требуется, ее можно без особого труда организовать самостоятельно. (Одно из возможных решений, основанное на специальной таблице блокировок `Locks`, приведено в моей предыдущей книге [Nilsson NED].)

Резюме

В этой главе были рассмотрены потребности в инфраструктуре сохраняемости. После краткого введения и определения ряда требований к такой инфраструктуре мы обсудили данный вопрос под разными углами зрения, включая классификацию и применение некоторых шаблонов архитектуры приложений для предприятия.

Но в целом такое обсуждение оказалось довольно абстрактным. Для большей конкретности требуются некоторые примеры. Именно этой цели и служит следующая глава, в которой рассмотрено применение преобразователя NHibernate на конкретных примерах.

Глава 9

Приведение в действие преобразователя NHibernate

Эта глава отличается от большинства других глав данной книги — пришло время применить на практике методы организации инфраструктуры сохраняемости, рассматривавшиеся в главе 8.

В начале главы дается краткое введение в преобразователь NHibernate, в том числе: с чего следует начинать, как обращаться с преобразованием и какой вид должен иметь интерфейс API.

Затем мы найдем место преобразователю NHibernate в классификации, рассматривавшейся в главе 8, а также обсудим реализацию в этом преобразователе различных шаблонов инфраструктуры, упоминавшихся в предыдущей главе.

И наконец, мы посмотрим, как преобразователь NHibernate вписывается в общую картину ППО.

Основания для выбора преобразователя NHibernate

Возникает резонный вопрос: почему я выбрал именно преобразователь NHibernate? Во всяком случае не потому, что это самое лучшее решение из всех имеющихся. Ведь все, как известно, зависит от конкретных обстоятельств.

Я выбрал преобразователь NHibernate специально для этой главы по следующим причинам.

- **Наследие Java**
  NHibernate — это переносимая версия преобразователя Hibernate [Hibernate], известного многим и хорошо описанного в литературе и других источниках. Кроме того, объектно-реляционное преобразование стало по-настоящему зрелым на платформе Java, в отличие от платформы .NET.
- **Открытость**
  NHibernate — открытый программный продукт, что делает его легко доступным и удобным для опробования.

Эта глава ни в коем случае не предназначается в качестве исчерпывающего руководства по преобразователю NHibernate. Ее назначение — перевести в более конкрет-

ное русло абстрактную дискуссию, начатую в главе 8. Я не собираюсь также перечислять все возможности данного преобразователя, поскольку это не очень интересно и, кроме того, может оказаться неактуальным уже через несколько недель после выхода данной книги в свет.

Эту главу можно было бы с тем же успехом посвятить и другому виду инфраструктуры сохраняемости, чтобы легче было сравнивать разные решения, но это не входило в мои планы, иначе пришлось бы написать еще один раздел специально для такого сравнения.

Кроме того, я не намерен рассматривать NHibernate в контексте такого уровня абстракции, как, например, NWorkspace [Nilsson Workspace]. Главный акцент в этой главе будет сделан на самом преобразователе NHibernate и его интерфейсе API.

Итак, начнем с введения в NHibernate.

Краткое введение в NHibernate

Мы рассмотрим здесь программный продукт, выполняющий преобразование метаданных по шаблону Metadata Mapping [Fowler PoEAA], т.е. объектно-реляционный преобразователь под названием NHibernate.

Как упоминалось выше, NHibernate — это переносимая версия Hibernate, широко распространенного преобразователя на платформе Java и первоначально созданного Гэвином Кингом (Gavin King) [Bauer/King HiA]. Переносимая версия этого преобразователя основана на версии 2.1 Hibernate, которая считается довольно старой. Но переносимая версия не является, строго говоря, результатом простого перенесения базы кода из данной конкретной версии Hibernate. Напротив, в нее включены свойства последующих версий Hibernate, а в отдельных местах и другие свойства.

Преобразователь NHibernate (как, впрочем, и Hibernate) является открытым программным продуктом. На момент написания этой книги существовала его версия 1.0. О ее загрузке см. в [NHibernate].

Как известно, все проверяется на практике, поэтому перейдем непосредственно к применению преобразователя NHibernate.

Подготовка

Допустим, что имеется модель предметной области и требуется написать для нее некоторый заказной код.

> ***На заметку***
>
> Еще одно требование состоит в том, чтобы сделать модель предметной области сохраняемой. Следовательно, требуется также СУБД, но мы не будем пока что обсуждать *схему* базы данных.

Прежде всего, необходимо указать на уровне потребителя ссылку на базовую структуру NHibernate (а именно на файл библиотеки `nhibernate.dll`). Сделать это не так уж трудно.

Затем необходимо настроить применение NHibernate на уровне потребителя модели предметной области. Это можно сделать в коде или же в файле конфигурации. Как правило, для этой цели используется файл конфигурации, поэтому выберем именно такой вариант. В ходе настройки указывается конкретная СУБД, строка, связывающая с базой данных, а также режим регистрации. Многочисленные примеры файла конфигурации можно найти на Web-сайте NHibernate [NHibernate], поэтому не будем вдаваться в подробности его настройки, а просто будем считать, что у нас уже имеется подходящий файл конфигурации.

Далее следует настроить фабрику сеансов работы (`SessionFactory`) для выполнения каждого приложения в отдельности из соображений рационального расходования ресурсов. В `SessionFactory` будут анализироваться все метаданные и строиться для этой цели соответствующие структуры в оперативной памяти. Как следует из названия `SessionFactory`, этот компонент служит для получения новых экземпляров, реализующих интерфейс `ISession`.

`ISession` представляет собой строку, выбираемую для выполнения операций в базе данных, а также интерфейс для постоянного взаимодействия на уровне потребителя.

Далее следует создать вспомогательный метод для управления интерфейсом `ISession`. Это можно сделать с разной степенью сложности, но в данном случае достаточно и самого простого варианта. Если допустить, что модель предметной области называется `ADDDP.Ordering.DomainModel`, то упрощенный вариант вспомогательного метода под названием `NHConfig` будет выглядеть следующим образом:

```
public class NHConfig
{
  private static ISessionFactory _sessionFactory;

  static NHConfig()
  {
    Configuration config = new Configuration();
    config.AddAssembly("ADDDP.Ordering.DomainModel");
    _sessionFactory = config.BuildSessionFactory();
  }

  public static ISessionFactory GetSessionFactory()
  {
    return _sessionFactory;
  }
}
```

Допустим также, что мы работаем в настоящий момент над клиентским приложением с расширенными функциональными возможностями, а следовательно, будем, хотя бы для начала, считать, что одного экземпляра интерфейса `ISession` должно

быть достаточно. В таком случае для запуска приложения или хотя бы одной его формы вполне подойдет следующий код:

```
_session = NHConfig.GetSessionFactory().OpenSession();
_session.Disconnect();
```

А для приведения в действие интерфейса `ISession` можно воспользоваться следующим небольшим фрагментом кода:

```
_session.Reconnect();

// Сделать что-нибудь полезное...

_session.Disconnect();
```

И наконец, по завершении работы приложения (или закрытии его формы — в зависимости от выбранной методики) сеанс связи по интерфейсу `ISession` закрывается следующим образом:

```
_session.Close();
```

На заметку

Выбор одного интерфейса `ISession` для каждого приложения или его формы соответствует лишь двум из нескольких возможных методик.

Все это, конечно, интересно, но в то же время совершенно бесполезно с точки зрения операций, выполняемых в базе данных. Открытия и закрытия сеанса связи по интерфейсу `ISession` явно недостаточно, ведь нужно еще и сохранить изменения в базе данных. А для этого нам потребуется дополнительная подготовка.

Метаданные для преобразования

Выполненная до сих пор подготовка вообще не затронула модель предметной области. Но нам нужно сделать так, чтобы не оказывать влияние на модель, а дополнить ее. С этой целью мы введем информацию о преобразовании, демонстрирующую взаимосвязь между моделью предметной области и базой данных. Как правило, такая информация формируется в файле XML для каждого объекта-сущности по шаблону Entity [Evans DDD], причем эти файлы хранятся в каталоге проекта для выбранной модели предметной области.

Мы пока еще не составили схему базы данных и поэтому вольны действовать как угодно. В данный момент у нас имеется лишь модель предметной области. Для формирования информации о преобразовании или же для составления схемы базы данных можно было бы воспользоваться специальными инструментами, но для целей данного введения в NHibernate важно лишь показать, каким образом может выглядеть содержимое файла конфигурации для определенного объекта-сущности из мо-

дели предметной области и таблицы из базы данных. Итак, выберем простой объект-сущность из нашей модели предметной области. Для этой цели вполне подойдет объект класса `Customer`. Этот объект приведен вместе с объектами-значениями по шаблону Value Object [Evans DDD] на рис. 9.1.

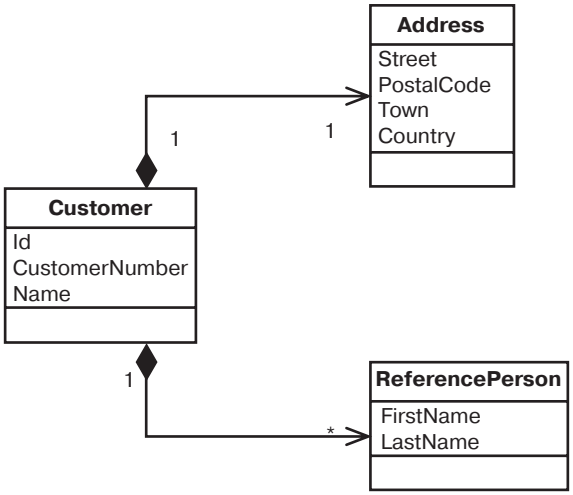


Рис. 9.1. Объект-сущность `Customer` и объекты-значения `Address` и `ReferencePerson`

В базе данных оба объекта `Customer` и `Address`, скорее всего, хранятся в одной таблице. На языке описания данных (DDL) оба этих класса могут выглядеть следующим образом:

```
create table Customers (
  Id UNIQUEIDENTIFIER not null,
   CustomerNumber INT not null,
   Name VARCHAR(100) not null,
   Street VARCHAR(50) not null,
   PostalCode VARCHAR(10) not null,
   Town VARCHAR(50) not null,
   Country VARCHAR(50) not null,
   primary key (Id)
)
```

На заметку

Наличие полей `Town` (Город) и `Country` (Страна) в этом описании данных кажется не вполне уместным. В реальной ситуации я бы вынес их за скобки, но для целей данного обсуждения это не так важно.

В приведенном выше описании данных в формате DDL отсутствует таблица `Reference Persons` (Ответственные лица), но ее можно проследить по рис. 9.1.

А теперь перейдем к недостающей и наиболее интересной для рассмотрения части — файлу преобразования.

На заметку

Следует еще раз подчеркнуть, что составлять таблицу в базе данных до файла преобразования совсем не обязательно. Порядок работы вы устанавливаете для себя сами.

Я лично предпочитаю начинать с модели предметной области, затем создавать файлы преобразования и далее автоматически формировать базу данных на основании информации о преобразовании. Для этой цели может служить следующий фрагмент кода:

```
Configuration config = new Configuration();
config.AddAssembly("ADDDP.Ordering.DomainModel");
SchemaExport se = new SchemaExport(config);
se.Execute(true, true, false, true);
```

Как показывает мой опыт, на базу данных зачастую приходится накладывать ряд ограничений в дополнение к тому, что мы можем описать в файле преобразования, и для этой цели я специально пишу заказной код. Такой подход не очень изящен, но вполне пригоден для решения данной задачи.

Имея все эти компоненты под рукой, я могу переформировывать базу данных на стадии разработки сколь угодно часто — даже перед выполнением каждого теста, при условии что это не замедляет процесс разработки.

Составим этот файл по частям. Прежде всего, нужно описать документ `Customer.hbm.xml` следующим образом:

```
<?xml version="1.0" encoding="utf-8" ?>
<hibernate-mapping xmlns="urn:nhibernate-mapping-2.0"
namespace="ADDDP.Ordering.DomainModel"
assembly="ADDDP.Ordering.DomainModel">
```

В данном случае документ описывается с помощью дескрипторов `namespace` (пространство имен) и `assembly` (сборка), чтобы выразить остальную информацию о преобразовании более лаконично, поскольку она не должна повторяться.

На заметку

Для того чтобы метаданные оказались в пределах сборки, не забудьте установить свойство `Embedded Resource` (Встроенный ресурс) файла XML.

Далее в описании необходимо указать имя класса для объекта-сущности по шаблону Entity, а также имя таблицы, например, следующим образом:

```
<class name="Customer" table="Customers">
```

После этого в описании указывается поле идентификации по шаблону Identity Field [Fowler PoEAA], причем не только его имя в классе и таблице (в приведенном ниже примере имена свойства и столбца одинаковы, поэтому дескриптор `column` не требуется), но и методика формирования нового значения. Такое описание может иметь следующий вид:

```
<id name="Id" access="field.camelcase-underscore"
  unsaved-value="00000000-0000-0000-0000-000000000000" >
  <generator class="guid" />
</id>
```

В данном конкретном примере идентификатор `guid` формируется преобразователем NHibernate, когда новый экземпляр объекта связывается с интерфейсом `ISession`.

Затем такие простые свойства, как `Name`, преобразуются следующим образом:

```
<property name="Name" type="AnsiString" length="100"
    not-null="true" />
```

Как правило, преобразователь NHibernate может самостоятельно подобрать подходящий тип для свойств, исследуя класс методом рефлексии, но для строковых свойств в данном случае специально указан дескриптор `type`, чтобы присвоить им тип `VARCHAR` вместо `NVARCHAR` при автоматическом формировании схемы базы данных на основании информации о преобразовании. Дескриптор `length` также служит для составления описания в формате DDL данных на основании информации о преобразовании, хотя ввод такой информации требует дополнительных усилий.

На заметку

Подобная информация очень полезна и для специальной проверки достоверности.

Кроме того, NHibernate позволяет преобразовывать не только свойства, но и поля. На самом деле `Customer.Name` — это не свойство (каковым NHibernate его считает по умолчанию и поэтому никак не выражает), а общедоступное поле. Следовательно, для того чтобы все было правильно, необходимо ввести еще и дескриптор `access` (доступ).

```
<property name="Name" access="field" type="AnsiString"
    length="100" not-null="true" />
```

Это распространяется и на все типы доступа, включая и частные, так что вы вольны выбирать наиболее подходящую для вас методику. Так, если при получении свойства раскрывается иное значение, чем сохраняемое, или же если выполняется перехват при вызове методов получения/установки свойств, то в этих случаях рекомендуется преобразование частного поля, а не свойства. Преобразование частного поля может выглядеть следующим образом:

```
<property name="_customerNumber"
    access="field" not-null="true" />
```

При преобразовании частного поля целесообразно установить и соблюдать правила присвоения имен. Благодаря этому в коде запроса придется указывать не имена частных полей (например, `_customerNumber`), а простые и удобные имена свойств (например, `CustomerNumber`). Это может выглядеть следующим образом:

```
<property name="CustomerNumber"
access="field.camelcase-underscore" not-null="true" />
```

На заметку

Разумеется, у такого подхода имеются свои недостатки, но, как правило, лучше преобразовать частное поле вместо того, чтобы вводить отдельные частные свойства только ради сохраняемости. Если данный способ действует, то и прекрасно, а в противном случае его можно всегда заменить, если возникнет такая необходимость.

Впрочем, по этому поводу ведется горячая полемика. Ведь многие отдают предпочтение частным свойствам, характерным для сохраняемости.

И наконец, нам нужно описать порядок преобразования используемых объектов-значений в единую таблицу `Customers` (хотя это можно было сделать и до описания простых свойств, но так уж получилось). В итоге файл преобразования для класса `Customer` (отдельная информация о преобразовании для класса `Address` отсутствует, поскольку он представлен шаблоном Value Object, а не шаблоном Entity) выглядит следующим образом:

```
<component name="Address" access="field">
  <property name="Street" access="field.camelcase-underscore"
    type="AnsiString" length="50" not-null="true" />

  <property name="PostalCode"
    access="field.camelcase-underscore"
    type="AnsiString" length="10" not-null="true" />

  <property name="Town" access="field.camelcase-underscore"
```

```
    type="AnsiString" length="50" not-null="true" />

  <property name="Country" access="field.camelcase-underscore"
    type="AnsiString" length="50" not-null="true" />
</component>
```

А список таких объектов-значений, как `ReferencePerson`, может выглядеть следующим образом:

```
<bag name="ReferencePersons" access="field.camelcase-underscore"
  cascade="all">
  <key column="OrderId" />
  <composite-element class="ReferencePerson">

  <property name="FirstName"
    access="field.camelcase-underscore" not-null="true"/>

  <property name="LastName"
    access="field.camelcase-underscore" not-null="true"/>
  </composite-element>
</bag>
```

Рассмотрим еще один пример. В предыдущих главах мы выяснили, что класс строки заказа (`OrderLine`) должен быть представлен шаблоном Value Object. В таком случае преобразование в классе `Order` по отношению к классу `OrderLine` может выглядеть так, как было только что показано.

Но нетрудно предвидеть, что у класса `OrderLine` может появиться собственный список, например, примечаний. А это служит веским основанием для преобразования класса `OrderLine` в объект-сущность по шаблону Entity. Таким образом, он преобразуется в объект-сущность не по концептуальным, а по техническим и инфраструктурным причинам. В связи с этим информация о преобразовании в классе `Order` должна измениться следующим образом (у класса `OrderLine` должна быть своя информация о преобразовании):

```
<bag name="OrderLines" access="field.camelcase_underscore"
  cascade="all">
  <key column="OrderId" />
  <one-to-many class="OrderLine" />
</bag>
```

Ради примера допустим также, что в классе `OrderLine` должно быть поле с обратной ссылкой на класс `Order` (несмотря на все, что было до сих пор сказано о стремлении весьма умеренно пользоваться двунаправленностью). Таким образом, в файле XML описания класса `OrderLine`, введенном при преобразовании этого класса в объект-сущность, теперь должна появиться следующая область, определяющая отношение “многие к одному”:

```
<many-to-one
  name="Order"
  access="field.camelcase-underscore"
  class="Order"
  column="OrderId" />
```

Следует, однако, заметить, что задачу двунаправленности приходится всякий раз решать самостоятельно. В методе `AddOrderLine()` класса `Order` это может выглядеть следующим образом:

```
// Класс Order
public void AddOrderLine(OrderLine ol)
{
  _orderLines.Add(ol);
  ol.Order = this;
}
```

И наконец, это также означает, что у класса `OrderLine` будет собственное поле идентификации независимо от того, используется оно в модели предметной области или нет.

После заполнения пустых областей в файлах преобразования можно переходить к правке кода потребителя с целью сделать его намного более интересным.

Небольшой пример интерфейса API

А теперь рассмотрим небольшие фрагменты кода, демонстрирующие, каким образом должен выглядеть код потребителя, когда преобразователь NHibernate вводится в действие для выполнения операций создания, чтения, обновления и удаления (CRUD). На такие операции зачастую приходится большая часть работы приложения, связанной с сохраняемостью. Первый из рассматриваемых ниже фрагментов кода связан с операцией создания.

Операция создания

Для создания нового экземпляра объекта (или строки таблицы) в базе данных достаточно получить экземпляр нового объекта `Customer`, установить его свойства и вызвать любые подходящие методы. Когда же все будет готово к сохранению, останется лишь воспользоваться рассмотренным выше фрагментом кода для воссоединения с интерфейсом `ISession`. После этого экземпляр объекта связывается с интерфейсом `ISession` и в нем вызывается метод `Flush()` для сохранения всех изменений. Вместе все это выглядит следующим образом:

```
// Код потребителя
Customer c = new Customer();
c.Name = "Volvo";

_session.Reconnect();
```

```
_session.Save(c);
_session.Flush();
_session.Disconnect();
```

В данном случае интерфейсу `ISession` было конкретно указано на необходимость выполнить в конечном итоге операцию `INSERT`, поскольку был вызван метод `Save()`. Вместо этого можно было бы вызвать метод `SaveOrUpdate()`, а затем предоставить преобразователю NHibernate самому решать, следует ли выполнять операцию `INSERT` или же операцию `UPDATE`. В качестве информации при этом используется значение из поля идентификации, которое сравнивается с тем, что указано в дескрипторе `unsaved-value` (несохраненное значение; при использовании идентификаторов `guid` это значение равно `"00000000-0000-0000-0000-000000000000"`). Если значения в поле идентификации и дескрипторе `unsaved-value` совпадают, значит, следует выполнить операцию `INSERT`, а иначе — операцию `UPDATE`.

Следует также заметить, что выполнение операции `INSERT` задерживается и происходит при вызове метода `Flush()`, а не метода `Save()`. Дело в том, что на уровне потребителя должна быть предоставлена возможность для внесения многочисленных изменений, а сохранение всех этих изменений должно быть отложено на как можно более долгий срок.

Насколько успешной оказалась операция записи в базу данных? Самый простой способ проверить это — прочитать обратно сохраненный экземпляр объекта, что мы и сделаем далее.

Операция однократного чтения

Во втором примере кода выполняется чтение одного экземпляра объекта (строки таблицы) из базы данных по его идентификатору в поле идентификации. Для этой цели допустим, что у нас имеется простой фрагмент кода воссоединения/разъединения и сосредоточим основное внимание на интересующем нас конкретном коде.

Как обычно, мы обращаемся к интерфейсу `ISession`. Вызвав метод `Load()/Get()`, можно указать, какой именно тип объекта требуется найти по его идентификатору, который передается в виде параметра из поля идентификации, следующим образом:

```
Customer c = (Customer)_session.Load(typeof(Customer), theId);
```

Следует, однако, иметь в виду, что это ни к чему не приведет, если сеанс связи по интерфейсу `ISession` не был закрыт с момента последнего вызова метода `Flush()`, поскольку при вызове метода `Load()/Get()` интерфейс `ISession` предоставит экземпляр из карты идентичности по шаблону Identity Map [Fowler PoEAA] вместо обращения к базе данных. Для принудительного обращения к базе данных (вместо открытия нового сеанса связи по интерфейсу `ISession`) можно вызвать метод `Evict()` по отношению к полученному экземпляру объекта, указав карте идентичности, чтобы она больше не отслеживала этот экземпляр, следующим образом:

```
_session.Evict(customer);
```

Все это хорошо, но что если значение идентичности объекта (т.е. его идентификатор) неизвестно? Если известна только часть имени клиента и требуется выбрать все экземпляры, которые совпадают с этим именем, то придется выполнить операцию многократного чтения, рассматриваемую в следующем примере.

Операция многократного чтения

Для формирования запросов в NHibernate имеются два специальных языка. Но в данном примере мы воспользуемся языком запросов NHibernate (HQL). Он очень похож на SQL, хотя и имеет свои отличия. Например, код для выборки всех клиентов, имена которых начинаются на "Vol", может иметь следующий вид:

```
// Код потребителя
string hql = "select from Customer where Name like 'Vol%'";
IList result = _session.CreateQuery(hql).List();
```

Таким образом, в переменной `result` сохраняется список клиентов, имена которых, как и предполагалось, начинаются на "Vol".

Как правило, код пишется не таким статическим, как в приведенном выше фрагменте, а с использованием параметризации. Но опять же, в данном примере продемонстрирован самый простой код, чтобы сразу же стало ясно, как это можно сделать.

Операция обновления

Итак, мы обнаружили немало экземпляров объектов, имена которых начинаются на "Vol", а также выяснили, что в один из них необходимо внести изменения. Затем мы внесли изменения в свойства такого экземпляра и теперь готовы сохранить эти изменения. И в этом случае мы могли бы воспользоваться методом `SaveOrUpdate()`, но поскольку нас интересует операция `UPDATE`, вместо этого выберем метод `Update()`, продолжив предыдущий фрагмент кода следующим образом:

```
Customer c2 = (Customer) result[0];
c2.Name = "Ford";
_session.Update(c2); // Можно опустить
_session.Flush();
```

На самом деле мы обнаружили экземпляр объекта благодаря чтению, и поэтому вызывать метод `Update()` совсем не обязательно. Этот экземпляр уже связан с интерфейсом `ISession`, а следовательно, изменения в нем будут все равно автоматически сохранены при вызове метода `Flush()`.

Операция удаления

Один из найденных экземпляров объектов оказался неверным, и поэтому он должен быть удален. Это можно сделать следующим образом:

```
// Код потребителя
Customer c3 = (Customer) result[1];
```

```
_session.Delete(c3);
_session.Flush();
```

Возможно, вы уже догадались, что удаление по методу `Delete()` задерживается до вызова метода `Flush()`. Как видите, удаление выполняется достаточно просто.

Транзакции

Прежде чем оставить данный пример интерфейса API, следует выяснить, каким образом можно контролировать транзакции, применяя NHibernate, поскольку пользоваться транзакциями приходится очень часто.

Если вы привыкли контролировать транзакции вручную, то вам нетрудно будет понять, как это делается с помощью NHibernate. Для этой цели выбирается экземпляр интерфейса `ITransaction`, который затем используется для вызова метода `Commit()` или `Rollback()` (при вызове метода `Commit()` по умолчанию вызывается метод `Flush()`). Весь фрагмент кода выглядит в данном случае следующим образом:

```
// Код потребителя
ITransaction tx = _session.BeginTransaction();
try
{
  tx.Commit();
}
catch (Exception ex)
{
  tx.Rollback();
  ...
```

На заметку

В приведенном выше фрагменте кода совершенно неожиданно использована обработка исключения, поскольку этого требует метод `Rollback()`. Конечно, обработку исключения можно было бы ввести и в остальные фрагменты кода, но это было бы отступлением от основной темы данной книги.

На этом, пожалуй, можно завершить краткое введение в преобразователь NHibernate. Дополнительные сведения о том, как им пользоваться, можно найти в [NHibernate].

А теперь перейдем к примеру выбранной инфраструктуры сохраняемости со структурой, представленной в главе 8. И прежде всего, рассмотрим общие требования к ней.

Требования к инфраструктуре сохраняемости

Итак, прежде чем углубляться в детали, рассмотрим общие требования, определенные в главе 8. Имеется в виду уровень неведения сохраняемости (НС), поддерживаемый жизненный цикл и обращение с базой данных.

Высокий уровень неведения сохраняемости

Модель программирования в NHibernate вполне соответствует НС во время выполнения, особенно в отношении самой модели предметной области. Но такая возможность дается не бесплатно. В частности, преобразователь NHibernate ничем не помогает в управлении свойством в обратном порядке, для чего требуется наличие двунаправленных отношений. Это приходится делать самостоятельно.

Кроме того, преобразователь NHibernate оказывается малопригодным, когда нужно проверить, изменилось ли состояние экземпляров объектов, например, с помощью свойства `IsDirty` или иным способом. (В момент сохранения преобразователь NHibernate может, конечно, определить, какие именно экземпляры объектов должны быть сохранены, но в данном случае имеется в виду совсем другое.)

Это лишь два примера того, чем приходится расплачиваться за высокий уровень НС. Помимо этого, расплачиваться приходится производительностью, добиться которой будет очень непросто из-за за высокого уровня НС.

Необходимо также предпринять что-то в отношении классов из модели предметной области, чтобы они вообще сохранялись преобразователем NHibernate. Ниже приведены характерные тому примеры.

- Ключевое слово `readonly` к полям неприменимо.

  Если в классе требуется свойство “только для чтения”, то чаще всего для этой цели используется ключевое слово `readonly`. Но в этом случае преобразователь NHibernate не сможет установить поле посредством рефлексии. Следовательно, это свойство придется преобразовать в частное поле плюс общедоступное получение свойства.

- У всех классов должны быть используемые по умолчанию конструкторы.

  Даже если такой конструктор и не требуется в модели предметной области, его придется ввести без параметров. Конструктор может быть внутренним и даже частным, но он все равно нужен.

- Следует избегать коллекций со строгим контролем типов.

  Зачастую это не такое уж и плохое требование. В частности, я бы предпочел воспользоваться интерфейсом `IList` вместо класса `CustomerList`. Для этой цели также подойдут обобщения.

  (К сожалению, на момент написания этой книги обобщения недостаточно поддерживались в NHibernate, в связи с чем приходилось как-то восполнять этот пробел.)

- Устанавливать поля идентификации нельзя.
  Как правило, в полях идентификации устанавливаются используемые по умолчанию значения, например, с помощью метода `Guid.NewGuid()`, но делать нечто подобное не стоит, если этого можно избежать при использовании NHibernate, поскольку в данном преобразователе значения полей идентификации служат для выбора между операциями `UPDATE` или `INSERT`. Если же предоставить используемое по умолчанию значение (при условии, что оно не установлено с помощью дескриптора `unsaved-value`), преобразователь NHibernate всегда будет выбирать операцию `UPDATE` в отсутствие явных указаний на иное. (Для конкретного указания операции `UPDATE` или `INSERT` можно также воспользоваться дескриптором `version`).

В целом дело обстоит не так уж плохо, хотя, по-прежнему, имеются факторы, понижающие уровень НС. Откровенно говоря, я даже не знаю, как избежать этого затруднения при использовании ключевого слова `readonly`, и поэтому не считаю, что какие-то другие решения окажутся лучше, если, конечно, они не повлияют на код.

Некоторые желательные свойства жизненного цикла постоянных объектов-сущностей

В главах 5 и 8 уже шла речь о простом жизненном цикле, который требуется для поддержки экземпляров модели предметной области. Приведем еще раз его сводку в табл. 9.1.

Таблица 9.1. *Сводка семантики жизненного цикла экземпляров объектов в модели предметной области*

Операция		Результат (временный/постоянный экземпляр объекта)
Вызов нового экземпляра объекта	→	Временный экземпляр объекта
`Repository.Add(`*`экземпляр`*`)` или `persistentInstance.Add(`*`экземпляр`*`)`	→	Постоянный экземпляр объекта в модели предметной области
`x.PersistAll()`	→	Постоянный экземпляр объекта в базе данных
`Repository.Get()`	→	Постоянный экземпляр объекта в модели предметной области (а также в базе данных)
`Repository.Delete(`*`экземпляр`*`)`	→	Временный экземпляр объекта (причем экземпляр удаляется из базы данных при вызове метода `x.PersistAll()`)

Этот жизненный цикл нетрудно привести к NHibernate, добавив еще один столбец в табл. 9.1 и получив таким образом табл. 9.2.

Таблица 9.2. *Сводка семантики жизненного цикла экземпляров объектов в модели предметной области с учетом NHibernate*

Операция	NHibernate		Результат (временный/ постоянный экземпляр объекта)
Вызов нового экземпляра объекта	Вызов нового экземпляра объекта	→	Временный экземпляр объекта
`Repository.Add(`*`экземпляр`*`)` или `persistentInstance.Add(`*`экземпляр`*`)`	`Isession.Save()`, `Update()`, `SaveOrUpdate()` или связывание с постоянным экземпляром объекта	→	Постоянный экземпляр объекта в модели предметной области
`x.PersistAll()`	`Isession.Flush()`	→	Постоянный экземпляр объекта в базе данных
`Repository.Get()`	`Isession.Load(),Get()` или `List()`	→	Постоянный экземпляр объекта в модели предметной области (а также в базе данных)
	`Isession.Evict()`, `Clear()` или `Close()`	→	Временный (но не удаляемый) экземпляр объекта; не подлежит контролю со стороны карты идентичности или единицы работы
`Repository.Delete(`*`экземпляр`*`)`	`Isession.Delete()`	→	Временный экземпляр объекта (причем экземпляр удаляется из базы данных при вызове метода `x.PersistAll()`)

В табл. 9.2 введена еще одна строка, в которой показано, как освобождается карта идентичности или единица работы экземпляра объекта (но не всех экземпляров). Такую ситуацию трудно заранее предусмотреть, но в действительности она иногда возникает, когда, например, требуется "перемещать" экземпляры объектов из одного сеанса связи по интерфейсу в другой.

Это, пожалуй, все, что можно пока что сказать по данному поводу.

На заметку

В данном вопросе имеется еще немало неразрешенных деталей, но мы не должны отступать от его рассмотрения в общей перспективе.

У вас может сложиться впечатление, будто соответствующий раздел главы 5 был изложен мной под сильным влиянием NHibernate, но аналогичные свойства жизненного цикла постоянных объектов-сущностей также поддерживаются большинством других объектно-реляционных преобразователей.

Аккуратное обращение с реляционной базой данных

И наконец, нам нужно, чтобы преобразователь NHibernate аккуратно обращался с реляционной базой данных. Это означает, что с базой данных следует обращаться, по крайней мере, так же, как и в том случае, когда программировать приходится вручную, т.е. очень аккуратно.

Если выполнить предыдущее требование, связанное с жизненным циклом постоянных объектов-сущностей, было относительно нетрудно, то удовлетворить данное требование будет намного труднее, что сильно зависит от типа объектно-реляционного преобразователя.

По поводу обращения преобразователя NHibernate с базой данных можно было бы привести немало критических замечаний, но я считаю, что данную задачу лучше решать вручную. Уже не раз подчеркивалось, что оптимизировать каждое обращение к базе данных нецелесообразно, поскольку это зачастую не имеет никакого значения. Вместо этого можно всегда обратиться непосредственно к коду и оптимизировать его в тех местах, где это действительно необходимо. Тем не менее, данное требование поставлено для того, чтобы не оптимизировать больше, чем требуется, а если и оптимизировать, то обращаться к обычным средствам SQL только при абсолютной необходимости.

На заметку

Надеюсь, что сказанное выше не будет воспринято как необходимость избегать SQL любой ценой или же пользоваться только объектно-реляционными преобразователями с развитыми функциональными возможностями.

Я отдаю должное средствам SQL и предпочитаю пользоваться ими, когда такое решение оказывается самым лучшим.

Какие же трудности в этой связи возникают при использовании NHibernate? Очевидно, что первая из них связана с переносом обработки из базы данных, например, с целью исключить появление логики в хранимых процедурах, а фактически — сами хранимые процедуры. Такая цель преследуется ради достижения большего удобства сопровождения, когда вся логика находится в модели предметной области и в ее окружении. (Опять же, при необходимости мы можем оптимизировать и сами хранимые процедуры в отдельных местах.)

Еще одно типичное затруднение возникает в связи с обработкой, которая обычно выполняется на уровне множеств, например, при увеличении в столбцах таблицы значений всех экземпляров объектов определенного класса. (Если это цена, то для всех товаров она увеличивается следующим образом: `Price = Price * 1.1`.) Для выполнения такой операции идеально подходит (и даже рекомендуется) SQL. А что касается обработки по отношению к базе данных, то в NHibernate множества рассматриваются как однострочные. Впрочем, это не так уж и плохо, поскольку правила

проверки достоверности чаще всего ориентированы на отдельные экземпляры объектов, и обработка по этим правилам обычно требуется для крупных обновлений, хотя и цена такой обработки может оказаться слишком высокой.

Третье затруднение состоит в том, что потребителя модели предметной области, скорее всего, вообще не интересует база данных. С другой стороны, он может создать сильную нагрузку на СУБД, даже не осознавая этого. В этом заключается один из недостатков той изоляции и абстракции, которую обеспечивает модель предметной области.

Четвертое затруднение связано с выборочным обновлением, если удовлетворяется определенный критерий, например:

```
UPDATE x SET y = 42 WHERE y >= 1
```

В таком случае уместно выполнить предварительное чтение данных.

Пятое затруднение связано с "чрезмерным чтением", когда считывается слишком много сложных типов данных, хотя на самом деле требуется совсем немного информации. Это особенно часто происходит тогда, когда не используется тщательно проработанная структура шаблона Aggregate или же шаблон Lazy Load.

Еще одно классическое затруднение связано с n+1-й выборкой. Допустим, что из базы данных выбираются все заказы, а все строки заказов загружаются по требованию. Эти операции затрагивают каждый заказ и его строки, причем для каждой строки заказа выдается новая команда `SELECT`. Если не использовать шаблон Lazy Load, данное затруднение окажется менее серьезным, но тогда в некоторых ситуациях возможно возникновение упомянутого выше затруднения, связанного с чрезмерным чтением.

Ни одно из рассмотренных выше затруднений не является характерным только для преобразователя NHibernate, напротив — оно присуще большинству объектно-реляционных преобразователей. Для преобразователя NHibernate более характерно затруднение, связанное со свойством `IDENTITY` или с последовательностями. В частности, применение полей идентификации приводит к более ранним обращениям к базе данных по команде `INSERT`, чем можно было бы предположить.

Преобразователь NHibernate вполне способен выполнить рассмотренные выше общие требования, особенно если вас как разработчика устраивает обычный режим работы объектно-реляционных преобразователей и связанные с этим компромиссы!

Классификация

Итак, после краткого введения в преобразователь NHibernate мы рассмотрели в общих чертах, как он удовлетворяет поставленным требованиям. Теперь можно продолжать изучение возможностей NHibernate, основываясь на классификации, приведенной в главе 8. Но вместо абстрактного обсуждения возможных альтернативных вариантов мы будем определять место NHibernate в каждой категории как пример

объектно-реляционного преобразователя и более подробно анализировать получающееся решение.

Начнем с вида модели предметной области, которую предполагается использовать в преобразователе NHibernate. Относительно данной категории следует дать дополнительные пояснения.

Вид модели предметной области

Как уже отмечалось в главе 8, к трем основным аспектам, определяющим виды моделей предметной области с точки зрения их использования в объектно-реляционных преобразователях, относятся следующие.

- Неведение сохраняемости.
- Наследование.
- Реализация интерфейса.

Выбор в данном случае сделать нетрудно, хотя он и не сводится только к двум полярным решениям. Преобразователь NHibernate позволяет добиваться высокого уровня НС. Кроме того, нам не требуется наследование от определенных базовых классов и реализация некоторых интерфейсов. Мы можем даже вообще обойтись без сборок NHibernate в модели предметной области.

Вид объектно-реляционного преобразователя

Что касается вида объектно-реляционного преобразователя, то, как упоминалось выше, преобразование на основе метаданных зачастую реализуется двумя разными способами:

- генерация кода;
- структурно-рефлексивный метод.

И в этой категории нетрудно найти место преобразователю NHibernate, поскольку он служит типичным примером применения структурно-рефлексивного метода. Разумеется, в NHibernate отчасти поддерживается генерация кода, но это обычно делается при формировании одной из трех составляющих модели предметной области, метаданных или схемы базы данных с учетом одной или двух других составляющих. Так, если имеются метаданные, то по ним можно составить схему базы данных. Под генерацией кода в данном случае не подразумевается определенный вид объектно-реляционного преобразователя. Этот вид определяется тем, как преобразование осуществляется во время выполнения, для чего в преобразователе NHibernate *не* используется генерация кода.

Исключением из этого правила может служить поддержка шаблона Lazy Load [Fowler PoEAA]. Так, во время выполнения преобразователь NHibernate будет подставлять вместо класса используемого списка некоторый его заменитель. Для этой цели придется раскрыть список в качестве интерфейса, а не конкретного класса.

На заметку

Применение интерфейсов не было упомянуто ранее в этой главе как средство понижения уровня НС, поскольку загрузка по требованию — это лишь свойство, а не что-то обязательное для использования.

По этому поводу существуют разные мнения, однако следует отметить, что загрузка по требованию является лишь методом оптимизации, а не чем-то обязательным для постоянного использования.

Если в преобразователе NHibernate применяется структурно-рефлексивный метод, то хорошо это или плохо? Я считаю, что когда речь идет об объектно-реляционных преобразователях, то в целом это хорошо.

Отправная точка

Для работы с определенным объектно-реляционным преобразователем зачастую можно выбрать одну или несколько отправных точек:

- схема базы данных;
- модель предметной области;
- информация о преобразовании.

Преобразователь NHibernate дает возможность начать работу с любой из перечисленных выше отправных точек, и это не просто особенность данного преобразователя, а свойство, которое поддерживается его структурой. Если же начинать с устаревшей схемы базы данных, изменять которую не разрешается, в таком случае целесообразно обратить внимание на базовую структуру преобразования данных iBATIS [iBATIS].

Акцент на интерфейсе API

Интерфейс API может быть представлен в двух разных вариантах:

- реляционные таблицы;
- модель предметной области.

Но, как было сказано выше, это на самом деле не касается большинства объектно-реляционных преобразователей, поскольку их основное назначение — дать модели предметной области представление о внешнем мире и в то же время использовать реляционную базу данных для целей сохраняемости. В преобразователе NHibernate модель предметной области служит в качестве акцента на интерфейсе API.

Тип языка запросов

Это уже более интересная категория. В категории формирования запросов различные решения могут сильно отличаться друг от друга. В главе 8 выделены следующие подкатегории формирования запросов:

- построчно в стиле SQL;
- на основании объектов запроса.

В преобразователе NHibernate реализованы оба способа формирования запросов под названием HQL и объектов Criteria. Выше уже приводился пример запроса HQL для поиска всех клиентов, имена которых начинаются на "Vol". Если же аналогичный запрос требуется выразить с помощью объектов Criteria, то в коде он будет иметь следующий вид:

```
// Код потребителя
IList result = _session.CreateCriteria(typeof(Customer))
    .Add(Expression.Like("Name", "Vol%"))
    .List();
```

Этот фрагмент кода служит также примером формирования цепочки вызовов нескольких методов. Он оказывается работоспособным, поскольку метод `Add()` возвращает экземпляр интерфейса `ICriteria` аналогично методу `CreateCriteria()`. Благодаря этому запросы критериев получаются более удобочитаемыми.

На заметку

В этой главе различные свойства преобразователя NHibernate рассматриваются лишь вкратце, и особенно это касается такой обширной темы, как формирование запросов. Более подробные сведения по данному вопросу можно найти в [Bauer/King HiA].

В главе 8 также упоминалось о том, что было бы неплохо иметь возможность обращаться непосредственно к SQL как к последнему средству, как правило, для решения проблем производительности, и это можно сделать в NHibernate. В этом преобразователе поддерживаются оба варианта, что дает возможность получать обратно объекты-сущности, обращаться к исходному соединению и далее делать все, что угодно.

Кроме того, преобразователь NHibernate предоставляет нечто среднее между механизмами обычных запросов и исходным SQL, т.е. так называемые запросы отчетов (или выравнивающие запросы). В коде это может выглядеть следующим образом:

```
// Класс потребителя
string hql = "select new CustomerSnapshot(c.Id, c.Name) " +
    "from Customer c";
IList result = _session.CreateQuery(hql).List();
```

В данном примере результатом обработки запроса будут не экземпляры объекта `Customer`, а экземпляры объекта `CustomerSnapshot`. Для того чтобы этот код стал работоспособным, необходимо предоставить согласующий конструктор в шаблоне Value Object (в данном примере `CustomerSnapshot`) и упомянуть класс

CustomerSnapshot в файле преобразования с помощью следующей директивы import:

```
<import class="CustomerSnapshot" />
```

Следует заметить, что согласно NHibernate в классе CustomerSnapshot поддерживаются только плоские данные (как правило, "только для чтения"). В то же время не отслеживается идентичность объектов в карте идентичности, а также изменения в единице работы.

С формированием запросов тесно связана следующая категория.

Улучшенная поддержка базы данных

В главе 8 приведены следующие примеры улучшенной (с точки зрения ООП, а не СУБД) поддержки базы данных:

- агрегирование;
- упорядочение;
- группирование;
- формирование скалярных запросов.

Все эти четыре операции поддерживаются в NHibernate на уровне HQL и объектов Criteria. Рассмотрим каждую из них на конкретном примере, начиная с агрегирующего запроса для поиска определенного числа экземпляров объекта Customer. В коде HQL это можно выразить следующим образом:

```
// Код потребителя
string hql = "select count(*) from Customer";
int numberOfCustomers =
    (int)_session.CreateQuery(hql).UniqueResult();
```

На заметку

В этом случае речь идет не об именах таблиц, а о классах, используемых в HQL.

Во втором примере код HQL для упорядочения результирующего множества в базе данных (чтобы не делать этого в модели предметной области, что зачастую имеет прямой смысл), будет выглядеть следующим образом (в данном случае упорядочивается операция выборки всех клиентов по имени):

```
select from Customer order by Name
```

Группирование чаще всего применяется в целях формирования отчетов, но его можно использовать и в приложении. Ниже приведен пример группирования по

имени и подсчета экземпляров объектов по каждому имени (в коде это выглядит сложнее, но в то же время наглядно показывает синтаксис):

```
// Код потребителя
string hql =
    "select c.Name, count(*) from Customer c group by c.Name";
IList result = _session.CreateQuery(hql).List();
```

И наконец, в качестве примера скалярного запроса можно было бы привести выборку только тех клиентов, имена которых начинаются на "Vol" (теперь надо признать, что это мой любимый критерий), но на самом деле этот пример уже был представлен выше при выполнении группирующего запроса. Что же касается результатов такого запроса, то переменная `result` типа `IList` содержит список массивов типа `object`. Ниже приведен фрагмент кода в продолжение приведенного далее примера, где перечисляются имена экземпляров объектов и их количество по каждому имени.

```
// Код потребителя, в продолжение предыдущего фрагмента кода
foreach (object[] o in result)
{
  Console.WriteLine("{0} {1}", (string)o[0], (int)o[1]);
}
```

Скалярные запросы особенно удобны в тех случаях, когда полная выборка экземпляров объекта `Customer` нецелесообразна, но требуется получить более простой результат, не определяя класс.

Разумеется, результат в этом случае не будет типизированным. Впрочем, можно было бы выбрать и другой вариант, а именно: упоминавшийся выше запрос отчета (или так называемый выравнивающий запрос).

Кроме того, можно было бы рассмотреть и другие варианты преобразования, но это привело бы к еще бо́льшим осложнениям. Ведь в таком случае, очевидно, нарушаются правила агрегирования, поскольку становится возможным обновление экземпляров без обязательной загрузки полных агрегатов, а это уже тревожный сигнал.

И в заключение этого раздела приведем пример, более "очевидный" с точки зрения ООП и менее понятный с точки зрения СУБД.

```
select o.Customer.Name, count(*) from Order o group by o.Customer.Name
```

Сам по себе этот запрос достаточно прост, но следует заметить, что данные выбираются из двух таблиц, и об этом (т.е. о соединении) никак не упоминается в предложении `from`. Разумеется, в данном случае используется информация о преобразовании, чтобы выяснить, что же нужно сделать с запросом HQL во время выполнения.

Другие функциональные возможности

И наконец, мы должны проверить преобразователь NHibernate на предмет поддержки других функциональных возможностей.

- **Поддержка конкретных СУБД**

  В этом случае обычно полагается приводить длинный перечень СУБД, но я не буду этого делать, а просто отмечу, что преобразователь NHibernate поддерживает самые разные базы данных (большинство коммерчески доступных и открытых СУБД). Подробнее об этом см. в [NHibernate]. Кроме того, вы можете написать собственный подключаемый модуль, если в NHibernate отсутствует поддержка используемой вами базы данных.

- **Не только функции объектно-реляционного преобразователя**

  NHibernate — это, главным образом, объектно-реляционный преобразователь, и ничего больше. Он построен по принципу делать что-то одно, но делать это как следует.

На заметку

Разумеется, NHibernate выполняет и другие функции, но они далеко не самые главные. Кроме того, этих функций немного и они не столь значительны.

- **Усовершенствованное кэширование**

  Что касается усовершенствованного кэширования, то преобразователю NHibernate не сравниться в этом отношении с Hibernate, хотя мне кажется, что он быстро восполнит этот пробел. С другой стороны, для меня лично это и не так важно, поскольку у кэширования второго уровня имеется немало недостатков, и поэтому данной возможностью можно и пренебречь.

- **Улучшенная поддержка контроля транзакций**

  В NHibernate достаточно хорошо реализована поддержка ручного контроля транзакций. Но если не быть начеку, то иногда возможны неожиданные результаты. Так, если запрос выполняется посредине транзакции, NHibernate сохраняет изменения по методу `Flush()` в базе данных перед выполнением запроса. Это своего рода компромисс между кэшированием и обычным контролем транзакций. Правда, этим процессом можно управлять в коде.

  В свете того, что было изложено в главе 8, типичным способом обращения с транзакциями в NHibernate является “ручной контроль”.

На заметку

Помимо этого, вы можете создать инструмент, в котором используются любые из дополнительных автоматических методов, упомянутых в главе 8.

- **Открытость исходного текста**
  Как упоминалось выше, NHibernate является открытым проектом, и поэтому вы можете исследовать его исходный текст и даже создать собственную разновидность этого объектно-реляционного преобразователя.
- **Версия (поколение)**
  Версию NHibernate определить нелегко. На момент написания этой книги она называлась версией 1, но в то же время ее качество было выше, чем обычно бывает в первой версии программного продукта. Кроме того, NHibernate является переносимой версией преобразователя Hibernate, который принадлежал ко второму, а теперь уже и к третьему поколению, хотя этот факт никак не отражен в NHibernate. Так или иначе, отнесем этот преобразователь к первому поколению.

А теперь рассмотрим место преобразователя NHibernate в классификации под другим углом зрения, т.е. с точки зрения шаблонов инфраструктуры.

Классификация по шаблонам инфраструктуры

Начнем непосредственно с типа метаданных.

Тип метаданных для преобразования

В главе 8 были определены три разных типа метаданных для шаблона Metadata Mapping [Fowler PoEAA]:

- документы XML или других форматов;
- атрибуты;
- исходный код.

Определить место NHibernate в этой категории нетрудно, поскольку его метаданные, как правило, представлены в формате XML-документа. Метаданные *можно* также описать и в исходном коде, но обычно этого никто не делает.

Имеются также методы представления метаданных в виде атрибутов, но и в этом случае метаданные формируются в виде XML-документов.

Поля идентификации

Мы уже обсуждали, как обращаться с полями идентификации в NHibernate, но мне бы хотелось рассмотреть этот вопрос чуть глубже. Ранее было показано, как преобразуется поле идентификации типа `Guid`. Возьмем для примера объект-сущность `Product`, у которого имеется поле идентификации типа `int`, а его значение формируется в базе данных с помощью свойства `IDENTITY` (т.е. в столбце с автоматическим приращением). В коде это может выглядеть следующим образом:

```
<id name="Id" access="field.camelcase-underscore"
  unsaved-value="0" >
  <generator class="identity" />
</id>
```

Как уже отмечалось в главе 8, когда дело доходит до новых объектов-сущностей, то значения в полях идентификации могут формироваться, по крайней мере, на четырех разных уровнях:

- потребителя;
- модели предметной области;
- базы данных;
- объектно-реляционного преобразователя.

Что касается глобально уникального идентификатора, то его можно формировать на всех уровнях, хотя делать это рекомендуется в объектно-реляционном преобразователе. В приведенном выше примере со свойством `IDENTITY` значение данного идентификатора формируется в базе данных, что фактически оказывает значительное влияние на модель программирования. Во-первых, переход к базе данных происходит раньше, чем можно было бы предположить. И во-вторых, вместо выборки значения `IDENTITY` может быть выполнена операция `INSERT`. В обоих случаях возможны осложнения, и поэтому я лично стараюсь пользоваться свойствами `IDENTITY` только в силу необходимости.

NHibernate поддерживает идентификаторы COMB

Применение идентификаторов `Guid` зачастую оказывается весьма удачным решением. Но, возможно, некоторые из вас уже читали мою статью о том, во что обходятся операции `INSERT` при использовании идентификаторов `Guid` в качестве первичных ключей [Nilsson COMB]. Дело в том, что производительность при выполнении операции `INSERT` оказывается во *много раз* ниже для таблиц с идентификаторами `Guid` в качестве первичных ключей, чем в том случае, если они имеют тип `int`. Как правило, это не вызывает особых осложнений, но может превратиться в серьезную проблему при наличии очень крупных таблиц и большой нагрузки на операции `INSERT`. В качестве решения этой проблемы я предложил в упомянутой выше статье использовать комбинированные идентификаторы, которые я сокращенно назвал COMB. По существу, это идентификаторы `Guid`, хотя и последовательного типа. Я был приятно удивлен, когда обнаружил, что в преобразователь NHibernate внедрена поддержка идентификаторов COMB в особой альфа-версии. Для того чтобы воспользоваться генератором идентификаторов COMB, необходимо придать метаданным следующий вид (единственное отличие состоит в том, что этот генератор называется `guid.comb`, а не просто `guid`):

```
<id name="Id" access="field.camelcase-underscore"
  unsaved-value="00000000-0000-0000-0000-000000000000" >

  <generator class="guid.comb" />
</id>
```

Проще всего предоставить преобразователю NHibernate самому формировать идентификаторы `Guid`, хотя такая возможность, конечно, имеется далеко не всегда. Иногда вместо этого приходится использовать назначенные поля идентификации, что немного труднее сделать. Например, преобразователю NHibernate нельзя позволять самому решать, следует ли выполнять операцию `UPDATE` или `INSERT` на основании дескриптора `unsaved-value`. Вместо этого преобразователю NHibernate обычно предписывается выполнять явные вызовы метода `Save()` или `Update()` вместо метода `SaveOrUpdate()`.

На заметку

Как уже отмечалось в этой главе, если вы отдаете предпочтение самостоятельной установке значений в полях идентификации, то можете воспользоваться для этой цели дескрипторами `version` и `unsaved-value`.

Итак, мы рассмотрели лишь некоторые методики применения полей идентификации. В NHibernate поддерживаются и многие другие методики. Кроме того, вы можете без особого труда внедрить собственные методики.

Преобразование внешних ключей

Ниже показано, как преобразование внешних ключей выглядит в файле преобразования `Order.hbm.xml`, где описывается заказ для определенного клиента.

```
<many-to-one name="Customer" access="field.camelcase-underscore"
  class="Customer" column="CustomerId" not-null="true" />
```

Не вдаваясь в детали, хочу отметить прочную поддержку в NHibernate многих видов отношений для удовлетворения большинства потребностей в данной области. И хотя научиться эффективно пользоваться всеми этими разновидностями отношений непросто, тем не менее, освоив даже самые элементарные отношения, можно добиться немалых успехов.

С другой стороны, вместе с преобразователем NHibernate очень просто используется шаблон Embedded Value (Внедренное значение) [Fowler PoEAA].

Внедренное значение

В преобразователе NHibernate дескриптор `component` служит для описания шаблона Embedded Value (как будто у этого дескриптора недостаточно других значений). Ниже показано, как выглядит в файле преобразования описание клиента, имеющего адрес.

```
<component name="Address" access="field">
  <property name="Street" access="field.camelcase-underscore"
    type="AnsiString" length="50" not-null="true" />

  <property name="PostalCode"
access="field.camelcase-underscore"
    type="AnsiString" length="10" not-null="true" />

  <property name="Town" access="field.camelcase-underscore"
    type="AnsiString" length="50" not-null="true" />

  <property name="Country" access="field.camelcase-underscore"
    type="AnsiString" length="50" not-null="true" />
</component>
```

Обратите внимание на наличие в таблице `Customers` всех столбцов, тогда как в классе `Customer` присутствуют только поля `Address`.

В этом случае потребителю будет нетрудно понять, как используется шаблон Embedded Value. Например, следующим образом:

```
Console.WriteLine(aCustomer.Address.Street);
```

Следует, однако, подумать и о том, должен ли объект-значение `Address` быть постоянным или непостоянным. В данном примере, пожалуй, стоило бы сделать его постоянным, хотя это далеко не всегда очевидно.

Если же сделать этот объект непостоянным, то вместо получения нового экземпляра объекта `Address` можно было бы написать следующий код:

```
aCustomer.Address.Street = "Large Street 42";
```

На заметку

Если внедренное значение должно использоваться во многих других классах, то было бы целесообразно реализовать интерфейс `IUserType`, чтобы не описывать всякий раз преобразование. Это уместно и для сохранения, которое отличается от раскрываемого типа данных. В этом случае преобразование выполняется посредством реализации интерфейса `IUserType`.

Недостаток такого подхода заключается в том, что из модели предметной области приходится обращаться к файлу библиотеки `nhibernate.dll`, иначе потребуется специальный код в отдельной сборке и наследование базового класса модели предметной области.

Я лично стараюсь избегать этого при всякой возможности. В простых случаях шаблон Embedded Value используется обычным способом, а в более сложных ситуациях преобразование свойств в частные поля при их получении или установке нередко может быть выполнено вручную, после чего эти частные поля приводятся к базе данных.

Применение типизированного шаблона перечисления [Bloch Effective Java] (например, для того чтобы наделить перечисления определенным поведением), служит характерным примером того, как я стараюсь избегать определенной конструкции или же пользоваться заказным кодом получения/установки свойств. Сначала я пытаюсь выяснить до конца, смогу ли я обойтись без обычного для C# типа `enum`, а затем пишу заказной код получения/установки свойств, чтобы не обращаться к интерфейсу `IUserType`. Это похоже на масштабное решение мелкой задачи.

И в заключение еще одно замечание: не следует преувеличивать значение попыток избежать обращения к библиотеке `nhibernate.dll`. Всякий раз нужно трезво оценивать, насколько это выгодно.

Решения проблемы наследования

В NHibernate поддерживаются все три решения проблемы наследования: Single Table Inheritance (Наследование с таблицей), Class Table Inheritance (Наследование с таблицей для каждого класса) и Concrete Table Inheritance (Наследование с таблицей для конкретного класса) [Fowler PoEAA]. Так, если классы `Customer` и `Vendor` наследуют от класса `Company`, то информация о преобразовании может иметь следующий вид:

```
<discriminator column="Type" type="AnsiString" length="8"
    not-null="true" />
```

Тогда для каждого подкласса потребуется следующее описание его особенностей, т.е. информация о преобразовании, не предназначенная для класса `Company`:

```
<subclass discriminator-value="Customer" name="Customer">
  <property name="CustomerNumber"
    access="field.camelcase-underscore"/>
</subclass>

<subclass discriminator-value="Vendor" name="Vendor">
  <property name="VendorNumber"
    access="field.camelcase-underscore"/>
</subclass>
```

Из приведенного выше должно быть ясно, что единственное изменение всякий раз будет связано с одним конкретным полем (`CustomerNumber` и `VendorNumber`).

В данном случае получится лишь одна таблица `Companies`, которая будет иметь следующий вид (при условии, что выбрано решение Single Table Inheritance):

```
create table Companies (
  Id UNIQUEIDENTIFIER not null,
   Type VARCHAR(8) not null,
   Version INT not null,
   Name VARCHAR(100) not null,
   Street VARCHAR(50) not null,
   PostalCode VARCHAR(10) not null,
   Town VARCHAR(50) not null,
   Country VARCHAR(50) not null,
   CustomerNumber INT null,
   VendorNumber INT null,
   primary key (Id)
)
```

Это означает, что для объектов класса `Vendor` значение в столбце `CustomerNumber` будет равно `NULL`, и наоборот. А значение в столбце `Type` будет равно "`Customer`" или "`Vendor`" (либо меньшему числу символов).

На заметку

Возможно, этот пример покажется не вполне очевидным тем, кто предпочитает пользоваться шаблоном Party Archetype (Групповой архетип) [Arlow/Neustadt Archetype Patterns] или же вообще не прибегать к излишнему наследованию, если единственное изменение связано только с одним полем.

Я же преследовал цель показать особенности наследования на ясном и простом примере.

Как и следовало ожидать, при правильно организованном преобразовании наследования программирующий потребитель может не думать о том, как сохраняется иерархия наследования, а вместо этого уделить основное внимание использованию данной иерархии.

Карта идентичности

В преобразователе NHibernate шаблон Identity Map используется на уровне интерфейса `ISession`. В этом нетрудно убедиться, если прочитать экземпляр объекта по его идентификационному номеру `Id` с помощью метода `Load()`, внести изменения непосредственно в строке таблицы из базы данных, а затем вновь прочитать экземпляр объекта с помощью метода `Load()`, используя тот же самый экземпляр интерфейса `ISession`. Изменения нельзя будет увидеть, поскольку преобразователь NHibernate не обратился к базе данных еще раз после вызова метода `Load()`, а вместо этого выбрал экземпляр объекта из карты идентичности.

Кроме того, преобразователь NHibernate обращается к карте идентичности, когда дело доходит до формирования запросов не с целью найти экземпляры объектов, но поместить в карте идентичности их идентичность для поддержки последующих вызовов метода `Load()` из этой карты.

Если же экземпляр объекта требуется удалить из карты идентичности, для этой цели достаточно вызвать метод `Evict()` в интерфейсе `ISession` или же метод `Clear()`, чтобы очистить карту идентичности от всех экземпляров объектов, либо просто закрыть сеанс связи по интерфейсу `ISession`, вызвав метод `Close()`.

Единица работы

В том, что в преобразователе NHibernate используется шаблон Unit of Work [Fowler PoEAA], вероятно, нет ничего удивительного, причем это вновь делается посредством интерфейса `ISession`, хотя реализация оказывается совсем не похожей на типичную. Вместо регистрации того, что произошло с единицей работы, преобразователь NHibernate делает моментальный снимок экземпляров объектов при их чтении из базы данных. А во время вывода из оперативной памяти экземпляры объектов, известные по интерфейсу `ISession`, сравниваются с моментальными снимками для формирования единицы работы в данный момент времени.

Любые удаления в известной степени регистрируются, а о новых экземплярах объектов уведомляется, например, при вызове метода `SaveOrUpdate()`.

Как всегда, у такого решения имеются определенные достоинства и недостатки. Очевидный его недостаток заключается в том, что когда дело доходит до анализа многих экземпляров объектов, то оказывается, что из оперативной памяти должен быть выведен только один экземпляр. Это одна из причин, по которой были внесены изменения в версию 3 Hibernate (но не в NHibernate — по крайней мере, на момент написания данной книги).

Загрузка по требованию или энергичная загрузка

Шаблон Lazy Load [Fowler PoEAA] без особого труда (и даже автоматически) применяется вместе с преобразователем NHibernate. Для этой цели достаточно объявить его в метаданных следующим образом:

```
<bag name="OrderLines" access="field.camelcase-underscore"
cascade="all" lazy="true">
    <key column="OrderId" />
    <one-to-many class="OrderLine />
</bag>
```

Теперь строки заказа загружаются по требованию, как только в них возникает необходимость. Но для этого нужно также установить соединение по интерфейсу `ISession`.

Если же в метаданных не используется объявление `lazy="true"`, то вместо загрузки по требованию, по существу, выполняется энергичная загрузка. В преобразователе

NHibernate применяются различные подходы к энергичной загрузке, включая выполнение оператора `OUTER JOIN` и ряда следующих друг за другом операторов `SELECT`.

Следует заметить, что при выполнении энергичной загрузки с помощью нескольких операторов `SELECT` существует вероятность получения несогласованных результатов чтения (что может вызвать еще бо́льшие осложнения при загрузке по требованию). Во избежание подобных осложнений осуществляется явный контроль транзакций или же уровень изоляции транзакций повышается до упорядочиваемого [Nilsson NED], но такие осложнения возникают редко, и поэтому не будем вдаваться в детали их устранения.

Кроме того, контроль транзакций можно осуществлять непосредственно в коде независимо от вида загрузки: по требованию или энергичной. Это, в частности, делается при формировании запросов, которому было уделено достаточно внимания в этой главе.

Управление параллельным выполнением операций

И наконец, нам потребуется поддержка управления параллельным выполнением операций. В книге [Fowler PoEAA] предлагается несколько решений этой проблемы:

- крупноструктурная блокировка;
- неявная блокировка;
- оптимистическая автономная блокировка;
- пессимистическая автономная блокировка.

В преобразователе NHibernate поддерживается пессимистическая автономная блокировка, если к объектам-сущностям добавить дескриптор `<version>` следующим образом:

```
<version name="Version" access="field.camelcase-underscore" />
```

Так, если на пересечении строки и столбца версии в таблице базы данных находится другое значение, чем то, что предполагается при сохранении, то, скорее всего, кто-то уже обновил данное значение — в итоге возникает конфликт параллельной обработки (появляется исключение `StaleObjectStateException`).

Остальные три механизма блокировки приходится поддерживать самостоятельно, поскольку такая возможность в преобразователе NHibernate изначально отсутствует.

Дополнительные возможности: перехватчики проверки достоверности

В качестве дополнительных возможностей, не упоминавшихся в главе 8, следует отметить перехватчики проверки достоверности. В NHibernate имеется интерфейс `IValidatable`, к которому этот преобразователь обращается при вызове метода `Flush()`.

Но для этой цели потребуется также обращение к библиотеке nhibernate.dll из модели предметной области, поэтому я лично предпочитаю решение, представленное в главе 7. Имеется в виду возможность обратиться к реализации интерфейса Iinterceptor (также поддерживаемого в NHibernate, но недоступного в модели предметной области) для неявной проверки правил и тем самым использовать любой интерфейс для классов из модели предметной области. (Разумеется, совсем не обязательно пользоваться интерфейсом для классов из модели предметной области.)

Для того чтобы обратиться к специальной реализации интерфейса IInterceptor, достаточно указать этот интерфейс в виде параметра при получении экземпляра интерфейса ISession следующим образом:

```
// Класс потребителя
_session = NHConfig.GetSessionFactory()
    .OpenSession(new MyCustomInterceptorImpl());
```

Следует, однако, иметь в виду, что такой подход не позволяет делать в коде проверки достоверности все, что угодно. В частности, нельзя пользоваться одним и тем же экземпляром интерфейса ISession для выполнения разных операций, чтобы не создавать, так или иначе, осложнения. Впрочем, мы уже рассматривали в главе 7 некоторые методики для преодоления подобных осложнений и поэтому не будем повторяться.

NHibernate и ППО

А теперь рассмотрим несколько примеров, которые помогут перенести все сказанное выше о преобразователе NHibernate в контекст ППО. Именно такими примерами мне бы и хотелось завершить эту главу.

И в этом случае я не собираюсь использовать уровень абстракции, аналогичный NWorkspace, хотя все примеры, приведенные ниже, в равной степени касаются и других объектно-реляционных преобразователей (пользоваться уровнем абстракции, на мой взгляд, было бы сильным упрощением).

Прежде всего, следует сказать несколько слов о том, как структурируются сборки.

Краткий обзор сборок

Концептуально, я предпочитаю рассматривать хранилища как неотъемлемую часть модели предметной области, но если выше объектно-реляционного преобразователя отсутствует уровень абстракции, я допускаю существование хранилищ в отдельной сборке. Благодаря этому образуются зависимости, приведенные на рис. 9.2.

Между моделью предметной области и преобразователем NHibernate пока что отсутствует какая-либо зависимость. Тем не менее в данной модели могут использоваться хранилища, если в этом есть потребность. Для этой цели, в частности, в хранилищах реализуются интерфейсы, присутствующие в модели предметной области, а реализации этих интерфейсов затем внедряются в подходящих местах экземпляров данной модели.

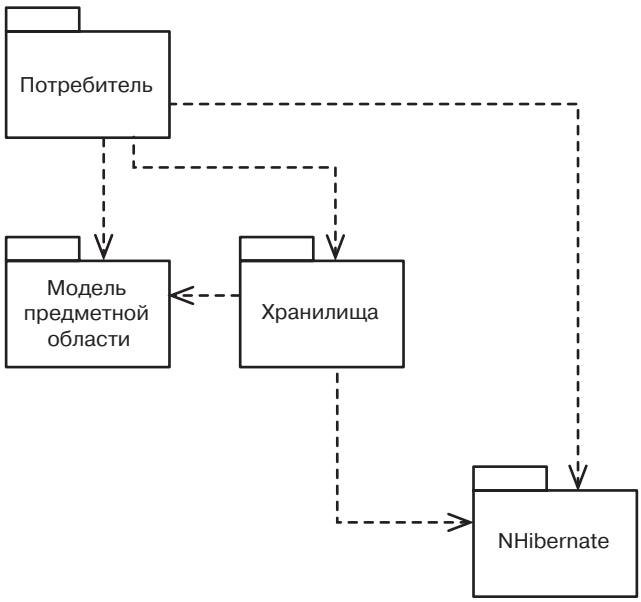


Рис. 9.2. Типичные зависимости между сборками

Интерфейс `ISession` и хранилища

Для того чтобы все это оказалось работоспособным, следует допустить внедрение на уровне потребителя экземпляров интерфейса `ISession`. Это означает, что фрагмент кода:

```
Customer c = (Customer)_session.Load(typeof(Customer), theId);
```

может быть преобразован в следующий фрагмент кода, если часть работы возьмет на себя хранилище (в данном простом примере это небольшая, но все же часть работы):

```
Customer c = customerRepository.GetCustomer(theId);
```

В этом фрагменте кода предполагается, что экземпляр объекта `customerRepository` создается следующим образом:

```
// Код потребителя
ICustomerRepository customerRepository =
    new CustomerRepository(_session);
```

С этого момента у хранилища будет интерфейс `ISession` для связи с СУБД.

Интерфейс `ISession`, хранилища и транзакции

И наконец, я предпочитаю управлять началом и завершением единицы работы на уровне потребителя, т.е. вне хранилищ. В приведенном ниже примере поясняется,

как это можно сделать с помощью очень компактного, хотя и *не* рекомендуемого кода транзакции.

```
// Код потребителя
customerRepository.Add(customer);
orderRepository.Add(order);
_session.BeginTransaction().Commit();
```

В этом фрагменте кода оба хранилища используют один и тот же интерфейс `ISession`.

Чего мы достигли

Сделаем небольшое отступление. Итак, мы выбрали объектно-реляционный преобразователь как наиболее очевидное решение и посмотрели, как он вписывается в концепцию хранилищ.

А теперь рассмотрим этот вопрос с совершенно другой стороны. Допустим, что у нас имеется хранилище и нам требуется наполнить его настоящим кодом. Допустим также, что часть этого кода решено написать вручную. Интерфейс API для хранилища может, конечно, оставаться неизменным в разных реализациях, но вместо следующей строки кода:

```
Customer c = (Customer)_session.Load(typeof(Customer), theId);
```

возможно, придется написать немало другого кода. В частности, для того чтобы определить оператор `SELECT` и затем получить все столбцы, необходимые для экземпляра объекта клиента, который требуется восстановить.

Далее необходимо создать экземпляр объекта клиента и переместить все значения из строки результирующего множества в этот экземпляр.

Конечно, мы можем все это сделать. Но дело в том, что это не столько сложная, сколько малоинтересная и трудоемкая работа, если, например, учесть, что экземпляру объекта клиента может сопутствовать список ответственных лиц, который требуется одновременно восстановить, не говоря уже о многих других деталях.

Всякое изменение в схеме базы данных или в модели предметной области приводит к дополнительной трудоемкой работе. А ведь в приведенном выше фрагменте кода мы лишь указали одну операцию получения единственного объекта.

Если же вместо этого воспользоваться объектно-реляционным преобразователем, изменения фактически будут касаться информации о преобразовании. Согласитесь, что это более эффективный способ внесения изменений по сравнению с малоинтересным написанием кода сохранения.

Эту идею можно рассматривать и в более широкой перспективе. Ведь если я знаю, что изменения не потребуют значительных затрат, то буду в большей степени склонен к рефакторингу кода, чем в противном случае!

И в завершение этого раздела мне бы хотелось привести пример из одного из своих недавних проектов. Как оказалось, изменения в этом проекте, повлиявшие только

на отдельные его составляющие (модель предметной области и схему базы данных) и касавшиеся только объектно-реляционного преобразователя, потребовали на порядок меньших затрат по сравнению с изменениями, повлиявшими на код сохранения, который пришлось писать вручную! Впрочем, не следует забывать, что объектно-реляционные преобразователи, безусловно, не являются *панацеей*. Им также присущи свои достоинства и недостатки.

Резюме

Применяя такой объектно-реляционный преобразователь, как NHibernate, можно очень точно выполнять требования к проекту, намеченные в главе 4, не внося существенные коррективы. Преобразователь NHibernate позволяет строить модель предметной области, придерживаясь принципа НС.

Объектно-реляционные преобразователи можно рассматривать как средства, дающие возможность делать акцент на ППО, но в то же время они должны допускать написание кода сохранения вручную, если в этом есть потребность.

Разумеется, я далек от того, чтобы рассматривать объектно-реляционные преобразователи лишь в розовом свете. В частности, мне бы очень хотелось, чтобы в таких преобразователях, как NHibernate, принимались во внимание принципы ППО, что упростило бы весь процесс проектирования и позволило бы достичь лучших результатов. Тем не менее, NHibernate как часть инфраструктуры можно считать большим шагом в правильном направлении.

Что же дальше? Ответам на этот вопрос посвящена последняя, четвертая часть этой книги.

В первой главе этой части дискуссия будет развернута в более широком контексте нескольких рассматриваемых моделей. После этого основное внимание будет уделено методам проектирования, представляющим интерес как в настоящем, так и в будущем. К ним относятся *архитектура, ориентированная на обслуживание* (АОС), *инверсия управления/внесение зависимостей*, а также *аспектно-ориентированное программирование* (АОП).

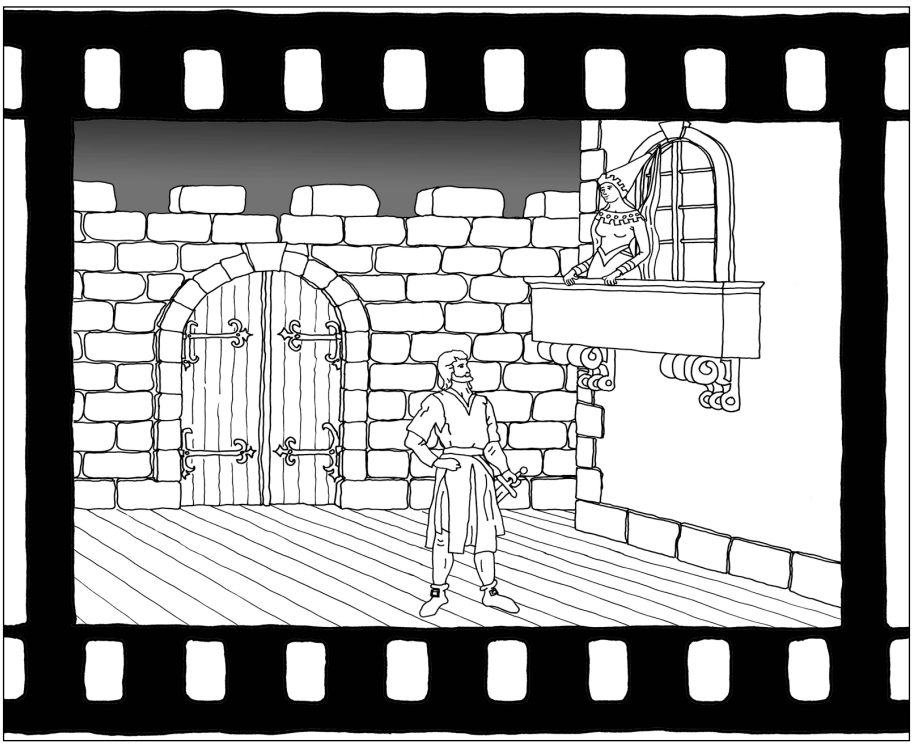
Часть IV

Что дальше

Глава 10. Рекомендуемые методы проектирования

Глава 11. Акцент на пользовательском интерфейсе

В этой части основное внимание уделяется другим методам проектирования, на которые следует обратить внимание, а затем и применять на практике. Кроме того, в ней показано, как действовать на уровне представления, когда требуется навести мосты между ним и моделью предметной области, и как организовать тестирование пользовательского интерфейса на стадии разработки. Эта часть почти полностью написана приглашенными авторами.



В этой части рассматриваются другие методы проектирования. Это не означает, что они чем-то лучше, но они так же отличаются от всего рассмотренного выше, как кино от театра

Глава 10

Рекомендуемые методы проектирования

Итак, мы уделили достаточно внимания применению основных принципов ППО. Для этой цели мы написали немало тестов, тщательно разработали модель предметной области и сделали ее сохраняемой с помощью объектно-реляционного преобразователя. Что еще осталось сделать?

По этому поводу уместно привести следующую историю об одном руководителе ведущей в своей отрасли многонациональной компании. Он назначил совещание после того, как прибыль, качество и прочие показатели компании оказались как никогда высокими. Совещание он открыл следующими словами: "Я хотел бы, чтобы все мы задумались на минуту над тем, чего мы достигли", а минуту спустя он продолжил совещание, говоря о насущных проблемах, критикуя и т.д. — в общем, как это обычно бывает на деловых совещаниях.

Итак, подумав немного над тем, чего мы уже достигли, зададимся вопросом: на чем нам теперь сосредоточить основное внимание?

Разумеется, вопросов, ожидающих рассмотрения, можно насчитать не один десяток. Прежде всего, мы рассмотрели лишь основы построения модели предметной области, оставив в стороне детали. Но ведь начало было положено совсем неплохое, и теперь предстоит решать вопросы производительности, параллельной обработки, безопасности и прочие. Их решение я оставляю вам, читатель, в качестве домашнего задания, поскольку надеюсь, что вы уже горите желанием опробовать освоенные методы ППО на практике, если уже не пытались сделать это.

Поэтому эту главу мне бы хотелось начать с краткого обсуждения перспектив возможного расширения использовавшихся до сих пор принципов ППО, сосредоточив основное внимание на шаблоне Bounded Context (Ограниченный контекст) [Evans DDD]. После этого мы рассмотрим различные методы проектирования, которые в настоящее время достойны внимания. К их числу относятся ориентация на службы, инверсия управления/внесение зависимостей и аспектная ориентация.

Я попросил двух моих коллег написать об этих методах, включив данный материал в соответствующие разделы этой главы, но первый ее раздел я написал сам.

Proekt_book.indb 391 27.08.2007 21:33:20

Все дело в контексте

Было бы вполне уместно обсудить в деталях значение контекста. Так, если разработчики рассматривают и применяют шаблоны проектирования [GoF Design Patterns], не имея предварительного опыта работы с ними, то они обычно забывают о контексте, для которого пригоден тот или иной шаблон.

Даже самая прекрасная идея, удачное решение или что-либо другое должно быть перенесено в подходящий контекст, иначе одно будет противоречить другому. Третьего, как говорится, не дано.

Сделаем небольшое отступление, прежде чем сосредоточить основное внимание на самом контексте. Прежде всего, рассмотрим слои и разделы.

Слои и разделы

В 1994 году вышла в свет книга *Object-Oriented Analysis and Design* [Booch OOAD] Гради Буча. Благодаря ей я смог лучше разобраться в понятиях, представленных в книге *Object Modeling Technique* [Rumbaugh OMT] Джима Рамбо, вышедшей несколько лет спустя. К числу таких понятий относятся слои и разделы. Понятие слоев было хорошо мне знакомо, чего нельзя было сказать о разделах, особенно в интерпретации Буча. Он рассматривал разделы как срезы иного измерения, чем слои.

Для большей конкретности рассмотрим пример с двумя типичными слоями: пользовательским интерфейсом и моделью предметной области. Двумя возможными разделами в данном примере могут быть подсистема регистрации заказов и подсистема административного реагирования на жалобы пользователей, причем у обоих разделов может быть свой пользовательский интерфейс и модель предметной области.

В настоящее время, как и в прошлом, разделению на слои уделяется намного больше внимания, чем разделам. Разделение на слои сыграло важную роль в “разделении ответственности”. Вероятно, вы уже догадались, что речь идет о соблюдении принципа *единственной ответственности* (ПЕО) [Martin PPP].

В известном смысле такой практически полный акцент на разделении на слои несколько изменился благодаря ППО — по крайней мере, по сравнению с тем, как в прошлом строго соблюдалось (и мной в том числе) разделение на слои [Nilsson NED]. (На самом деле мы по-прежнему уделяем немало внимания отделению инфраструктуры от модели предметной области.)

> ***На заметку***
>
> О разделении на слои в связи с ППО речь уже шла в главе 4.

В то же время все большее понимание находит то обстоятельство, что невозможно создать одну крупную модель предметной области для крупномасштабных решений, чтобы она не оказалась, как правило, сложной, неэффективной и дорогостоящей. Что

же касается возросшего интереса к разбиению на разделы, то он подогревается, в основном, ориентацией на службы, поскольку она проясняет весь этот вопрос в целом.

В связи с возросшим интересом к разбиению на разделы интерес к разделению на слои может упасть. Впрочем, это не столь важно для проектирования со строгим разделением на слои и мелким разбиением на разделы.

А теперь рассмотрим более подробно основания для разбиения на разделы.

Основания для разбиения на разделы

Если для разделения на слои имеются технические основания, то для разбиения на разделы — организационные. Разумеется, разбиение на разделы может быть сделано и по техническим причинам. Например, определенный раздел может быстро устареть, и его придется впоследствии заменить новой реализацией. Поэтому такая замена должна быть очевидной и просто осуществимой, когда для нее настанет время.

Но все же организационные основания очень важны для разбиения на разделы, в частности потому, что одна группа специалистов отвечает за одну часть работы, а другая — за другую. Это согласуется с выводами, которые я сделал из дискуссии с моими коллегами по поводу мотиваций для разделения на слои. Ни один из них не считает, что разделение на слои годится для организации группы разработчиков. Но именно это я делал время от времени и, как мне кажется, довольно успешно. Они же имели в виду, что группу разработчиков лучше организовать по разделам, и тогда разделение на слои окажется менее выраженным в каждом разделе. Разумеется, необходимо принимать во внимание и то обстоятельство, что люди обладают разным опытом и преследуют разные цели. Ведь нельзя же создать один крупный отдел разработки пользовательских интерфейсов для всех приложений, другой отдел для разработки модели предметной области и еще один раздел для разработки базы данных. Если выбирать между техническими задачами и задачами предметной области, то приоритет следует отдавать последним. Об этом стоит всегда помнить при организации группы разработчиков.

Рассматривая необходимость разбиения на разделы, следует учитывать и объемы проектных работ. Так, для небольшой системы может оказаться вполне достаточно и одного раздела, тогда как для крупной системы очень важно построить подходящую структуру разделов.

Подводя итог, следует сказать, что сначала следует уделить внимание разделам, а затем внутри самих разделов рассматривать слои, но не наоборот.

А теперь посмотрим, как все сказанное выше согласуется с ППО, что приводит нас к шаблону Bounded Context [Evans DDD].

Ограниченный контекст

Очень часто возникает вопрос: “Как бороться со сложностью?” На этот вопрос часто дается следующий ответ: “Разделяй и властвуй”. А для того чтобы разделять и властвовать, нужно пользоваться иерархиями.

ППО имеет непосредственное отношение к иерархиям. Мы уже не раз упоминали выше об одном из наиболее полезных для этой цели инструментов — шаблоне Aggregate, который представляет собой довольно крупную структурную единицу. Еще одной структурной единицей являются объекты-сущности и объекты-значения (соответствующие им шаблоны Entity и Value Object подробно описаны в книге [Evans DDD]).

Если мы будем двигаться в противоположном направлении, т.е. вверх от агрегатов, то обнаружим, что ограниченные контексты отличаются от агрегатов и в большей степени похожи на подсистемы или подмодели.

Шаблон Bounded Context построен по принципу максимальной ясности контекста, в котором применяется модель. Кроме того, в пределах шаблона Bounded Context необходимо очень строго придерживаться модели и ее концепций, а все, что находится за его пределами, не имеет особого значения.

В узком смысле веским основанием для разделения на несколько ограниченных контекстов может стать возможность анализировать все детали в одной модели, соблюдая строгость ее структуры. А в более широком смысле — возможность мыслить категориями "крупных абстракций", которые нетрудно понять и рассматривать в самых общих чертах.

Одной из сильных сторон шаблона Aggregate являются его четкие границы. Он упрощает разработку, уменьшает связывание и делает связи более явными. Такие же границы, но на более высоком уровне можно обнаружить и у шаблона Bounded Context.

На заметку

Несмотря на большую ценность границ, они обходятся не бесплатно. С точки зрения обмена данными и производительности они требуют значительных издержек. Поэтому, как обычно, следует стремиться к простоте, а усложнять проектирование стоит лишь в том случае, если выгоды перевешивают издержки.

Если связать все сказанное выше с примером заказа на закупку (см. предыдущие главы), то в нем можно выделить один ограниченный контекст для составления заказов и другой — для отгрузки товаров. Если же перейти от этого небольшого примера к более сложной ситуации, встречающейся в реальной жизни, то для разных отделов организации могут быть выделены отдельные бизнес-процессы, и поэтому разделение оказывается целесообразным с технологической точки зрения.

Взаимосвязь между ограниченными контекстами и разделами

Итак, мы рассмотрели разделы и ограниченные контексты, но это не одно и то же, хотя вполне возможно, что ограниченный контекст может совпадать с типичным разделом. С другой стороны, ограниченный контекст может состоять из нескольких разделов, но вряд ли наоборот.

Расширение масштабов проектов ППО

Типичным основанием для применения ограниченных контекстов служит возможность расширения масштабов ППО до более крупных контекстов. Вместо того чтобы иметь одну крупную модель предметной области, над которой работает большая группа разработчиков, модель разделяется на несколько мелких моделей и в итоге становится более понятной, а разработчикам легче сосредоточить усилия на своих моделях.

Кроме того, ограниченные контексты позволяют сохранить универсальный язык от нечеткости понятий и расплывчатости определений. Вместо этого в употребление входит несколько универсальных языков (по одному на каждый ограниченный контекст). С одной стороны, это недостаток, а с другой — каждый такой язык становится более эффективным и точным. Таким образом, преимуществ у данного подхода оказывается больше, чем недостатков.

Если концепции исходной модели впоследствии приобретают другое значение, а универсальный язык теряет свою четкость и ясность, это может служить явным признаком того, что одна модель фактически превратилась в две. Вместо того чтобы както противостоять этому явлению, благоразумнее способствовать его проявлению и определить два разных ограниченных контекста.

А теперь сместим немного акцент и рассмотрим еще одно основание для разбиения моделей предметной области на разделы.

Основание для разбиения модели предметной области на разделы — ориентация на службы

Если уже имеется крупная модель предметной области, то зачем разбивать ее на разделы? Этот вопрос не нов, однако благодаря ориентации на службы он стал еще более очевидным по сравнению с другими решениями для распределенных систем, включая EJB и COM+. В таких системах разбиение на разделы было полезным, но чаще всего оно не соблюдалось, тогда как в системах, ориентированных на службы, оно стало нормой.

На заметку

Я отдаю себе отчет в том, что многие сторонники ориентации на службы вообще не берут в расчет модель предметной области, но начинают проектирование непосредственно с интерфейсов. В проектировании всегда присутствует немного и того, и другого.

Эта книга не посвящена интеграции и обмену сообщениями, а только архитектуре и проектированию *одного* приложения или *одной* службы. Тем не менее в следующем разделе, написанном Уди Даханом, основное внимание будет уделено введению в архитектуру, ориентированную на службы (AOC — Service Oriented Architecture (SOA)).

Введение в AOC

Уди Дахан

ППО и AOC являются относительно новыми понятиями, причем первое определено более полно, чем последнее. И хотя некоторые считают, что понятие AOC известно еще со времен CORBA, оно никогда не рассматривалось в данной отрасли "только как архитектура".

Что собой представляет AOC

Несмотря на то что был предложен целый ряд определений AOC, ни одно из них не было признано самым лучшим. Большинство определений основано на том, что системы состоят из независимых служб. Но не совсем ясно, из чего состоит сама служба. Одно из определений, которое нельзя назвать несостоятельным, обозначает AOC как способ разработки программного обеспечения. В частности, AOC имеет отношение к разработке систем, которые являются или же могут быть распределенными. Заявление о том, что распределение является главным вопросом архитектуры прикладных систем, считается одним из основополагающих для AOC.

Назначение AOC

Программное обеспечение, предназначенное для выполнения в распределенной среде, должно разрабатываться по-другому. В этом состоит фундаментальное отличие от представлений о распределенных объектах, где приложения должны использовать нелокальные ресурсы "прозрачно", даже не подозревая о месте их расположения. Недостатки распределенных объектных архитектур кроются, главным образом, в производительности. Семантика работы с локальными и удаленными ресурсами должна быть разной. Этот с трудом усвоенный урок превратился в следующий девиз разработки распределенных систем: "Блочность превыше многословности".

Блочные интерфейсы, называемые также крупноструктурными, требуют большего количества операций при одиночном вызове, чем мелкоструктурные интерфейсы, требующие многих вызовов для выполнения такого же количества операций. Явное согласование и блокировка ресурсов, которые необходимы для мелкоструктурных интерфейсов, полностью инкапсулированы в одном блочном вызове. В итоге клиентский код оказывается менее связанным с реализацией серверного кода.

В действительности на развитие архитектуры распределенных систем повлияли новые идеи и отрезвляющее действие реальности — производительность. Несмотря на что закон Мура продолжает действовать уже много лет подряд и вряд ли перестанет действовать в обозримом будущем, область разработки крупномасштабных систем по-прежнему испытывает затруднения, связанные с производительностью. Но особый интерес представляет не столько сама производительность как некий статический показатель, сколько масштабируемость как динамический показатель произ-

водительности при переменной нагрузке, который разработчики архитектуры стремятся сделать максимально высоким.

AOC является следующим шагом на пути развития архитектуры, позволяющим создавать новые системы и эксплуатировать старые на основе масштабируемых решений.

Отличия AOC

Если в процессе разработки распределенные системы проектируются слоями и развертываются на физических ярусах, то в AOC эти понятия объединены в "службы", которые одновременно являются единицами проектирования и развертывания. AOC отличается от настоящей практики разработки распределенных систем еще и способом кодирования связи между службами. Если на связь между слоями или ярусами системы не существует ограничений, то связь между службами должна осуществляться по образцам обмена сообщениями, определенным по некоторой схеме. Кроме того, существует теоретическая и практическая поддержка только одностороннего асинхронного обмена сообщениями для связи между службами, хотя она пока еще не нашла широкого признания в данной отрасли.

На заметку

Следует заметить, что "схема" совсем не обязательно подразумевает XML, а скорее, ряд сообщений, которые циркулируют между службами.

Если XML (и протокол SOAP в более широком смысле) не является предварительным условием для AOC, то расширяемость XML упрощает управление версиями схемы.

Что такое служба

Помимо того что службы являются одновременно единицами проектирования и развертывания, они, как ни странно, довольно неясно определены, принимая во внимание большой интерес к AOC. Вместо определения самой службы в большей части литературы на данную тему описываются свойства служб. Примером тому могут служить четыре следующих принципа ориентации на службы:

- службы автономны;
- службы имеют явные границы;
- службы раскрывают схему и контракт, но не класс или тип;
- совместимость служб определяется устанавливаемыми правилами.

Несмотря на то что все эти свойства действительно направляют проектирование масштабируемых служб (которые более подробно будут рассмотрены далее в этом

разделе), они, очевидно, малопригодны в качестве руководящих принципов разработки распределенных систем. Разработчики архитектуры, только начинающие внедрять в свои проекты методы, ориентированные на службы, недостаточно осведомлены о тех элементах, которые должны быть объединены в службы, о выявлении и определении границ обслуживания, а также о том, как реализовывать службы, используя классические объектно-ориентированные шаблоны. Помимо этого, технологии, связанные с "чисто" архитектурными аспектами AOC, продолжают оставаться в состоянии постоянного изменения, а различные предложения поставщиков оказываются не более чем переупаковкой текущих программных продуктов. Поэтому и не удивительно, что шумиха, поднятая вокруг AOC, постепенно стихает.

Что входит в службу

Поднятый выше вопрос относительно тех элементов проектирования, которые должны быть объединены в единую службу, не является новым для большинства разработчиков приложений их архитектуры. Этот вопрос уже возникал при внедрении компонентов и объектов. Объединение нескольких объектов или классов в единый компонент кажется вполне логичным, чего нельзя сказать о компонентах и службах. Ведь служба является *не* столько совокупностью компонентов, сколько фундаментальным шагом в направлении изменения наших представлений о том, как следует структурировать программное обеспечение.

Принимая во внимание тот факт, что связь между службами основывается на обмене сообщениями, мы обнаруживаем, что службы состоят из разнотипных элементов, а не только из "компонентов" и "объектов". Для раскрытия функциональных возможностей служб используется схема сообщений. Такая схема состоит из нескольких элементов, к самым основным из которых относятся "типы данных" (например, `Customer`), определяющие объекты-сущности; сообщения, определяющие операции, которые можно выполнять (но не обязательно над каждым объектом-сущностью в отдельности); а также контракт на службу, устанавливающий взаимодействия с определенными сообщениями (например, запрос на изменение клиента (`CustomerChangeRequest`) приводит к ответу об изменении клиента (`CustomerChangeReply`)). Ясно, что схема сообщений не является ни объектом, ни компонентом, хотя и необходима как составляющая службы.

Еще одной составляющей службы, часто упускаемой из виду, является обслуживающий хост. Служба не может предоставлять свои функциональные возможности в отсутствие определенного рода среды, в которой она работает. Так, в качестве обслуживающего хоста Web-служб может быть выбран IIS, Apache, WebSphere или другой Web-сервер. А другие службы могут размещаться в среде COM+, Windows Service или даже в простом консольном приложении. Несмотря на наметившуюся в данной отрасли тенденцию к переходу на независимые от транспорта службы, некоторые детали реализации хостов (например, протокол) оказывают существенное влияние на службы. Так, HTTP или другие протоколы типа TCP полностью ориентированы на

установление соединения. В распределенных системах, где отдельные элементы могут выйти из строя, перезагружаться или просто медленно реагировать, протоколы с установлением соединения способны вызывать простои, что сказывается на производительности. А протоколы без установления соединения (например, UDP) вызывают другого рода затруднения, связанные с отсутствием встроенных механизмов, выявляющих, что сообщение не достигло места своего назначения. Как бы там ни было, но хост является важной составляющей любой службы.

Помимо рассмотренных выше нетипичных элементов, службы, безусловно, должны состоять из компонентов или объектов. Но для того чтобы эти составляющие службы удовлетворяли четырем перечисленным выше руководящим принципам AOC, требуется правильное понимание значения этих принципов. Поэтому мы рассмотрим их более подробно.

О значении четырех руководящих принципов AOC

Два первых принципа (автономность и явные границы службы) тесно взаимосвязаны. Каждая служба должна быть независима от остальных служб, и кроме того, должно быть ясно, где оканчивается одна служба и начинается другая. Вследствие этого аварийный сбой одной службы приводит к аварийному сбою другой службы. Следовательно, на самом элементарном уровне различные службы должны работать в разных процессах, чтобы поток не входил в другую службу в результате вызова метода. А на более глубоком уровне одна служба не должна блокировать ресурсы другой службы — выполнение транзакций (типа АНИД) должно быть ограничено областью действия одной службы.

На заметку

Термин “транзакция” первоначально служил для описания порядка работы с реляционными базами данных, а четыре свойства транзакций обозначались сокращением АНИД (ACID) — атомарность, непротиворечивость, изоляция, долговечность. Типичной реализацией базы данных, удовлетворяющей этим требованиям, являются соответственно строки, страницы и таблицы, ограничиваемые областью действия транзакции.

На первый взгляд, третий принцип (раскрытия схемы, но не класса) имеет отношение к способности к взаимодействию, т.е. для связи между службами в качестве наименьшего общего знаменателя используется сетевой вариант XML. Но на самом элементарном уровне данный принцип имеет отношение к разделению ответственности. При этом класс, реализующий требуемое поведение, отделяется от внешних клиентов, которые его “вызывают”. Этим данный принцип отличается от организации удаленных вызовов в Java RMI, COM или .NET, где в клиентском коде должны присутствовать ссылки на серверные типы данных. При использовании схемы

приходится выполнять явное преобразование из форматов внешних сообщений во внутренние вызовы классов. Это дает нам возможность изменять внешнюю реализацию службы до такой степени, чтобы вызывать отдельный класс или интерфейс, не оказывая влияния на внешних клиентов.

У третьего принципа имеется и более глубокое значение, связанное с управлением версиями. Если типы данных могут образовывать иерархии таким образом, чтобы один тип был разновидностью другого типа, то практика доказывает, что управление версиями типов может оказаться весьма затруднительным, что обычно ассоциируется с кошмаром, связанным с управлением библиотеками DLL. Одна из целей управления версиями служб состоит в том, чтобы избежать обновления уже развернутого клиентского кода. Это означает, что развернутые клиенты будут и далее связываться с теми же конечными пунктами, что и раньше. Преимущество схем, в отличие от типов данных, заключается в том, что они могут быть совместимы — даже если отличаются друг от друга. Именно для этого и требуется свойство расширяемости XML.

Например, после развертывания версии 1 службы и клиента может потребоваться переход к версии 2 этой же службы, чтобы развернуть ряд новых клиентов (версии 2), требующих более развитых функциональных возможностей от службы. Очевидно, что было бы желательно ни нарушать работу уже имеющихся клиентов вследствие таких изменений ни обновлять этих клиентов. А для этого версия 2 схемы службы должна быть совместима с версией 1, и решить данную задачу без связи по схеме будет намного труднее и неудобнее для разработчика. Достаточно лишь представить себе необходимость отслеживать до конца сигнатуру каждого метода типа “`params object[] extraData`”.

Четвертый принцип AOC (совместимость на основе устанавливаемых правил) еще больше подчеркивает значение двух первых принципов — служба сама решает, должна ли она реагировать на данный запрос, но на более высоком уровне, ожидая передачи метаданных вместе с запросом. Эти метаданные и составляют правила. В большинстве видов представления правила описываются на примерах возможностей шифрования или аутентификации. В основу данного принципа положено разделение ответственности. Схема, определяющая логическую связь, не зависит от правил шифрования. Так, если имеется информация, которую требуется передать из одной службы в другую, не выражая ее в схеме, это можно сделать, в частности, с помощью правил.

Еще раз о том, что такое служба

Итак, рассмотрев свойства и принципы организации служб, попробуем окончательно выяснить, что же собой представляет служба. Когда-то мы разрабатывали приложения, с которыми взаимодействовал пользователь. А ныне мы можем назвать такие приложения “службами”, причем они в состоянии взаимодействовать не только с пользователями, но и с другими “службами”. Следует заметить, что распределенные приложения уместнее называть не “распределенными службами”, а скорее, смоделированными в виде целого ряда работающих вместе служб по следующему эмпиричес-

кому правилу: один исполняемый файл на каждую службу или же один корневой каталог Web-приложения на каждую службу. Обратная сторона этой медали связана с базами данных. Если наличие одной базы данных на каждую службу не является обязательным, то у служб не должно быть общих баз данных. На самом деле общие базы данных выполняют роль черного хода для служб, позволяя обходить проверку достоверности и соответствующую логику, а по существу, нарушая автономность службы и повышая степень связывания служб.

Место объектной ориентации в AOC

А теперь, когда стало ясно, что приложение на самом деле является (частично или полностью) службой, необходимо выяснить, каким образом объектная ориентация как общепринятый практический прием разработки приложений вписывается в AOC. Несмотря на заявления о том, что AOC — это следующая стадия разработки приложений, по существу, заменяющая объектную ориентацию, ничего подобно до сих пор не произошло. Ведь для перехода типичной разработки от процедурного к объектно-ориентированному программированию потребовалось более двух десятков лет. Поэтому вряд ли переход от объектной ориентации к AOC произойдет быстрее, если он вообще станет возможным.

В действительности AOC и объектная ориентация действуют на разных уровнях. Если объектная ориентация имеет отношение к проектированию и программированию одной единицы развертывания, то AOC — к решению, состоящему из нескольких единиц развертывания. AOC не предполагает новые или альтернативные способы проектирования или программирования внутренней логики данного приложения или службы. Нет оснований полагать, что AOC и объектная ориентация должны стать совместно используемыми методиками. Вопрос, который ставит AOC при разработке объектно-ориентированных приложений, состоит в том, как такие приложения будут принимать и реагировать на сообщения, поступающие из других источников, кроме самого пользователя, — и даже параллельно с действиями пользователя. Тот факт, что приложение может принимать запросы и в то же время выдавать собственные запросы другим службам, заставляет прийти к выводу, что такие взаимодействия уже не следуют классическому образцу “клиент-сервер”.

Архитектура “клиент-сервер” и AOC

Ранее бо́льшая часть разработок основывалась на архитектуре “клиент-сервер”. В частности, клиентское программное обеспечение “посылало запрос” на сервер базы данных и получало от него ответ. В дальнейшем на смену архитектуре “клиент-сервер” пришла трехъярусная (или N-ярусная) модель, а вместе с ней появился и сервер приложений. И в этом случае запросы пересылались в одном направлении, а ответы — в другом.

По мере развития систем и взаимосвязей между ними N-ярусная концепция стала расплываться, как только серверы приложений начали принимать запросы от других

аналогичных серверов как от своих клиентов. Упорядоченность, присущая многоярусной архитектуре, уже перестала наблюдаться на практике. И наконец, клиентское программное обеспечение начало воспринимать события от серверов асинхронно, по существу, реагируя на запросы, для чего оно, вообще говоря, не предназначалось.

Помимо теоретических положений AOC, устанавливающих, что клиентское программное обеспечение должно быть также разработано в виде службы, существует реальная потребность в клиентском коде, способном принимать сообщения. Рассмотрим случай, когда выполнение задания на сервере отнимает очень много времени — больше, чем длится простой по протоколу HTTP, возможно, даже не один день. Это типичное задание, требующее вмешательства человека. Если клиент отправит на сервер запрос о завершении задания, такое требование к фоновой задаче клиента может оказать непомерную нагрузку на сервер. Если же организовать при выполнении той же самой фоновой задачи прослушивание сообщений, уведомляющих о завершении задания, то сервер будет избавлен от подобной нагрузки. Такое решение вполне осуществимо и реализуется с помощью современных технологий.

Подобным образом клиент может принимать разнотипные сообщения, по существу, становясь службой. В действительности, обмен сообщениями лучше описывается в виде диалога между инициатором диалога и его целевым объектом, чем в виде пар запросов и ответов между клиентом и сервером. При этом не существует ограничений на то, какие именно службы могут начать диалог и какие из них — продолжить его.

Односторонний асинхронный обмен сообщениями

Как было показано выше, при продолжительном времени реакции приходится отделять ответное сообщение от потока запрашивающих сообщений. Но может ли инициирующая служба знать, сколько времени потребуется целевой службе для ответа? Даже если предыдущий ответ пришел очень быстро, это еще не гарантирует быструю реакцию в дальнейшем. По существу, если бы все взаимодействия со службой основывались на таком принципе, клиенты данной службы не были бы подвержены влиянию нагрузки на саму службу. У данного принципа имеются и другие преимущества.

Если инициатор диалога больше не требует синхронного ответа, то целевой объект диалога может свободно выполнить обработку запроса в любое удобное для него время. Так, если целевая служба сильно нагружена, она может запланировать обработку новых сообщений на более отдаленный строк. Иными словами, мы можем программно выравнивать нагрузку во времени. В частности, мы можем написать код, принимающий во внимание характер запроса при планировании его обработки на будущее.

В соответствии с синхронным принципом запросов-ответов инициатор диалога может лишь принимать ответ от целевого объекта диалога. Теперь, когда инициатор диалога прослушивает ответы, единственное ограничение связано с тем, что ответ может возвращать только целевой объект диалога. Это дает целевой службе возможность направлять ответы другим службам.

У одностороннего обмена сообщениями по асинхронному принципу имеется немало преимуществ. К ним, в частности, относится повышение гибкости и масштабируемости решения. Разумеется, все эти преимущества даются не бесплатно, но, как будет показано ниже, эта плата не столь уж велика.

Как AOC улучшает масштабируемость

Используя односторонний асинхронный обмен сообщениями, служба способна усреднять нагрузку, ставя запросы в очередь. В итоге производительность служб не падает при увеличении нагрузки на них, поскольку для этого не требуются такие ресурсы, снижающие производительность, как потоки. Сохраняя использование ресурсов на уровне, близком к постоянному, при любой дополнительной нагрузке, службы, разработанные по принципу одностороннего асинхронного обмена сообщениями, обеспечивают лучшие показатели производительности, как показано на рис. 10.1.

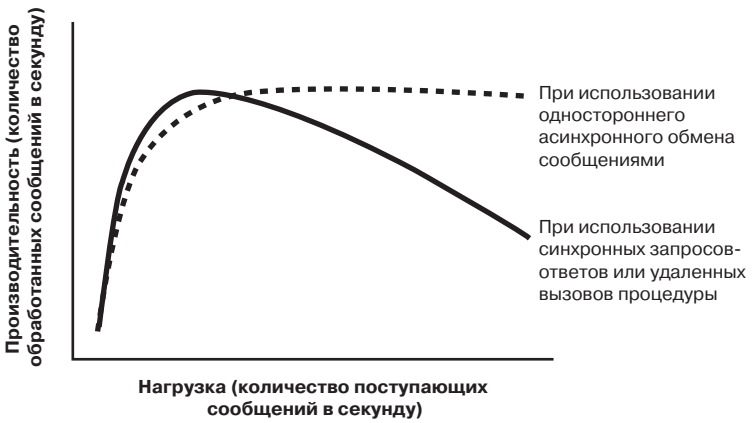


Рис. 10.1. Зависимость производительности от нагрузки при одностороннем асинхронном обмене сообщениями и синхронных запросах-ответах или удаленных вызовах процедуры

Следует заметить, что при малой нагрузке время обработки одного сообщения при одностороннем асинхронном обмене может оказаться больше, чем при использовании классического удаленного вызова процедуры (RPC). Это и есть плата за более высокую производительность при большой нагрузке.

Структура службы AOC

На самом высоком уровне структура службы, реализующей принцип одностороннего асинхронного обмена сообщениями, может выглядеть так, как показано на рис. 10.2.

В этой структуре принятое сообщение не обрабатывается сразу, а помещается в очередь, чтобы свести к минимуму число потоков, одновременно открытых сервером приложений, независимо от нагрузки или времени обработки. Затем в назначенный

заранее срок поток обработки выбирает сообщение из очереди и обрабатывает его. Если такая обработка требует отправки сообщений другим службам, ответы будут получены таким же образом, как и запросы. Более того, потоку обработки не придется ждать этих ответов, но он может перейти к обработке новых сообщений, а состояние обработки предыдущих сообщений сохраняется в определенном запоминающем устройстве, чтобы его можно было восстановить при поступлении ответов. Следует также заметить, что потоки могут быть заменены процессами, если используется внепроцессная очередь, аналогичная MSMQ, чтобы лучше обеспечить горизонтальное масштабирование системы благодаря выполнению процессов на отдельных машинах.

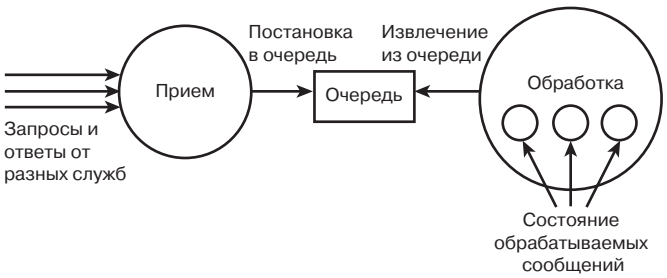


Рис. 10.2. Гипотетическая структура службы на самом высоком уровне с использованием одностороннего асинхронного обмена сообщениями

Взаимодействие службы с другими службами

Рассмотрим на конкретном примере особенности обработки запроса в службе, включая получение информации от других служб. (Примечание: если на приведенных ниже рисунках одна горизонтальная стрелка оказывается ниже другой, то нижняя стрелка обозначает событие, наступившее позднее, чем событие, обозначаемое верхней стрелкой.)

Сначала в рассматриваемую службу поступает запрос, затем он помещается в очередь и далее обрабатывается. Для обработки этого запроса требуется получить информацию от трех других служб. После отправки запросов другим службам сохраняется состояние обработки исходного запроса (в ожидании ответов 1, 2 и 3), как показано на рис. 10.3.

Ответ, поступивший от службы #2, ожидает в очереди до тех пор, пока до него не дойдет обработчик сообщений, который выбирает состояние исходного запроса из запоминающего устройства, восстанавливает и обновляет его (в ожидании ответов 1 и 3). А поскольку обработка исходного запроса не завершена, в данный момент завершается обработка ответа 2 (рис. 10.4).

Ели же очередной запрос поступит до завершения обработки предыдущего запроса, он не окажет влияния на обработку исходного запроса. В то же время может на-

чаться отдельная обработка этого запроса и затем отправлены запросы любым другим службам аналогичным образом. Обработка нескольких запросов одними и теми же ресурсами может перемежаться (рис. 10.5).

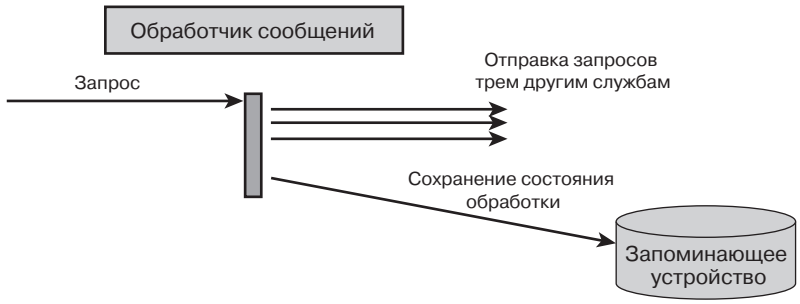


Рис. 10.3. Отправка запросов другим службам и сохранение состояния обработки

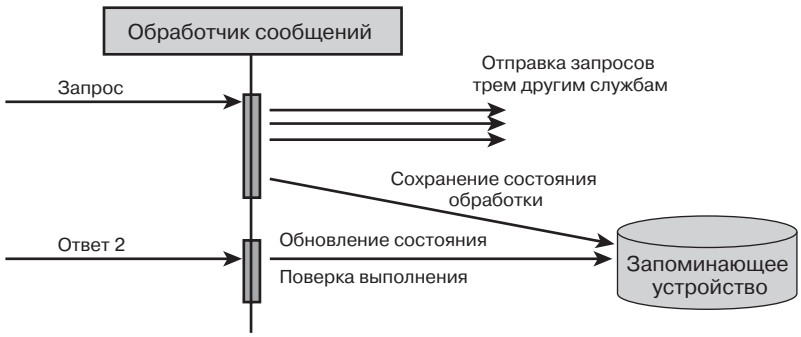


Рис. 10.4. Обработка одного ответа

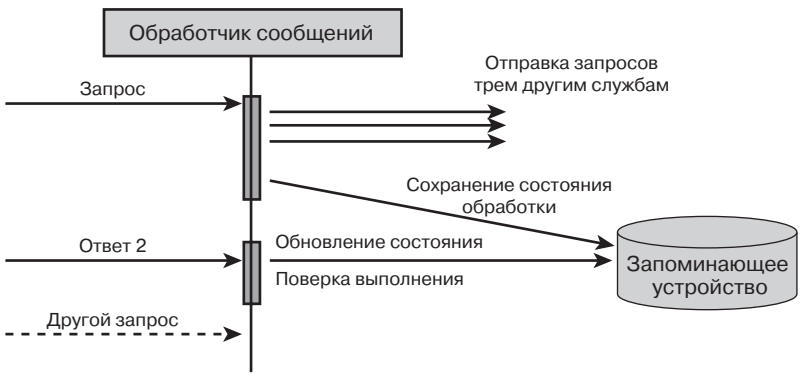


Рис. 10.5. Очередной поступающий запрос не будет оказывать влияния на первый запрос, который находится в состоянии обработки

Ответ, поступивший от службы #3, обрабатывается таким же образом, как и ответ от службы #2. А поскольку получены не все необходимые ответы, инициатору диалога по-прежнему не посылается ответ на исходный запрос (рис. 10.6).

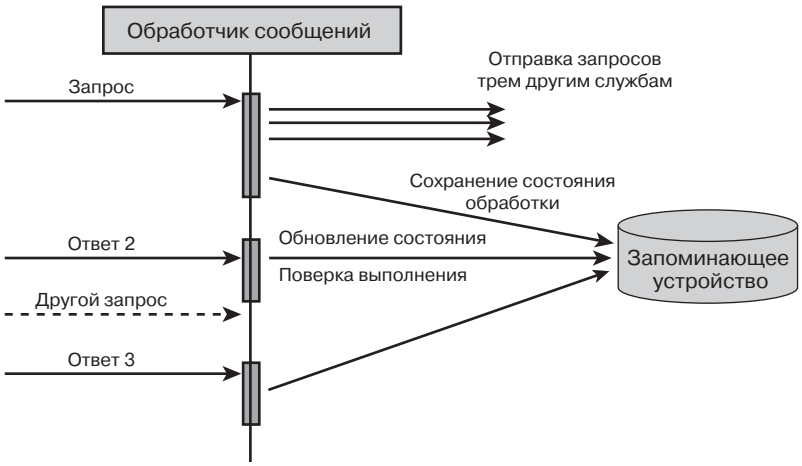


Рис. 10.6. Поступает очередной ответ, но осталось получить еще один ответ

И наконец, обработка исходного запроса может быть завершена только после того, как поступит ответ от службы #3. Обновив состояние исходного запроса в запоминающем устройстве, служба теперь знает, что все ответы поступили и далее можно отправить ответ инициатору исходного запроса (рис. 10.7).

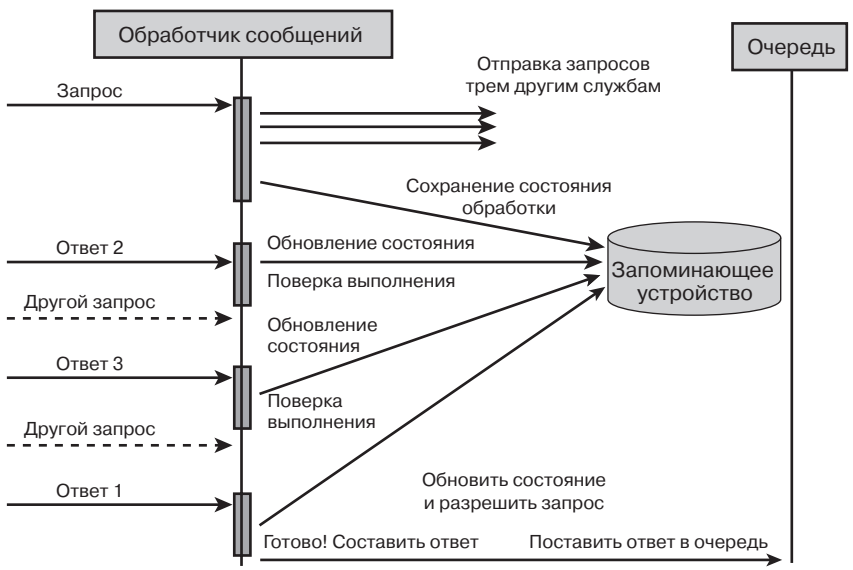


Рис. 10.7. Поступает последний ответ на первый запрос, и результат может быть помещен в очередь

Следует заметить, что логика обработки сообщений не требует отправки ответа инициатору исходного запроса непосредственно, но допускает постановку этого ответа в очередь. Преимущество обработки сообщений отдельно от возврата ответов заключается в снижении нагрузки на логику обработки сообщений. А с точки зрения разделения ответственности порядок и время возврата ответов оказывается вне компетенции логики обработки сообщений.

Несмотря на то что мы приняли во внимание тот факт, что службам может потребоваться некоторое время, чтобы отреагировать на запросы, мы еще не рассмотрели ситуацию, когда возврат ответа не предполагается. В текущей структуре службы такая ситуация не предусмотрена, поскольку в ней не учитываются, в частности, простои, присущие протоколам с установлением соединений.

AOC и недоступные службы

При проектировании распределенных систем в классическом стиле удаленного вызова процедуры (RPC) попытка соединения с удаленным сервером, который не работает, может привести к появлению исключения. Но в рассмотренной до сих пор структуре службы получение любого исключения или аналогичной реакции вообще невозможно. Поэтому службе, пославшей запрос, нужно каким-то образом дать знать, что обработка этого запроса задерживается. С этой целью можно просто назначить таймер для каждого отправленного запроса. Но это не самая интересная часть задачи.

Намного интереснее выяснить, что следует делать, когда время ожидания запроса истечет? Это еще один аспект проектирования, требующий логики предметной области. Ниже перечислены типичные вопросы, которые призвана разрешить логика предметной области.

- Следует ли возвращать сообщение об ошибке?
- Следует ли возвращать частичный результат?
- Следует ли еще раз пытаться послать ответ?
- Следует ли использовать все эти варианты в определенном сочетании?
- Следует ли использовать то же самое поведение независимо от вида сообщения?

Еще одно преимущество предыдущей структуры службы над проектированием в классическом стиле удаленного вызова процедуры заключается в том, что логика предметной области, выполняемая при недоступности удаленной службы, явно отделена от основной цепи обработки сообщений. При проектировании в стиле удаленного вызова процедуры исключение может появиться в ходе выполнения кода обработки сообщений, а код, который требуется написать для обработки исключения разными способами, описанными в поставленных выше вопросах, может сделать недостаточно ясным назначение логики предметной области.

Как показано на блок-схеме взаимодействия, приведенной на рис. 10.8, код, обрабатывающий простои, отделен от кода обработки сообщений не только в пространстве, но и во времени. А логика обработки запросов и сбоев инкапсулирована.

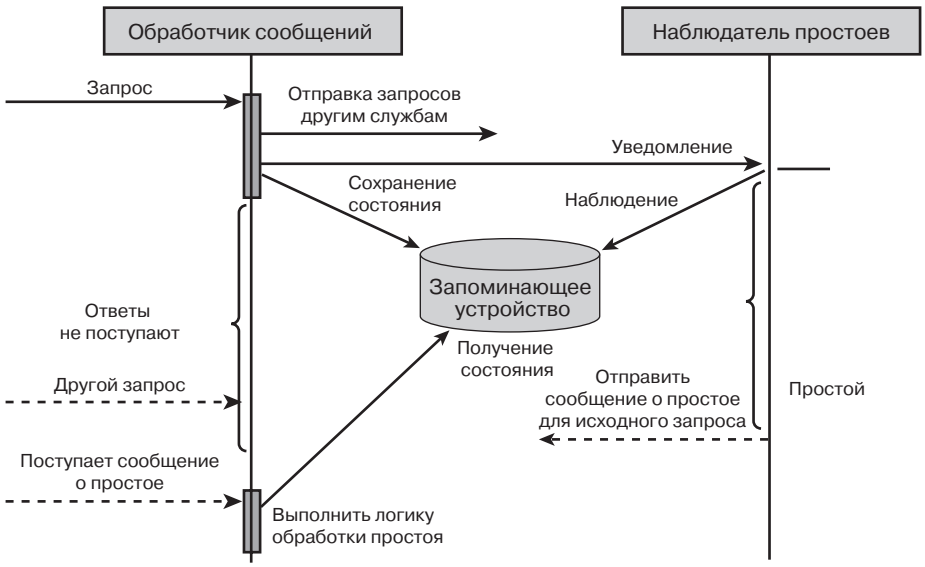


Рис. 10.8. Применение наблюдателя простоев

Сложные процессы обмена сообщениями

При разработке служб с использованием приведенной выше структуры мы можем также составлять сложные и надежные последовательности операций для взаимодействия служб. В частности, при получении запроса A соответствующие запросы посылаются службам Альфа и Бета. Если при получении ответа от службы Альфа или Бета ответное значение оказывается больше некоторого порогового значения, то посылается запрос с пороговым значением службе Гамма. В противном случае посылается запрос с ответным значением службе Дельта. Если же служба Дельта недоступна, должны быть возвращены частичные результаты. А если служба Альфа недоступна или возвращает сообщение об ошибке, аналогичное сообщение должно быть возвращено инициатору запроса и т.д. В течение всего этого процесса могут быть обработаны и другие сообщения, а также начато параллельное выполнение многих других сложных процессов. Это позволяет не только выполнять обработку параллельно, но и эффективно использовать ресурсы — даже под нагрузкой.

Масштабирование служб

Как было показано выше, благодаря реализации одностороннего асинхронного обмена сообщениями улучшается масштабируемость АОС. Но способны ли службы, разработанные подобным образом, эффективно использовать увеличенное число процессоров в одном системном блоке? А как насчет увеличенного числа серверов?

Рассмотрим сначала обработку сообщений. Процесс или поток обработки сообщений не имеет состояния — для каждого запроса он находит подходящее состояние в за-

поминающем устройстве и соответственно обрабатывает результаты. Следовательно, мы можем увеличить число параллельно работающих обработчиков сообщений без ущерба для правильности результатов обработки. А поскольку каждый обработчик сообщений действует независимо от остальных аналогичных обработчиков, то блокировка в оперативной памяти не требуется и данному обработчику не нужно ждать (если, конечно, у него нет сообщений для обработки).

При наличии одного системного блока с несколькими процессорами можно увеличить как число обработчиков сообщений, так и производительность. А при наличии нескольких серверов прием, постановку в очередь и сохранение сообщений можно организовать на одном сервере, разместив дополнительные обработчики сообщений на отдельных серверах.

В конечном итоге преимущество описанного выше варианта горизонтального масштабирования заключается в том, что оно позволяет вводить в действие дополнительные серверы во время выполнения. Контролируя число запросов в очереди, мы можем выяснить, когда система выйдет на максимальную мощность, и затем ввести дополнительные ресурсы, не снижая производительность системы.

Выводы

Главный вывод относительно AOC состоит в том, что данная архитектура представляет собой не более чем логическое продолжение развития наилучших практических приемов разработки распределенных систем. Базовые принципы разделения ответственности и независимости интерфейса от реализации, присущие объектной ориентации, находят свое отражение и в практических методах ориентации на службы. Несмотря на то что в этом введении в AOC были затронуты разные темы, более углубленное их рассмотрение выходит за рамки данного введения. Остается только надеяться, что читатель уяснил, для чего AOC *не* предназначается и какие инструменты не следует использовать для взаимодействия служб. Что же касается внутренней структуры логики службы, то на данный вопрос вряд ли можно вообще дать исчерпывающий ответ, но мне кажется, что Джимми направит вас, читатель, на путь истинный.

Благодарю, Уди!

Тех читателей, которые хотели бы узнать больше об AOC и, в частности, об обмене сообщениями, я отсылаю к отличной книге [Hohpe/Woolf EIP].

Одной из побудительных причин для AOC стало сокращение связывания. Еще одному методу сокращения связывания посвящен следующий раздел.

В этой книге уже не раз упоминалось о внесении зависимостей, я лично применял данный метод только вручную. Для многих внесение в конструктор нужного экземпляра объекта является вполне естественным. Такой метод удобно применять в мелком масштабе и анализировать результаты при выполнении тестов, но в крупном масштабе во время выполнения дело обстоит намного сложнее, особенно это касается повторного использования кода.

Итак, в следующем разделе слово предоставляется Эрику Дёрненбургу, чтобы представить инверсию управления как метод и внесение зависимостей как шаблон.

Инверсия управления и внесение зависимостей

Эрик Дёрненбург

В предыдущих главах было показано, как проектируются и разрабатываются объекты предметной области и соответствующая инфраструктура. А в этом разделе мне бы хотелось представить особый метод проектирования, или шаблон, который может быть использован для компоновки объектов во время выполнения. В его основу положен общий принцип, называемый *инверсией управления* (Inversion of Control), а соответствующий шаблон получил название Dependency Injection (Внесение зависимостей).

Ни один из объектов не изолирован

Практически любой объект в приложении, обеспечивающий функционирование предметной области, зависит от других объектов. Это могут быть объекты, предоставляющие данные или услуги в определенной форме. При проектировании и разработке объектов в стиле ППО мы обычно явно указываем требуемые объекты и зачастую пользуемся вместо них заглушками и имитациями для изоляции тестируемого объекта. Но во время выполнения нашим объектам требуется настоящая реализация, и поэтому все объекты, так или иначе, должны быть представлены в виде экземпляров и связаны.

Наиболее распространенный метод разрешения зависимости одного объекта от другого состоит в том, чтобы возложить ответственность за создание или получение зависимого объекта на объект, имеющий зависимость. Для этого достаточно получить экземпляр объекта, который требуется объекту, имеющему зависимость.

В качестве примера рассмотрим создание калькулятора цен, т.е. объекта, рассчитывающего цены для такого финансового инструмента, как облигация. Разумеется, для расчета цены калькулятору цен требуется сама облигация, но будем считать, что она не должна быть частью состояния калькулятора цен и ее следует передавать лишь при каждом запросе калькуляции цены. Следовательно, класс калькулятора цен будет иметь такой вид:

```
public class BondPricer {
  public double GetPrice(Bond bond) {
    /* выполнить здесь что-то действительно сложное */
  }
}
```

Но цена облигации зависит от других факторов и, в частности, от так называемой кривой скидок. (Знание предметной области в данном примере особого значения не имеет. Даже если функционирование калькулятора зависит от расположения звезд

на небе, это все равно ничего не меняет.) Эта кривая служит для расчета цен самых разных облигаций, оставаясь относительно постоянной. Поэтому она должна быть частью состояния калькулятора цен и храниться в переменной-члене, инициализируемой в конструкторе.

```
// Класс BondPricer (Калькулятор цен облигаций)
private readonly DiscountCurve curve;

public BondPricer() {
  curve = new DiscountCurve();
}
```

Следует заметить, что кривую скидок можно было бы сформировать и по требованию, но тогда решение о полезности получения экземпляра такой кривой по требованию было бы совершенно иным. В любом случае калькулятор цен теперь напрямую зависит от кривой скидок. Вполне возможно, что и сам объект кривой скидок зависит от других объектов, экземпляры который он получает в нужный момент. На первый взгляд, такой подход кажется вполне жизнеспособным, но при более тщательном анализе возникают вопросы, подобные следующему: у нас имеется ряд функционирующих объектов для калькуляции цен облигаций, но насколько гибким и удобным для сопровождения оказывается наше решение?

Допустим, что объект кривой скидок считывает отдельные точки данной кривой из базы данных. Именно это и требовалось пользователям первоначально. Но допустим, что они затем попросили нас ввести свойство, позволяющее анализировать сценарии альтернатив. Для этой цели пользователям нужно иметь возможность указывать точки на кривой в пользовательском интерфейсе. Такая возможность реализуется достаточно просто. Сначала мы создаем интерфейс из класса `DatabaseDiscountCurve` и предоставляем две отдельные реализации: `DatabaseDiscountCurve` и `InMemoryDiscountCurve`, т.е. кривую скидок в базе данных и в оперативной памяти. Затем мы изменяем класс `BondPricer` таким образом, чтобы использовать в нем только методы интерфейса, а следовательно, не связывать его больше с реализацией для базы данных.

Несмотря на то что код расчета в калькуляторе цен облигаций зависит только он интерфейса кривой скидок, нам, к сожалению, придется получить в какой-то момент конкретную реализацию интерфейса `IDiscountCurve` (в данном примере — конструктор класса `BondPricer`). В данный момент нужно решить, какая из двух реализаций нам потребуется, но независимо от нашего решения класс `BondPricer` будет связан как с интерфейсом, так и с выбранной реализацией. На блок-схеме, приведенной на рис. 10.9, представлена структура данного класса, наглядно показывающая, как *образуется* связь между классом `BondPricer` и реализацией `DatabaseDiscountCurve`.

Еще один вопрос, связанный с прямым получением экземпляров объектов, состоит в том, что в рассматриваемом приложении отсутствует единое место для управления формированием кривых скидок, что дает возможность неоднократно использовать

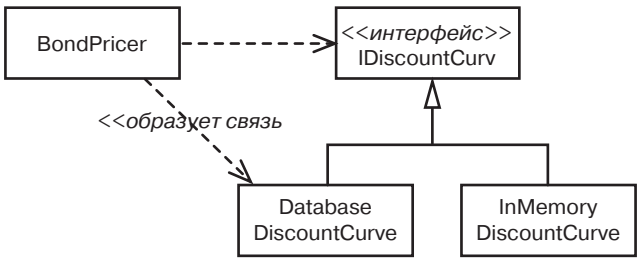


Рис. 10.9. Зависимости, возникающие в результате прямого получения экземпляров объектов

один и тот же интерфейс в разных целях. Решить этот вопрос непросто. Ведь если принять во внимание, что в исходном варианте объект кривой загружает информацию из базы данных, то становится ясно, что воссоздание новых экземпляров может оказать существенное влияние на производительность данного приложения.

Прежде чем переходить к инверсии управления и внесению зависимостей, рассмотрим еще один распространенный подход к решению упомянутых выше вопросов.

Фабрики, реестры и указатели служб

Один из способов решения вопроса связывания объектов состоит в использовании шаблона Factory. В этом случае мы создаем класс фабрики, предоставляющий требуемые объекты (в нашем примере — объект, реализующий интерфейс кривой скидок).

```
public class DiscountCurveFactory {
  public static IDiscountCurve GetCurve() {
    return new DatabaseDiscountCurve();
  }
  }

public class BondPricer {
  private readonly IDiscountCurve curve;

  public BondPricer() {
    curve = DiscountCurveFactory.GetCurve();
  }
}
```

Благодаря такой фабрике калькулятор цен уже не связан тесно с реализацией кривой скидок, чего мы, собственно, и добивались. Но, к сожалению, сам класс фабрики теперь оказывается тесно связанным с конкретной реализацией интерфейса `IDiscountCurve`, и поэтому использование различных реализаций кривой скидок в разных приложениях калькулятора цен по-прежнему остается затруднительным. Для того чтобы преодолеть данное затруднение, мы могли бы создать интерфейс фабрики и передать ему различные реализации фабрики, но это вынудило бы нас решать задачу получения подходящей фабрики, которая аналогична получению подходящих

объектов. Иными словами, мы ввели бы уровень косвенности, обеспечивающий дополнительную гибкость, но так и не решили бы главную задачу.

В приведенном выше фрагменте кода не показано, что шаблон Factory позволяет также решить вопрос управления экземплярами объектов. Центральное место в этом коде принадлежит методу `GetCurve()`, отвечающему за предоставление экземпляров объекта кривой, поэтому мы можем выбрать реализацию альтернативных правил, распространяющихся на все приложение в целом. Например, вместо того чтобы формировать каждый раз новую кривую, мы могли бы поддерживать единственный экземпляр ее объекта или же организовать пул таких экземпляров, если приложение должно быть многопоточным.

Еще одним шагом в направлении полной развязки может стать замена фабрики реализацией шаблона Registry (Реестр) [Fowler PoEAA]. Аналогично фабрике, реестр предоставляет объекты заданного типа, но вместо жесткого кодирования типа объекта, он облекает данное решение во внешнюю форму и создает экземпляры объектов, используя рефлексию. Для наших целей такой подход является усовершенствованием шаблона Factory, поскольку он позволяет разделить ответственность. В частности, на один класс (реестра) мы возлагаем ответственность за предоставление объектов, а код, в котором всякий раз принимается решение об использовании конкретного класса, можно переместить в другое, более подходящее место в базе кода.

Реализация такого реестра достаточно проста, но эффективна и в конечном итоге дает нам возможность добиться желаемой цели: отделить класс `BondPricer` от реализации интерфейса `IDiscountCurve`.

```
public class DiscountCurveRegistry {
  private Type curveType;

  public static void RegisterCurveType(Type aType) {
    curveType = aType;
}
  public static IDiscountCurve GetCurve() {
    return (IDiscountCurve)Activator.CreateInstance(curveType);
  }
}
```

На рис. 10.10 приведена итоговая структура класса калькулятора цен. Она наглядно показывает, что объект калькулятора цен зависит только от интерфейса, но в то же время привлекает наше внимание к двум важным последствиям применения реестра: объект калькулятора цен зависит от класса реестра, и поэтому нам требуется другой код (компоновщика), обеспечивающий конкретные реализации реестра.

Объектами кривой, которые мы используем в данном примере, являются источники данных, но нетрудно заметить, что мы могли бы использовать те же самые шаблоны для обнаружения служб, предоставляющих объекты (например, аудиторской службы, записывающей рассчитанную цену в журнал регистрации). Для этой цели обычно служит шаблон Service Locator (Указатель службы) [Alur/Crupi/Malks Core J2EE Patterns].

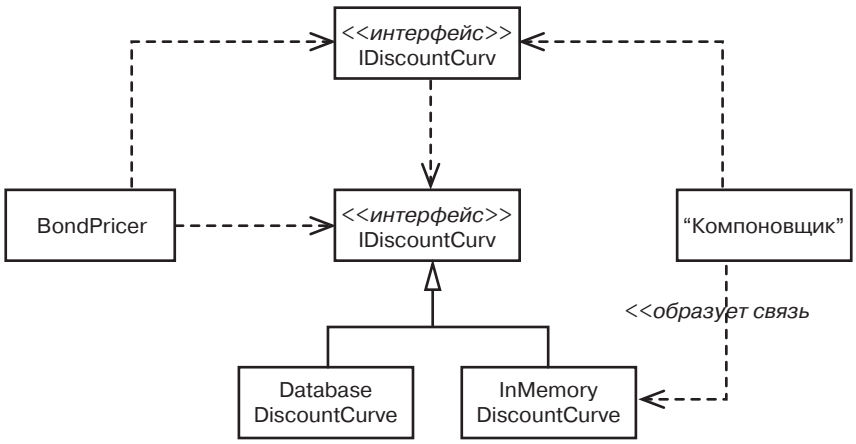


Рис. 10.10. Зависимости, возникающие при использовании шаблона Registry

В большинстве реализаций указателей служб вместо отдельных совокупностей объектов реестра и методов для каждой службы, предоставляется единый набор общих методов и динамическая система условных обозначений.

```
public class Locator {
  public static void RegisterType(String name, Type aType) { ... }
  public static Object GetInstance(String name) { ... }
}
```

Используя шаблон Service Locator, мы регистрируем конкретную реализацию кривой с помощью указателя по имени, которое, скорее всего, обозначает интерфейс. Теперь калькулятор цен может получить кривую скидки, запросив ее у указателя служб.

```
public class BondPricer {
  private readonly IDiscountCurve curve;

  public BondPricer() {
    curve = (IDiscountCurve)Locator.GetInstance("MyBank.IDiscountCurve");
  }
}
```

В нашей системе условных обозначений мы пользуемся строками, что, как правило, и делается, но для этой цели вполне подойдут и типы объектов. Преимущества типов объектов заключаются в том, что они обеспечивают полную поддержку автоматического рефакторинга кода, предусмотренного в используемой интегрированной среде разработки, а также способствуют более тщательной проверке ошибок, хотя эти преимущества достигаются за счет некоторой потери гибкости. Независимо от выбранного подхода шаблон остается неизменным и обеспечивает ту же самую степень развязки.

Внесение зависимостей в конструктор

С помощью реестров и указателей служб нам удалось отделить объект от тех объектов, от которых он зависел, используя косвенность (в данном случае — пресловутую таблицу поиска). Код настройки отвечает за выбор конкретной реализации, а ответственность за поиск зависимых объектов возлагается на объект, имеющий зависимости. Он использует объект пассивного указателя служб и активно извлекает из этого указателя нужные объекты, как только они ему потребуются.

В отличие от этого, внесение зависимости и соответствующий шаблон делают пассивным объект, имеющий зависимости. Такой объект объявляет свои зависимости, в противном случае он вообще не обращает внимания на зависимости. Процесс разрешения зависимостей и поиска зависимых объектов предоставляется внешнему механизму, который может быть реализован самыми разными способами. Как правило, в приложениях имеется код общей инфраструктуры, загружающий файл конфигурации или же использующий внутренний реестр для хранения списка всех объектов и их зависимостей. Но в шаблоне Dependency Injection этот аспект не рассматривается. Код инфраструктуры оказывается в этом случае строго дополнительным, а для удовлетворения зависимостей можно даже передать ему объекты из кода приложения.

Существует несколько разновидностей шаблона внесения зависимостей. Рассмотрим сначала шаблон Constructor Dependency Injection (Внесение зависимостей в конструктор). По этому шаблону объект объявляет свои зависимости в собственном конструкторе, а код, создающий данный объект, обеспечивает получение действительных экземпляров этого объекта во время выполнения.

```
public class BondPricer {
  private readonly IDiscountCurve curve;

  public BondPricer(IDiscountCurve aCurve) {
    curve = aCurve;
  }
}
```

Важно отметить, что приведенный выше код ни от чего не зависит, кроме концепций, связанных с предметной областью. Он не требует ссылки на фабрику или реестр и не занимается поиском или созданием экземпляров объектов. Следовательно, внесение зависимостей способствует изящной разработке программного обеспечения, которая обычно отличается хорошим разделением ответственности между предметной областью и инфраструктурой.

На рис. 10.11 представлена структура класса калькулятора цен, использующая внесение зависимостей. В нее включен объект компоновщика, отвечающий за создание калькулятора цен и формирование кривой скидок, от которой этот калькулятор зависит. Эта блок-схема наглядно показывает, что в такой структуре отсутствует зависимость любого объекта из предметной области от кода инфраструктуры, например, компоновщика.

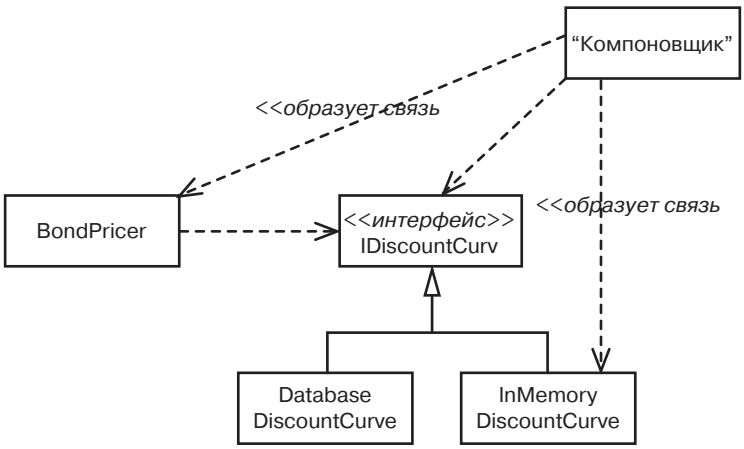


Рис. 10.11. Зависимости, возникающие при использовании шаблона Dependency Injection

Тот факт, что калькулятор цен облигаций вообще не связан с каким-либо кодом инфраструктуры, способствует улучшению тестируемости кода. Рассмотрим первую реализацию калькулятора цен, в которой используется исходная реализация кривой скидок, загружающей свои точки из базы данных. Для тестирования такого класса всегда требуется обращение к базе данных, но, как было показано в предыдущих главах, это не самая лучшая идея. Благодаря внесению зависимостей в конструктор мы можем без труда подставить реализацию зависимости в виде заглушки или имитации, что намного упрощает тестирование калькулятора цен отдельно от базы данных.

```
[Test]
public void testPriceIncludesValuesForOutstandingCoupons () {
  IMock curveMock = new DynamicMock(typeof(IDiscountCurve));
  IDiscountCurve curve = curveMock.MockInstance as IDiscountCurve;
  BondPricer pricer = new BondPricer(curve);
  /* выполнить конкретный тест */
}
```

Такой подход оказывается более простым, чем любое решение, которое может быть получено с помощью других, рассмотренных выше шаблонов. Так, если выбрать для проектирования реестр, то для тестирования придется создать сам реестр и зарегистрировать с его помощью имитацию. А если выбрать внесение зависимостей, то в этом случае нужно лишь просто передать требуемый объект. Такая простота объясняется высоким уровнем развязки в шаблоне Dependency Injection. Благодаря этому упрощается использование объекта в нескольких средах — по крайней мере, в обычной среде выполнения и в среде тестирования.

Мы даже не попытались заменить все функциональные возможности шаблона Registry, внедрив шаблон Dependency Injection. В предыдущей структуре реестр предоставлял еще одну важную услугу — преобразование среди абстрактных типов, ин-

терфейса `IDiscountCurve` и конкретной реализации, предназначенной для использования в данном приложении или контексте.

Как ни странно, такое преобразование достаточно запрограммировать жестко в коде, создающем объект. В нашем примере мы могли бы воспользоваться двумя разными методами, которые вызываются из разных мест в пользовательском интерфейсе: один — для расчета цены выбранной облигации на основании информации, полученной из базы данных, а другой — для запроса у пользователя нескольких точек, формирования кривой скидок в оперативной памяти по указанным точкам и построения калькулятора цен по сформированной кривой. В обоих случаях калькулятор цен может быть построен в соответствующем методе. Но на самом деле использование реестра в данном примере оказывается затруднительным.

```
public class BondInfoForm : Form {
  private TextBox priceTextBox;

  private Bond SelectedBond { get { … } }

  public void CalculateActualPrice() {
    DatabaseDiscountCurve curve = new DatabaseDiscountCurve();
    BondPricer pricer = new BondPricer(curve);
    priceTextBox.Text = pricer.GetPrice(this.SelectedBond);
  }
  public void CalculateScenarioPrice() {
    InMemoryDiscountCurve curve = new InMemoryDiscountCurve();
    this.GetCurvePointsFromUser(curve);
    BondPricer pricer = new BondPricer(curve);
    priceTextBox.Text = pricer.GetPrice(this.SelectedBond);
  }
}
```

Как правило, любое упрощение представляет собой некоторый компромисс, причем простые структуры зачастую неспособны удовлетворить более сложным требованиям. Правда, простота создания компонентов, проектируемых по шаблону Dependency Injection, не препятствует составлению сложных схем преобразования и конфигурации, причем имеется целый ряд легких контейнеров инверсии управления (IoC), способных выполнить роль реестра. (Подробнее об этих контейнерах речь пойдет далее.) Это означает, что мы можем выбирать между простым жестко кодируемым решением, как было показано выше, и более гибким преобразованием, которое обеспечивается некоторым кодом инфраструктуры.

Еще одно интересное соображение связано с объектом, имеющим необязательные зависимости. В примере калькулятора цен облигаций точность некоторых расчетов может быть увеличена, если принять во внимание местные праздники. Это означает, что праздничные дни в календаре составляют необязательную зависимость. Используя внесение зависимостей в конструктор, мы можем отразить это обстоятельство в двух конструкторах: по одному на каждую действительную совокупность зависимостей.

```
public class BondPricer {
  private readonly IDiscountCurve curve;
  private readonly IHolidayCalendar calendar;

  public BondPricer(IDiscountCurve aCurve) {
    curve = aCurve;
  }
  public BondPricer(IDiscountCurve aCurve
    , IHolidayCalendar aCalendar) : this(aCurve) {
    calendar = aCalendar;
  }
}
```

Кроме того, можно было бы воспользоваться конструктором со всеми зависимостями, передаваемыми в виде аргументов, а необязательными зависимостями — в виде пустых значений аргументов, но в этом случае контракт оказался бы менее явным, особенно при увеличении числа зависимостей. В связи с этим рекомендуется всегда предоставлять один конструктор на каждую действительную совокупность зависимостей и организовывать конструкторы в цепочку, если такая возможность имеется. Если же это невозможно, то, скорее всего, у рассматриваемого класса имеется слишком много зависимостей, а значит, он должен быть разбит на более мелкие составляющие.

Внесение зависимостей в механизм установки

Как подразумевает название шаблона Setter Dependency Injection (Внесение зависимостей в механизм установки), для внесения зависимостей в данном случае используется не конструктор, а механизм установки. Это означает, что у калькулятора цен по умолчанию имеется конструктор, но ему требуется дополнительное свойство, которое задается с помощью механизма установки.

```
public class BondPricer {
  private IDiscountCurve curve;

  public BondPricer() {
  }
  public IDiscountCurve DiscountCurve {
    set { curve = value; }
  }
}
```

Внесение зависимостей в механизм установки обеспечивает ту же самую степень развязки, что и внесение зависимостей в конструктор, да и объекты калькулятора цен создаются в блочных тестах так же просто, как и при внесении зависимостей в конструктор. Главное отличие данного подхода заключается в том, что калькулятор цен находится в недействительном состоянии до тех пор, пока не будет вызван соответствующий метод механизма установки, что практически невозможно при внесении зависимостей в конструктор. Кроме того, мы не можем объявить переменную,

хранящую кривую скидок, с типом `readonly`, поскольку она не инициализируется в конструкторе. И наконец, при внесении зависимостей в механизм установки немного труднее реализовать без ошибок окончательную инициализацию функциональных возможностей, которые требуется активизировать после внесения всех зависимостей.

Несмотря на все эти недостатки, внесение зависимостей в механизм установки все же находит применение, поскольку оно лучше распространяется на компоненты со многими зависимостями. Так, для компонента с тремя необязательными зависимостями и строгим представлением о внесении зависимостей в конструктор потребуется восемь конструкторов. Можно, конечно, спорить, требует ли такой компонент рефакторинга кода, хотя это не всегда оказывается практичным. Поэтому в подобных случаях удобно иметь в своем распоряжении более подходящий шаблон. Аналогично, у многих существующих объектов по умолчанию уже имеется конструктор и свойства, а следовательно, мы можем использовать их вместе с контейнерами IoC, поддерживающими внесение зависимостей в механизм установки.

Имеются и другие разновидности шаблона Setter Dependency Injection, в том числе шаблон Interface Driven Setter Dependency Injection (Внесение зависимостей в механизм установки посредством интерфейса), в котором разметочные интерфейсы служат для того, чтобы отличить методы механизма установки, описывающие зависимости, от других общедоступных методов механизма установки. Такие разновидности данного шаблона используются редко и не изменяют радикально основные свойства внесения зависимостей в механизм установки, поэтому мы и не будем их рассматривать.

Инверсия управления

Итак, рассмотрев шаблон Dependency Injection и конкретные его реализации, перейдем к шаблону Inversion of Control (Инверсия управления), называемому также Hollywood Principle (Принцип Голливуда: “Не зовите нас, мы сами позовем вас, если понадобится!”). Строго говоря, инверсию управления следует называть не шаблоном, а общим принципом, который гласит: объекты должны опираться на свою среду, чтобы предоставлять другие объекты, а не активно получать их.

Раньше термином *инверсия управления* (IoC) нередко обозначался метод, применявшийся в некоторых контейнерах для создания компонентов во время выполнения. Рассматривая такие контейнеры, Мартин Фаулер поставил следующий вопрос: “Какой аспект управления они подвергают инверсии?” Он предположил, что в таких контейнерах подходы, основанные на внесении зависимостей в механизм установки и конструктор, реализуются в виде конкретного шаблона и поэтому его уместнее назвать Dependency Injection [Fowler Inversion-OfControl]. Это позволило также провести четкое различие между шаблоном Dependency Injection и другими видами инверсии управления.

Широко применяемый шаблон, в котором соблюдается принцип инверсии управления, а не внесения зависимостей, получил название Contextualized Dependency

Lookup (Привязанный к контексту поиск зависимостей). Если применить такой шаблон к нашему калькулятору цен облигаций, это будет означать, что мы по-прежнему используем класс типа реестра, который в данном шаблоне нередко называется *контекстом*, но в то же время мы даем контейнеру возможность предоставить такой контекст нашему объекту, что уже считается аспектом инверсии управления. В коде это может выглядеть следующим образом:

```
public interface IServiceContext {
  public Object GetInstance(String name);
}

public interface IComponent {
  public void Initialize(IServiceContext context);
}

public class BondPricer : IComponent {
  private IDiscountCurve curve;

public void Initialize(IServiceContext context) {
  curve = (IDiscountCurve)context
    .GetInstance("MyBank.IDiscountCurve");
  }
}
```

При тщательном анализе оказывается, что этот шаблон похож на некий гибрид шаблонов Service Locator и Setter Dependency Injection. И это не случайно, поскольку исторически сложилось так, что он был одним из первых шаблонов инверсии управления и стал ступенькой на пути к более изящным шаблонам Dependency Injection.

Внесение зависимостей в среде Spring.NET

Spring.NET является средой разработки (или же каркасом) приложений [Spring.NET], перенесенной из широко распространенной среды Java Spring [Spring] и предоставляющей очень гибкий контейнер IoC, поддерживающий различные разновидности шаблона Dependency Injection. Такой контейнер является обычным компонентом, а по существу, интерфейсом с несколькими реализациями, отвечающим за загрузку конфигурации и соответствующую настройку объектов. Раньше термин *контейнер* служил для иллюстрации того факта, что контейнер IoC может стать альтернативой стандартному контейнеру J2EE на платформе Java.

Вернемся к рассмотренному выше примеру внесения зависимостей в конструктор, чтобы показать, как контейнер, предоставляемый средой Spring.NET, используется для получения настроенного экземпляра калькулятора цен.

Наиболее предпочтительный способ описания компонентов и их зависимостей средствами Spring.NET состоит в использовании XML-файла конфигурации. Сначала мы объявляем объекты по типу, включая компоновку, из которой они загру-

жаются, а затем разрешаем зависимость калькулятора цен от кривой скидок с помощью вложенного элемента `constructor-arg` в объявлении калькулятора цен.

```
<objects>
  <object name="DbCurve" type="MyBank.DatabaseDiscountCurve, MyBank"/>
  <object name="DbPricer" type="MyBank.BondPricer, MyBank">
    <constructor-arg><ref local="DbCurve"/></constructor-arg>
  </object>
</objects>
```

Для доступа к коду мы можем создать экземпляр интерфейса `IApplication Context` (Интерфейс контекста приложения), указать ему загрузить конфигурации из файла конфигурации, а затем запросить у него объект по имени. Благодаря контексту гарантируется создание и внесение там, где требуется, всех зависимых объектов.

```
IApplicationContext context = new XmlApplicationContext("spring.xml");
BondPricer pricer = (BondPricer)context.GetObject("DbPricer");
```

В реальных приложениях их контекст нередко используется непосредственно, поскольку именно так код, создающий объект, например, калькулятор цен, оказывается зависимым от контекста. Это очень похоже на зависимость между классами и реестром в шаблоне Registry, которой нам так хотелось избежать. Если допустить, что класс, для которого требуется калькулятор цен, является формой Windows, тогда можно воспользоваться внесением зависимостей в самой форме и ввести ее в XML-конфигурацию. Теперь при создании формы калькулятор цен вносится в нее контейнером, а самой форме не нужно обращаться непосредственно к контексту приложения.

Но в этом случае потребуется дополнительный код, создающий форму и с этой целью осуществляющий поиск формы по контексту, а следовательно, связывающийся с контекстом. Ясно, что он не может быть слишком пассивным. В качестве выхода из этого положения в большинстве контейнеров предоставляется возможность для интеграции со средами разработки приложений (например, со средой Spring.Web), чтобы в исходных компонентах приложения, включая формы Windows и Web-страницы, можно было просто использовать внесение зависимостей в то время, как среда разработки приложений отвечала бы за их создание по контексту. В отсутствие интеграции для конкретного сочетания контейнера и среды имеет смысл максимально использовать возможности внесения зависимостей и ограничить зависимость от контекста одним классом.

Если же предпочтение отдается внесению зависимостей в механизм установки, в таком случае можно воспользоваться реализацией на основе данного метода аналогично рассмотренному выше примеру калькулятора цен и изменить соответствующий элемент конфигурации Spring.NET для данной реализации. Код, получающий такую реализацию, остается неизменным, поскольку созданием объектов занимается контейнер.

```
<objects>
  <object name="DbCurve" type="MyBank.DatabaseDiscountCurve, MyBank" />
  <object name="DbPricer" type="MyBank.BondPricer, MyBank" >
    <property name="DiscountCurve">
      <ref local="DbCurve"/>
    </property>
  </object>
</objects>
```

Разработчики среды Spring.NET поддерживают внесение зависимостей в механизм установки, хотя в контейнерах поддерживаются оба метода внесения зависимостей. На самом деле нам ничего не мешает объединить внесение зависимостей в конструктор и в механизм установки. Так, если объявить обязательную зависимость калькулятора цен от кривой скидок в конструкторе и предоставить устанавливаемое свойство для необязательной зависимости от праздничных дней по календарю, то калькулятор цен может быть настроен следующим образом:

```
<objects>
  <object name="DbCurve" type="MyBank.DatabaseDiscountCurve, MyBank" />
  <object name="Calendar" type="MyBank.DefaultHolidayCalendar, MyBank" />
  <object name="DbPricer" type="MyBank.BondPricer, MyBank" >
    <constructor-arg><ref local="DbCurve"/></constructor-arg>
    <property name="HolidayCalendar">
      <ref local="Calendar"/>
    </property>
  </object>
</objects>
```

Это лишь один из способов расширения возможностей Spring.NET, но даже столь простые возможности оказывают немалую помощь в настройке компонентов, разрабатываемых по шаблону Dependency Injection. Можно, конечно, спорить, насколько гибким оказывается размещение в файле конфигурации слишком большого количества сложной логики, которая требуется редко, поскольку это обычно делается за счет удобства сопровождения кода.

Автосвязывание в среде PicoContainer.NET

Файлы конфигурации, упоминавшиеся в предыдущем разделе, нетрудно понять, а их назначение состоит в объявлении нескольких классов и зависимостей между ними. Тем не менее они относительно многословны и дублируют информацию, которая доступна также из системы типов в .NET. Например, тот факт, что калькулятор цен зависит от кривой скидок, выражается в объявлении конструктора в классе этого калькулятора. В действительности вся информация, которая требуется в данном примере, доступна в системе типов.

В большинстве контейнеров IoC поддерживается разрешение зависимостей на основе оперативной информации, доступной во время выполнения. Это свойство назы-

вается *автосвязыванием*, поскольку установка зависимых компонентов и их связывание происходит автоматически и без дополнительной настройки. В среде Spring.NET автосвязывание поддерживается в виде альтернативы XML-файлам, но ради разнообразия мы рассмотрим данное свойство в PicoContainer.NET [PicoContainer], поскольку для этой среды оно является предпочтительным.

Для активизации контейнера с целью разрешить зависимости и для создания объектов в PicoContainer.NET достаточно зарегистрировать классы конкретной реализации с помощью контейнера.

```
IMutablePicoContainer pc = new DefaultPicoContainer();
pc.RegisterComponentImplementation (typeof(DatabaseDiscountCurve));
pc.RegisterComponentImplementation ("pricer", typeof(BondPricer));
```

Помещая объявления в код, мы исключаем избыточность и получаем большую поддержку со стороны интегрированной среды разработки, поскольку в этом случае она может выявить ошибочно набранные имена типов данных и благополучно переименовать их в режиме автоматического рефакторинга кода.

Объекты получаются из контейнера практически так же, как из контекста приложения в Spring.NET, и поэтому для зависимости от контейнера справедливы те же соображения: почти во всех классах должно быть использовано внесение зависимостей и лишь центральный класс оставляется для взаимодействия с контейнером для получения исходных компонентов.

```
BondPricer pricer = (BondPricer)container.GetComponentInstance("pricer");
```

В приведенном выше примере кода показано, что помимо использования системы типов, в PicoContainer.NET поддерживаются также именованные компоненты (в данном случае — калькулятор цен). Это дает возможность искать объекты, не указывая их тип, что означает введение еще одной степени развязки.

Необязательные зависимости обрабатываются автоматически, поскольку в PicoContainer.NET используется самый сложный конструктор, для которого могут быть удовлетворены зависимости. В приведенном выше примере кода для класса `BondPricer` в PicoContainer.NET используется конструктор с единственным аргументом в качестве кривой скидок, но если ввести регистрацию праздничных дней по календарю, то для получения объекта калькулятора цен в PicoContainer.NET потребуется конструктор с двумя аргументами.

```
IMutablePicoContainer pc = new DefaultPicoContainer();
  pc.RegisterComponentImplementation (typeof(DatabaseDiscountCurve));
pc.RegisterComponentImplementation(typeof(HolidayCalendar));
pc.RegisterComponentImplementation ("pricer", typeof(BondPricer));
```

Что же касается зависимостей объектов, то мы ничего не теряем в смысле гибкости, используя типизированное автосвязывание вместо подхода, основанного на файлах конфигурации. В то же время мы извлекаем выгоду из расширения опыта

разработки. Автосвязывание действует не так хорошо, если в конструкторе класса ожидаются такие параметры, как имя хоста, номер*а* портов и время простоев, поскольку примитивные типы не предоставляют достаточной информации о своем назначении. Так, если зарегистрировать с помощью контейнера два целых типа: один — для обозначения номера порта, а другой — для обозначения времени простоя, то контейнер будет рассматривать оба типа данных как целые числа, но не сможет определить, какой из этих типов следует использовать в конструкторе, воспринимающем целочисленный аргумент.

Основной вопрос состоит в следующем: должен ли контейнер поддерживать внесение требуемых компонентов таким образом, как и при установке параметров конфигурации, или же оба требования следует рассматривать как разные виды ответственности, к каждой из которых требуется особый подход. Безусловно, применение типизированного автосвязывания дает немало преимуществ по сравнению с использованием файлов конфигурации, когда дело доходит до обработки зависимостей, поэтому для этих целей больше подходит автосвязывание. С другой стороны, параметры конфигурации могут быть заданы с помощью механизмов установки для именованных свойств, что, безусловно, является более интуитивным подходом к решению данного вопроса.

Третьим контейнером IoC, доступным на платформе .NET, является Windsor как составляющая проекта Castle [Castle]. В базовой реализации контейнера в проекте Castle, которую разработчики называют микроядром, применяется типизированное автосвязывание, хотя и допускаются подключаемые модули для поддержки дополнительных методик разрешения зависимостей. Контейнер Windsor [Windsor], основанный на микроядре, обеспечивает дополнительную поддержку файлов конфигурации, позволяя разработчикам выбирать наиболее подходящий механизм. Такой подход впервые был успешно применен в Java-контейнере NanoContainer [NanoContainer], который располагается на один уровень выше версии контейнера PicoContainer для Java.

Вложенные контейнеры

Тот факт, что контейнер в Spring.NET называется `ApplicationContext` (Контекст приложения), свидетельствует о том, что он определяет контекст для объектов, выбирающих компоненты из этого контейнера. Такие объекты взаимодействуют с возвращаемыми объектами, а их поведение зависит от конкретных реализаций, предоставляемых контейнером. Следовательно, контейнер можно рассматривать как источник контекста, в котором действуют объекты. Кроме того, название данного контейнера подразумевает, что контекст, образуемый из файла конфигурации, характерен для всего приложения в целом и является общим для всех видов поиска в приложении.

В Web-приложениях используется несколько видов контекста, причем большинство Web-сред предоставляют такой контекст в явной форме приложения, сеанса работы и запрашиваемых объектов. Код обработки запросов, например, код функ-

ционирования Web-страницы, обычно зависит от нескольких компонентов. Одни компоненты являются частью контекста приложения, поскольку они могут совместно использоваться всеми обработчиками запросов, а другие — частью контекста отдельных запросов, потому что они должны применяться в едином контексте.

Такая модель поддерживается в некоторых реализациях IoC контейнеров, позволяя им быть вложенными. Это означает, что у контейнера может быть родительский контейнер и, как следствие, порожденные контейнеры. Если контейнер не способен удовлетворить все зависимости, необходимые для создания компонента, он пытается получить недостающие компоненты из своего родительского контейнера, который, в свою очередь, может запрашивать компоненты у своего родительского контейнера. Такая организация контейнеров напоминает структуру шаблона Chain of Responsibility (Цепочка ответственности) [GoF Design Patterns].

В качестве примера рассмотрим создание обработчика запросов или страницы, имеющей зависимость от соединения с базой данных и регистратором. Соединение с базой данных должно устанавливаться только поочередно при каждом запросе, тогда как регистратор может коллективно использоваться всеми запросами. Впрочем, это не имеет отношения к обработчику запросов.

```
public ResultPage : Page {
  private Logger logger;
  private IDbConnection connection;

  public ResultPage(Logger aLogger, IDbConnection aConnection) :
base() {
    logger = aLogger;
    connection = aConnection;
  }
}
```

В данном случае контейнер ассоциируется с приложением. Кроме того, для каждого сеанса работы выделяется порожденный контейнер, а для каждого запроса в сеансе работы создается соответствующий сеансовый контейнер. В рассматриваемом примере приложения запрос URL приводится к типу страницы и содержит следующий общий код, создающий новые обработчики запросов для данного типа страницы:

```
public RequestHandlerFactory {
  IPicoContainer sessionContainer;

  public Page CreatePage(Type pageType) {
    IMutablePicoContainer reqContainer = CreateRequestContainer();
    reqContainer.RegisterComponentImplementation(pageType);
    return (Page)reqContainer.GetComponentInstance(pageType);
 }
  private IMutablePicoContainer CreateRequestContainer() {
    IMutablePicoContainer pc = new DefaultPicoContainer(sessionContainer);
    pc.RegisterComponentImplementation (typeof(SqlDbConnection));
```

```
    /* зарегистрировать здесь все остальные компоненты,
       которые должны стать доступными для запросов
       по очереди. */
    return pc;
  }
}
```

После создания контейнера в методе `CreatePage()` приведенный выше код регистрирует тип страницы с помощью контейнера запросов, чтобы использовать внесение зависимостей на самой странице. Когда в третьей строке этого кода выбирается конкретный экземпляр страницы, контейнер запросов пытается создать страницу, но при условии, что для этого требуется класс `ResultPage` (Итоговая страница), зависимость от класса `Logger` (Регистратор) не может быть разрешена, поскольку не зарегистрирована с помощью контейнера запросов. В этом случае предпринимается попытка получить недостающий компонент из родительского, сеансового контейнера, который также не содержит регистратор, а значит, запрашивает его у своего родительского контейнера приложения. При условии, что этот контейнер зарегистрирует регистратор, все зависимости для страницы могут быть удовлетворены, после чего она создается контейнером запросов.

Преимущество такого подхода заключается в том, что, с одной стороны, компоненты могут регистрироваться и совместно использоваться на подходящем уровне, а с другой — объекты, выбирающие компоненты из контейнера, должны обращаться только к одному контейнеру для поиска требуемых компонентов в цепочке контейнеров.

Сравнение шаблонов Service Locator и Dependency Injection

До сих пор мы рассматривали два разных шаблона, решающих вопросы связывания и управления экземплярами объектов, упомянутые в начале этого раздела. Остается выяснить, когда следует применять каждый из этих шаблонов.

Можно с уверенностью сказать, что шаблон Service Locator используется в настоящее время чаще, поскольку он лучше подходит для большинства приложений, хотя это и не обязательно. Следует также иметь в виду, что шаблон Service Locator применяется уже довольно давно, тогда как Dependency Injection — относительно новый и пока еще недостаточно хорошо известный шаблон. В действительности, разработчики пришли к шаблону Dependency Injection, пытаясь улучшить шаблон Service Locator.

Полная развязка компонентов от механизма разрешения зависимостей, которая достигается с помощью шаблона Dependency Injection, улучшает тестируемость кода, поскольку в тестах не нужно устанавливать специальные версии указателя служб для подстановки заглушек или имитирующих объектов. Кроме того, это способствует повторному использованию компонентов в крупных системах, поскольку позволяет избежать затруднительных ситуаций, в которых компоненты, применяемые в системе, связаны с разными реализациями указателя служб.

Еще одно преимущество шаблона Dependency Injection, и особенно его разновидности Constructor Dependency Injection, заключается в том, что все зависимости ясно видны из кода. Современные интегрированные среды разработки способны выявлять все случаи использования указателя служб в данном классе (нажатием комбинации клавиш <Ctrl+Shift+F7>), но даже в развитой интегрированной среде разработки сделать это труднее, чем просто посмотреть содержимое конструктора.

Нередко обращается внимание на то, что шаблон Dependency Injection связывает жизненный цикл объекта *A*, имеющего зависимости, с жизненными циклами объектов *B* и *C*, от которых он зависит, поскольку объекты *B* и *C* должны быть созданы до объекта *A*, чтобы их можно было внести. Для шаблона Service Locator это не проблема, поскольку создание объектов *B* и *C* можно отложить до тех пор, пока объект *A* не запросит их у указателя службы. Правда, того же результата можно добиться (возможно, и не таким естественным путем) с помощью шаблона Dependency Injection, как показывает опыт применения классов `Component Adapter` (Адаптер компонента) в контейнере PicoContainer. Вместо внесения экземпляра объекта конкретного класса в этом контейнере вносится замещающий объект, который создает настоящий экземпляр, когда используется в первый раз, а затем передает этому экземпляру все вызовы методов. Это выглядит совершенно прозрачно для объекта, имеющего зависимости, но в то же время эффективно развязывает их жизненные циклы.

Выводы

В этом разделе были рассмотрены два вопроса, которые возникают при создании любой системы программного обеспечения: связывание объектов, зависящих друг от друга, и управление экземплярами компонентов. Сначала было показано, как эти вопросы решаются с помощью распространенных шаблонов Factory, Registry и Service Locator. Принимая во внимание определенные недостатки этих шаблонов, мы рассмотрели шаблон Dependency Injection, позволяющий решать подобные вопросы по-новому. Этот шаблон, действуя по принципу инверсии управления, обеспечивает лучшую развязку и вполне пригоден для разрешения как простых, так и сложных зависимостей. Кроме того, он улучшает тестируемость, что очень важно для РПТ, и позволяет полностью отделить модель предметной области от кода инфраструктуры, что делает его полезным инструментом для ППО.

Как и любой другой шаблон, Dependency Injection не является панацеей, и поэтому вряд ли целесообразно всегда заменять им шаблон Service Locator. Для того чтобы найти этим шаблонам подходящее применение, нужно ясно представлять отличия и достоинства каждого из них. Кроме того, мы рассмотрели некоторые разновидности шаблона Dependency Injection как компромиссные варианты выбора. Правда, их выбор не оказывает влияния на тестируемость, что очень важно для разработки, а контейнеры, используемые во время выполнения, как, правило, поддерживают все основные варианты. Следовательно, мы можем выбирать (и даже комбинировать) по мере надобности различные разновидности этого шаблона.

При проектировании систем необходимо, прежде всего, разделить ответственность (в данном случае — конфигурацию и использование компонентов), а затем выбрать из богатого арсенала наиболее подходящий шаблон.

Благодарю, Эрик!

Мне бы хотелось лишь подчеркнуть еще раз то, на чем остановился Эрик. Шаблон Dependency Injection можно рассматривать так, как об этом говорит Мартин Фаулер в своей статье о языке DSL [Fowler LW], где он предлагает взглянуть на разработку как на создание подходящих абстракций и последующее их конфигурирование. В этом смысле шаблон Dependency Injection позволяет решить часть задачи конфигурирования.

Мне пришлось воздержаться от фразы "аспект конфигурирования" в предыдущем предложении, чтобы это не выглядело попыткой как-то связать предыдущую тему со следующей. Но на самом деле внесение зависимостей и аспектно-ориентированное программирование (АОП) часто существуют в некотором симбиозе, как станет ясно из дальнейшего.

Если вернуться к вопросам проектирования по шаблону Repository, то можно выделить несколько возможных подходов к данной задаче.

1. Конкретные хранилища.
2. Базовый класс.
3. Базовый класс с обобщениями.

Недостаток подходов 2 и 3 (помимо нехватки типовой безопасности, если считать это недостатком) состоит в том, что они приводят к созданию определенного рода автоматической, неестественной абстракции, присущей всем хранилищам. В частности, возможность удаления должна быть не у всех хранилищ.

Для преодоления этого недостатка можно было бы создать более мелкий базовый класс и написать специальный код для разных вариантов. Если пойти по этому пути, то можно обнаружить, что у базового класса будет, скорее всего, только два или три метода (`GetById()`, `MakePersistent()` и т.д.), а это мало чем отличается от подхода 1.

Дублирование кода можно попытаться сократить, перенеся общий код во вспомогательные классы, которым отдельные хранилища передают определенные функции. Но тогда придется писать трудоемкий код передачи таких функций, что малоинтересно.

Вместо того чтобы начинать разработку с базового класса, вообще не имеющего достаточной степени детализации, можно выбрать другой подход: создать хранилище, исходя из определенного сочетания нескольких мелких аспектов. Это совсем иной способ осмысления подобной задачи. И хотя он не всегда оказывается правильным (впрочем, единого правильного подхода все равно не существует), тем не менее, определенно заслуживает внимания. В связи с этим я попросил своего коллегу Александара Сеовича написать краткое введение в АОП.

Аспектно-ориентированное программирование

Александар Сеович

За последние несколько лет АОП наделало немало шуму в среде разработчиков программного обеспечения. Метод АОП был изобретен в конце 1990-х годов Грегором Кичалешом [Gregor Kiczales] и его группой из лаборатории Xerox PARC при попытке решить проблему дублирования кода в объектно-ориентированных системах посредством инкапсуляции *взаимно пересекающихся отношений* системы в повторно используемые *аспекты*.

AspectJ [AspectJ] как эталонная реализация АОП была выпущена в 2001 году и послужила толчком для активных разработок на платформе Java, включая и альтернативные реализации АОП.

Несмотря на то что АОП по-прежнему применяется, в основном, на платформе Java, существует несколько реализаций АОП для .NET, благодаря которым появляется возможность воспользоваться преимуществами АОП и на этой платформе. К этим реализациям относятся Spring.NET AOP [Spring.NET], Loom.NET [Loom.NET] и AspectSharp [AspectSharp].

Из-за чего столько шума

Основной предпосылкой для АОП послужило то обстоятельство, что хотя ООП и позволяет существенно сократить дублирование кода по сравнению с процедурными языками программирования, в этом отношении оно оставляет желать много лучшего. Такие основные свойства ООП, как наследование и полиморфизм, а также документированные шаблоны проектирования, включая Template Method (Шаблонный метод) [GoF Design Patterns], действительно способствуют сведению к минимуму дублирования кода в базовой логике приложения, но код, реализующий такие взаимно пересекающиеся отношения, как регистрация или безопасность, по-прежнему очень трудно (если вообще возможно) сделать блочным, используя только средства ООП.

Несмотря на то что в большинстве встречающихся статей, посвященных АОП, приводится один и тот же малоинтересный пример регистрации вызовов методов, мы воспользуемся аналогичным примером, чтобы высветить те проблемы, которые пытается решить АОП.

Допустим, что мы реализовали простую службу калькулятора, выполняющую четыре основные арифметические действия: сложение, вычитание, деление и умножение, как показано в приведенном ниже коде. Обратите внимание, насколько простым и удобочитаемым оказывается код методов благодаря тому, что они содержат базовую логику.

```
public class Calculator : ICalculator
{
  public int Add(int n1, int n2)
```

```
  {
    return n1 + n2;
  }

  public int Subtract(int n1, int n2)
  {
    return n1 - n2;
  }

  public int Divide(int n1, int n2)
  {
    return n1 / n2;
  }

  public int Multiply(int n1, int n2)
  {
    return n1 * n2;
  }
}
```

Ради удобства обсуждения допустим, что от нового пользователя поступает запрос на регистрацию каждого вызова методов данной службы. Этой службе передается имя класса, имя метода, значения параметров, а она, в свою очередь, возвращает значение, которое должно быть зарегистрировано.

Если воспользоваться только ООП, то столь простой, на первый взгляд, запрос вынудит нас видоизменить все четыре метода в приведенном выше фрагменте кода, чтобы ввести в них код регистрации. В конечном итоге код примет следующий вид:

```
using System;

namespace LoggingAspectDemo
{
  public class LoggingCalculator : ICalculator
  {
    public int Add(int n1, int n2)
    {
      Console.Out.WriteLine("-> LoggingCalculator.Add
        (" + n1 + ", " + n2 + ")");
      int result = n1 + n2;
      Console.Out.WriteLine("<- LoggingCalculator.Add
        (" + n1 + ", " + n2+ ") returned " + result);

      return result;
    }

    public int Subtract(int n1, int n2)
    {
      Console.Out.WriteLine("-> LoggingCalculator.Subtract
```

```
        (" + n1 + ", " + n2 + ")");
      int result = n1 - n2;
      Console.Out.WriteLine("<- LoggingCalculator.Subtract
        (" + n1 + ", " + n2 + ") returned " + result);

      return result;
    }

    public int Divide(int n1, int n2)
    {
      Console.Out.WriteLine("-> LoggingCalculator.Divide
        (" + n1 + ", " + n2 + ")");
      int result = n1 / n2;
      Console.Out.WriteLine("<- LoggingCalculator.Divide
        (" + n1 + ", " + n2 + ") returned " + result);

      return result;
    }

    public int Multiply(int n1, int n2)
    {
      Console.Out.WriteLine("-> LoggingCalculator.Multiply
        (" + n1 + ", " + n2 + ")");
     int result = n1 * n2;
     Console.Out.WriteLine("<- LoggingCalculator.Multiply
       (" + n1 + ", " + n2 + ") returned " + result);

      return result;
    }
  }
}
```

Как видите, сходный код регистрации присутствует в каждом методе.

На заметку

Очевидно, что приведенный выше пример кода можно было бы составить лучше и без АОП, используя, в частности, шаблон Decorator (Декоратор) [GoF Design Patterns], однако этим примером мы преследуем цель дать общее представление об АОП, поэтому будьте снисходительны.

Кроме того, шаблон Decorator будет оказывать влияние на потребителя, поскольку он должен создавать экземпляр декоратора вместо декорированного класса. Если же требуется ввести несколько служб, для этого придется реализовать и выстроить в цепочку несколько декораторов.

В связи с рассматриваемым подходом возникает немало осложнений.

- Код часто дублируется, что является плохим признаком. В данном случае код можно было бы упростить, создав для регистрации статические обслуживающие методы, воспринимающие имя класса и параметры или возвращаемые значения в виде аргументов, но само дублирование кода нельзя исключить средствами ООП.
- Код регистрации отвлекает читающего от базовой логики метода, в результате чего код метода оказывается менее удобным для чтения и сопровождения в долгосрочной перспективе.
- Код, реализующий базовую логику, оказывается длиннее, чем нужно. Методы, которые нельзя сделать однострочными, должны быть разбиты на строки присвоения и возврата результирующих значений, чтобы вывести результат в коде регистрации.
- Если в какой-то момент потребуется исключить код регистрации, это придется делать вручную, превратив в комментарий весь код регистрации в соответствующих классах.
- Если же имя класса потребуется зарегистрировать полностью, включая и пространство имен, то придется искать все методы, содержащие код регистрации, и вводить вручную недостающее пространство имен.
- То же самое придется делать и в том случае, если потребуется зарегистрировать исключения, возникающие, например, при переполнении или делении на нуль. Для этого придется ввести блок `try/catch` и оператор регистрации исключений во многих местах базы кода.

Конечно, если выбрать для регистрации более развитое решение, например, Log4Net [Log4Net] или Enterprise Library Logging Application Block [Enterprise Library Logging], вместо простых операторов `Console.Out.WriteLine`, то для отключения режима регистрации достаточно будет лишь изменить параметры конфигурации, даже не касаясь классов.

Все это в целом верно, но не решает проблему по сути. Ведь базовая логика будет перемежаться вспомогательным кодом регистрации, а многие методы придется попрежнему править вручную, чтобы внести изменения в логику регистрации. Более того, если рассмотреть вместо регистрации пример соблюдения правил безопасности или выполнения транзакций, то включение или выключение такого режима окажется непростым делом.

С другой стороны, АОП предлагает разделение взаимно пересекающегося кода, например, регистрации и безопасности, на отдельные аспекты и удобное применение этих аспектов ко всем классам и методам, которые того требуют. Разумеется, возможности АОП этим не исчерпываются. Оно позволяет также отделить бизнес-правила от базовой логики и изменить существующие объекты, дополнив их состоянием и поведением. Соответствующие примеры будут представлены в последующих подразделах.

Самое главное, что АОП не заменяет собой ООП, а является лишь его расширением, которое позволяет добиваться результатов, которых трудно достичь только средствами ООП. Испытанные принципы и приемы ООП по-прежнему находят себе применение. В действительности, теперь как никогда важно иметь, прежде всего, солидную подготовку в объектно-ориентированном проектировании, поскольку это намного облегчает применение аспектов.

Терминология АОП

Прежде чем переходить непосредственно к рассмотрению АОП, необходимо дать пояснение основным его терминам, которые не так уж и сложны, как кажется на первый взгляд. Ниже приведены определения этих терминов.

- **Извещение** — фрагмент кода, который требуется инкапсулировать или использовать повторно. Например, код регистрации может быть реализован в виде извещения и применен там, где требуется. Существует несколько видов извещений: *предшествующие*, *последующие* и *окружающие* извещения.
- **Введение** — это особого рода извещение, которое применяется только в классах. Введения позволяют внедрять в существующие классы новые члены (как состояние, так и поведение) и могут служить для извлечения максимальной пользы из множественного наследования в тех языках программирования, где оно не поддерживается, без обычных для этого компромиссов. (В некоторых языках введение называется смешением.)
- **Точка соединения** — любое место в коде, где *может* быть применено извещение. Теоретически, в качестве точки соединения может служить все, что угодно — переменная экземпляра, цикл `for`, оператор `if` и т.д. Но на практике в качестве точек соединения чаще всего служат классы, методы и свойства.
- **Срез** — обозначает ряд точек соединения, в которых *должно* быть применено извещение. Так, если извещение о транзакции требуется применить ко всем методам, помеченным атрибутом `Transaction`, то придется объявить срез, обозначающий эти методы.
- **Аспект** — группирует извещения и срезы, в которых они применяются, что очень похоже на группирование данных и связанного с ними поведения в классах.

АОП в .NET

А теперь перейдем от теории к практике и посмотрим, как АОП помогает решать реальные задачи, с которыми нам приходится сталкиваться практически каждый день в процессе разработки приложений.

Примеры, приведенные в этом разделе, основываются на реализации АОП в Spring.NET, поэтому для их опробования на практике вам придется загрузить и установить последнюю версию Spring.NET.

На заметку

Следует иметь в виду, что приведенные ниже примеры составлены на реализации АОП, поддержка которой в Spring.NET только планировалась на момент написания этой книги. По поводу возможных изменений обращайтесь на Web-сайт, посвященный данной книге (`www.jnsk.se/adddp`), или же по адресу: `www.springframework.net`.

Блочное составление кода регистрации с помощью аспекта

Итак, посмотрим, насколько эффективно мы можем воспользоваться АОП для разрешения затруднений, связанных с упомянутым выше кодом регистрации.

Формирование извещения о регистрации

Для того чтобы сформировать извещение, прежде всего нужно инкапсулировать код регистрации следующим образом:

```
using System;
using System.Text;
using AopAlliance.Intercept;

namespace LoggingAspectDemo
{
  public class MethodInvocationLoggingAdvice
    : IMethodInterceptor
  {
    public object Invoke(IMethodInvocation invocation)
    {
      Console.Out.WriteLine("-> " +
        GetMethodSignature(invocation));
      object result = invocation.Proceed();
      Console.Out.WriteLine("<- " +
        GetMethodSignature(invocation)
        + " returned " + result);
      return result;
    }

    private string GetMethodSignature
      (IMethodInvocation invocation)
   {
      StringBuilder sb = new StringBuilder();
      sb.Append(invocation.Method.DeclaringType.Name)
        .Append('.')
        .Append(invocation.Method.Name)
        .Append('(');
string separator = ", ";
```

```
      for (int i = 0; i < invocation.Arguments.Length;
          i++)
      {
        if (i == invocation.Arguments.Length - 1)
        {
          separator = "";
        }
        sb.Append(invocation.Arguments[i])
          .Append(separator);
      }
      sb.Append(')');

      return sb.ToString();
    }
  }
}
```

В данном извещении регистрируется имя класса, имя метода, значения параметров и возвращаемые значения для каждого вызова метода, к которому это извещение применяется.

В нем используется метод `ToString()` для вывода значений параметров и возвращаемых значений. И хотя такой подход вряд ли может удовлетворить все возможные ситуации, тем не менее, он вполне пригоден для большинства методов, особенно если позаботиться о реализации метода `ToString()` в классах приложения. Что же касается исключения, то данное извещение лишь распространяет его, и поэтому для обработки такого исключения может подойти любой уже имеющийся код обработки ошибок.

Самое главное происходит внутри метода `Invoke()`, где требуется зарегистрировать начало вызова метода, зафиксировать результат, присвоив переменной, используемой для хранения результата, значение, возвращаемое после вызова метода `invocation.Proceed()`, и наконец, зарегистрировать результат перед его возвратом. Вызов метода `invocation.Proceed()` имеет решающее значение, ведь именно так организуется работа калькулятора.

Вспомогательный метод `GetMethodSignature()` служит для формирования сигнатуры метода обычным образом, т.е. путем проверки объекта `invocation`. Возможно, вы обратили внимание на то, что метод `GetMethodSignature()` вызывается дважды, хотя данное извещение можно было бы немного оптимизировать, присвоив его результат локальной переменной и затем воспользовавшись этой переменной вместо повторного вызова метода `GetMethodSignature()`. Но дело в том, что, вызывая дважды данный метод, мы можем контролировать любые изменения используемых параметров в извещаемом методе. Это особенно полезно для регистрации вызовов тех методов, у которых отсутствуют параметры.

Итак, мы определили извещение. А если потребуется зарегистрировать любые исключения, которые формирует извещаемый метод? Для этого достаточно внести в код несложное изменение — добавить логику в извещаемый метод `Invoke()` следующим образом:

```
public Object Invoke(IMethodInvocation invocation)
{
  Console.Out.WriteLine("-> "
    + GetMethodSignature(invocation));
  try
  {
    object result = invocation.Proceed();
    Console.Out.WriteLine("<- "
      + GetMethodSignature(invocation)
      + " returned " + result);
    return result;
  }
  catch (Exception e)
  {
    Console.Out.WriteLine("!! "
      + GetMethodSignature(invocation)
      + " failed: " + e.Message);
    Console.Out.WriteLine(e.StackTrace);
    throw e;
  }
}
```

Изменения в методе `GetMethodSignature()`, связанные с включением пространства имен класса, сделать еще проще, поэтому оставляем это читателю в качестве домашнего задания.

Применение извещения о регистрации

Сформировать извещение — это только полдела. Ведь если извещение не применяется, то оно бесполезно. В данном случае нам нужно применить извещение о регистрации к классу `Calculator` из первого фрагмента кода. В среде Spring.NET это делается двумя способами.

Первый способ состоит в использовании кода. В данном примере это нетрудно сделать, но на самом деле такой способ далек от идеала, поскольку он предполагает применение извещения к конкретному объекту вручную, а не автоматически с помощью контейнера там, где это необходимо, на основании конфигурации аспекта. Тем не менее, покажем, как извещение о регистрации можно применять в коде.

```
using Spring.Aop.Framework;
...
ProxyFactory pf = new ProxyFactory(new Calculator());
pf.AddAdvice(new MethodInvocationLoggingAdvice());
ICalculator calculator = (ICalculator) pf.GetProxy());
```

Таким образом, использовав лишь три строки кода, мы применили извещение о регистрации ко всем методам экземпляра класса `Calculator` и получили ссылку на него. По этой ссылке теперь можно вызвать любой метод и убедиться, что вызовы регистрируются надлежащим образом.

Другой декларативный подход к АОП в Spring.NET оказывается более эффективным и вообще не предполагает написание кода. Тем не менее, он требует, чтобы для управления объектами использовалась среда Spring.NET, но рассмотрение этого вопроса выходит за рамки данного раздела. Для того чтобы дать некоторое представление о данном способе, ниже показано, каким образом может выглядеть определение декларативного извещения в рассматриваемом примере, а за подробностями отсылаем читателя к справочному руководству по Spring.NET.

```
<aop:aspect name="MethodInvocationLoggingAspect">
  <aop:advice
    type="LoggingAspectDemo.MethodInvocationLoggingAdvice"
    pointcut=
      "class(LoggingAspectDemo.Calculator) and method(*)"/>
</aop:aspect>
```

Такое объявление вынуждает контейнер Spring.NET применять сформированное извещение ко всем методам всех экземпляров класса `Calculator`.

Для того чтобы применить извещение о регистрации ко всем методам всех классов в пространстве имен `LoggingAspect`, достаточно внести следующие изменения в приведенное выше объявление (этим демонстрируется эффективность декларативного подхода в частности и АОП в целом):

```
<aop:aspect name="MethodInvocationLoggingAspect">
  <aop:advice
    type="LoggingAspectDemo.MethodInvocationLoggingAdvice"
    pointcut="class(LoggingAspectDemo.*) and method(*)"/>
</aop:aspect>
```

Ввод состояния и поведения в существующий класс

Во втором примере, который мы собираемся теперь рассмотреть, *введение* используется в определенном сочетании с *предшествующим извещением* для реализации и осуществления блокировки объектов.

Допустим, что в приложении имеются объекты предметной области, которые требуется блокировать в течение сеанса работы с этим приложением. Если объект заблокирован, то любая попытка изменить его состояние в других сеансах работы должна привести к формированию исключения `LockViolationException` (Исключение в связи с проверкой достоверности блокировки).

Если применить только подход ООП, то классы, которым требуется такое поведение, скорее всего, придется дополнить необходимым состоянием и методами напрямую либо посредством их наследования от некоторого базового класса, реализующего основные функции блокировки и разблокировки.

Первое решение не совсем удачно, поскольку приводит в конечном итоге к массовому дублированию кода. А второе решение оказывается вполне приемлемым при условии, что классы, которым требуется блокирующее поведение, принадлежат к одной и той же иерархии классов, причем все они должны поддерживать блокировку.

Если же они принадлежат к разным иерархиям или если внедрение в них блокирующего поведения вообще не требуется, то все возвращается на круги своя.

Намного более эффективный подход заключается в использовании свойств АОП и декларативном вводе поддержки блокировки только в те классы, которым она действительно требуется. Этого нетрудно добиться с помощью определенного сочетания предшествующего извещения и внедрения.

Реализация введения для блокировки

В этом случае сначала необходимо сформировать введение, чтобы добавить состояние или поведение к извещаемым классам. С этой целью нам нужно определить интерфейс, который требуется внедрить.

```
namespace LockableAspectDemo
{
  public interface ILockable
  {
    void Lock();
    void Unlock();
    bool IsLocked { get; }
  }
}
```

Далее необходимо создать класс введения, реализующий этот интерфейс. В данном примере предполагается, что мы создаем Web-приложение в среде ASP.NET и собираемся использовать `HttpContext.Current.Session.SessionID` в качестве идентификатора сеанса работы для целей блокировки.

```
using AopAlliance.Aop;

namespace LockableAspectDemo
{
  public class LockableIntroduction : ILockable, IAdvice
  {
    private bool isLocked;
    private string sessionId;

    public void Lock()
    {
      isLocked = true;
      sessionId = HttpContext.Current.Session.SessionID;
    }

    public void Unlock()
    {
      isLocked = false;
      sessionId = null;
    }
```

```
    public bool IsLocked
    {
      get
      {
        return isLocked && sessionId
          != HttpContext.Current.Session.SessionID;
      }
    }
  }
}
```

Итак, мы сформировали требуемое введение. Теперь нужно сформировать извещение для осуществления блокировки, реализуемой с помощью данного введения.

Реализация извещения для осуществления блокировки

В данном конкретном случае нам не требуется такое же полноценное *окружающее извещение*, как и в рассмотренном выше примере регистрации. Оно должно лишь проверять, заблокирован объект или нет, и если он заблокирован, то формируется соответствующее исключение.

Для этой цели можно воспользоваться простым *предшествующим извещением*, чтобы не вызывать метод `IMethodInvocation.Proceed()`:

```
using System;
using System.Reflection;
using Spring.Aop;

namespace LockableAspectDemo
{
  public class LockEnforcerAdvice : IMethodBeforeAdvice
    { public void Before(MethodBase method, object[] args,
    object target)
    {
      ILockable lockable = target as ILockable;
      if (lockable != null && lockable.IsLocked)
      {
         throw new LockViolationException
        ("Попытка видоизменить заблокированный объект.", target);
      }
    }
  }
}
```

Применение аспекта блокировки к объектам предметной области

И наконец, определенное выше введение и извещение необходимо применить к тем объектам предметной области, которые требуется сделать блокируемыми. Для целей данного примера допустим, что нам требуется сделать блокируемыми объекты класса `Account` (Счет) и все порожденные ими объекты.

Прежде всего, создадим класс `Account`.

```
namespace LockableAspectDemo.Domain
{
  public class Account
  {
    private string name;
    private double balance;

    public Account() {}

    public Account(string name, double balance)
    {
      Name = name;
      Balance = balance;
    }

    public virtual string Name
    {
      get { return name; }
      set { name = value; }
    }

    public virtual double Balance
    {
      get { return balance; }
      set { balance = value; }
    }

    [StateModifier]
     public virtual void Withdraw(double amount)
    {
      balance -= amount;
   }

    [StateModifier]
    public virtual void Deposit(double amount)
    {
      balance += amount;
    }
  }
}
```

Ради краткости опустим порожденные объекты, хотя можно вполне допустить, что блокировка потребуется также объектам классов `CheckingAccount`, `SavingsAccount` и `MoneyMarketAccount`, наследующим от базового класса `Account`.

Для того чтобы этот класс оказался пригодным для заместителя АОП, необходимо соблюсти ряд условий. Во-первых, свойства должны быть реализованы с помощью определенного сочетания механизмов получения и установки значений в част-

ных полях. Если вместо этого использовать общедоступные поля, то заместителю АОП не удастся перехватить доступ к свойству.

Во-вторых, все свойства и методы должны объявляться как виртуальные. Это позволяет создать в Spring.NET динамический заместитель путем наследования класса `Account` и внесения кода перехвата. Если бы методы приняли окончательный вид, в классе `Account` пришлось бы реализовать интерфейс для его замещения составным заместителем. А если бы интерфейс и виртуальные методы отсутствовали, создать заместитель для класса `Account` и применить к нему извещения АОП было бы невозможно.

И наконец, следует упомянуть о необходимости использовать атрибут `StateModifier` (Модификатор состояния). Мы определили этот атрибут в основном для того, чтобы показать, каким образом извещения могут применяться к атрибутам, хотя использование атрибута в данном случае вполне уместно и по той причине, что он предоставляет дополнительные метаданные, явно указывающие, какие именно методы изменяют состояние объекта.

Мы могли бы использовать атрибут `StateModifier` прежде механизмов установки свойств, но это было бы излишне. Помимо этого, опуская атрибут `StateModifier` при установке свойств, мы можем показать, каким образом определяется составной срез.

Несмотря на то что аспект блокировки может быть сконфигурирован только программными средствами, это не самый удобный способ, поэтому покажем в данном случае лишь декларативный способ.

```
<aop:aspect name="LockEnforcementAspect">
  <aop:pointcut name="StateModifiers"
    expression="class(Account+)
      and (setter(*) or attribute(StateModifier))"/>
    <aop:advice type="LockableAspectDemo.LockableIntroduction"
      pointcut="class(Account+)"/>
    <aop:advice type="LockableAspectDemo.LockEnforcerAdvice"
      pointcut-ref="StateModifiers"/>
</aop:aspect>
```

Большая часть этого объявления не требует пояснений. Главного внимания заслуживает следующее.

- Знак “+” после имени класса в определении среза обозначает, что срез должен соответствовать указанному классу и производным от него классам.
- Выражение `"setter(*)"` служит для согласования всех механизмов установки свойств независимо от их типа и имени.
- Выражение `"attribute(StateModifier)"` служит для согласования всех членов, помеченных атрибутом `StateModifier`.

И наконец, нужно создать клиентскую часть приложения, чтобы использовать в нем внедренный интерфейс `ILockable` так же эффективно, как и при его непосредственной реализации в классе `Account`.

```
using LockableAspectDemo.Domain;

namespace LockableAspectDemo.Services
{
  public class AccountManager
  {
    public void TransferFunds(Account from, Account to
      , double amount)
    {
      ILockable acctFrom = from as ILockable;
      ILockable acctTo = to as ILockable;

      try
      {
        acctFrom.Lock();
        acctTo.Lock();

        from.Withdraw(amount);
        to.Deposit(amount);
      }
      finally
      {
        acctFrom.Unlock();
        acctTo.Unlock();
      }
    }
  }
}
```

Таким образом, применяя АОП, действительно можно составить свойство блокировки объектов по блокам и применить его там, где это уместно, преодолевая ограничения, присущие ООП.

На заметку

Несмотря на то что приведенный выше пример формально ничем не отличается от применения операторов `lock` для синхронизации доступа к объектам `acctFrom` и `acctTo` из нескольких потоков, оба подхода заметно отличаются по семантике. С помощью интерфейса `ILockable` мы можем сохранить состояние блокировки на время действия многих запросов по протоколу HTTP, а не только одного подобного запроса.

Перенос бизнес-правил в аспекты

И под конец этого раздела рассмотрим более сложный пример применения АОП. Начинать экспериментирование над АОП с переноса всех бизнес-правил в аспекты

не рекомендуется. Внедрять АОП в конкретные проекты лучше всего неспеша, чтобы решение таких задач взаимно пересекающихся отношений, как регистрация, безопасность и транзакции, выглядело ясным и простым. Следует, однако, вполне осознавать весь потенциал АОП, чтобы применять его эффективно, как только станут более понятными особенности этого метода разработки приложений.

У каждого коммерческого приложения имеются свои бизнес-правила, которые обычно встраиваются в логику этого приложения. И хотя данный принцип справедлив для большинства правил и приложений, тем не менее, благодаря именно ему приложения оказываются менее гибкими, чем нужно, и зачастую требуют модификации кода при изменении бизнес-правил. Перенося некоторые бизнес-правила в аспекты, мы намного упрощаем специальную настройку приложения. В отличие от предыдущих примеров, основной причиной для переноса бизнес-правил в аспекты обычно служит не исключение дублирования кода, а повышение гибкости приложения.

Для такого переноса лучше всего подходят правила, требующие реализации вспомогательной логики наряду с базовой. Например, базовая логика метода `Account.Withdraw()` — снятие со счета, из предыдущего примера состоит в уменьшении остатка на счете. А вспомогательная логика состоит в соблюдении требований к минимальному остатку на счете или отправке по электронной почте предупреждения владельцу счета, когда остаток на его счете окажется меньше определенной суммы.

Посмотрим, как будет выглядеть класс `Account`, если ввести в него логику соблюдения минимального остатка на счете и выдачи соответствующего предупреждения.

```
namespace BusinessRulesDemo.Domain
{
  public class Account
  {
    private AccountHolder accountHolder;
    private string name;
    private double balance;
    private double minimumBalance;
    private double alertBalance;

    // взаимодействующие компоненты, вносимые
    // контейнером IoC в Spring.NET

    private INotificationSender notificationSender;

    public Account() {}

    public Account(string name, double balance)
    {
      Name = name;
      Balance = balance;
    }

    public virtual Person AccountHolder
```

```
{
  get { return accountHolder; }
  set { accountHolder = value; }
}

public virtual string Name
{
  get { return name; }
  set { name = value; }
}

public virtual double Balance
{
  get { return balance; }
  set { balance = value; }
}

public virtual double MinimumBalance
{
  get { return minimumBalance; }
  set { minimumBalance = value; }
}

public virtual double AlertBalance
{
  get { return alertBalance; }
  set { alertBalance = value; }
}

// вносится контейнером IoC в Spring.NET
public virtual INotificationSender NotificationSender
{
  set { notificationSender = value; }
}

[StateModifier]
public virtual void Withdraw(double amount)
{
  if (balance - amount < minimumBalance)
  {
    throw new MinimumBalanceException();
  }

  balance -= amount;

  if (balance < alertBalance)
  {
    notificationSender.SendNotification
    (accountHolder.Email,
```

```
        "Low balance",
        "LowBalance.vm", this);
      }
    }

    [StateModifier]
    public virtual void Deposit(double amount)
    {
      balance += amount;
    }
  }
}
```

Конечно, можно спорить, должна ли логика минимальной проверки остатка на счету оставаться в классе `Account`, но совершенно ясно, что код отправки уведомления не относится к основным функциям метода `Withdraw()`. Это означает, что его можно вполне перенести в аспект. Реорганизуем этот код.

Формирование извещения об уведомлении

Прежде всего, создадим общее извещение для отправки уведомлений.

```
using System.Reflection;
using BusinessRulesDemo.Domain;
using Spring.Aop;

namespace BusinessRulesDemo
{
  public class AccountBalanceNotificationAdvice :
    IAfterReturningAdvice
  {
    private INotificationSender sender;
    private string subject;
    private string templateName;

    public INotificationSender Sender
    {
      get { return sender; }
      set { sender = value; }
    }

    public string Subject
    {
      get { return subject; }
      set { subject = value; }
    }

    public string TemplateName
    {
```

```
      get { return templateName; }
      set { templateName = value; }
    }

    public void AfterReturning(object returnValue,
      MethodBase method, object[] args, object target)
   {
      Account account = (Account) target;
      if (account.Balance < account.AlertBalance)
      {
        sender.SendNotification(
           account.AccountHolder.Email,
           subject, templateName,
           target);
      }
    }
  }
}
```

В этом примере *предшествующее извещение* используется для того, чтобы проверить, находится ли новый остаток на счете ниже критического уровня, и если это так, то отправить предупреждающее уведомление. Кроме того, в классе извещения определен ряд свойств, чтобы показать, каким образом извещение настраивается с помощью контейнера IoC в Spring.NET.

Тесная связь между извещением и классом `Account` не вызывает в данном случае никаких осложнений, поскольку мы знаем, что извещение об уведомлении все равно будет применяться только к экземплярам класса `Account`, ведь это извещение служит лишь для выноса правила уведомления за пределы базовой логики класса `Account`.

В этом примере переноса бизнес-правил особое внимание привлекает гибкость. Так, если продать готовое приложение другому заказчику, который потребует удалить из него уведомление, для этого достаточно исключить из файла конфигурации применение соответствующего извещения. Аналогично, если заказчик попросит реализовать логику таким образом, чтобы перенос со счета кредитного лимита происходил автоматически, когда остаток на счету окажется ниже минимально допустимой суммы, для этого достаточно сформировать еще одно извещение и применить его к классу `Account` или даже в дополнение к извещению об уведомлении.

Применение извещения

Как и в предыдущем примере, мы покажем в завершение данного примера лишь декларативное конфигурирование аспекта.

```
<aop:aspect name="BalanceNotificationAspect">
  <aop:advice
    type="BusinessRulesDemo.AccountBalanceNotificationAdvice"
    pointcut="class(Account+) and method(Withdraw)">
```

```
    <aop:property name="Sender" ref="NotificationSender"/>
    <aop:property name="Subject" value="Low Balance"/>
    <aop:property name="Template"
    value="~/Templates/LowBalance.vm"/>
    </aop:advice>
</aop:aspect>
```

Остается лишь заметить, что свойства контейнера IoC в Spring.NET можно использовать для конфигурирования извещений аналогично конфигурированию любого другого объекта. В данном примере мы воспользовались такой возможностью, чтобы указать тему уведомления и имя файла для шаблонного формирователя сообщений NVelocity, а также экземпляр используемого отправителя уведомлений.

Выводы

АОП позволяет решать ряд задач, с которыми разработчику приходится сталкиваться едва ли не каждый день. И хотя поддержку АОП в .NET на уровне языка программирования и общеязыковой исполняющей среды (CLR) можно только приветствовать, для платформы .NET уже имеется немало открытых инструментов, предоставляющих немало возможностей для разработки приложений средствами АОП.

Вероятно, самый лучший способ узнать больше об АОП — загрузить один из таких инструментов и поэкспериментировать с ним, а возможно, и попробовать внедрить его в реальные проекты как вспомогательное средство для применения в самых простых ситуациях, включая регистрацию или профилирование. Изучив подробнее АОП и ознакомившись поближе с выбранным инструментом, можно обнаружить немало других областей, в которых аспекты окажутся весьма полезными.

Благодарю, Александар!

Итак, решая задачу проектирования по шаблону Repository, о которой шла речь до раздела, посвященного АОП, с использованием средств АОП, можно внедрить в класс `CustomerRepository` такие интерфейсы, как `GetByIdable`, `Deletable` и т.д.

Согласитесь, что решение данной задачи получается весьма изящным. Конечно, для применения АОП можно найти более очевидные примеры, включая и рассмотренную выше типичную регистрацию. И хотя этот пример, как отмечалось выше, мало интересен, тем не менее, он наглядно показывает эффективность метода АОП в процессе решения подобных задач.

Резюме

В этой главе были рассмотрены три разных и очень важных метода проектирования, которые следует взять на заметку или же начать изучать. В последней главе речь пойдет о не менее важных службах представления.

Глава 11

Акцент на пользовательском интерфейсе

Тестировать базу данных и пользовательский интерфейс непросто. Тестирование базы данных мы уже обсуждали в главе 6, а в этой главе речь пойдет о тестировании пользовательского интерфейса.

Мы также рассмотрим вопросы проектирования пользовательского интерфейса в уже привычном для нас контексте ППО — по крайней мере, в сравнении с *Быстрой разработкой приложений* (БРП — Rapid Application Development (RAD)) в .NET. С этой целью мы исследуем шаблоны Model-View-Controller (Модель–Представление–Контроллер — MVC) и Model-View-Presenter (Модель–Представление–Презентатор — MVP).

Кроме того, мы рассмотрим в этой главе другие виды преобразования — на этот раз не объектно-реляционного, а объектно-объектного, а также преобразования пользовательского интерфейса.

Но прежде всего, необходимо настроить вас, читатель, на нужный лад, что постарается сделать мой давний коллега Дан Бистрём, которого я попросил написать краткий эпилог к моей книге, но по зрелом размышлении я решил, что он больше подойдет в качестве введения к последней главе, и поэтому данный раздел получил название “Предварительный эпилог”.

Предварительный эпилог

Дан Бистрём

Представьте себе, что вы создали самую идеальную модель предметной области, которую когда-либо видел свет. Вам удалось сделать то, что казалось просто невозможным: преобразовать бо́льшую часть сферы деятельности заказчика в модель предметной области. При этом вам удалось избежать целого ряда специальных случаев и дать заказчику намного более общее представление о сфере его деятельности, чем он мог вообще предположить.

Более того, интерфейс получился ясным, а еще более ясным — код. У кого из тех, кто воспользуется такой моделью предметной области, не перехватит дыхание от изумления и не зародится втайне зависть к вашим достижениям? Большего, кажется, и пожелать нельзя. Теперь и помереть не жалко!

"А кто же все-таки те, для кого предназначена ваша модель предметной области?"

"Пожалуй, — ответите вы, — это все, кто программирует на языке, поддерживаемом в среде .NET. К их числу можно также отнести всех разработчиков, работающих на платформе Java, или других разработчиков в зависимости от конкретной реализации".

"Значит, сотни тысяч программистов могут воспользоваться вашей моделью предметной области? Это нечто!"

"Да, здорово, правда? — ответите вы. — Мне удалось инкапсулировать в этой модели большую часть бизнес-модели и все, что связано с сохранением данных, чтобы ею можно было легко пользоваться, не вдаваясь во все эти малоприятные детали".

"Вы уверены, что никто, кроме программистов, не заинтересуется вашей моделью предметной области?"

"Что вы имеете в виду?"

"Я просто любопытствую. Не кажется ли вам, что существует немало внешних *пользователей*, которые могли бы извлечь пользу из вашей работы? В самом деле, не являются ли *пользователями*, прежде всего, те, кто заказал вам модель предметной области?"

"Да, это так. И что же вы предлагаете?"

"Не кажется ли вам, что для установления связи с вашей новенькой моделью предметной области *пользователям* потребуется *пользовательский интерфейс*?"

"О..."

Мне посчастливилось работать вместе с Джимми над проектом, в котором акцент был сделан не только на модели предметной области и объектно-реляционном преобразовании, но и на пользовательском интерфейсе. Работа над этим проектом сложилась настолько удачно, что мы почувствовали себя победителями, да и заказчик, думаю, не остался внакладе.

Когда я разрабатываю приложение, то рассматриваю пользовательский интерфейс в качестве центральной его части, поскольку без пользовательского интерфейса приложение оказывается малопригодным. Это же справедливо и для представления и сохранения данных в большинстве коммерческих приложений. Таким образом, вся разработка сводится к личным *представлениям*, *пристрастиям* и *интересам* разработчика.

У некоторых разработчиков не вызывает особых трудностей внедрение в пользовательский интерфейс прямого доступа к базе данных. Характерным тому примером служит классическая привязка данных, в которой элементы управления пользовательского интерфейса привязываются непосредственно к полям базы данных. Всякий раз, когда я пробовал это сделать, я давал себе слово больше не делать этого! (С другой стороны, я всегда стараюсь найти самый простой способ непосредственной привязки к модели предметной области!)

По мере разрастания масштабов приложения, которое приходится сопровождать, логические слои становятся настоящим благом.

Еще одним значительным усовершенствованием можно считать логическое выделение данных пользовательского интерфейса в отдельное представление в виде шаблона Presentation Model (Модель представления) [Fowler PoEAA2].

Нельзя сказать, что я придерживаюсь такого подхода в отношении простых пользовательских интерфейсов, но что касается сложных интерфейсов, то я стараюсь все же это делать — пускай не полностью и не всегда. К сожалению, мне не приходилось еще работать над проектами с неограниченным бюджетом. Но с другой стороны, простая задача, как правило, требует простого и только простого решения.

В действительности, когда я впервые столкнулся с механизмом SQL Database (до этого я всегда разрабатывал собственные механизмы сохранения данных), то сразу же осознал, что мне понадобится некий промежуточный слой между базой данных и пользовательским интерфейсом, хотя мне потребовалось более десяти лет, чтобы прийти к понятию *модели предметной области*.

Итак, модель предметной области *вполне* подходит для программирующих пользовательский интерфейс. Но это только начало. Самое интересное следует дальше!

Отлично! Благодарю, Дан!

А теперь рассмотрим службы представления в контексте модели предметной области с развитыми функциональными возможностями сначала с точки зрения клиентской части приложения с расширенными функциональными возможностями и с акцентом на тестировании.

Как отмечал Мартин Фаулер [Fowler PoEAA], MVC относится к самым неверно понимаемым шаблонам, хотя он весьма эффективен и широко распространен. Назначение и основной принцип действия этого шаблона нетрудно понять, но, как обычно, его суть кроется в деталях. В следующем разделе Кристофер Скьолдборг попробует пролить свет на применение шаблона MVC, подробно рассмотрев его на конкретном примере.

Шаблон Model-View-Controller

Кристофер Скьолдборг

Назначение шаблона Model-View-Controller (MVC) — разделить поведение пользовательского интерфейса на отдельные части, чтобы улучшить повторное использование и тестируемость кода. Этими тремя частями являются модель, представление и контроллер.

Модель служит некоторым обозначением состояния объекта-сущности. Она может быть самой разной: от простой структуры данных (XML-документа, набора данных и объекта передачи данных) до полноценной модели предметной области, если таковая построена.

Представление должно состоять *только* из логики, непосредственно связанной с “раскрашиванием пикселей на экране”. Оставить представление “как можно более немым” — зачастую основная цель применения шаблона MVC. Объясняется это тем, что виды и экраны обычно проверяются “на глаз”. Хотя этот процесс не лишен ошибок и связан с бо́льшими издержками по сравнению с автоматическим блочным

тестированием. Если свести к минимуму, прежде всего, логику представления, то столь же минимальным окажется и риск внести программные ошибки, которые могут быть выявлены только в результате тестирования вручную. Иными словами, мы сводим к минимуму объем необходимого тестирования вручную.

И наконец, *контроллер*, составляющий основу шаблона MVC, связывает вместе модель и представление. Контроллер получает сообщения, когда пользователь взаимодействует с представлением, преобразует эти сообщения в действия, выполняемые над базовой моделью, а затем обновляет представление соответственно.

Своими корнями шаблон MVC уходит в среду языка Smalltalk, получившего распространение в 70-е и 80-е годы XX века [MVC History].

Как уже отмечалось, основной причиной для внедрения шаблона MVC послужила потребность в улучшении тестируемости. Ниже перечислены другие достоинства этого шаблона.

- Сам процесс явного извлечения кода из представлений требует большего структурирования кода пользовательского интерфейса, чем это обычно происходит в противном случае. Многие современные инструменты разработки приложений упрощают данный процесс до одного щелчка, которого оказывается достаточно, чтобы ввести код обработки различных событий, не думая слишком долго. Для развертывания шаблона MVC разработчику требуется приложить чуть больше усилий, но в конечном итоге ему будет совершенно ясно, что происходит в самом пользовательском интерфейсе.
- Упрощается процедура изменения всего внешнего вида и поддержки новых пользовательских интерфейсов и даже платформ. Повторному использованию кода в шаблоне MVC иногда придают слишком большое значение, особенно это касается межплатформенной переносимости (по цепочке “клиентское приложение с расширенными функциональными возможностями — Web — карманное устройство”). Для максимального использования возможностей целевой платформы зачастую приходится существенно видоизменять контроллер шаблона MVC.
- Если используется система управления блокировками доступа к файлам исходного текста, то разные члены группы разработчиков могут одновременно работать над представлением и контроллером, поскольку эти части теперь легко отделяются друг от друга.

Пример: обувной магазин Джо

Шаблон проектирования MVC сформулирован довольно свободно. Это означает, что при его использовании приходится принимать немало решений относительно деталей реализации. А поскольку это может оказаться не очень приятным сюрпризом в ходе проектирования, то прежде чем рассматривать разновидности шаблона MVC, приведем для начала простой пример реализации, чтобы дать общее представление

об этом шаблоне (хотя этот пример не так уж далек от действительности, т.е. от того, что приходится делать при разработке реального приложения аналогичной сложности). Далее мы рассмотрим основные направления для усовершенствования на основании, заложенном в данном примере, и дадим ряд рекомендаций относительно того, как это лучше сделать.

В данном примере используется язык C# и среда Windows Forms, сам код будет достаточно простым и ясным — даже если вы не знакомы с C# и Windows Forms.

Итак, рассмотрим каталог товаров обувного магазина Джо. В этом магазине продается обувь только двух видов: обувь для танцев и защитная обувь. Если обувь для танцев обладает свойством жесткости, то защитная обувь характеризуется максимальным давлением. У каждой модели обуви имеется свой идентификационный номер, название и цена. На рис. 11.1 приведен пример экрана для ведения каталога товаров обувного магазина Джо.

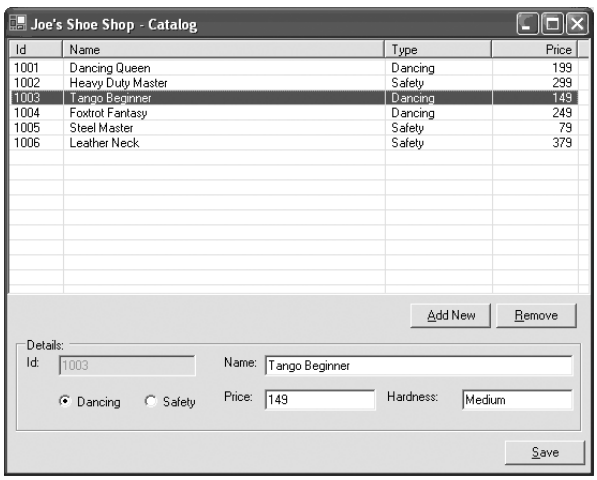


Рис. 11.1. Экран для ведения каталога товаров обувного магазина Джо

Отдельные модели обуви можно просматривать, вводить, удалять и обновлять. После сохранения новой модели обуви ее идентификационный номер не меняется. Когда выбирается обувь для танцев, отображается ее жесткость, а когда выбирается защитная обувь, то отображается максимальное давление, которое данная обувь способна выдерживать. Несмотря на то что подобное представление кажется далеко не самым сложным, для него характерны такие общие элементы, как сеточная форма и части экрана, которые должны изменяться динамически в зависимости от действий, происходящих в других частях экрана.

Для того чтобы отделить логику от представления, необходимо сначала наметить в общих чертах события и действия пользователя, которые требуют обработки и соответствующей реакции. В данном случае это будут следующие события.

- `LoadView` — загрузка данных и создание экземпляра всего представления при первом его вызове.
- `SelectedShoeChanged` — изменение пользователем активной позиции обуви.
- `AddNewShoe` — выбор пользователем кнопки Add New (Добавить новую позицию).
- `RemoveShoe` — выбор пользователем кнопки Remove (Удалить).
- `ChangeType` — изменение описания метки свойства обуви: `Hardness` (Жесткость) или `Max Pressure` (Максимальное давление), а также очистка значения соответствующего свойства.
- `Save` — сохранение изменений, внесенных пользователем.

Все эти события должны быть в первую очередь обработаны в методах класса `Controller`. Несмотря на то что в шаблоне MVC формально установлено, что контроллер должен воспринимать события и воздействовать на представление, на практике чаще всего принято организовывать подписку на события в представлении и затем передавать их контроллеру для обработки. (Такая разновидность основного шаблона MVC названа Model-View-Presenter Мартином Фаулером [Fowler PoEAA2].)

Итак, мы составили довольно ясное представление о логике, которую требуется ввести в контроллер, теперь нужно организовать взаимодействие контроллера с представлением. Напомним, что представление должно быть как можно более немым, поэтому всю основную работу должен выполнять контроллер, а представлению можно поручить ряд простых операций, не требующих дополнительной обработки. Для этой цели определим интерфейс `ICatalogView`, который должен быть реализован в представлении.

```
public interface ICatalogView
{
void SetController(CatalogController controller);
void ClearGrid();
void AddShoeToGrid(Shoe shoe);
void UpdateGridWithChangedShoe(Shoe shoe);
void RemoveShoeFromGrid(Shoe shoe);
string GetIdOfSelectedShoeInGrid();
void SetSelectedShoeInGrid(Shoe shoe);
string Id{get;set;}
bool CanModifyId{set;}
string Name{get;set;}
ShoeType Type{get;set;}
string Price{get;set;}
string VariantName{get;set;}
string VariantValue{get;set;}
}
```

На первый взгляд, имена некоторых методов в интерфейсе `ICatalogView` раскрывают их сложные функции, но в действительности эти функции строго привязаны к раскраске пикселей на экране, взаимодействуя с различными реквизитами пользовательского интерфейса, активизируя и отключая кнопки и т.д. В приведенном ниже фрагменте кода показана частичная реализация этих функций.

На заметку

В этом разделе основное внимание уделяется загрузке в представление списка обуви. Это же относится и ко всему приведенному ниже коду представления, контроллера и теста.

Полную базу кода для данного примера можно посмотреть, загрузив ее из посвященного этой книге Web-сайта по адресу: `www.jnsk.se/adddp`.

```
public void SetController(CatalogController controller)
{
  _controller=controller;
}

public void ClearGrid()
{
  // Определить столбцы сеточной формы
  this.grdShoes.Columns.Clear();
  this.grdShoes.Columns.Add("Id", 50
    , HorizontalAlignment.Left);
  this.grdShoes.Columns.Add("Name", 300
    , HorizontalAlignment.Left);
  this.grdShoes.Columns.Add("Type", 100
    , HorizontalAlignment.Left);
  this.grdShoes.Columns.Add("Price", 80
    , HorizontalAlignment.Right);
  this.grdShoes.Items.Clear();
}

public void AddShoeToGrid(Shoe shoe)
{
  ListViewItem parent;
  parent=this.grdShoes.Items.Add(shoe.Id);
  parent.SubItems.Add(shoe.Name);

  parent.SubItems.Add(Enum.GetName(typeof(Shoe.ShoeType),
  shoe.Type));

  parent.SubItems.Add(shoe.Price.ToString());
}
```

```
public void SetSelectedShoeInGrid(Shoe shoe)
{
  foreach(ListViewItem row in this.grdShoes.Items)
  {
    if(row.Text==shoe.Id)
    {
      row.Selected=true;
      break;
    }
  }
}
```

Метод `SetController()` позволяет сообщить представлению, какому именно экземпляру контроллера оно должно передавать события, а все обработчики событий просто вызывают метод, соответствующий событию, в классе `Controller`.

Возможно, вы заметили, что в ряде методов интерфейса `ICatalogView` используется объект `Shoe`. Это объект класса `Shoe`, относящегося к модели, благодаря которому представлению становится известно о модели. В данном примере `Shoe` — очень простой класс предметной области без поведения, тогда как у классов из настоящей модели предметной области, скорее всего, будет намного больше функциональных возможностей.

```
public class Shoe
{
  public enum ShoeType
  {
    Dancing=1,
    Safety=2
  }
  public string Id;
  public string Name;
  public ShoeType Type;
  public decimal Price;
  public string VariantValue;

  public Shoe(string id, string name, ShoeType type,
    decimal price, string variantValue)
  {
    Id=id;
    Name=name;
    Type=type;
    Price=price;
    VariantValue=variantValue;
  }
}
```

На заметку

Возникает резонный вопрос: почему для обозначения класса используется название `Shoe` (Обувь), а не `PairOfShoes` (Пара обуви)? Дело в том, что в торговле обувью чаще используется первое, а не второе название. Я лично обратил внимание на то, что торговцы всегда говорят "эта обувь" и "та обувь", тогда как другие чаще употребляют фразы "эта пара обуви" или "эти туфли". Кроме того, название `Shoe` получается более простым и кратким, чем название `PairOfShoes`.

Теперь нужно реализовать остальные части шаблона MVC: представление и модель. Обе эти части связаны, в основном, с состоянием. Так, модель хранит в структурированном формате состояние в оперативной памяти, а в представлении хранится визуальное представление и средства для изменения состояния. Свяжем эти части шаблона MVC посредством контроллера. Мы уже наметили в общих чертах виды событий, которые требуют обработки, но ради большей конкретности нам нужно написать тесты для каждого события. В этих тестах используется среда NMock [NMock].

Конкретный код приведен ниже, а здесь даются некоторые комментарии к нему. Так, `getTestData()` является закрытым вспомогательным методом для составления списка обуви в оперативной памяти. В остальной части набора тестов он используется в качестве модели.

На каждый метод, определенный в контроллере, приходится один тест. Однако основное внимание в текущий момент уделяется загрузке данных в представление, и поэтому соответствующий тест называется `LoadView_test()`.

Сначала создается имитация `viewMock`, обозначающая представление в тесте, затем составляется список `shoes` (т.е. модель), и наконец, создается контроллер, который и требуется протестировать. Обратите внимание на то, как конструктору контроллера передается реализованная имитация представления. Контроллер будет оперировать этой имитацией как настоящим представлением, поскольку интерфейс `ICatalogView` — это все, что контроллеру известно о представлении.

Далее имитации `viewMock` сообщается, вызовы каких именно методов и с какими параметрами следует ожидать. Эти вызовы обозначают "заказы", которые предполагается получить в немом представлении от контроллера. После этого вызывается обработчик событий, который требуется проверить в контроллере. И наконец, имитации `viewMock` предписывается проверить, насколько оправдались предположения. Внутри имитации `viewMock` утверждается, что каждый предполагаемый вызов был получен и сделан с правильными параметрами. Соответствующий код приведен ниже.

```
private IList getTestData()
{
  IList shoes=new ArrayList();
  shoes.Add(new Shoe("1001","Dancing
```

```
  Queen",Shoe.ShoeType.Dancing,199m,"Soft"));

  shoes.Add(new Shoe("1002",
  "Heavy Duty Master",Shoe.ShoeType.Safety,299m,"500 lbs"));

  shoes.Add(new Shoe("1003",
  "Tango Beginner",Shoe.ShoeType.Dancing,149m,"Medium"));

  return shoes;
}

[Test]
public void LoadView_test ()
{
  DynamicMock viewMock=new DynamicMock(typeof(ICatalogView));
  IList shoes=getTestData();
  CatalogController ctrl = new CatalogController
    ((ICatalogView)viewMock.MockInstance, shoes);

  // Задать предположения для имитации представления
  viewMock.Expect("ClearGrid");
  viewMock.Expect("AddShoeToGrid",new object[]{shoes[0]});
  viewMock.Expect("AddShoeToGrid",new object[]{shoes[1]});
  viewMock.Expect("AddShoeToGrid",new object[]{shoes[2]});
  viewMock.Expect("SetSelectedShoeInGrid"
    , new object[]{shoes[0]});
  ctrl.LoadView();
  viewMock.Verify();
}
```

Итак, все тесты готовы (хотя в приведенном выше коде показан лишь один из них), и теперь остается показать фактическую реализацию класса `CatalogController` (см. приведенный ниже код).

```
public class CatalogController
{
  ICatalogView _view;
  IList _shoes;
  Shoe _selectedShoe;

  public CatalogController(ICatalogView view, IList shoes)
  {
    _view=view;
    _shoes=shoes;
    view.SetController(this);
  }

  public void LoadView()
  {
```

```
    _view.ClearGrid();
    foreach(Shoe s in _shoes)
      _view.AddShoeToGrid(s);

    _view.SetSelectedShoeInGrid((Shoe)_shoes[0]);
  }
}
```

Упрощение интерфейсов представления с помощью адаптеров

В примере приложения для обувного магазина Джо интерфейс `ICatalogView` подробно раскрывает свойство, доступ к которому требуется контроллеру. Но это в идеале, а на практике интерфейсы могут оказаться слишком крупными, если в форме используются 10 полей ввода и требуется управлять их видимостью, доступностью, шрифтовым оформлением, цветом заднего и переднего плана, стилем рамки и прочими свойствами. Во избежание подобных осложнений можно использовать шаблон Adapter/Wrapper (Адаптер/Оболочка) [GoF Design Patterns] для каждого вида элементов управления пользовательского интерфейса. В частности, адаптер для класса `TextBox` может раскрыть упомянутые выше свойства, а интерфейсу представления останется только раскрыть одно свойство типа данных `TextBoxAdapter` для каждого текстового поля. При получении экземпляра внутри представления адаптеру предоставляются ссылки на конкретные элементы пользовательского интерфейса, а сам адаптер просто передает получаемый вызов конкретной реализации.

Развязка контроллера от представления

Иногда целесообразно развязывать контроллер от представления. Это можно сделать несколькими способами. Самый простой способ состоит в том, чтобы создать интерфейс для контроллера и дать представлению знать только об этом, но не о конкретной реализации. А более сложный способ состоит в том, чтобы представления инициировали события для анонимных подписчиков, используя, в частности, шаблон Observer (Наблюдатель) [GoF Design Patterns]. Я лично не большой сторонник такого подхода, поскольку мне трудно себе представить, зачем одному представлению нужно сообщать о действиях пользователя многим подписчикам, особенно подписчикам неизвестного типа! Если другим контроллерам или даже представлениям требуется знать об изменениях в конкретном представлении, об этом следует уведомить контроллер данного представления. Это можно, в частности, сделать с помощью событий. Главное, чтобы контроллер полностью следил за тем, кто и что должен знать.

Независимо от конкретных деталей реализации преимущество развязки контроллера от представления заключается в том, что контроллер может меняться незаметно для представления. Это, в частности, удобно в том случае, если представление должно проявляться в разных контекстах, и для этого нецелесообразно связывать его с конкретным контроллером. Примером тому может служить специальная логика, применяемая в представлении для конкретных клиентов. В этом случае может

потребоваться фабричный метод, выполняющий роль соответствующего контроллера в зависимости от конкретного клиента, работающего с системой.

Сочетание представлений и контроллеров

Сложные представления обычно состоят из множества простых представлений. Примерами тому служат мастера, в которых пользователь переходит от одного экрана к другому с помощью кнопок перемещения вперед и назад, а возможно, и сразу на несколько экранов вперед или назад. В ходе этого перемещения сохраняется соответствующая информация, хотя она не фиксируется до тех пор, пока пользователь не щелкнет на кнопке Save (Сохранить) на последнем экране.

Другим примером могут служить основные и детальные виды, где изменения в детальном виде приводят к изменениям в основном виде.

Оба примера могут быть формально реализованы с помощью одного крупного контроллера, но тогда сведение к минимуму повторного использования кода превратится в малоприятное занятие. Так, отдельные этапы процедуры мастера могут потребоваться в другом месте приложения, а основной вид, возможно, придется показывать без детального вида. Очевидно, что в таком случае представление и контроллеры следует разделить на более мелкие части по принципу: “Если нужно съесть слона, это придется сделать по частям.” В примере мастера можно было бы создать класс `FlowController` для управления основной последовательностью операций перемещения вперед и назад по мастеру, причем у каждого этапа процедуры мастера могли бы быть свои контроллер и представление.

В примере основного и детального видов можно было бы инициировать в контроллере события при всяком изменении детального вида. А любой другой контроллер, в том числе и контроллер основного вида, мог бы тогда подписаться на данное событие.

Чего все это стоит

Главное преимущество применения шаблона MVC заключается в том, что он упрощает тестирование кода. Кроме того, он вынуждает применять весьма структурированный подход к процессу программирования и разработки пользовательского интерфейса, что само по себе дает в конечном итоге более ясный код. Иными словами, шаблон MVC дисциплинирует мышление.

С другой стороны, следует заметить, что *возможность* тестировать каждую строку кода совсем не означает, что это *обязательно* делать. Применяя шаблон MVC, можно написать обширный набор тестов для каждого представления, чтобы получить на первых порах отрицательные результаты (зеленые полосы) тестирования логики пользовательского интерфейса. Но ценность этих простых тестов, скорее всего, значительно убавится при последующих изменениях в коде из-за постепенного ослабления зависимости от других частей системы. Это особенно справедливо, если наборы тестов пользовательского интерфейса накладывают большие издержки на сопровождение базы

кода. Достаточно немного изменить пользовательский интерфейс, чтобы затем вносить существенные коррективы в соответствующие тесты этого интерфейса.

Благодарю, Крис!

Независимо от потребности проверять пользовательский интерфейс с помощью строгих тестов, Кристофер продемонстрировал нам очень ценный метод рациональной разработки видовой части приложения в среде Windows Forms. При этом основное внимание уделяется выводу форм на экран, а логика их представления выносится в отдельный слой.

Данной теме посвящена и бо́льшая часть следующего раздела, но на этот раз она рассматривается в контексте Web-приложения. В этом разделе Ингмар Лундберг поделится своим опытом применения шаблона MVP [Fowler PoEAA2] в качестве средства тестирования Web-приложения.

Разработка Web-формы посредством тестирования

Ингмар Лундберг

Как выполнить блочное тестирование графического пользовательского интерфейса, применяя PPT?

В этом кратком разделе я попытаюсь дать общее представление о том, как выполняется блочное тестирование графического пользовательского интерфейса. А по ходу дела будут представлены такие методы, как самошунтирование [Feathers Self-Shunt] и имитация. При этом особенности PPT не поясняются. Самошунтирование вкратце означает метод, позволяющий тестовому классу действовать также в качестве заглушки или имитации.

Предпосылки

Типичный код графического пользовательского интерфейса состоит из целого ряда методов обратного вызова, которые вызываются в результате определенных действий пользователя в этом интерфейсе, например, после нажатия кнопки. Как правило, метод обратного вызова выполняет определенные операции над объектом, а возможно, и некоторые операторы SQL. Но дело в том, что методы обратного вызова обычно вызываются из внешней среды, включая WinForms или ASP.NET, которую трудно связать с тестами в среде NUnit, а по существу, это просто невозможно.

Основная цель, которая преследуется при разработке приложений, — свести к минимуму непроверенный код. Для этой цели следует вынести как можно больше кода из методов обратного вызова. Если код методов обратного вызова не очень запутан, значит, для его проверки можно создать блочный тест.

Но этого недостаточно. Необходимо также учитывать состояние пользовательского интерфейса. Так, если пользователь выделил отдельную строку таблицы, то долж-

на быть отключена соответствующая кнопка. Этого нельзя добиться одним только перемещением кода за пределы методов обратного вызова. Следует ясно осознавать, что это лишь первый шаг в правильном направлении.

А далее необходимо воспользоваться самым острым инструментом, который только имеется в арсенале средств разработчика: *абстракцией*. Следовательно, нужна модель взаимодействия пользователя с интерфейсом и абстракция его формы для программирования. С этого момента мы будем вести речь не о графическом пользовательском интерфейсе в общих чертах, а о конкретной форме и, в частности, о Web-форме, разрабатываемой в среде ASP.NET. Впрочем, рассматриваемый здесь метод применим и в среде Windows Forms. Его можно даже считать независимым от конкретной платформы.

Пример

Когда приходится писать о программировании, то самое трудное — найти подходящий пример, который мог бы проиллюстрировать обсуждаемый вопрос, не отвлекая читателя на второстепенные вопросы. В данном случае требуется подходящий пример, в котором изменялось бы состояние графического пользовательского интерфейса. Иными словами, необходимо отключить какой-нибудь элемент этого интерфейса, как только пользователь выполнит определенное действие.

В Швеции существует традиция позволять детям лакомиться леденцами по субботам. Допустим, что у нас есть одна коробка с плитками шоколада, другая коробка с леденцами на палочке и третья коробка с фасованным арахисом. У детей имеются три вида сластей на выбор, но не более двух одинаковых видов. Упрощенно такая модель предполагает, что сласти никогда не закончатся.

Это, без сомнения, очередной простоватый пример, но будем надеяться, что он сделает свое дело. На рис. 11.2 приведен конечный результат. Можно было привести сделанный от руки набросок графического пользовательского интерфейса, но и этого будет достаточно. В левом подокне представлены все имеющиеся виды сластей. Именно в этом подокне пользователь (т.е. ребенок любого возраста) выбирает сласти про запас на субботу, как показано в правом подокне. Как видите, выбрать третью плитку шоколада уже нельзя, но можно выбрать арахис и леденцы.

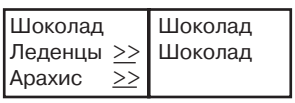


Рис. 11.2. Варианты выбора, ограничения и выбранное

Прошу читателя не обращать внимание на столь упрощенное оформление пользовательского интерфейса, поскольку данный вопрос выходит за рамки этого раздела.

Модель предметной области

Для рассматриваемого примера нам требуется модель предметной области. Она представлена на рис. 11.3.

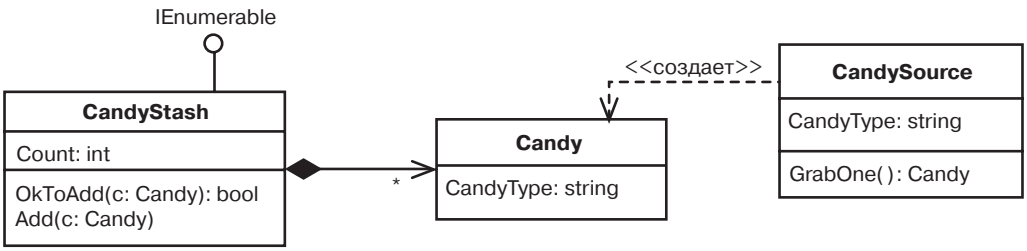


Рис. 11.3. Модель предметной области

Я проверил эту модель отдельно. В частности, метод `OkToAdd()` проверяет установленные выше бизнес-правила: ввести можно три позиции сластей, но не более двух одинакового вида (`CandyType`). Как видите, в этом нет ничего сложного.

PPT графического пользовательского интерфейса

И наконец, мы подошли к самому главному. Но прежде мне бы хотелось напомнить следующее: несмотря на то что мы создаем абстракцию формы, мы должны иметь достаточно полную картину происходящего, как показано на рис. 11.4.

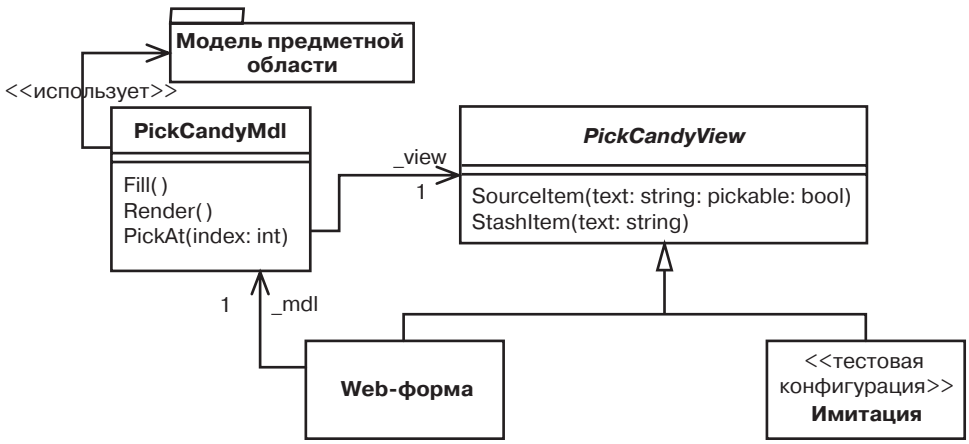


Рис. 11.4. Модель пользовательского интерфейса

Отображение видов сластей

При отображении формы в первый раз должна быть предоставлена возможность сначала для заполнения левого подокна названиями всех видов сластей, а затем и для

выбора сластей из всех трех источников. Ниже приведен тест этих функциональных возможностей пользовательского интерфейса.

```
[Test] public void FillSourcePanel()
{
  mdl.Fill();
  mdl.Render();
  // использовать A=NUnit.Framework.Assert
  A.AreEqual(SrcPanel, "a> b> c> ");
}
```

Этот тест даже не компилируется. В класс `PickCandyMdl` (Модель выбора сластей) необходимо ввести методы заполнения и визуализации формы. Сделать это нетрудно. Нужно лишь ввести два пустых общедоступных метода `Fill()` и `Render()`. Но что выражает данный тест? Что он утверждает? Утверждение выражает вид формы — в данном случае абстрактный. Строка `SrcPanel` (Панель источников) используется вместо HTML-таблицы в левом подокне формы, которая и есть нашей конечной целью.

Откуда же появилась строка `SrcPanel`? Эта строка из тестовой конфигурации (класса), которой оперирует интерфейс `PickCandyView` (Представление выбора сластей), реализованный методом самошунтирования [Feathers Self-Shunt]. Это шаблон, повторяющийся при имитации представления.

```
[TestFixture] public class PickCandyTests: PickCandyView
{
  void PickCandyView.SourceItem(string text, bool   pickable)
  {
    SrcPanel += text + (pickable ? ">"
      : string.Empty) + " ";
  }

  PickCandyMdl mdl;
  string SrcPanel;

  [SetUp]
  protected void Setup()
  {
    mdl = new PickCandyMdl();
    SrcPanel = string.Empty;
  }
```

Тест компилируется, но дает положительный результат (красную полосу). Чего же недостает? Пожалуй того, что модели формы типа `PickCandyMdl` должно быть известно ее представление. Для этого достаточно ввести в модель метод `SetView()`. (Между прочим, это один из тех редких случаев, когда реализацию интерфейса можно использовать аналогично методу `PickCandyView.SourceItem()` в формате {имя_интерфейса}.{имя_члена}.)

На заметку

Реализация методов может существовать только в классах. Так, если класс `Foo` реализует интерфейс `Bar`, то методы этого интерфейса становятся частью данного класса. Как правило, члены интерфейса `Bar` реализуются в классе `Foo` как общедоступные. Не следует, однако, забывать, что интерфейс самого класса состоит из всех его членов. А раз `Foo` — это `Bar`, то и члены интерфейса `Bar` являются частью класса `Foo`. Иногда же члены интерфейса реализуются как закрытые для разрешения конфликтов имен, как, например, при имитации методом самошунтирования. Но в рассматриваемом здесь примере интерфейсы (`PickCandyView` и `CandySourceRetriever`) не являются частью интерфейса тестовой конфигурации.

В форме модели должны быть известны все источники для получения сластей. Для этого можно было бы снабдить модель списком источников сластей. Но вместо этого мы рассмотрим другой прием, улучшающий тестируемость кода в целом. Он состоит в том, чтобы предоставить модели формы собственный интерфейс, представляющий требуемую операцию выборки.

```
public interface CandySourceRetriever
{
  IList Retrieve();
}
```

Почему обозначение интерфейсов не всегда начинается с буквы I

Такое обозначение было принято в среде, где само представление интерфейса было обозначением, правилом, но не понятием языка высокого уровня, хотя это положение, так или иначе, изменилось с появлением компилятора midl. При попытке получить экземпляр интерфейса с помощью оператора `new` компилятор C# (по крайней мере, в той версии, которой пользуюсь я) интерпретирует абстрактные классы и интерфейсы одинаково.

```
public interface Bar {}
public class Foo { public static void Main() {new Bar();} }
ingo.cs(2,47): error CS0144: Невозможно создать экземпляр
абстрактного класса или интерфейса
'Bar'
```

Почему же обозначение абстрактных классов не начинается с буквы A? Потому что при обращении к типу данных, имеющему определенные методы и свойства, совершенно не важно, является ли он интерфейсом, классом или конкретной копией. По-видимому, обозначение интерфейсов, начинающееся с буквы I, было принято в стандартной библиотеке классов (Framework Class Library — FCL) без каких-то дополнительных соображений.

Другое, более сложное пояснение

Обозначение интерфейсов, начинающееся с буквы I, оказалось уместным в среде COM, которая на самом деле была (и есть) не объектно-ориентированной, в компонентно-ориентированной. По этой причине, вероятно, не имело смысла интерпретировать интерфейсы компонентов как нечто, не присущее самому компоненту. Но тогда возникает вопрос: являются интерфейсы в FCL интерфейсами компонентов или же являются ли они компромиссным решением трудной (для компилятора) задачи множественного наследования? Если это так, то интерфейс в среде CLR очень близок к абстрактному классу. А раз отсутствует обозначение абстрактных классов, начинающееся с буквы A, то можно не придерживаться обозначения интерфейсов, начинающегося с буквы I, поскольку в этом нет никакой надобности. Опять же, при использовании или обращении к типу данных, имеющему определенные методы и свойства, совершенно не важно, является ли он интерфейсом, классом или конкретной копией.

Этот интерфейс имитируется также методом самошунтирования. Разумеется, модели формы должно быть известно об операции выборки. С учетом всего сказанного выше тестовая конфигурация начинается со следующего:

```
[TestFixture] public class PickCandyTests: PickCandyView,
CandySourceRetriever
{
  void PickCandyView.SourceItem(string text, bool pickable)
  {
    SrcPanel += text + (pickable ? ">"
      : string.Empty) + " ";
  }

  IList CandySourceRetriever.Retrieve()
  {
    return sources;
  }

  PickCandyMdl mdl;
  ArrayList sources;
  string SrcPanel;

  [SetUp] protected void Setup()
  {
    mdl = new PickCandyMdl();
    mdl.SetView(this);
    mdl.SetRetriever(this);
    sources = new ArrayList();
    sources.Add(new CandySource("a")); // выбрать шоколадку
```

```
    sources.Add(new CandySource("b")); // выбрать леденец
    sources.Add(new CandySource("c"));
    SrcPanel = string.Empty;
  }
```

Этот тест по-прежнему дает положительный результат (красную полосу), поскольку мы еще ничего не реализовали в классе `PickCandyMdl`. Сделаем это следующим образом:

```
[Serializable] public class PickCandyMdl
{
  public void Fill()
  {
    _retrieved = _retriever.Retrieve();
  }

  public void Render()
  {
    foreach(CandySource s in _retrieved)
      _view.SourceItem(s.CandyType, true);
  }

  public void SetView(PickCandyView v)
  {
    _view = v;
  }

  public void SetRetriever(CandySourceRetriever r)
  {
    _retriever = r;
  }

  [NonSerialized] PickCandyView _view;
  CandySourceRetriever _retriever;
  IList _retrieved;
}
```

Как видите, большая часть этого кода достаточно проста. В методе `Fill()` выбираются доступные источники сластей, а в методе `Render()` циклически просматривается информация для вывода на экран. Следует, однако, иметь в виду, что до сих пор реализация лишь имитировалась, а сами источники сластей не всегда должны быть выбираемыми (на что указывает константа `true`). Код методов `SetView()` и `SetRetriever()` также достаточно прост. Атрибут `[NonSerialized]` интерфейса `_view` требуется в том случае, если обработка состояния `Session` выполняется вне процесса в приложении ASP.NET.

Теперь тест дает наконец-то отрицательный результат (зеленую полосу)!

Выбор сластей

Далее нам нужно организовать выбор сластей из списка. Выразим это в виде теста следующим образом:

```
[Test] public void PickingCandy ()
{
  mdl.Fill();
  mdl.PickAt(0);
  mdl.Render();
  A.AreEqual(SrcPanel, "a> b> c> ");
  A.AreEqual(StashPanel, "a ");
}
```

`StashPanel` — это представление в тестовой конфигурации правого подокна, в котором отображаются результаты выбора сластей (в виде HTML-таблицы или настоящей формы). Для этой цели требуется новый метод `PickAt()`, как, впрочем, и новый метод для отображения результатов.

```
public interface PickCandyView
{
  void SourceItem(string text, bool pickable);
  void StashItem(string text);
}
```

Метод `PickAt()` служит удобным местом для внедрения в модель формы перечня выбранных про запас сластей, обозначаемого как `_stash` в следующем фрагменте кода:

```
public void PickAt(int index)
{
  Candy c = ((CandySource)_retrieved[index]).GrabOne();
  _stash.Add(c);
}
```

Кроме того, метод `Render()` необходимо дополнить следующим кодом:

```
public void Render()
{
  foreach(CandySource s in _retrieved)
    _view.SourceItem(s.CandyType, true);
  foreach(Candy c in _stash)
    _view.StashItem(c.CandyType);
}
```

Разумеется, метод `StashItem()` можно реализовать таким же образом, как и метод `SourceItem()`.

Этот тест вновь дает отрицательный результат (зеленую полосу). Но прежде чем двигаться дальше, дополним первый тест утверждением о том, что правое подокно `StashPanel` пусто (`string.Empty`).

Динамическое отображение результатов выбора сластей

Если выбраны две сласти одного вида, этот вид сластей уже не должен выбираться (см. рис. 11.2). Выразим это в виде теста следующим образом:

```
[Test] public void PickingTwoOfSameKind()
{
  mdl.Fill();
  mdl.PickAt(0); mdl.PickAt(0);
  mdl.Render();
  A.AreEqual(SrcPanel, "a b> c> ");
  A.AreEqual(StashPanel, "a a ");
}
```

Перечень выбранных про запас сластей теперь содержит две сласти вида `"a"`. Обратите внимание на отсутствие знака `>` после `"a"` в выражении `A.AreEqual(SrcPanel, "a b> c> ")`. В данном тесте не введены новые методы, тем не менее, он дает положительный результат (красную полосу) из-за того, что метод `Render()` реализован ранее в сокращенной форме. Напомним о константе `true` и в связи с этим дополним код перечня выбранных про запас сластей методом `OkToAdd()` следующим образом:

```
public void Render()
{
  foreach(CandySource s in _retrieved)
    _view.SourceItem(s.CandyType, _stash.OkToAdd(s.GrabOne()));
  foreach(Candy c in _stash)
    _view.StashItem(c.CandyType);
}
```

В этом фрагменте кода сласть извлекается из источника (`s.GrabOne()`) и передается методу `OkToAdd()`, который определяет, можно ли ее выбрать. `CandySource` — это бесконечный источник (а по существу, объект фабрики). В реальной жизни такой запас шоколадок оказался бы практически неисчерпаемым, но это не реальная жизнь, а всего лишь программа.

Итак, если выбрано три сласти разного вида, то выбирать сласти больше нельзя. Выразим и это условие в тесте следующим образом:

```
[Test] public void PickingThreeMix()
{
  mdl.Fill();
  mdl.PickAt(0); mdl.PickAt(0); mdl.PickAt(1);
  mdl.Render();
  A.AreEqual(SrcPanel, "a b c ");
  A.AreEqual(StashPanel, "a a b ");
}
```

Этот тест дает отрицательный результат (зеленую полосу). Проверка сластей на предмет их выбора в методе `OkToAdd()` оказалась удачной идеей. Впрочем, это не

удивительно, поскольку данный метод воплощает саму суть бизнес-правила в данном примере приложения.

Реализация Web-формы

Обработка состояния в Web-приложении может быть организована несколькими способами. В данном примере состояние отображения в форме документа отключено, а для обработки состояния используется только объект `Session` на сервере. Страница формируется после выбора Web-формы и затем переформируется непосредственно перед завершением обработки (события `PreRender`) для отражения текущего состояния модели. Напомним, что состояние модели меняется при выборе сластей про запас.

Для этой цели введем сначала вызов метода `MyInit()` в стандартный метод `OnInit()` в Web-форме, созданной средствами Visual Studio .Net.

```
void MyInit()
{
  if (!IsPostBack)
  {
    _mdl = new PickCandyMdl();
    Session["PickCandyMdl"] = _mdl;
    _mdl.SetRetriever(CreateRetriever());
    _mdl.Fill();
  }
  else
  {
    _mdl = (PickCandyMdl)Session["PickCandyMdl"];
  }
  _mdl.SetView(this);
  MdlRender();
}

PickCandyMdl _mdl;
HtmlTable _srcTbl, _stashTbl;

void MdlRender()
{
  _srcTbl = new HtmlTable();
  srcpnl.Controls.Add(_srcTbl);
  _srcTbl.Width = "100%";

  _stashTbl = new HtmlTable();
  stashpnl.Controls.Add(_stashTbl);
  _stashTbl.Width = "100%";

  _mdl.Render();
}
```

```
private void WebForm1_PreRender(object sender, System.EventArgs e)
{
  srcpnl.Controls.Clear();
  stashpnl.Controls.Clear();
  MdlRender();
}
```

Панель источников сластей (`srcpnl`) и панель выбранных про запас сластей (`stashpnl`) относятся к элементам управления типа `Panel` в пространстве имен `WebControls` и введены с помощью конструктора форм в среде Visual Studio.

Разумеется, в данной форме реализован интерфейс `PickCandyView`. Его код достаточно прост. Любое решение, принимаемое с помощью оператора `if`, касается только внешнего вида формы, но не состояния взаимодействия с ней пользователя. Это очень важно, поскольку за последнее должна отвечать модель.

```
void PickCandyView.SourceItem(string text, bool pickable)
{
  int index = _srcTbl.Rows.Count;
  HtmlTableRow tr = AddTR(_srcTbl);
  HtmlTableCell td = AddTD(tr);
  td.InnerText = text;

  td = AddTD(tr);
  if (pickable)
  {
    LinkButton lb = new LinkButton();
    td.Controls.Add(lb);
    lb.ID = "pick@" + index.ToString();
    lb.Text = ">>";
    lb.Click += new EventHandler(Pick_Click);
  }
}

void PickCandyView.StashItem(string text)
{
  AddTD(AddTR(_stashTbl)).InnerText = text;
}
```

И наконец, организуем обращение к методу обратного вызова следующим образом:

```
private void Pick_Click(object sender, EventArgs e)
{
  _mdl.PickAt(IndexFromID(sender));
}
```

Думаю, что этот код особых пояснений не требует. Вам же остается самое простое — реализовать метод `CreateRetriever()`. Все, что для этого требуется, можно обнаружить в рассмотренной выше тестовой конфигурации.

На рис. 11.5 приложение для выбора сластей представлено в виде последовательности иллюстраций к комиксу. Дети выбирают сначала две плитки шоколада, а затем пакетик арахиса.

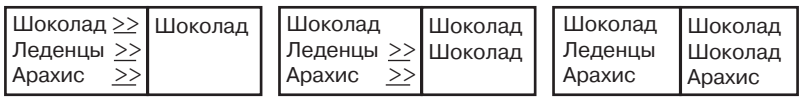


Рис. 11.5. Приложение для выбора сластей

Выводы

Большая часть кода для графического пользовательского интерфейса находится в модели взаимодействия с пользователем. И только небольшая часть простого кода остается непроверенной. Но такой подход предполагает тестирование не графического представления, а динамики работы графического пользовательского интерфейса.

В примере, рассмотренном выше не до конца, мы обошлись самой минимальной обработкой в модели формы. Заключение объекта `CandyItem` в оболочку может стать одним из путей дальнейшего усовершенствования данного приложения, если, например, потребуется добавить возможность отказа от выбора (и возврата сласти в источник).

Имитация средствами NMock

В качестве небольшого дополнения мне хотелось бы продемонстрировать еще один способ имитации средствами NMock [NMock] во время тестирования. Я буду краток, предоставив тестовому коду самому говорить за себя, а вам — сравнить его с предыдущими тестами. Следует только заметить, что в классе тестовой конфигурации не реализуются интерфейсы `PickCandyView` и `CandySourceRetriever`, как это было сделано ранее в тестовой конфигурации, полученной методом самошунтирования.

```
PickCandyMdl mdl;
IMock data, view;
const bool Pickable=true, NotPickable=false;

[SetUp] protected void Setup()
{
  data = new DynamicMock(typeof(CandySourceRetriever));
  view = new DynamicMock(typeof(PickCandyView));

  mdl = new PickCandyMdl();
  mdl.SetView((PickCandyView)view.MockInstance);
  mdl.SetRetriever((CandySourceRetriever)data.MockInstance);
  ArrayList sources = new ArrayList();
  sources.Add(new CandySource("a")); // выбрать шоколадку
  sources.Add(new CandySource("b")); // выбрать леденец
  sources.Add(new CandySource("c"));
```

```
  data.SetupResult("Retrieve", sources);
}
```

Главное в этом коде заключается в создании экземпляров объектов имитации `DynamicMock` и объектов, принадлежащих к указанным интерфейсам. Динамически "кодируемым и компилируемым" объектом имитации в данном случае оказывается `IMock.MockInstance`. С помощью метода `SetupResult()` имитация данных получает указание возвратить источники после вызова метода `Retrieve()`.

Ниже представлен эквивалент рассмотренного ранее первого теста.

```
[Test] public void FillSourcePanel()
{
  // Предположения
  view.Expect("SourceItem", "a", Pickable);
  view.Expect("SourceItem", "b", Pickable);
  view.Expect("SourceItem", "c", Pickable);
  view.ExpectNoCall("StashItem", typeof(string));
  // Выполнение
  mdl.Fill();
  mdl.Render();
  // Проверка
  view.Verify();
}
```

Комментарии к этому коду обозначают лишь основные стадии тестирования (кроме настройки) средствами NMock. В объявлении указывается, что метод `SourceItem()` интерфейса `PickCandyView` предполагается вызвать три раза и при этом получить соответствующие значения. Кроме того, указывается, что метод `StashItem()` не должен вызываться.

На стадии выполнения происходит обращение к тестируемому объекту посредством вызова некоторых его методов. И наконец, имитация представления получает указание проверить, насколько оправдались наши предположения во время выполнения.

Рекомендуется придерживаться известной формулы РПТ: красная полоса, зеленая полоса, рефакторинг кода. По крайней мере, если забыть в приведенном выше тесте вызвать метод `Verify()`, тест в конечном итоге даст отрицательный результат (зеленую полосу) независимо от того, оправдались предположения или нет. Если же сначала тест дает положительный результат (красную полосу), то можно быть уверенным, что кое-что все же проверяется (утверждается).

Благодарю, Ингмар!

Следует еще раз подчеркнуть, что тестирование имеет решающее значение для разработки приложений, и поэтому к нему нужно готовиться и соответственно настраиваться на него. Итак, выше были представлены два разных подхода к разработке пользовательского интерфейса, хотя в их применении можно обнаружить некоторое сходство.

И в заключение будут рассмотрены не менее важные вопросы представления данных для пользовательского интерфейса, в отличие от данных в модели предметной области.

Рассмотрение вопросов преобразования для служб представления может показаться мелочным и излишним, поскольку преобразование является весьма распространенным решением для служб сохранения. Однако если сначала сделать главный акцент на базовой модели предметной области, а затем выработать простое, гибкое решение таких задач инфраструктуры, окружающей эту модель, как ее преобразование в другую модель для иного яруса приложения, то это будет вполне в стиле ППО. Именно поэтому я усматриваю немалые потенциальные возможности в объектно-объектном преобразовании и преобразовании пользовательского интерфейса, особенно для решения типичных задач, на которые не хотелось бы тратить время, выполняя кодирование вручную.

В следующем разделе Матс Хеландер представит на обсуждение некоторые аспекты преобразования пользовательского интерфейса, сосредоточив основное внимание на автоматическом преобразовании удобной для представления модели, например, по шаблону Presentation Model [Fowler PoEAA2], в модель предметной области, и обратно. Итак, предоставим слово Матсу.

Преобразование и заключение в оболочку

Матс Хеландер

С помощью объектно-реляционного преобразования мы можем связать нашу модель предметной области с реляционной базой данных. Вместе с методами, которые относятся к конкретной сфере деятельности и вводятся в классы модели предметной области, а возможно, и в шаблон Service Layer (Уровень обслуживания) [Fowler PoEAA], мы получаем довольно прочную архитектуру в качестве структурного и функционального основания для приложения.

Это приводит нас к последней стадии — *уровню представления*. Если разработчики считают модель предметной области сердцевиной приложения, то для конечных пользователей такой сердцевиной, скорее всего, является то, что им представлено.

Это означает, что независимо от того, насколько хорошо построена модель предметной области, включая и поддерживающий ее шаблон Service Layer, никто не оценит эту модель по достоинству (кроме коллег-разработчиков), если не предоставить доступ к ней посредством удобного пользовательского интерфейса.

Если уровень представления удастся связать с моделью предметной области, то все будет отлично, а если не удастся, то пользоваться ею будет очень трудно.

В простейшем случае модель предметной области может уже быть "представляемой", и тогда остается лишь привязать данные непосредственно к объектам из этой модели. Но для интерфейса приложения нередко требуется представление информа-

ции в более удобном для пользователя виде, чем тот, в котором она находилась в модели предметной области.

Преобразование и заключение объектов в оболочку

Всякий раз, когда требуется немедленно сделать представляемыми объекты из модели предметной области (МПО), приходится выбирать между *заключением* этих объектов *в оболочку* или их *преобразованием* в новый набор объектов.

Допустим, что в МПО имеется класс `Person` (Физическое лицо) со свойствами `FirstName` (Имя) и `LastName` (Фамилия). А в пользовательском интерфейсе для правки Ф.И.О. физического лица предусмотрено лишь одно текстовое поле.

В целях решения этой задачи можно, в частности, написать для пользовательского интерфейса код, просто читающий имя и фамилию из объекта `Employee` (Служащий) в МПО и записывающий результат в текстовое поле, а затем читающий результат правки из этого поля и записывающий его обратно в свойства объекта `Employee`, когда пользователь нажимает кнопку Save.

В действительности именно так чаще всего требования к пользовательскому интерфейсу и удовлетворяются, но при этом накапливается масса неупорядоченного кода.

В конечном итоге такой путь не ведет к удобному сопровождению приложения, что заставляет искать проектные решения, допускающие расширение по мере увеличения сложности и масштабов приложения.

Одно из современных решений состоит в том, чтобы дополнить объекты МПО новым набором объектов, нередко называемым *моделью представления* (МП).

Объекты модели представления являются частью уровня представления, и поэтому должны иметь структуру и поведение, соответствующие этому уровню (а по существу, потребностям пользовательского интерфейса). В данном примере у класса `Person` МП должно быть свойство `FullName` (Ф.И.О.), а не свойства `FirstName` и `LastName`, доступные в классе `Person` МПО.

А почему бы не ввести в класс `Person` МПО метод `GetFullName()`? Конечно, мы могли бы это сделать, если бы это было целесообразно для выполнения некоторой логической операции в предметной области, но если метод `GetFullName()` требуется только на уровне представления, он, скорее, подойдет для МП, чем для МПО.

Модель предметной области должна быть свободна от таких аспектов, не связанных с самой предметной областью, как представление и сохраняемость. Это так же важно, как и некогда освобождение уровня бизнес-логики от логики представления и логики доступа к данным, по одной простой причине: сердцевина приложения должна быть понятной и удобной для сопровождения.

Итак, допустим в данном примере, что метод `GetFullName()` применяется только для представления и поэтому пригоден, скорее, для МП, чем для МПО.

Остается решить следующий вопрос: как связать классы МП с классами МПО? Ответ на этот вопрос приводит нас к необходимости выбора между *преобразованием* и *заключением в оболочку*.

Заключение МПО в оболочку МП

Как правило, заключение в оболочку оказывается более простым решением, не требующим дополнительной базовой структуры для управления состоянием в МП. Такое решение основывается на том, что объект `Employee` МПО передается правилу-конструктору объекта `Employee` МП. В объекте `Employee` МП сохраняется внутренняя ссылка на объект МПО, а все вызовы передаются свойствам МПО, как показано в приведенном ниже фрагменте кода.

```
// Объект-оболочка из модели представления
namespace MyCompany.MyApplication.Presentation
{
  public class EmployeeWrapper
  {
    private Employee employee;

    public EmployeeWrapper (Employee employee)
    {
      this.employee = employee;
    }

    public int Id
    {
      get { return this.employee.Id; }
      set { this.employee.Id == value; }
    }

    public string FullName
    {
      get
      {
        return this.employee.FirstName + " " +
        this.employee.LastName;
      }
      set
      {
      // Этот код должен быть, конечно, дополнен
      // более развитой логикой, а также соответствующей
      // проверкой, хотя это всего лишь пример...
      string[] names = value.Split(" ".ToCharArray()
        , 2);
      this.employee.FirstName = names[0];
      this.employee.LastName = names[1];
      }
    }
  }
}
```

Применение класса `EmployeeWrapper` (Оболочка объекта служащего), по существу, означает выборку объекта `Employee` из МПО и последующую его передачу конструктору класса `EmployeeWrapper`, как показано в приведенном ниже фрагменте кода.

```
// Применение объекта-оболочки из модели представления
Employee employee = employeeRepository.GetEmployeeById(42);
EmployeeWrapper employeeWrapper = new EmployeeWrapper(employee);

SomeControl.DataSource = employeeWrapper;
```

Главное преимущество метода заключения в оболочку — в его простоте и гибкости. В объекты-оболочки можно заключить практически любые преобразования, причем намного более сложные, чем такое относительное простое преобразование, как Ф.И.О. в приведенном выше примере.

Главный недостаток метода заключения в оболочку — в необходимости писать немало повторяющегося кода в классах оболочки для передачи функций свойствам, не требующим никакого преобразования, как, например, свойство `Id` в приведенном выше примере.

При этом может возникнуть искушение просто предоставить объектам МП возможность *наследовать* от объектов МПО вместо переопределения любых свойств, которым требуется преобразование, оставив без изменения все остальные свойства. Но дело в том, что тогда приходится привязываться к общедоступному интерфейсу API МПО, поскольку все общедоступные члены суперкласса наследуются подклассом.

Так, в упомянутом выше примере можно было бы предоставить объекту `Employee` МП возможность наследовать от объекта `Employee` МПО и ввести свойство `FullName` в объект `Employee` МП, но тогда объекту `Employee` МП пришлось бы *также* раскрыть свойства `FirstName` и `LastName` объекта `Employee` МПО, поскольку эти общедоступные свойства наследуются.

Заключение объектов МПО в оболочку вместо наследования от них обеспечивает более высокий уровень инкапсуляции, хотя и за счет дополнительной и, скорее всего, трудоемкой работы по передаче функций между свойствами МП и МПО.

Впрочем, код передачи функций свойствам, не требующим дополнительных преобразований, можно благополучно сформировать посредством генерации кода или даже шаблонов фрагментов кода. В противном случае написание шаблонного кода следует поручить стажирующемуся программисту!

Преобразование МП в МПО

Альтернативой заключения объектов МПО в оболочку объектами МП, по существу, является копирование данных из объектов МПО в объект МП и обратно. Для этой цели может подойти следующий код:

```
// Перемещение данных вручную из МП в МПО и обратно
Employee employee = employeeRepository.GetEmployeeById(42);

EmployeeView employeeView = new EmployeeView();
employeeView.Id = employee.Id;
employeeView.Salary = employee.Salary;
employeeView.FullName = employee.FirstName + " " +
employee.LastName;
```

Как и при заключении в оболочку, код для копирования данных может быть заменен конструктором объекта МП. В этом случае конструктор воспринимает объект МПО в виде параметра. В приведенном выше фрагменте кода ради ясности показаны только строки кода операций перемещения данных.

Если вы подвержены *мании абстракции*, как и я, то сразу же оцените потенциальные возможности абстрагирования такой операции до некоторой базовой структуры. Иными словами, мы должны суметь указать (например, в XML-файле), какие именно свойства МП преобразуются в соответствующие свойства МПО, а затем использовать в базовой структуре рефлексию для перемещения данных.

Такая базовая структура должна преобразовать один набор объектов в другой набор объектов, и поэтому ее было бы логично назвать объектно-объектным преобразователем!

Если попытаться создать подобную базовую структуру, то станет ясно, что разрешение таких ситуаций, как преобразование одного свойства `Id` в другое свойство `Id` и одного свойства `Salary` в другое свойство `Salary` в приведенном выше фрагменте кода, превращается в достаточно простую задачу. А вот преобразовать свойство `FullName` в свойства `FirstName` и `LastName` будет сложнее.

Конечно, эта задача вполне разрешима, вопрос только в том, насколько такой подход приемлем для встраивания развитых служб преобразования в базовую структуру объектно-объектного преобразователя.

Но вместо того, чтобы наделять объектно-объектный преобразователь функциями структурных преобразований, можно сосредоточить основное внимание только на перемещении данных между объектами, разделяющими одну и ту же или сходную структуру.

В рассматриваемом примере это означает, что в класс `Employee` МП вводится свойство `FullName` в дополнение к свойствам `FirstName` и `LastName`, но в то же время два последних свойства делаются защищенными.

Объектно-объектный преобразователь должен без особых затруднений получать доступ к защищенным свойствам класса МП посредством рефлексии, чтобы преобразовывать в них соответствующие свойства `FirstName` и `LastName` класса МПО, но с помощью объекта МП клиенту будет раскрыто только свойство `FullName`.

Объектно-объектный преобразователь может даже выполнить запись непосредственно в закрытые поля объектов МП, чтобы разработчику вообще не нужно было реализовывать защищенные свойства. На самом же деле объектно-объектный преобразователь не только может, но и *должен* получать непосредственный доступ

к закрытым полям во избежание побочных эффектов в механизмах установки и получения значений свойств. Такие механизмы должны вызываться только при доступе клиентского кода к этим свойствам, но не в том случае, когда базовой структуре потребуется перемещение данных.

Некоторые объектно-объектные преобразователи могут предоставлять дополнительные возможности (например, загрузку по требованию), которые зависят от имеющихся свойств. Это позволяет перехватывать доступ к свойствам и сохранять их без изменения по мере надобности.

В приведенном ниже фрагменте кода показано, как в объекте МП применяется объектно-объектное преобразование.

```
// Объект МП с применением объектно-объектного преобразования
namespace MyCompany.MyProject.Presentation
{
  public class EmployeeView
  {
    private int id;
    private decimal salary;
    private string firstName;
    private string lastName;

    public EmployeeView() {}

    public int Id
    {
      get { return this.id; }
      set { this.id = value; }
    }

    public decimal Salary
    {
      get { return this.salary; }
      set { this.salary = value; }
    }

    protected string FirstName
    {
      get { return this.firstName; }
      set { this.firstName = value; }
    }

    protected string LastName
    {
      get { return this.lastName; }
      set { this.lastName = value; }
    }

    public string FullName
```

```
    {
      get
      {
         return this.firstName + " " +
         this.lastName;
      }
      set
      {
        string[] names = value.Split(" ".ToCharArray()
          , 2);

        this.firstName = names[0];
        this.lastName = names[1];
      }
    }
  }
}
```

Разумеется, все, что делается в базовой структуре, может быть сделано и вручную. Так, если для записи данных в защищенные поля требуется скопировать их вручную из МПО в МП без помощи базовой структуры объектно-объектного преобразования, то для этой цели можно воспользоваться рефлексией или же создать сокращенные методы доступа к свойствам по имени. Ниже приведен код, реализующий второй подход.

```
// Перемещение данных между МП и МПО без преобразования
Employee employee = employeeRepository.GetEmployeeById(42);
EmployeeView employeeView = new EmployeeView();

employeeView.Id = employee.Id;
employeeView.Salary = employee.Salary;

// Нам придется использовать метод для доступа к
// защищенным свойствам по имени для записи данных
// с помощью защищенных методов.
//С другой стороны, мы могли бы воспользоваться рефлексией.
employeeView.SetPropertyValue("FirstName", employee.FirstName);
employeeView.SetPropertyValue("LastName", employee.LastName);
```

Благодаря такому подходу задача перемещения данных между МП и МПО существенно упростилась бы, если бы не досадные свойства ссылки.

Организация отношений между объектами

Если принять во внимание отношения между объектами, то рассматриваемая задача вновь усложняется. Графы объектов (т.е. группы объектов, связанных определенными отношениями) могут быть довольно разветвленными, и поэтому если предложить объектно-объектному преобразователю заполнить объект `Employee` МП данными из

объекта Employee МПО, то в конечном итоге будет заполнено множество связанных с ним объектов, что незамедлительно скажется на производительности приложения.

Не следует также забывать о возврате к объектам МП из свойств ссылки МП. Так, после чтения данных из свойства AssignedToProject (Назначено для проекта) объекта Employee МП необходимо вернуться к объекту Project МП, а не к объекту Project МПО. Ниже приведена наивная реализация объекта-оболочки со свойством ссылки.

```
// Наивная реализация свойства ссылки в объекте-оболочке
namespace MyCompany.MyApplication.Presentation
{
  public class EmployeeWrapper
  {
    private Employee employee;

    public EmployeeWrapper (Employee employee)
    {
      this.employee = employee;
    }

    public Project AssignedToProject
    {
      get { return this.employee.Project; }
      set { this.employee.Project = value; }
    }
  }
}
```

Недостаток такой реализации заключается в том, что свойство Project объекта EmployeeWrapper возвращает объект Project МПО вместо объекта ProjectWrapper МП. Но обычно требуется совсем другое, поэтому свойства ссылки МП придется реализовать следующим образом:

```
// Несколько менее наивная реализация
// свойства ссылки в объекте-оболочке
namespace MyCompany.MyApplication.Presentation
{
  public class EmployeeWrapper
  {
    private Employee employee;

    public EmployeeWrapper (Employee employee)
    {
      this.employee = employee;
    }

    public ProjectWrapper AssignedToProject
    {
```

```
      get
      {
        return new
        ProjectWrapper(this.employee.Project);
      }
      set
      {
        this.employee.Project =
        value.GetDomainObject();
      }
    }

    public Employee GetDomainObject()
    {
      return employee;
    }
  }
}
```

Обратите внимание на то, что метод `GetDomainObject()` класса `ProjectWrapper` используется в методе установки `AssignedToProject()`. При записи данных в свойство `AssignedToProject` объекта `EmployeeWrapper` МП методу установки передается объект `ProjectWrapper`, но при этом объект `Project` МПО приходится передавать свойству `Project` заключенного в оболочку объекта `Employee`. Следовательно, нам требуется способ доступа к требуемому объекту `Project` из свойства в переданном объекте `ProjectWrapper`, и эту роль выполняет метод `GetDomainObject()`.

Таким образом, при реализации свойств ссылки мы совершенно неожиданно обнаруживаем, что требуется метод для доступа к объекту МПО, к которому объект МП обращается по внутренней ссылке, что в противном случае может и не понадобиться.

Более того, для реализации соответствующего свойства `AssignedEmployees` в классе `ProjectWrapper` мы также предоставили метод `GetDomainObject()` в классе `EmployeeWrapper`.

Но и этого оказывается недостаточно. Если критически проанализировать приведенный выше код, то нельзя не заметить, что всякий раз, когда выполняется чтение данных из свойства `AssignedToProject`, создается новый объект `ProjectWrapper`. Если допустить, что в слое модели предметной области действует отображение идентичности (Identity Mapping), то каждый объект `ProjectWrapper`, создаваемый при неоднократном чтении данных из одного и того же свойства, будет заключать в оболочку тот же самый объект `Project` МП, а следовательно, исключается риск несогласованности и порчи данных, хотя такое решение далеко от идеала.

В конечном счете нам может потребоваться полноценное решение, предусматривающее также отображение идентичности в МП, чтобы исключить появление в одном сеансе работы двух разных экземпляров МП, представляющих один и тот же экземпляр МПО.

Для этого, скорее всего, придется переписать код свойства `AssignedToProject`, чтобы вместо создания нового экземпляра самого объекта `ProjectWrapper` запрашивалось хранилище МП с картой идентичности для объекта `ProjectWrapper`.

Это означает, что объекту `EmployeeWrapper` потребуется ссылка на объект `ProjectRepository`, а следовательно, придется распрощаться с удобной простотой, которой до сих пор отличалась модель представления. И в конечном итоге мы вновь столкнемся со множеством тех же самых трудностей и осложнений, неприятные воспоминания о которых остались у нас в связи с манипулированием свойствами ссылки в слое предметной области.

Один из способов решения подобных затруднений, связанных со свойствами ссылки, состоит в том, чтобы просто исключить такие свойства в объектах МП, используя вместо этого "плоские" объекты, раскрывающие только простейшие свойства, возможно, даже из ссылочных объектов. Ниже приведен код реализации "плоских" объектов.

```
// "Плоский" объект МП, раскрывающий
// только простейшие свойства
 namespace MyCompany.MyApplication.Presentation
{
  public class EmployeeWrapper
  {
    private Employee employee;

    public EmployeeWrapper(Employee employee)
    {
      this.employee = employee;
    }

    public int Id
    {
      get { return this.employee.Id; }
      set { this.employee.Id == value; }
    }

    public string AssignedToProjectName
    {
      get { return this.employee.Project.Name; }
      set { this.employee.Project.Name = value; }
    }
  }
}
```

Такой подход нередко оказывается очень удобным, особенно потому, что многие элементы управления пользовательского интерфейса предназначены для отображения строк таблиц, а для этого вполне приспособлены плоские объекты. Но для представления сложных структур с возможностью перемещения по всей их глубине такой подход не годится. Впрочем, полностью разделенную модель представления можно

всегда дополнить рядом плоских объектов, специально предназначенных для табличного представления данных в приложении.

Плоские объекты МП служат характерным примером того, как модель представления нередко оказывается совершенно непохожей на модель предметной области, и, конечно, чем больше отличие между этими моделями, тем больше у нас оснований иметь две отдельные модели вместо того, чтобы вводить лишь в модель предметной области новые возможности, необходимые на уровне представления.

Вопросы состояния

Выбирая между преобразованием и заключением в оболочку, особое внимание следует уделять отличиям обоих методов в отношении обработки состояния.

При заключении в оболочку объектов из модели предметной области в приложении используется только один экземпляр данных. Так, если имеются два вида, одновременно отображающих одного и того же служащего и в одном из них обновляется имя этого служащего, то в объекте, заключенном в оболочку, обновляется лишь один экземпляр данных. Если же затем восстанавливается другой вид, он должен отображать обновленные данные, поскольку он также выполняет преобразование непосредственно к тому же самому обновленному объекту.

А при преобразовании объектов МП в объекты МПО может быть получено несколько разных наборов объектов МП, представляющих один и тот же объект МПО, но с разным состоянием в этих наборах.

Даже при наличии карты идентичности для МП в этой модели могут присутствовать два класса `Employee`, имеющих некоторые внешние отличия для разных целей представления. И в этом случае карта идентичности вообще не приносит никакой пользы.

С одной стороны, это может привести к конфликтам, а с другой — открывает путь для более изощренного управления разъединенными данными.

Так, работая очень долго с отдельным преобразованным набором объектов МП в некотором мастере, пользователь может на последнем шаге процедуры мастера решить зафиксировать результаты своей работы в МПО, а затем зафиксировать обновление МПО в базе данных, или отменить изменения. Если бы пользователь работал с набором объектов МП, непосредственно заключавших в оболочку объекты МПО, у него не было бы такой возможности отменить изменения, поскольку объекты МПО непрерывно обновлялись бы в процессе работы пользователя с мастером!

Безусловно, возможность для отмены с целью исключить передачу изменений в базу данных имеется и без МП, при условии, что в МПО поддерживаются разъединенные операции, например, с помощью шаблона Unit of Work [Fowler PoEAA]. Но если нажать кнопку отмены в конце процедуры мастера, работающего непосредственно с МПО, эта модель все равно изменится, и на следующем экране мастера будут видны внесенные изменения, хотя они и не коснутся базы данных.

Это не вызовет особых осложнений, если в приложении предусмотрено отбрасывание МПО при отмене пользователем внесенных изменений или же если в едини-

це работы поддерживается полный возврат в исходное состояние МПО. Но если речь идет о создании клиентского приложения с развитыми функциональными возможностями, где МПО постоянно существует в области действия приложения и не отбрасывается при отмене пользователем внесенных изменений, а в единице работы не поддерживается полный возврат в исходное состояние, то возможны серьезные осложнения в связи со сменой состояния модели.

В одних случаях более подходящим для нужд проектирования оказывается заключение в оболочку, в других случаях — преобразование и в третьих — заключение в оболочку применяется в приложении по умолчанию, а преобразование используется только для мастеров или других пакетных заданий, где допускается отмена без сохранения изменений.

Заключительные соображения

Иногда судьба бывает благосклонна к разработчику. Так, ситуация может сложиться настолько благоприятно, что он сможет предоставить модель предметной области пользователю непосредственно, не преодолевая трудности связывания МП с МПО.

Этих трудностей оказывается значительно меньше, если используются плоские объекты МП, поскольку в результате манипулирования свойствами ссылки сразу же возникает немало серьезных осложнений.

Выбор между заключением в оболочку и преобразованием зависит от целого ряда факторов, в том числе от способа обработки состояния и возможностей сократить объем шаблонного кода в приложении посредством генерации кода (при заключении в оболочку) или с помощью базовой структуры объектно-объектного преобразования (при преобразовании).

Если попытаться вынести структурные преобразования за пределы любой применяемой базовой структуры или генератора кода (вместо того чтобы вводить логику всех подобных преобразований в код внутри классов МП и скрывать непреобразуемые члены в МП, делая их защищенными), то можно существенно упростить требования, предъявляемые такой базовой структурой или генератором кода, а следовательно, повысить шансы написать нечто полезное или же найти что-нибудь подходящее в оперативном режиме.

Если же необходимость применять все эти дополнительные архитектурные издержки в виде моделей представления, объектно-объектных преобразователей и методов преобразования логики вызывает сомнение, то, по крайней мере, можно попытаться самостоятельно найти подходящее представление для модели предметной области.

Главный вывод заключается в том, чтобы всячески избегать видоизменения модели предметной области, с тем чтобы подогнать ее под требования уровня представления, и не пытаться использовать ее не только как модель предметной области, но и как модель представления, если, конечно, без этого нельзя обойтись, не изменяя модель предметной области.

Как только станут ясны отличия между способом представления модели предметной области внутри приложения и способом ее представления пользователю, станет очевидной и потребность в МП — по крайней мере, если следовать рекомендации *никогда не отягощать модель предметной области аспектами уровня представления*.

Я лично стараюсь делать все возможное, чтобы избавить модель предметной области от аспектов уровня представления. Так, если моя модель предметной области связана непосредственно с данными и только некоторые атрибуты добавляются к свойствам (например, атрибуты для указания категории и используемого по умолчанию значения для привязки к сетке свойств), я воздерживаюсь от ввода подобных атрибутов уровня представления в объекты МПО. Вместо этого я дополняю свою модель предметной области моделью представления — даже если она едва ли не в каждой детали повторяет модель предметной области, за исключением атрибутов. Впрочем, такая крайность может показаться чрезмерной.

Тем не менее я иду на такую крайность, поскольку, имея в своем распоряжении МП и инфраструктуру, поддерживающую заключение в оболочку или преобразование в МПО, я обычно получаю немало возможностей для уточнения аспектов представления в МП, и тогда обе модели очень скоро начинают все больше и больше отличаться друг от друга, а следовательно, появляется больше оснований для дополнения модели предметной области моделью представления.

Благодарю, Матс!

Резюме

Итак, мои коллеги раскрыли преимущества модели предметной области — на этот раз на примере служб представления.

Они показали, что помимо самой модели предметной области, большое значение имеет и служба ее представления. Именно поэтому нужно очень тщательно прорабатывать модель предметной области. Чем больше абстрагироваться от уровня представления, который только отвлекает от контекста, тем проще создать качественный пользовательский интерфейс.

Эпилог

Итак, мы рассмотрели разработку приложений в стиле ППО. По этому же принципу построена структура книги. Имеется в виду следующее.

1. Сначала мы пытаемся сформулировать требования и понять задачу.
2. Далее мы обдумываем метод структурирования основной логики. Согласно Фаулеру, мы можем выбирать среди шаблонов Transaction Script, Table Module

и Domain Model [Fowler PoEAA]. Если же требования свидетельствуют о том, что это слишком сложный (в отношении поведения и/или отношений) и долгий путь разработки проекта, мы, скорее всего, останавливаем свой выбор на модели предметной области.

3. Затем мы выбираем определенный вид модели предметной области, чтобы не только избежать скрытых препятствий, но и создать нечто полезное и эффективное. Именно для этой цели и служит ППО. Мы пытаемся сформировать универсальный язык [Evans DDD] и тщательно прорабатываем модель предметной области, чтобы обогатить ее соответствующими знаниями.
4. После этого мы отвлекаемся на время от самой модели, чтобы выбрать подходящую инфраструктуру. Самый главный вопрос на данном этапе проектирования — это вопрос сохраняемости. Чаще всего в проектах, разрабатываемых в стиле ППО, применяется объектно-реляционное преобразование. (Вместо несколько перегруженного термина "инфраструктура" этот процесс удобнее описывать в виде преобразования модели в другие ярусы приложения, включая службы представления, сохраняемости и интеграции.)
5. И наконец, остается решить много других вопросов, в том числе и тех, что касаются представления.

Нельзя сказать, что книга полностью отражает водопадный процесс, но она имеет последовательный характер, такой же характер чаще всего имеет и описание. Очень важно отметить и подчеркнуть повторяющийся и постепенный стиль разработки, который позволяет свести к минимуму риск и успешно завершить проект.

Итак, книга подошла к концу, но, как заметил Дан, это только начало.

И еще одно замечание: в начале книги я упоминал о том, насколько полезно соблюдать равновесие, чтобы все было в самый раз, или "лагом", как говорят шведы. Удалось ли мне соблюсти данный принцип? Это уже судить читателям — в конечном счете, все зависит от контекста и уравновешенности взглядов самого читателя.

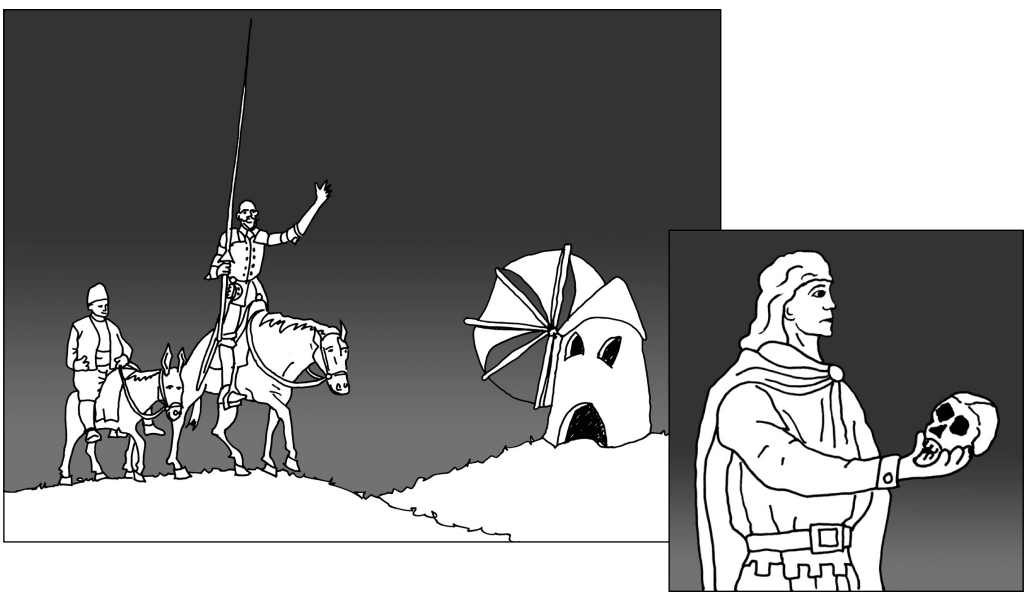
Часть V

Приложения

Приложение А. Другие виды моделей предметной области

Приложение Б. Перечень рассмотренных шаблонов

В двух приложения к книге представлены дополнительные примеры видов модели предметной области и перечень шаблонов с кратким описанием.



В приложениях представлены другие виды моделей и наборы шаблонов

Приложение А

Другие виды моделей предметной области

Шаблон Domain Model может быть использован самыми разными способами. В этой связи я попросил трех моих коллег поделиться своими методами применения такого шаблона в частности и моделей предметной области в целом. Они получили список требований, приведенных в главе 4, решения представлены в этом приложении[1]. Приведем эти требования еще раз.

1. Список клиентов, формируемый с применением гибкого и сложного фильтра.
   Персоналу, обслуживающему клиентов, необходимо предоставить возможность для удобного поиска клиентов. В частности, они должны пользоваться метасимволами в многочисленных полях, включая имя, адрес, ответственное лицо и т.д. Кроме того, у них должна быть возможность искать клиентов с заказами определенного типа, определенной величины, на определенные виды продукции и т.д. Таким образом, здесь речь идет о полноценной утилите поиска, которая в итоге формирует список клиентов со своими номерами, именами и фамилиями, адресами.
2. Список заказов, формируемый при просмотре отдельного клиента.
   В списке должна отображаться общая стоимость заказа, а также его состояние, тип, дата составления, имя и фамилия ответственного лица.
3. Заказ может состоять из самых разных строк.
   Заказ может быть многострочным, причем в каждой строке описывается заказанная продукция и ее номер.
4. Большое значение имеет выявление конфликтов параллельной обработки.
   В данном случае достаточно выбрать оптимистический вариант управления параллельным выполнением операций, т.е. считать вполне приемлемым, когда пользователь уведомляется о конфликте с предыдущим результатом сохранения после того, как он завершит свою работу и попытается сохранить ее. При этом конфликтами считаются только те, что приводят к настоящим противоречиям. Следовательно, в данном решении следует предусмотреть блок

[1] Следует иметь в виду, что вначале предполагалось использование сервера приложений, но этот сценарий был затем изменен в главе 4.

Proekt_book.indb 491 27.08.2007 21:33:35

управления версиями для клиентов и заказов. (Это может оказать некоторое влияние на ряд других возможностей.)

5. У клиента может быть долг, не превышающий определенную сумму.

   У каждого клиента этот лимит свой. Он устанавливается при первоначальном вводе клиента, а впоследствии может быть изменен. Наличие неоплаченных заказов на общую сумму, превышающую данный лимит, считается противоречием. Но мы можем допустить это противоречие в том случае, если пользователь уменьшит лимит. Пользователь, уменьшивший лимит, получает соответствующее уведомление, но операция сохранения разрешается. Тем не менее, заказ не может быть введен или изменен с превышением лимита.

6. Заказ не может быть составлен на сумму свыше 1 млн крон.

   Этот лимит, в отличие от предыдущего, является правилом, распространяющимся на всю систему.

7. У каждого заказа и клиента должен быть уникальный и удобный для использования номер.

   Допускается следование номеров не по порядку.

8. Прежде чем новый клиент будет признан приемлемым, должен быть проверен его кредит в соответствующем кредитном учреждении.

   Это означает, что лимит, рассматривавшийся в п.5 данного списка и установленный для клиента, будет проверен на обоснованность.

9. У заказа должен быть клиент, а у строки заказа — заказ.

   Заказов с неопределенными клиентами быть не должно. Это же относится и к строкам заказа — они должны принадлежать определенному заказу.

10. Сохранение заказа и его строк должно быть атомарным.

    Откровенно говоря, я не уверен в том, что такая возможность действительно нужна. Ведь вполне допустимо, чтобы строки заказа вводились после его первоначального составления. Но мне требуется какое-то правило для данного примера, чтобы иметь возможность, связанную с защитой транзакций.

11. У заказов должно быть состояние подтверждения, изменяемое пользователем.

    Это состояние может изменяться пользователями при переходе от подтверждения заказа к неподтверждению и обратно. Что же касается других состояний, то их смена должна происходить косвенно другими методами, доступными в модели предметной области.

В следующем разделе Матс Хеландер поделится своими соображениями по поводу применения модели предметной области.

Объектно-ориентированная модель данных, уровень развитого обслуживания и документы

Матс Хеландер

Долгими зимними вечерами я иногда придаюсь воспоминаниям о том, как я раньше обходился без модели предметной области. Налив немного коньяку в чашку горячего шоколада и подложив очередное полено в огонь, я усаживаюсь поудобнее у камина.

Эволюция архитектуры в моих приложениях в конечном итоге привела к необходимости использовать модель предметной области в соответствии с шаблоном, который, вероятно, известен многим читателям. Я попытаюсь вкратце изложить эту эволюцию, поскольку она поможет лучше понять мой нынешний подход к применению модели предметной области в определенном контексте.

В начале

Я начинал с разработки приложений "клиент-сервер", в которых весь код приложения находился на стороне клиента и обращался непосредственно к базе данных. Затем я занялся разработкой Web-приложений, состоявших из ASP-страниц и всего кода приложения, включая и код обращения к базе данных.

В конечном итоге архитектура таких приложений превращалась в ужасно беспорядочную смесь, как только приложение становилось немного сложнее примера "Здравствуй, мир", и поэтому я вскоре научился переносить бо́льшую часть кода из уровня представления на *уровень бизнес-логики* (УБЛ).

Я сразу же почувствовал, что разрабатывать и сопровождать приложения стало легче, а кроме того, уровень бизнес-логики можно было повторно использовать в промежутке между разными уровнями представления. Следует также заметить, в то время бо́льшую часть кода Web-приложения приходилось компилировать в компонент из соображений производительности и масштабируемости.

Следующий этап состоял в переносе всего кода, ответственного за обращение к базе данных, из УБЛ на отдельный *уровень доступа к данным* (УДД). Я помню, что был немного удивлен тем, что УДД зачастую оказывался значительно более крупным, чем УБЛ, т.е. логика приложения в большей степени была связана с обращением к базе данных, чем с конкретной предметной областью.

В то время я уже был практически готов к тому, чтобы перейти к модели предметной области, поскольку архитектура моих приложений строилась, в основном, по следующему принципу: уровень представления был связан с УБЛ, который, в свою очередь, был связан с УДД, а последний — с базой данных. Но вся информация из базы данных передавалась приложению в виде наборов записей, представлявших в оперативной памяти строки из базы данных (рис. А.1).

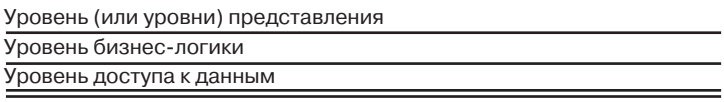


Рис. А.1. Классическая многоуровневая архитектура, предшествовавшая модели предметной области

Если с данными требуется работать в реляционной форме, то именно для этой цели и подходит описанная выше архитектура. Но со временем я осознал, что реляционные структуры данных не всегда оказываются удобными для бизнес-логики.

Объектная ориентация и реляционные структуры данных

Некоторый опыт работы с объектно-ориентированной базой данных Jasmine компании Computer Associates заставил меня сделать вывод о том, что объектно-ориентированная структура данных зачастую оказывается намного более удобной для разработки бизнес-логики, чем соответствующая реляционная структура данных.

Во-первых, более простыми оказываются отношения между объектами-сущностями. Это особенно справедливо для отношений типа "многие ко многим", поскольку для этой цели в реляционной структуре данных требуется дополнительная таблица, тогда как в объектно-ориентированной структуре данных дополнительный класс не нужен.

Во-вторых, дополнительная таблица требуется в реляционной структуре данных для свойств совокупностей объектов, тогда как в объектно-ориентированной структуре данных дополнительный класс не нужен. И наконец, объектно-ориентированные структуры данных типизированы и поддерживают наследование.

Модель предметной области и объектно-реляционное преобразование

Главная трудность заключалась в том, что объектно-ориентированные базы не нашли широкого распространения. Большинство проектов, с которыми мне пришлось иметь дело, строилось на основе реляционной базы данных. А для этой цели требовалось объектно-реляционное преобразование.

Объектно-реляционное преобразование позволяет взять объектно-ориентированную структуру данных и преобразовать ее в таблицы из реляционной базы данных. Преимущество такого подхода — в возможности работать с объектно-ориентированной структурой данных как с объектно-ориентированной базой данных типа Jasmine, хотя в действительности для хранения всей информации используется реляционная база данных.

Кроме того, в классы объектно-ориентированной структуры данных можно вводить бизнес-логику в виде бизнес-методов. Нетрудно догадаться, что объектно-ориентированная структура данных вместе с бизнес-методами представляет собой модель предметной области.

Освоив объектно-реляционное преобразование, я смог сделать решающий шаг к тому, чтобы включить модель предметной области в архитектуру своих приложений. В частности, введя уровень модели предметной области (УМПО) в промежутке между уровнями УБЛ и УДД, я связал, наконец, бизнес-логику приложения с объектно-ориентированной структурой данных (рис. А.2.)

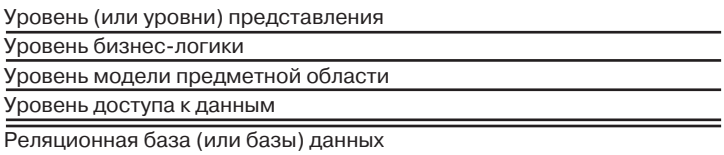


Рис. А.2. Обновленное многоуровневое приложение, включающее модель предметной области

Это означает, что теперь для работы с данными на УБЛ вместо набора записей, состоящих, например, из рядов таблицы `Employees`, используется класс `Employees` из модели предметной области. В итоге я добился типовой безопасности, упрощения и ясности кода, не говоря уже о возможности перемещаться по структуре данных во время разработки, используя технологию IntelliSense компании Microsoft!

С тех пор я стал преданным сторонником модели предметной области и уже не оглядывался назад, разве что предаваясь воспоминаниям в долгие, холодные зимние вечера.

Как упоминалось выше, еще одно преимущество модели предметной области состоит в том, что она дает возможность распределять бизнес-логику среди ее классов. Я немало экспериментировал с конкретными местами для размещения бизнес-логики, перенося бо́льшую ее часть из УБЛ на УМПО и обратно.

Уровень обслуживания

Попутно я пришел к выводу, что термин *уровень обслуживания* (Service Layer) [Fowler PoEAA] точнее, чем термин *уровень бизнес-логики* (Business Logic Layer), отражает размещение большей части бизнес-логики на уровне модели предметной области. Когда бизнес-логика оказывается, в основном, на уровне обслуживания, я не вижу причин возвращаться к термину *уровень бизнес-логики*, хотя я иногда называю такой уровень обслуживания "толстым", чтобы отличить его от чаще применяемого "тонкого" уровня обслуживания.

В настоящее время я практически всегда размещаю бо́льшую часть логики на "толстом" уровне обслуживания. Конечно, можно спорить по поводу того, стоит ли объединять в одном слое структуру и поведение, но я довольно часто нахожу практически целесообразным оставлять на уровне обслуживания поведение, связанное с бизнес-логикой.

Как показывает мой опыт, бизнес-правила, определяющие поведение модели предметной области, обычно меняются не с такой периодичностью и не в те моменты

времени, что и правила, определяющие структуру этой модели. Кроме того, модель предметной области может без труда использоваться повторно при разных наборах бизнес-правил.

Как правило, в сами объекты из модели предметной области я ввожу только те бизнес-методы, которые я считаю в достаточной степени основополагающими, а следовательно, действительными при *любых* наборах бизнес-правил и изменяющимися с той же периодичностью и в те же моменты времени, что и структура модели предметной области.

Когда вся бизнес-логика находится на уровне обслуживания, архитектура приложения оказывается практически готовой для перехода к новому методу проектирования — AOC. Но прежде чем сделать это на конкретном примере приложения Джимми, рассмотрим еще одно преимущество такого подхода к проектированию.

Объединение на одном уровне

Как уже отмечалось, объектно-ориентированная структура данных, которую предоставляет модель предметной области, подходит не для *всякой* выполняемой операции. Имеется немало веских оснований для того, чтобы отдавать предпочтение реляционным базам данных, а не объектно-ориентированным. Ведь для решения многих задач оказывается достаточно реляционной структуры данных и операций над множествами, доступных в SQL.

В таких случаях ввод в объекты из модели предметной области методов, по существу, выполняющих операции над множествами непосредственно в реляционных структурах данных, может показаться не вполне естественным.

С другой стороны, методы, получающие непосредственный (или посредством УДД) доступ к базе данных, могут вполне сосуществовать на уровне обслуживания с методами, манипулирующими моделью предметной области. Кроме того, на уровне обслуживания нетрудно переориентировать любой метод с одной из двух указанных функций на другую.

Иными словами, на уровне обслуживания можно свободно реализовать любой походящий метод, минуя уровень модели предметной области, где этот метод может оказаться бесполезным, и используя его в полной мере там, где он действительно полезен, что, как показывает мой опыт, происходит довольно часто.

Приложение Джимми и AOC

Итак, сделав предварительные замечания, обратимся к приложению Джимми и новому методу проектирования — AOC. До сих пор я лишь описал в общих чертах архитектуру приложения на сервере, чего может оказаться вполне достаточно для Web-приложения, но поскольку далее речь пойдет о клиентской части приложения с развитыми функциональными возможностями, мне бы хотелось сосредоточить основное внимание на серверах приложений как на одном из возможных вариантов разработки.

В тех случаях, когда практически вся бизнес-логика распределена по модели предметной области, возникает искушение попытаться экспортировать вместе данные и поведение в клиентскую часть приложения с развитыми функциональными возможностями, предоставив ей доступ к модели предметной области. В используемой архитектуре, где поведение находится, в основном, на уровне обслуживания, более естественным было бы раскрытие поведения и данных по отдельности.

Если быть более конкретным, то уровень обслуживания внешне дополняется интерфейсом Web-службы (в виде шаблона Remote Façade (Интерфейс удаленного доступа) [Fowler PoEAA]), который, собственно, выполняет роль "тонкого" слоя для раскрытия методов бизнес-логики на уровне обслуживания в качестве Web-служб. Клиентская часть приложения с развитыми функциональными возможностями связывается с сервером, обращаясь к Web-службам, и обменивается XML-документами, содержащими объекты из модели предметной области, преобразованные в последовательную форму.

В данном примере приложения это означает, что всякий раз, когда клиентской части приложения с развитыми функциональными возможностями требуется новая информация, она обращается на сервер к Web-службе, которая затем возвращает информацию в формате XML-документа. Итак, рассмотрим сначала формирование списка клиентов с применением фильтра.

Прежде всего, создадим на уровне обслуживания метод `GetCustomersBy Filter()`, фактически реализующий фильтрацию. Этот метод воспринимает параметры фильтра в виде аргументов и возвращает отфильтрованные объекты из модели предметной области. Затем введем метод Web-службы `GetCustomersByFilter()`, чтобы раскрыть метод уровня обслуживания для клиентской части приложения с развитыми функциональными возможностями.

Метод Web-службы воспринимает те же самые параметры фильтра в виде аргументов и передает их вызываемому методу уровня обслуживания. Этот метод возвращает объекты из модели предметной области, которые затем могут быть преобразованы в последовательную форму и составлены в виде XML-документа, возвращаемого в конечном итоге Web-службой. Полученная таким образом информация может быть использована в клиентской части приложения с развитыми функциональными возможностями для отображения списка клиентов по требованию пользователя (рис. А.3).

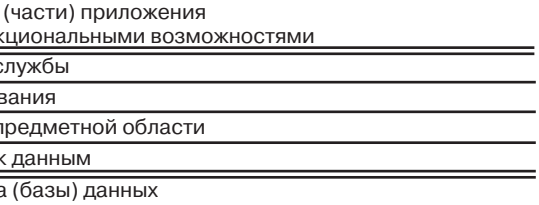


Рис. А.3. Многоуровневая архитектура приложения "клиент-сервер" с развитыми функциональными возможностями, включающая модель предметной области

Создание клиентской части модели предметной области, которую можно заполнить данными из XML-документа, оставляется на усмотрение разработчиков клиентской части приложения с развитыми функциональными возможностями. Они могут также принять решение о сохранении информации с целью использовать ее в автономном режиме. С одной стороны, информацию можно сохранять в виде XML-документа на диске, а с другой — хранить клиентские модели предметной области в автономной базе данных на клиентской машине.

Следует заметить, что в окончательном виде модель предметной области на клиентской стороне совсем не обязательно должна совпадать с такой же моделью на серверной стороне. Так, если модель предметной области на клиентской стороне хранится в автономной базе данных, клиентская база данных, скорее всего, должна иметь структуру, соответствующую этой модели, а следовательно, она не должна быть похожа на серверную базу данных.

На заметку

Схема серверной базы данных отличается от схемы клиентской базы данных главным образом тем, что если в ней таблица используется автоматически, то в клиентской базе данных этого, как правило, не происходит.

Допустим, что в серверную базу данных вводится новый служащий. При этом в таблицу `Employees` вводится новая строка и ей автоматически присваивается значение в столбце автоматического увеличения идентификационного номера. Затем данные о служащем преобразуются в последовательную форму и передаются на сторону клиента, где этими данными заполняется модель предметной области, которая затем сохраняется в клиентской базе данных для использования в автономном режиме.

Новой строке таблицы на стороне `Employees` клиента не должен присваиваться новый идентификационный номер, а вместо него должен использоваться идентификационный номер, присвоенный в серверной базе данных. Это означает, что свойства столбцов идентификационного номера на стороне клиента и сервера могут отличаться.

Таким образом, разработчики клиентской части приложения с развитыми функциональными возможностями вольны решать, следует ли им вообще использовать модель предметной области, и если следует, то они могут выбрать наиболее подходящую ее структуру. Ведь обмен данными между клиентом и сервером осуществляется исключительно на основе документов, причем ни одна из сторон не строит никаких предположений относительно моделей предметной области на другой стороне.

Допустим, что клиентская часть приложения с развитыми функциональными возможностями обращается к Web-службам самых разных организаций, и все эти службы способны возвращать списки клиентов, но с некоторыми отличиями в полях

XML-документов. Для разрешения подобного затруднения можно было бы, в частности, создать в модели предметной области на стороне сервера объект "супермножества" со свойствами самых разных полей, возвращаемых после обращения к службам различных организаций.

Далее нам требуется, чтобы пользователь клиентской части приложения с развитыми функциональными возможностями мог просматривать заказы, принадлежащие конкретному клиенту. В XML-документ был, конечно, включен идентификационный номер, возвращенный первой Web-службой, поэтому нам известен идентификационный номер интересующего нас клиента.

Итак, необходимо реализовать Web-службу, принимающую идентификационный номер клиента и возвращающую заказы данного клиента. В этом случае начнем с метода `GetOrdersByCustomerID()` уровня обслуживания, воспринимающего идентификационный номер клиента и возвращающего объекты заказов из модели предметной области, принадлежащие указанному клиенту.

Затем мы вводим метод Web-службы `GetOrdersByCustomerID()`, принимающий идентификационный номер клиента, передающий его методу уровня обслуживания и, наконец, преобразующий в последовательную форму возвращаемые объекты заказов и составляющий из них XML-документ, который данная Web-служба возвращает.

Но на этот раз следует уделить больше внимания преобразованию в последовательную форму при составлении XML-документа. В этот документ требуется включить итоговую сумму по каждому заказу, но в объекте заказа из модели предметной области отсутствует соответствующее свойство `TotalValue`, поскольку итоговая сумма заказа не сохраняется ни в одном из полей базы данных.

Вместо этого итоговая сумма рассчитывается всякий раз, когда в этом возникает потребность, путем сложения соответствующих сумм из строк заказа. В свою очередь, сумма в каждой строке заказа рассчитывается путем умножения цены приобретенного товара в момент его закупки на количество единиц этого товара.

В XML-документе, возвращенном клиентской части приложения, отсутствуют строки заказа, и поэтому она не в состоянии рассчитать итоговые суммы по отдельным заказам. Это означает, что подобный расчет должен быть произведен на сервере, а его результат — включен в XML-документ.

Методы расчета суммы в строке заказа и итоговой суммы заказа служат характерными примерами тех методов, которые обычно вводятся в объекты заказа и строки заказа из модели предметной области, но я бы предпочел ввести их на уровне обслуживания.

Структура уровня обслуживания

Теперь самое время упомянуть о том, что я обычно организую уровень обслуживания вокруг тех же самых объектов-сущностей, которые используются в модели предметной области, а не исходя из конкретной задачи, что чаще всего делают разработчики. Так, если на уровне модели предметной области имеется объект `Employee`, я создаю класс `EmployeeServices` (Службы для служащего) на уровне обслуживания.

Для тех служб, которым не требуется контроль транзакций, я также реализую статические методы на уровне обслуживания, чтобы вызывать их, не создавая экземпляры объектов из классов уровня обслуживания.

В данном примере в класс `OrderLineServices` (Службы для строк заказа) уровня обслуживания следует ввести статический метод `GetValue()` — получить сумму, а в класс `OrderServices` — статический метод `GetTotalValue()` — получить итоговую сумму. Оба метода воспринимают в виде параметров объекты, с которыми они должны работать. Это означает, что вместо вызова метода в самом объекте заказа:

```
myOrder.GetTotalValue();
```

вызывается метод уровня обслуживания:

```
OrderServices.GetTotalValue(myOrder);
```

От такого процедурного вида синтаксиса некоторые пуристы ООП вознегодуют, не пожалев для меня самых сильных выражений, но я считаю, что столь далекий от изящества синтаксис стоит употребить ради преимуществ, которые дает отделение поведения от структуры модели предметной области.

Разумеется, методы уровня обслуживания совсем не обязательно должны быть статическими, а иногда они и не могут быть таковыми — например, если эти методы должны инициировать декларативные транзакции или если требуется настроить классы уровня обслуживания с помощью шаблона Dependency Injection (см. главу 10).

Но когда статические методы оказываются пригодными, я делаю их таковыми просто ради экономии на строке кода инициализации объекта уровня обслуживания при каждом вызове службы. Именно по этой причине здесь и далее будут использоваться статические методы уровня обслуживания.

Главное — не упустить возможность для организации методов так же изящно, как и при их вводе в классы из модели предметной области. А поскольку каждому такому классу в модели предметной области соответствует класс на уровне обслуживания, я могу разделить бизнес-логику таким же образом, как и при ее распределении в самой модели предметной области.

Упомянутые выше преимущества проявляются, когда начинают меняться правила для метода `GetTotalValue()`, несмотря на то что базовая структура данных не меняется — например, в том случае, если неожиданно требуется внедрить системное правило, устанавливающее, что сумма заказа не может превышать 1 млн крон. В таком случае достаточно видоизменить соответствующие методы уровня обслуживания, не прибегая к перекомпилированию кода и даже не касаясь уровня модели предметной области, который может и дальше оставаться протестированным и выверенным.

Возвращаясь к методу Web-службы `GetOrdersByCustomerID()`, просто введем в него вызов метода `OrderServices.GetTotalValue()` уровня обслуживания, которому по очереди передается каждый заказ в процессе преобразования в последовательную форму, а затем в XML-документ вводится поле для хранения результата.

Это еще один характерный пример того, как модель предметной области на стороне клиента, если таковая реализуется разработчиком приложения, может отличаться от аналогичной модели на стороне сервера. Объект заказа из модели предметной области на стороне клиента может вполне содержать свойство для итоговой суммы, представленной в XML-документе. Такое свойство отсутствует у аналогичного объекта на стороне сервера.

Следуя этим же путем, мы можем включить в XML-документ дополнительные поля со значениями, рассчитанными в методах уровня обслуживания.

Рассмотрим, например, системное правило, устанавливающее, что клиент не может иметь долг, превышающий определенную сумму. Конкретная сумма долга устанавливается и рассчитывается индивидуально в некотором методе уровня обслуживания. При составлении XML-документа, возвращаемого методом Web-службы `GetCustomersByFilter()`, в него, возможно, потребуется включить максимальную сумму долга для каждого клиента или, по крайней мере, указать, превысил ли клиент свой лимит.

Этим демонстрируется, что документы, предоставляемые сервером, совсем не обязательно должны соответствовать модели предметной области, поскольку обмен данными между клиентской и серверной стороной осуществляется исключительно на основе документов.

Итак, на данный момент пользователь клиентской части приложения с развитыми функциональными возможностями должен иметь возможность просматривать отфильтрованные списки клиентов, а также список заказов, принадлежащих каждому клиенту. Далее нам требуется предоставить пользователю возможность проверять строки, принадлежащие конкретному заказу.

В реализации такой возможности нет ничего особенного. Клиентская часть приложения обращается к Web-службе по идентификационному номеру заказа, а Web-служба передает вызов методу уровня обслуживания, преобразует полученный результат в последовательную форму и составляет соответствующий XML-документ. Итак, мы уже знаем, как реализовать выборку данных на стороне клиента. А как насчет предоставления данных клиентской стороной?

Предоставление данных

Допустим, что пользователю требуется обновить состояние заказа “Не проверен” состоянием “Утвержден” или “Отклонен”. На первый взгляд, такой запрос может показаться простым, но на самом деле он приводит к целому ряду проблем параллельной обработки, поскольку теперь требуется доступ для чтения и записи данных вместо доступа “только для чтения”.

Итак, начнем с самого общего подхода, не вдаваясь в подробности, и решим, как нам дальше поступить.

Прежде всего, необходимо создать метод `UpdateOrderStatus()` — обновление состояния заказа, как на уровне обслуживания, так и для Web-службы. Этот метод должен воспринимать идентификационный номер заказа и его состояние в виде

параметров и возвращать логическое значение, указывающее, насколько успешно завершилась операция. (Мы не будем здесь касаться вопросов межплатформной обработки исключений, хотя они представляют немалый интерес с точки зрения разработки приложений!)

Реализовать упомянутый выше метод уровня обслуживания несложно, а в Web-службе это всего лишь метод-оболочка, передающий вызов методу уровня обслуживания. А теперь рассмотрим затруднения, которые возникают в связи со сменой состояния.

Допустим, что два пользователя пытаются одновременно обновить состояние одного и того же заказа. Пользователь А выбирает заказ, замечает, что он находится в состоянии "Не проверен", и переходит к его проверке. Некоторое время спустя пользователь Б выбирает тот же самый заказ, замечает то же самое и также решает проверить его. При этом пользователь А замечает, что у него возникает небольшое затруднение с переводом заказа в состояние "Отклонен" после вызова метода Web-службы `UpdateOrderStatus()`.

Как только пользователь А обновит состояние заказа, пользователь Б обратится к той же самой Web-службе для обновления состояния этого же заказа. Но пользователь Б не обратил внимания на затруднения, связанные с обновлением состояния этого заказа, и стремится перевести его в состояние "Утвержден". Следует ли позволить пользователю Б обновить состояние заказа, присвоенное пользователем А и сохраненное в базе данных?

Обычно на такой вопрос дается следующий ответ: "Нет, поскольку пользователю Б не было *известно* об изменениях, внесенных пользователем А". Если бы пользователь Б знал об изменениях, сделанных пользователем А, и, тем не менее, решил пренебречь этим, то ответ был бы следующим: "Да, поскольку можно надеяться, что пользователь Б не перезапишет данные пользователя А по ошибке". В связи с этим возникает другой вопрос: как определить, что именно известно пользователю Б или любому другому пользователю?

Для этого можно, в частности, отслеживать обновление исходных значений в полях базы данных, а в момент сохранения видоизмененной информации в базе данных проверять, совпадают ли исходные значения с теми, что теперь хранятся в базе данных. Если эти значения в базе данных изменились, значит, клиентской части приложения *не было известно* о подобных изменениях, и поэтому ее данные должны быть отвергнуты.

Такой метод называется *оптимистическим параллелизмом*. Рассмотрим простейший вариант реализации оптимистического параллелизма в данном примере.

Прежде всего, клиентская часть приложения должна быть запрограммирована таким образом, чтобы при изменении состояния заказа по-прежнему сохранялось состояние, первоначально возвращенное из базы данных посредством XML-документа.

Затем необходимо видоизменить метод Web-службы `UpdateOrderStatus()` и одноименный метод уровня обслуживания, чтобы и тот и другой воспринимал в виде дополнительного параметра значение, определяющее исходное состояние заказа.

Следовательно, при обращении клиентской части приложения к Web-службе передаются три следующих параметра: идентификационный номер заказа, новое состояние заказа и исходное состояние заказа.

И наконец, метод Web-службы `UpdateOrderStatus()` следует обновить таким образом, чтобы ввести в него проверку на совпадение текущего состояния заказа в базе данных с исходным его состоянием, переданным в виде параметра, и разрешить выполнение соответствующей операции обновления только при совпадении обоих состояний заказа.

В таком варианте оптимистического параллелизма совпадение исходных значений с теми, что находятся в обновленных полях базы данных, обеспечивает хорошую производительность и масштабируемость, но его реализация может быть затруднена — по крайней мере, в клиентской части приложения с развитыми функциональными возможностями. Другой вариант оптимистического параллелизма состоит в том, чтобы ввести поле контроля версий в таблицы базы данных.

В тех случаях, когда методы оптимистического параллелизма не годятся, приходится выбирать решения, исходя из особенностей конкретной предметной области. Как правило, для этой цели служит определенного рода система входного и выходного контроля, в которой пользователь помечает объект как "прошедший выходной контроль" или "заблокированный", когда он выбирается для доступа с целью записи, что препятствует другим пользователям обновлять этот объект до тех пор, пока данный пользователь не завершит свои действия. Другие, более сложные системы предполагают слияние параллельно обновляемых данных.

Для выбора наиболее подходящего метода в каждом конкретном случае рекомендуется обращаться к специалисту по данной предметной области, который хорошо представляет себе последствия подобных решений для соответствующей сферы применения и знает, какое именно поведение было бы желательно. Ведь единой методики выбора метода параллелизма не существует.

Тем не менее в рассматриваемом примере, на мой взгляд, уместно использовать оптимистический параллелизм как вместе с полем контроля версий, так и без такового. Что же касается предполагаемого поведения, то решение этого вопроса целесообразно предоставить тем, кто лучше знает конкретную сферу применения.

Вероятно, самый большой недостаток подхода, примененного в данном примере, заключается в том, что отслеживание исходных значений предоставлено клиентской части приложения. Для разработчика такого приложения это может оказаться чрезмерным требованием, поэтому более приемлемым вариантом было бы хранение исходных значений на сервере в переменной `Session` или же в аналогичной конструкции.

Степень структурирования

Следующий на повестке вопрос связан с тем, насколько мелкоструктурной должна быть Web-служба. Следует ли реализовать единый метод Web-службы `UpdateOrder()` вместо отдельных Web-служб для каждого обновляемого поля заказа?

В данном примере Web-служба может принимать XML-документ, представляющий обновленный заказ, вместо каждого обновляемого поля в виде параметра. Это означает, что вместо кода:

```
[WebMethod]public bool UpdateOrder(int orderID, int orderStatus
, DateTime orderDate, int customerID)
```

можно использовать следующий код:

```
[WebMethod]
public bool UpdateOrder(string xmlOrder)
```

В этом случае клиентская часть приложения принимает и предоставляет данные в виде XML-документов. Такой подход, строго ориентированный на обмен документами, позволяет сохранить крупноструктурным интерфейс API Web-служб и в то же время избежать чрезмерной "многословности" при обмене данными между клиентом и сервером (рис. А.4).

Рис. А.4. Общий вид архитектуры приложения

Несколько слов о транзакциях

И наконец, о транзакциях. Как уже отмечалось, я делаю методы уровня обслуживания статическими, если не требуются транзакции, но в данном примере приложения, управляющего заказами, я склонен считать, что транзакции должны стать частью общего решения.

Это, в частности, подразумевает транзактный характер метода `UpdateOrderStatus()`, т.е. данный метод должен быть не статическим, а действующим на уровне экземпляра объекта, и поэтому он должен быть обозначен в .NET атрибутом `[AutoComplete]`. А поскольку я собираюсь обозначить атрибутом `[Transaction]` весь класс, к которому принадлежит этот метод и который должен наследовать от класса `ServicedComponent`, то данный метод следует перенести из класса `OrderServices` в класс `OrderServicesTx`, где `Tx` означает "транзактный".

Таким образом, метод Web-службы `UpdateOrderStatus()` теперь должен создавать экземпляр объекта класса `OrderServicesTx` и вызывать метод `UpdateOrderStatus()` этого объекта вместо прежнего метода того же наименования в классе `OrderServices`. При выполнении метода `OrderServicesTx.UpdateOrderStatus()` выполняются распределенные транзакции, которые автоматически фиксируются, если выполнение данного метода завершается без формирования исключений, а иначе происходит откат транзакций.

На мой взгляд, уровень обслуживания является отличным местом для осуществления контроля транзакций в приложении, а также для проверки безопасности, регистрации и других аналогичных аспектов приложения.

Следует, однако, иметь в виду, что не *все* методы уровня обслуживания должны быть организованы вокруг объектов-сущностей из уровня модели предметной области. В частности, применение классов `EmployeeServices`, `CustomerServices` и прочих совсем не означает, что на уровне обслуживания не должны быть также классы `LoggingServices` и `SecurityServices`.

Выводы

Я отдаю предпочтение такому стилю проектирования, который предполагает размещение бо́льшей части бизнес-логики на уровне обслуживания, где осуществляются операции над объектно-ориентированной структурой данных, представленной моделью предметной области. Обмен данными между клиентом и сервером осуществляется только на основе документов, что исключает любые попытки экспортировать модель предметной области на сторону клиента или даже доступ с его стороны к серверной модели предметной области (например, посредством удаленных вызовов).

Надеюсь, что мне удалось дать читателю некоторое представление о том, как и почему я пользуюсь уровнем модели предметной области и, в частности, при разработке приложения Джимми.

Модель базы данных как модель предметной области

Франс Боума

Для работы с данными на уровне семантики нередко оказывается полезно снабжать общие определения элементов соответствующими порциями логики. Так, система обработки заказов оперирует, среди прочего, элементами *заказа*. Для того чтобы выяснить, как работает логика, целесообразно дать определение понятия *заказа*. Это позволит нам описать функции системы, указав операции над элементами заказа, и снабдить ими определение данного элемента.

Такой элемент заказа (`Order`) содержит другие элементы, как, например, идентификационный номер заказа (`OrderID`) и дата отгрузки (`ShippingDate`), а также тесно связан с другим элементом — строкой заказа (`OrderRow`), который, в свою очередь, содержит другие элементы. Можно даже сказать, что заказ *состоит*, с одной стороны, из ряда элементов `OrderRow`, а с другой стороны, из элементов значений. В чем же отличие содержимого элементов значений `OrderID` от содержимого ряда элементов `OrderRow`? Ответ на этот вопрос имеет большое значение для дальнейшего воплощения понятия заказа в конкретном программном коде, реализующем систему обработки заказов.

Место обитания сущностей

При анализе реляционной модели, разработанной доктором Е.Ф. Коддом [Codd Relational Model], обращают на себя внимание два отдельных элемента: `Order` и `OrderRow`, между которыми существует *связь*, а сами элементы образуют *отношение* на основании значений, которые они содержат, например, `OrderID`.

Но анализ модели предметной области, впервые разработанной Мартином Фаулером и другими [Fowler PoEAA], показывает, что это не совсем так. В частности, элементы `OrderRow` можно рассматривать среди ряда прочих элементов как сумму заказа и работать с заказом как с единым целым, включая и элементы `OrderRow`.

Что такое сущность

В 1970 году доктор Питер Чен определил понятие объекта-сущности (или просто *сущности*) для своей модели "сущность-связь" [Chen ER], основанной на реляционной модели Кодда. Понятие сущности очень полезно для определения элементов `Order` и `OrderRow` в реляционной модели. Итак, Чен определил сущность следующим образом.

> "*Сущность* и *множество сущностей*. Это понятие обозначает сущность, существующую в нашем воображении. Сущности разделяются на три разных множества, например, EMPLOYEE, PROJECT и DEPARTMENT. С каждым множеством сущностей связан предикат, по которому тестируется принадлежность сущности к этому множеству. Так, если известно, что сущность находится во множестве сущностей EMPLOYEE, то известно также, что у нее имеются свойства, общие и для

других сущностей во множестве EMPLOYEE. К этим свойствам относится и упомянутый выше тестовый предикат".

Используя модель "сущность-связь", мы можем определить такие объекты-сущности, как `Order` и `OrderRow`, и найти им подходящее место в реляционной модели, определяющей тип базы данных. Но если мы воспользуемся следующим определением сущности, данным Эриком Эвансом: "Объект, прослеживаемый по разным состояниям и даже по разным реализациям", то отклонимся от реляционной модели. Главное отличие определения сущности, данного Ченом, от аналогичного определения Эванса заключается в элементе абстракции сущности, допускающем ее существование без состояния и даже без физического представления. Согласно Эвансу, сущность физически существует и представляет собой объект с определенным состоянием и поведением. А по абстрактному определению сущность не зависит от контекста, в котором используются ее данные, поскольку интерпретация данных сущности производится не самой сущностью (из-за отсутствия у нее поведения), а внешней логикой.

Во избежание недоразумений, связанных с неверной интерпретацией данного понятия, мы будем в дальнейшем придерживаться определения, данного доктором Питером Ченом, поскольку оно позволяет абстрагироваться от реальных предметов (как физических, так и виртуальных элементов), не привязываясь к контексту или конкретному содержимому, что очень важно. Следовательно, такое определение идеально подходит для описания таких элементов реляционной модели, как `Customer` или `Order`. В этом случае физический элемент `Customer` называется *экземпляром* объекта-сущности.

Где находится сущность

В каждом приложении приходится иметь дело с таким явлением, как *состояние*. Но состояние — это слишком общее понятие. У большинства приложений имеются самые разные состояния. К наиболее важным из них относятся *состояние пользователя* и *состояние приложения*. В частности, состояние пользователя можно рассматривать как состояние хранения всех объектов или данных каждого пользователя в определенный момент времени и в пределах области действия пользователя. Примером состояния пользователя может служить содержимое объекта, определяющего сеанс работы пользователя с приложением ASP.NET в определенный момент времени. Состояние приложения отличается от состояния пользователя тем, что его можно рассматривать как состояние хранения всех объектов или данных в области действия приложения. Примером такого состояния может служить содержимое базы данных, общее для всех пользователей Web-приложения в конкретный момент времени. Рассматривать состояние пользователя в качестве подмножества состояния приложения было бы неразумно. Ведь когда пользователь находится на пятом шаге процедуры мастера, состояние пользователя хранит данные о предыдущих шагах этой процедуры, тогда как состояние приложения не меняется. Оно может измениться лишь *после* завершения процедуры мастера.

Очень важно выяснить, где именно находится сущность: в состоянии приложения или же в состоянии пользователя. Если сущность находится в состоянии пользователя, она оказывается локальной для пользователя, которому принадлежит данное состояние, а другим пользователям эта сущность недоступна, и поэтому они не могут оперировать ею. После своего *создания* экземпляр объекта-сущности (например, заказ в упомянутой выше системе обработки заказов) существует в конкретном приложении и является неотъемлемой частью состояния приложения. Но в процессе составления заказа, когда, например, пользователь заполняет форму заказа, сам заказ фактически не создается, а вместо этого в состоянии пользователя находится временный набор данных, который в конечном счете превратится в заказ по завершении его составления. Иными словами, после своего создания экземпляр объекта-сущности *сохраняется* в состоянии приложения.

Имеются разные виды состояний приложения: разделяемое, в котором экземпляры объектов-сущностей находятся в оперативной памяти, или сохраняемое, в котором экземпляры объектов-сущностей находятся в базе данных. В большинстве приложений, работающих с экземплярами объектов-сущностей, используется так называемое *постоянное хранение* данных этих экземпляров на случай прекращения подачи электроэнергии и прочих непредвиденных ситуаций, приводящих к аварийному выключению компьютера, а следовательно, и к потере содержимого оперативной памяти. Если в приложении используется постоянное хранение, то данные, находящие на постоянном хранении, можно приравнять к конкретному состоянию приложения. Так, при закрытии приложения, например, для целей сопровождения, оно не теряет своего состояния, которое остается доступным после повторного открытия приложения. Следовательно, экземпляр объекта-сущности в оперативной памяти является *зеркальным отображением* конкретного экземпляра объекта-сущности, находящегося на постоянном хранении, и логика приложения использует такое зеркальное отображение для изменения того экземпляра объекта-сущности, который находится на постоянном хранении.

Преобразование классов в таблицы в сравнении с преобразованием таблиц в классы

Объектно-реляционное преобразование осуществляет преобразование объектной модели в реляционную модель, и обратно. Иными словами, оно преобразует объекты в экземпляры объектов-сущностей, находящиеся на постоянном хранении, и обратно. В целом его можно определить следующим образом: поле класса из объектной модели связывается с атрибутом сущности в реляционной модели, и наоборот.

В связи с этим возникает вопрос: что за чем следует? Нужно ли сначала определить классы сущностей (т.е. классы, представляющие определения сущностей, например, класс `Order`, представляющий сущность `Order`) и создать с их помощью сущности реляционной модели с соответствующими атрибутами или же определить, прежде всего, реляционную модель, чтобы затем использовать ее для определения классов сущностей?

Как и в остальном, однозначного ответа на этот вопрос не существует, поскольку все зависит от конкретных обстоятельств. Так, если придерживаться модели предметной области, то начинать, скорее всего, следует с предметной области для определения классов, а возможно, и некоторой иерархии наследования. Для такой модели классов потребуется реляционная модель, чтобы хранить данные. Она может быть представлена в виде одной таблицы с первичным ключом, состоящим из идентификатора объекта, поля большого двоичного объекта и нескольких элементов метаданных, описывающих этот объект. В таком случае вполне естественно преобразовать класс в элементы реляционной модели, предварительно убедившись в том, что реляционная модель построена наиболее подходящим для объектной модели образом.

Если же начать с реляционной модели и затем построить модель "сущность-связь", то сущность, скорее всего, придется преобразовать в соответствующий класс реляционной модели. Этим данный подход отличается, в частности, от модели предметной области, поскольку реляционная модель не поддерживает иерархии наследования. Так, иерархию "Физическое лицо <– Служащий <– Руководитель" нельзя смоделировать таким образом, чтобы она представляла также саму иерархию. Конечно, можно построить реляционную модель, семантически интерпретируемую как иерархию наследования, но по определению она не будет представлять такую иерархию.

В этом состоит главное отличие между обоими подходами. Если начинать с классов и затем перейти к базе данных, то реляционная модель и база данных будут использоваться как место для хранения данных. А если начинать с реляционной модели и затем перейти к классам, то классы будут использоваться как средство для работы с реляционной моделью в стиле ООП.

В связи с тем что мы отдали ранее предпочтение определению сущностей, данному Ченом, остановим свой выбор на подходе, предполагающем определение сначала реляционной модели, а затем переход к классам. В заключение этого раздела будет показано, как навести мосты между двумя упомянутыми выше подходами.

Манипулирование данными в стиле ООП

Классы, представляющие сущности, дают разработчику возможность определить сущности в коде аналогично тому, как в физически реализованной модели "сущность-связь" сущности определяются с помощью таблиц в месте постоянного хранения. Используя упоминавшийся ранее метод объектно-реляционного преобразования, разработчик может манипулировать экземплярами объектов-сущностей в месте постоянного хранения, используя находящиеся в оперативной памяти зеркальные отображения экземпляров классов сущностей. Такой процесс всегда носит групповой характер, поскольку разработчик манипулирует данными отдельно от места их постоянного хранения. В качестве управляющей среды в данном случае служит объектно-реляционный преобразователь, контролирующий связь между экземплярами объектов-сущностей в месте их постоянного хранения и зеркальными отображениями, находящимися в оперативной памяти.

Разработчик может, например, указать объектно-реляционному преобразователю загрузить определенную совокупность экземпляров заказов в оперативную память. В итоге каждому экземпляру заказа в месте постоянного хранения будет соответствовать зеркальное отображение внутри экземпляра соответствующего класса сущности в оперативной памяти. Таким образом, разработчик может манипулировать каждым зеркальным отображением экземпляра объекта-сущности посредством экземпляра класса сущности, представлять такие зеркальные отображения в форме заказа или же предоставлять их в качестве результата обслуживания. Манипулируемые зеркальные отображения экземпляров объектов-сущностей должны быть сохранены, чтобы внесенные изменения стали постоянными. С точки зрения разработчика этот процесс похож на сохранение манипулируемых данных экземпляров объектов-сущностей, находящихся внутри объектов, в месте их постоянного хранения аналогично сохранению в файле фрагмента текста, набранного пользователем в текстовом редакторе. Объектно-реляционный преобразователь выполняет сохранение вместо разработчика. Но поскольку он работает с зеркальными отображениями экземпляров объектов-сущностей, вместо сохранения фактически выполняется *обновление* экземпляра объекта-сущности в месте его постоянного хранения, причем изменения, сохраненные в зеркальном отображении, получаются из кода, написанного разработчиком.

Связь между сущностями в реляционной модели должна быть представлена в коде с помощью функций, предоставляемых объектно-реляционным преобразователем. Это позволяет разработчику прослеживать связи от одной сущности к другой. Например, после загрузки в оперативную память экземпляра объекта `Customer` в системе обработки заказов разработчик может проследить все экземпляры объектов `Order`, связанных с объектом `Customer`, используя функции объектно-реляционного преобразователя, будь то объект, представляющий совокупность в объекте `Customer`, или же новый запрос у объектно-реляционного преобразователя экземпляров `Order`, связанных с данным экземпляром `Customer`.

Такой режим работы с сущностями достаточно статичен: построение сущностей во время выполнения с помощью определенного сочетания атрибутов из нескольких связанных сущностей отнюдь не приводит к появлению классов, представляющих сущности, поскольку такие классы должны присутствовать во время компиляции. Но это совсем не означает, что экземпляры объектов-сущностей, создаваемые во время выполнения с помощью определенного сочетания атрибутов (например, путем выбора с внешним соединением), не могут быть загружены в оперативную память, хотя они будут при этом представлять не сохраняемые, а, скорее, виртуальные сущности. Этот дополнительный уровень абстракции чаще всего используется в таких операциях “только для чтения”, как составление отчетов и постоянных списков, где обычно требуется определенное сочетание атрибутов из связанных сущностей. В качестве примера определения подобного списка может служить сочетание всех атрибутов сущности `Order` и атрибута “наименования организации” из сущности `Customer`.

Самое главное для успешного манипулирования данными в стиле ООП — достаточная гибкость функций, контролирующих связь между зеркальными отображени-

ями экземпляров объектов-сущностей в оперативной памяти и физическими их экземплярами. Это означает, что функции составления отчетов и списков атрибутов должны определяться и загружаться в оперативную память, не прибегая к помощи другого приложения только ради более динамичного использования данных в экземплярах объектов-сущностей.

Функциональный анализ как основа разработки приложений

Для эффективной подготовки реляционной модели, т.е. преобразования определений сущностей из реляционной модели в сущности, представляющие классы, а также в сами эти классы, очень важно иметь возможность использовать результаты работы, проделанной ранее на стадии функционального анализа в рамках проекта разработки программного обеспечения. Эта стадия является типичной для большинства методов разработки программного обеспечения. На данной стадии определяются в самом общем виде и документально оформляются функциональные требования и возможности системы. В течение ряда лет был разработан ряд методов, оказывающих помощь в работе над проектом на данной стадии. К ним, в частности, относится метод NIAM [Halpin/Nijssen Conceptual Schema], который в дальнейшем был доработан Т.А. Халпином [Halpin IMRD] и стал методом ORM (Object Role Modeling — Моделирование "объект-роль"). Методы NIAM и ORM упрощают интерпретацию результатов функционального анализа на понятном для заказчика языке в виде таких несложных предложений, как "У клиента имеется заказ" или "Заказ принадлежит клиенту". Подобные предложения затем используются для определения сущностей и связей между ними в абстрактной модели NIAM/ORM. Как правило, для этой цели применяются такие визуальные средства, как Microsoft Visio.

О значении результатов функционального анализа

Преимущество моделирования результатов функционального анализа с помощью таких методов, как NIAM или ORM, заключается в том, что подобные результаты документально оформляются в абстрактной модели на стадии функционального анализа, и в то же время она служит источником для построения реляционной модели, которую предполагается внедрить в разрабатываемое приложение. Используя такие инструменты, как Microsoft Visio, можно построить реляционную модель на основании модели "сущность-связь", сформированной из модели NIAM/ORM, которая может также служить для построения физической реляционной модели в СУБД. Метаданные, образующие определение реляционной модели в СУБД, могут быть затем использованы для формирования классов и преобразований конструкций.

Таким образом, под иерархию классов, с которой работают разработчики, закладывается солидное теоретическое обоснование, сложившееся в ходе функционального анализа на начальной стадии проекта. Это означает, что если в разработке приложения что-то меняется (например, часть его функциональных возможностей), такое изменение происходит по следующей цепочке: изменение модели NIAM/ORM,

коррекция реляционной модели с учетом новой модели "сущность-связь" и обновленной модели NIAM/ORM, а затем коррекция классов по новой модели "сущность-связь". Возможен и обратный порядок, когда требуется найти основания для применения конструкций кода, с которыми работает разработчик. Так, для выяснения причин выбора конструкций кода, прослеживающих связи между экземплярами объектов-сущностей, достаточно вернуться назад по цепочке от соответствующего класса к результатам функционального анализа, чтобы обнаружить теоретическое обоснование для применения таких конструкций. Тесная связь между теоретическим обоснованием и конкретным кодом имеет решающее значение для успешной разработки качественного и удобного для сопровождения программного обеспечения.

Функциональные процессы в качестве потребителей данных и размещение бизнес-логики

Если настоящие определения сущностей находятся в реляционной модели, а по существу, в базе данных, то экземпляры объектов-сущностей, находящиеся в оперативной памяти, являются всего лишь зеркальными отображениями настоящих экземпляров объектов-сущностей, находящихся в базе данных, и поэтому в этих сущностях нет места для поведения или правил бизнес-логики. Разумеется, ввести поведение в классы сущностей несложно. Но насколько это будет логично для классов сущностей, представляющих определения таких сущностей в реляционной базе данных? Все зависит от той категории бизнес-логики, которую требуется ввести в сущность в виде поведения. Таких категорий можно выделить три:

- Бизнес-логика, ориентированная на атрибуты.
- Бизнес-логика, ориентированная на одну сущность.
- Бизнес-логика, ориентированная на многие сущности.

Бизнес-логика, ориентированная на атрибуты, представляет собой категорию, состоящую из правил, аналогичных выражению `OrderId > 0`. Это довольно простые правила, действующие подобно ограничениям, накладываемым на одно поле сущности. Правила данной категории могут соблюдаться при установке значения в поле сущности.

Бизнес-логика, ориентированная на одну сущность, представляет собой категорию, состоящую из правил, аналогичных выражению `ShippingDate >= OrderDate` и действующих подобно ограничениям. Правила этой категории могут соблюдаться при загрузке сущности в экземпляр объекта-сущности, находящийся в оперативной памяти, сохранении сущности в месте постоянного хранения или же при проверке сущности на соответствие конкретному контексту.

Бизнес-логика, ориентированная на многие сущности, представляет собой категорию, состоящую из правил, распространяющихся на несколько сущностей, например, правило проверки того, является ли клиент покупателем золота. Для соблюдения этого правила бизнес-логика должна обратиться к сущностям заказов и связанными с ними сущностям подробностей заказов.

Все три категории бизнес-логики зависят от *контекста*, в котором используется сущность, хотя не все правила отдельной категории являются контекстно-зависимыми. В категории бизнес-логики, ориентированной на атрибуты, используются, в основном, правила, не связанные с контекстом, и поэтому такую бизнес-логику уместно ввести в класс сущности в виде поведения. Бизнес-логика, ориентированная на одну сущность, обычно не годится для ввода в класс сущности в виде поведения, поскольку в этой категории большая часть правил, проверяющих соответствие сущности конкретному контексту, может и должна изменяться, когда сущность используется в другом контексте. Правила из категории бизнес-логики, ориентированной на многие сущности, распространяются на несколько сущностей и поэтому не могут быть введены в одну сущность, кроме тех случаев, когда они слишком тесно связаны с контекстом, в котором они используются.

Внедряемые правила

Для того чтобы сущность можно было и далее использовать в качестве концепции, не связанной с конкретным контекстом, проблему контекстно-зависимых правил из категорий бизнес-логики, ориентированной на атрибуты и на одну сущность, можно решить с помощью *внедряемых правил*. К таким правилам относятся объекты, содержащие правила бизнес-логики и внедряемые в объект-сущность во время выполнения. Преимущество такого подхода заключается в том, что классы сущностей не связаны с контекстом, в котором они используются, но могут применяться в любом контексте, который предусмотрен в разрабатываемой системе. Достаточно составить ряд внедряемых правил на каждый контекст или даже на каждую сущность и применять эти правила к сущности в зависимости от состояния контекста, просто устанавливая ссылку на соответствующий объект внедряемых правил. Процессы, определяющие объекты правил для внедрения в сущности, предназначены для сохранения контекста, в котором используется сущность. Они представляют конкретные бизнес-процессы и называются *функциональными процессами*.

Функциональные процессы

Ранее в подразделе “О значении результатов функционального анализа” была рассмотрена стадия функционального анализа и подчеркнуто особое значение, которое имеет сохранение тесной связи между проанализированными функциональными возможностями и конкретной их реализацией. В разрабатываемой системе зачастую предусматривается автоматизация определенных бизнес-процессов, а функциональный анализ описывает эти процессы в абстрактной форме. Для сохранения как можно более тесной связи между анализом и реализацией функциональных возможностей в ходе проекта нередко выполняется моделирование конкретной реализации после абстрактного описания бизнес-процесса, в результате чего появляются классы, которые можно назвать *функциональными процессами*, поскольку они практически не содержат данных и только выполняют определенные функции.

Функциональные процессы идеально подходят для реализации правил из категории бизнес-логики, ориентированной на многие сущности. В приведенном выше

примере с покупателем золота процесс превращения клиента в покупателя золота может быть реализован как функциональный процесс, потребляющий объект-сущность `Customer`, и связанные с ним объекты-сущности `Order`, обновляющий некоторые поля в объекте-сущности `Customer` и сохраняющий этот объект-сущность по завершении обновления. А поскольку функциональные процессы фактически выполняют определенные этапы бизнес-процесса, то они служат именно тем местом, где присутствует контекст, в котором сущности потребляются, а также единственным подходящим местом, где следует принимать решение о тех правилах, которые должны быть внедрены в объект-сущность в конкретный момент времени для определенного состояния контекста.

Идеальные условия для применения модели NIAM/ORM к классам и базам данных

Для большинства людей реальность далеко не всегда соответствует предполагаемому идеалу, и в этом отношении разработка программного обеспечения не является исключением. Как отмечалось, метаданные физической реляционной модели используются для формирования классов и преобразований на основании определений классов сущностей, с которыми приходится иметь дело разработчику. Более идеальным подходом было бы применение модели NIAM/ORM для непосредственного формирования классов сущностей, исключая преобразование из метаданных в определение класса. Благодаря такому подходу можно приблизиться еще на один шаг к заветной цели: сделать проектирование таким же полезным, как и само приложение.

Трудно сказать, когда такой идеал станет реальностью, если это вообще возможно, поскольку слишком много факторов, оказывающих влияние на успешное проектирование программного обеспечения, выходят за рамки вычислительной техники. Тем не менее было бы интересно посмотреть, чего можно добиться уже сегодня, применяя такие современные методы проектирования, как разработка, основанная на модели.

Прагматизм и нетрадиционный подход

Ингмар Лундберг

Когда Джимми предложил мне написать этот раздел в качестве приглашенного автора, я сразу же согласился. Должен признать, что это мне польстило. Но вскоре у меня возникло сомнение: о чем писать и как изложить свои соображения на данную тему кратко и точно, а самое главное — как вписать архитектуру приложения, разработанного мной в .NET, в концепцию модели предметной области?

Я попытаюсь дать общее представление об архитектуре приложения, созданного мной для моего бывшего работодателя. Теперь я могу без ложной скромности назвать эту архитектуру своей, хотя я и не пользуюсь ею в текущих проектах. Кроме того, я рассмотрю ряд вопросов, поставленных Джимми в перечне требований к приложению и представляющих определенный интерес с точки зрения сильных и слабых сто-

рон моей архитектуры. В ограниченных рамках одного раздела невозможно представить всю картину в целом, но поскольку назначение этого раздела — показать разные подходы к построению архитектуры приложений, я надеюсь, что приведенный ниже краткий материал достигнет своей цели. В контексте этого раздела рассматривается конкретная архитектура, причем такой подход не является общей догмой.

Предпосылки и история вопроса

Для того чтобы пояснить, почему моя архитектура построена так, а не иначе, требуется некоторая исходная информация и историческая справка. На мое нынешнее представление об архитектуре приложений повлияли следующие факторы. Во-первых, до .NET я разработал архитектуру в среде MTS/COM+. С появлением .NET некоторые (доморощенные) концепции старой архитектуры уже настолько устоялись в представлении разработчиков, что отказываться от них было бы слишком накладно[2]. Во-вторых, моя архитектура не обязательно является самым лучшим вариантом из того, что я мог придумать. На мой взгляд, она воплощает удачное сочетание современных методов проектирования и разработки приложений, опыта эксплуатации приложений и требуемого соотношения качества и затрат.

Краткий обзор архитектуры

На каких же концепциях построена моя архитектура? Тем, кто знаком со средой MTS/COM+, должны быть хорошо известны концепции ООП без сохранения состояния. В частности, объект процесса [Ewald TxCOM+] не имеет состояния и представляет единый *бизнес-объект* в одном экземпляре. Он содержит полный набор операций для одного или целого ряда бизнес-объектов определенного типа. Отсутствие состояния дает такую возможность.

В основу рассматриваемой архитектуры положена базовая структура взаимодействующих классов. Она сама по себе полезна, но для более эффективного ее использования служит вспомогательный инструмент BODef (на рис. А.11 приведен моментальный снимок части интерфейса этого инструмента), в котором определяются бизнес-объекты и генерируется шаблонный код. Генератор кода представляет собой нечто большее, чем обычный мастер, поскольку он позволяет время от времени генерировать код, не теряя фрагменты кода, введенные вручную. Для этой цели служит шаблон Generation Gap (Разрыв в генерации кода) [Vlissides Pattern Hatching]. На рис. А.5 представлено одно (частичное) определение бизнес-объектов для рассматриваемого примера приложения.

Стандартный способ разделения классов приложения приведен на рис. А.6. Все модули и сборки, кроме тех, что обозначены меткой <<базовая структура>>, характерны для конкретного приложения. Так, поддержка бизнес-объектов (в виде сборки

[2] Если у вас есть опыт внедрения технических новшеств в отделе информационных технологий, вы не могли не заметить, что предлагаемый вами переход к новой технологии не сразу встречает понимание у ваших коллег.

BoSupport) должна быть доступна не только в среднем ярусе приложения, но и для клиентских частей приложения с удаленным вызовом. В рассматриваемом примере приложения Джимми его клиентская часть с развитыми функциональными возможностями должна связываться со средним ярусом приложения посредством удаленных вызовов[3].

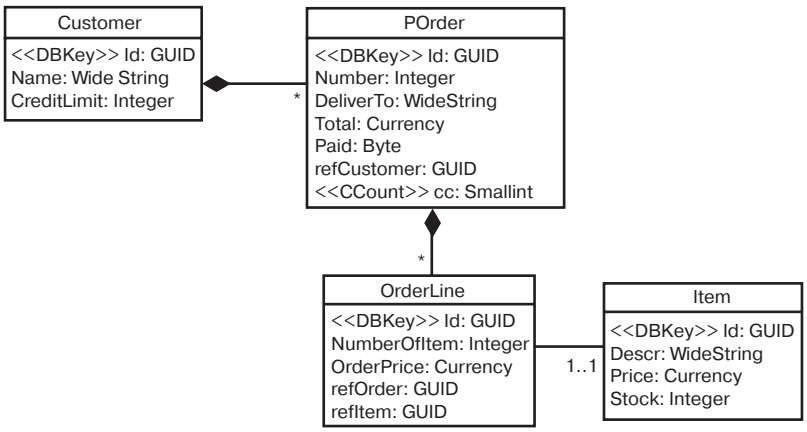


Рис. А.5. Бизнес-объекты для рассматриваемого примера приложения

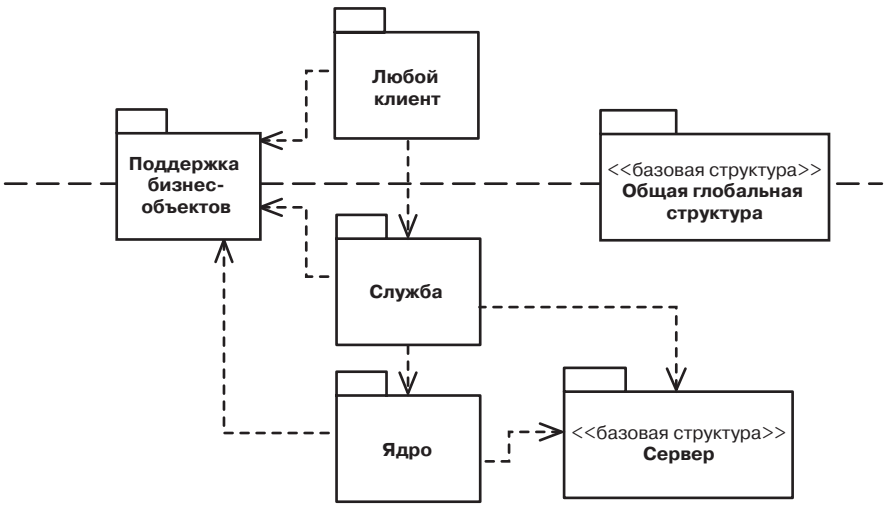


Рис. А.6. Общий вид архитектуры приложения

Сборка службы (Service) должна содержать, по крайней мере, интерфейсный объект, реализующий некоторые элементарные интерфейсы для поддержки основ-

[3] Как упоминалось в начале этого приложения, наличие сервера приложений было первоначальным требованием, но затем данное требование было снято. — *Примеч. авт.*

ных операций создания, чтения, обновления и удаления (CRUD), и самым важным среди них является интерфейс `IBvDBOReq` (рис. А.7).

Вместо формирования такого набора интерфейсов наследуются все базовые функции (от объекта `FacadeBase`), включая реализацию этих интерфейсов.

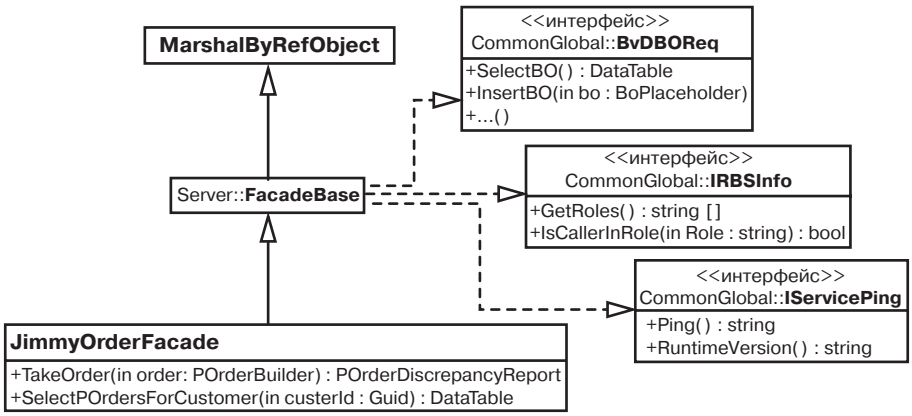


Рис. А.7. Архитектура для интерфейсов приложения

Что же касается постоянного хранения, то наличие системы управления реляционной базой данных (СУРБД) столь же очевидно, как и наличие самой файловой системы на месте эксплуатации приложения. В действительности, наличие СУРБД является безусловным требованием, поскольку именно так лучше всего организовать сохранение и восстановление данных приложения.

Выборка данных

Данные могут быть возвращены из среднего яруса приложения в его клиентскую часть в любом формате: `DataTable`, `DataSet`, `DataSet` со строгим контролем типов, XML или графов объектов в последовательной форме. Самым простым и чаще всего используемым для этой цели форматом является `DataTable`, поскольку для получения массива данных бизнес-объекта достаточно вернуть их построчно.

Заполнитель бизнес-объекта

Строку из таблицы `DataTable` нетрудно преобразовать в заполнитель бизнес-объекта (`BoPlaceholder`). Класс `BoPlaceholder`, расположенный в сборке `BoSupport`, полезен сам по себе, но в то же время он является базовым классом для конкретных заполнителей бизнес-объектов.

Такие заполнители формируются инструментом BODef (один из них — `POrderPh` (Заполнитель заказа на поставку) приведен на рис. А.8). Всякий раз, когда выбирается режим формирования, образуется класс `Impl`. Краевой класс образуется только один раз, что составляет саму суть шаблона Generation Gap.

Объекты-заполнители можно рассматривать в качестве носителей информации или объектов передачи данных, хотя они пригодны для применения и в более сложных сценариях поведения.

И разумеется, производные класса `BoPlaceholder` находятся в сборке `BoSupport`, которая, по-видимому, присутствует как в среднем ярусе, так и в клиентском ярусе приложения (т.е. в клиентской части приложения с развитыми функциональными возможностями).

Данные в формате `DataTable` могут быть возвращены из среднего яруса клиентскому ярусу, где выполняется преобразование экземпляров `DataRow` в экземпляры `BoPlaceholder`. Кроме того, можно возвратить ряд объектов `BoPlaceholder` (например, в виде списка `ArrayList`) для непосредственного использования в клиентской части приложения. Объекты `BoPlaceholder` являются упорядочиваемыми. Впрочем, можно найти немало примеров привязки данных формата `DataTable`, тогда как примеры привязки данных в формате `BoPlaceholder` весьма редки.

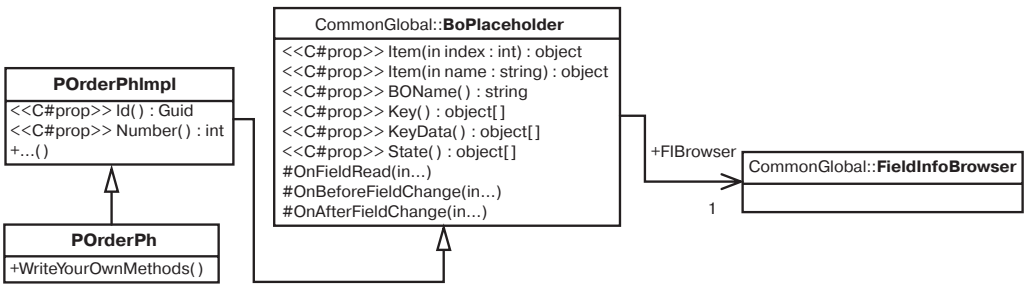


Рис. А.8. Применение класса `BoPlaceholder`

Объекты процесса

Окончательное исполнение и соблюдение бизнес-правил является задачей базовых бизнес-объектов, называемых также объектами процесса. Такие объекты не имеют состояния и существуют недолго. Пример объекта процесса `POrder` приведен на рис. А.9. Как видите, такие объекты следуют шаблону Generation Gap. Они реализованы в сборке `Core` (Ядро). Авторизация составляет ядро бизнес-правил, и поэтому соответствующими функциями наделены общедоступные методы объектов процесса.

Несмотря на то что за соблюдение бизнес-правил отвечают объекты ядра, один из них — `POrder` приведен на рис. А.9, реализовать бизнес-правила можно и в сборке `BoSupport`, чтобы сделать их доступными для клиентской части приложения с развитыми функциональными возможностями. Но сделав это, следует перепроверить бизнес-правила в объектах процесса перед тем, как сохранить внесенные изменения. Ведь клиентская часть приложения может вполне переопределить бизнес-правило, например, верхний предел суммы заказа, посредством рефлексии перед тем, как предоставить его интерфейсному объекту в среднем ярусе приложения.

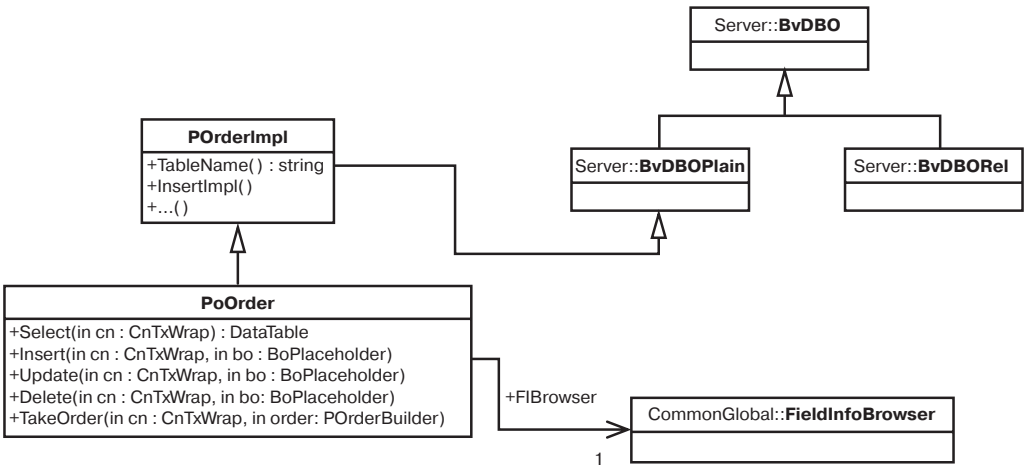


Рис. А.9. Размещение специальных правил в объекте `POrder` (в данном примере — в методе `TakeOrder()`)

Объекты процесса зачастую передаются объектам `BoPlaceholder` и "скрываются" внутри интерфейсного объекта. Концептуально объект процесса и заполнитель, например, `POrder` и `POrderPh`, являются составными частями одного и того же (логического) бизнес-объекта.

Метаданные

Возможно, вы уже обратили внимание, что на некоторых приведенных выше рисунках немало метаданных присутствует в классах базовой структуры. Более полное, хотя и не поясняемое подробно, представление метаданных приведено на рис. А.10. Метаданные придают динамичность таким операциям, как сохранение данных и обращение к пользовательскому интерфейсу.

Операторы SQL, находящиеся в среднем ярусе приложения, динамически формируются из метаданных, и в итоге получается очень удачная и надежная реализация. Если же в ней обнаружатся критические элементы, то обработку средствами SQL можно реализовать самостоятельно, например, с помощью хранимых процедур.

Пример приложения Джимми для обработки заказов

А теперь перейдем непосредственно к примеру приложения Джимми для обработки заказов. Допустим, что пользователь такого приложения связывается с клиентом для его обслуживания по телефону. Назовем такого пользователя принимающим заказы, а соответствующую часть приложения — клиентской с развитыми функциональными возможностями. В этом можно усматривать стремление разработать приложение электронной коммерции, но даже если и пойти по такому пути, это повлияет не на само проектирование, а, скорее, на функции пользователя и администратора системы. В частности, пользователь может выполнять бо́льшую часть операций в системе,

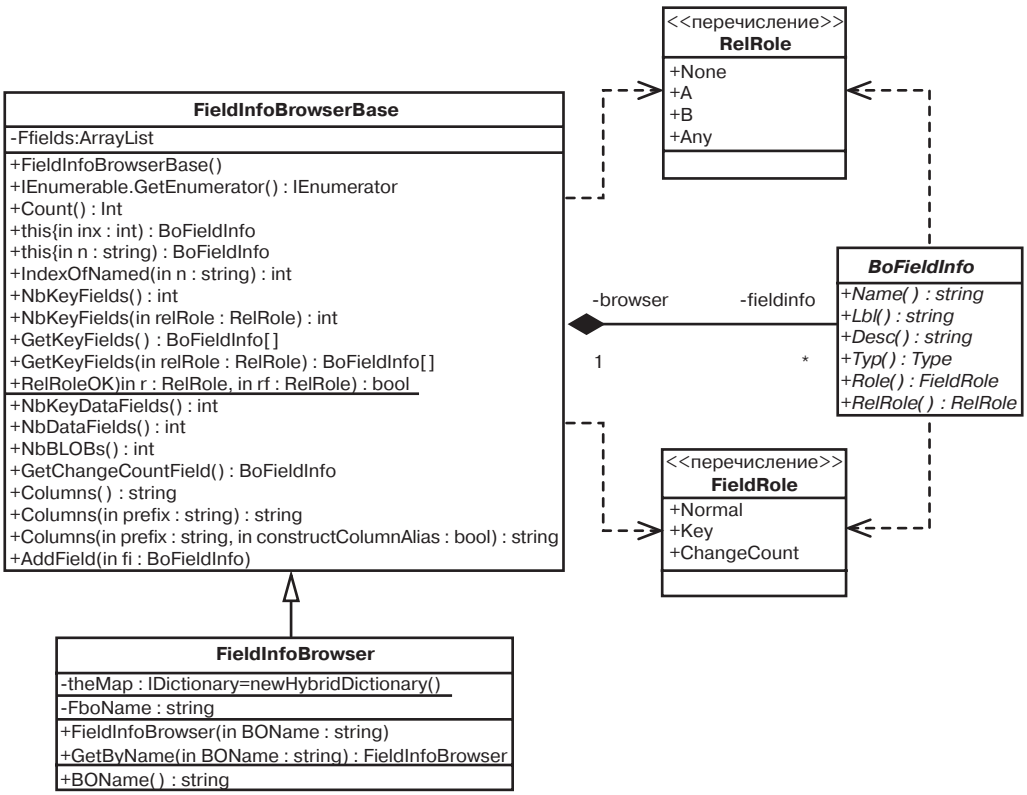


Рис. А.10. Полное представление метаданных

а администратор — почти все, что угодно. (В приложении электронной коммерции могут потребоваться функции пользователя с общим доступом.)

Ради краткости мы не будем рассматривать бо́льшую часть атрибутов. В частности, обработка адреса клиента или адреса доставки заказанных товаров не должна вызвать особых затруднений. На ранних стадиях разработки приложения основное внимание зачастую приходится уделять взаимосвязям между объектами (и, разумеется, их поведению и взаимодействию друг с другом), а не конкретным атрибутам.

Простая обработка реестра

Решение этой задачи предполагает обработку данных клиента. Для этого потребуются самые простые функции обработки реестра. Далее я представлю собственное решение этой задачи, чтобы поделиться своими идеями и раскрыть потенциальные возможности включения метаданных в бизнес-объекты.

На рис. А.11 приведен моментальный снимок экрана инструмента BODef, предназначенного для определения бизнес-объектов (в данном случае показан объект `Customer`). Как видите, помимо имени поля и его типа к этому полю присоединена дополнительная информация.

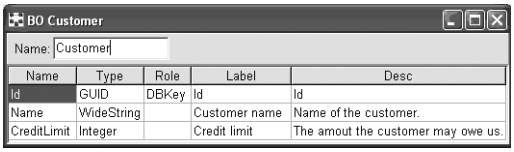


Рис. А.11. Определение объекта класса `Customer` в BODef

На рис А.12 приведена форма из приложения обработки заказов с метками, взятыми из метаданных. В такой сеточной форме можно вводить, править и удалять клиентов. Код, необходимый для этой цели, достаточно компактен (см. приведенный ниже фрагмент кода). Главная роль в нем принадлежит классу `TableHandler` (Обработчик таблиц), в котором используются метаданные бизнес-объекта `Customer`, общий интерфейс `IBvDBOReq`, а также взаимосвязь между классами `BoPlaceholder` и `DataRow` в архитектуре приложения.

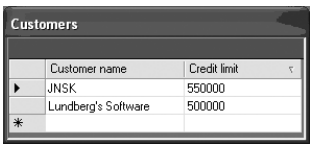


Рис. А.12. Стандартная реализация класса `Customer`

```
public CustomerFormList()
{
  InitializeComponent();

  _tabHndlr = new TableHandlerGuid(SvcFactory.Facade,
    CustomerPh.FieldInfoBrowser.BOName);
  _tabHndlr.TableChanged += new
    TableHandler.TableHandlerEventHandler
    (TableHandlerEventHandler);
  _tabHndlr.HookUpGrid(grd);
  _tabHndlr.FillGrid();
}
TableHandler _tabHndlr;

string[] confirmString = new string[] {
  "Do you want to add Customer?",
  "Do you want to save changes to Customer?",
  "Do you want to delete Customer?" };

void TableHandlerEventHandler(object sender,
  TableHandler.TableHandlerChangedEventArgs args)
{
```

```
  args.AcceptChanges = MessageBox.Show(
    confirmString[(int)args.TableHandlerChangeType],
    "Confirm", MessageBoxButtons.YesNo) == DialogResult.Yes;
}
```

Подробно анализировать этот код не имеет смысла. Он дает лишь самое общее представление о том, как обращаться с метаданными. С помощью метаданных можно также описать взаимосвязи между бизнес-объектами. (Ассоциативным объектам в моей архитектуре отведена ведущая роль.)

Бизнес-правила размещения заказов

Постановка задачи в примере приложения Джимми сформулирована так, чтобы ее можно было без труда расширить, рассмотрев под другими углами зрения. Именно это он поручил нам, приглашенным авторам.

Прежде всего, следует упомянуть о том, что не будет далее рассматриваться в постановке задачи, хотя это не значит, что такие возможности приложения нельзя реализовать. Это касается вопросов поиска заказов, составления списка заказов клиента и кредитоспособности нового клиента. Все эти вопросы не будут обсуждаться из-за нехватки места. Кроме того, уникальный номер заказа должен присваиваться во время выдачи заказа, поскольку до этого он не требуется принимающему заказы.

Процесс приема заказа и заполнения соответствующей формы должен иметь основательную поддержку. Для этой цели служит специальный формирователь[4], манипулирующий формой заказа. Кроме того, такой формирователь может стать упорядочиваемым объектом после составления заказа и его выдачи из клиентской части в средний ярус приложения.

В связи с многопользовательским режимом работы приложения возникает целый ряд непростых вопросов. Так, если во время разговора между принимающим заказ и потенциальным клиентом меняется цена на товар, то следует ли сразу же уведомить об этом принимающего заказ? Проверка цены на товар в хранилище мало что дает, поскольку если выбрать цену из хранилища в 10:55 и вывести ее на экран, она может измениться к 10:56. Аналогичная проблема возникает и с наличием товаров на складе: имеется ли заказанный товар на складе или его нужно заказать у поставщика?

Для решения данной проблемы можно было бы, в частности, организовать в среднем ярусе приложения немедленное уведомление всех клиентов (пользователей, принимающих заказы) обо всех изменениях в цене или наличии товаров на складе, чтобы клиент принял соответствующие меры. Но это слишком сложное решение. Ведь придется обеспечивать обмен данными между средним ярусом и всеми активными клиентскими частями приложения, а следовательно, синхронизировать этот процесс.

Попробуем найти более простое решение данной задачи, обратившись к бизнес-процессу. Принимающий заказ разговаривает с клиентом по телефону. Они обсуждают товары, необходимые клиенту, их цену и наличие на складе. В конечном ито-

[4] В частности, для формирования строки может служить класс `StringBuilder`. — *Примеч. авт.*

ге клиент решается на покупку и принимающий заказ выдает его в средний ярус приложения. В ответ из среднего яруса приложения возвращается отчет, в котором указываются окончательные цены и наличие товаров на складе. Последнее оказывает непосредственное влияние на дату доставки товаров. Все расхождения с устно оговоренными условиями заказа, если таковые имеются, отмечаются в отчете. Принимающий заказ обсуждает их с клиентом, и если клиента что-то не устраивает, заказ отменяется. Отмена заказа представляет собой отдельную операцию — транзакцию отмены или удаления.

Ясно, что если сообщить клиенту цену на товар, то придется в какой-то степени придерживаться ее[5]. Это положение нетрудно исправить, предоставив, например, возможность устанавливать цены только некоторым лицам по завершении рабочего дня. Кроме того, в системе можно отложить изменение цен до следующего удобного момента. Если клиентская часть приложения буферизует каким-то образом данные о ценах, она должна знать, как обновить или же очистить эту буферизованную информацию при удобном случае или же в стратегических целях.

В отличие от цен, наличие товаров на складе имеет динамический характер. Следовательно, в качестве реакции на переданный заказ требуется выдать отчет. В этом отчете на всякий случай могут быть также отражены любые обнаруженные расхождения в ценах, если реализованы соответствующие правила. Кроме того, в отчете должно быть явно указано, не превысила ли сумма заказа глобально установленный максимальный предел и не исчерпан ли кредит клиента.

Когда заказ сохраняется, он привязывается к заданной цене, чтобы не "обрадовать" клиента счетом-фактурой с более высокими ценами, чем предложенные ранее. Это правило вынуждает сохранять предложенные цены (свойство `OrderPrice` на рис. А.5) вместе с заказом вместо ссылки на них. С другой стороны, можно вести предысторию изменения цен на товары, хотя это слишком хлопотно. И разумеется, выданные заказы нельзя вновь открывать, чтобы дополнить их товарами в том случае, если, например, после разговора с принимающим заказы до клиента наконец-то дойдет, что ему требуются не две, а три единицы товара. В подобной ситуации заказ должен размещаться без стоимости доставки и/или выписки счета-фактуры[6]. Изменение количества товаров в строках заказа — еще одна несложная задача, хотя в реальной ситуации она может повлечь за собой значительные скидки.

Класс `POrderBuilder`

Поддержка приема и обработки заказов на поставку осуществляется в классе `POrderBuilder`. Этот класс предназначен для использования как в среднем ярусе приложения, так и в клиентской части приложения, и поэтому он передается по каналам связи с преобразованием в последовательную форму. Класс `POrderBuilder`

[5] Если принимающий заказ пропустит два нуля в цене на товар, то кто за это будет отвечать? — *Примеч. авт.*

[6] Это одна из многих задач, которые недостаточно полно сформулированы в данном несложном примере. — *Примеч. авт.*

взаимодействует со складом в нескольких случаях, например, при формировании строки заказа в течение вызова метода `AddItemToOrder()`. Такое взаимодействие фиксируется в абстрактном классе `ItemStore`, состоящем из единственного метода `GetItem()`.

Отчет о расхождениях формируется в классе `POrderBuilder` при вызове метода `CheckDiscrepancies()` — проверка расхождений. Как правило, принимающий заказы пользуется локальным вариантом класса `ItemStore` (подробнее об этом — в следующем подразделе) и разными его реализациями для клиента, сервера и теста, чтобы получить цены и запасы на складе во время разговора с клиентом по телефону. Как только заказ будет составлен, принимающий заказы выдает его в средний ярус приложения, где в определенный момент вызывается метод `CheckDiscrepancies()` в переданном классе `POrderBuilder` для формирования отчета. В этот момент `ItemStore` представляет собой объект, находящийся в среднем ярусе приложения ближе к базе данных.

С точки зрения блочного тестирования весьма привлекательно иметь `CheckDiscrepancies()` в качестве метода класса `POrderBuilder`. Рассмотрим приведенный ниже фрагмент тестового кода. Объект `bldr` класса `POrderBuilder` создается на стадии, помеченной атрибутом `[SetUp]`, где он также заполняется несколькими строками заказа. Объект `itmStore` служит в качестве заглушки класса `ItemStore`. Он также создается и связывается с объектом `bldr` на стадии `[SetUp]`. В первой строке тестового кода имитируется изменение ситуации в классе `ItemStore` по отношению к запасам товаров, когда наступает момент для формирования отчета о расхождениях.

```
[Test]
public void Stock()
{
  itmStore.Items[0].Stock = 1;

  POrderDiscrepancyReport r = bldr.CheckDiscrepancies();
  Assert.AreEqual(0, r.PriceCount);
  Assert.AreEqual(1, r.StockCount);
  Assert.AreEqual(1, r.GetStockAt(0).NumberOfMissingItems);
}
```

Таким образом, отчет о расхождениях создается на ранней стадии обработки выданного заказа перед его фиксацией. Но при этом возникает затруднение, связанное с обработкой транзакций и часто упускаемое из виду или же слишком упрощаемое. Если не проявить бдительность, то проверка может оказаться недостаточно точной, т.е. без расхождений, хотя при фиксации заказа последняя единица товара на складе назначается, например, для другого заказа и клиента. Иными словами, проверка расхождений должна происходить в той же транзакции, что и фиксация заказа, и без блокировок, препятствующих другим пользователям выбирать со склада те товары, которые собирается продать пользователь, принимающий текущий заказ.

Реализация класса `ItemStore` для клиента, сервера и теста

Было бы интересно рассмотреть класс `ItemStore` более подробно. Ранее уже упоминался его локальный вариант для клиентской части приложения, а также транзактный вариант для среднего яруса приложения. Покажем, что же все это означает.

Как следует из рис. А.13, в классе `POrderBuilder` находится ссылка на класс `ItemStore`. Эта ссылка реализована в виде общедоступного свойства, присваиваемого в коде приложения. В клиентской части приложения используется реализация класса `ItemStore`, которая совершенно отличается (и это следует еще раз подчеркнуть) от транзактной реализации, применяемой в среднем ярусе приложения в момент фиксации заказа. Класс `ItemStore` — это не столько интерфейс, реализуемый для целого ряда методов, сколько центральный элемент для класса `POrderBuilder`. В частности, он используется всеми методами, приведенными на рис. А.13.

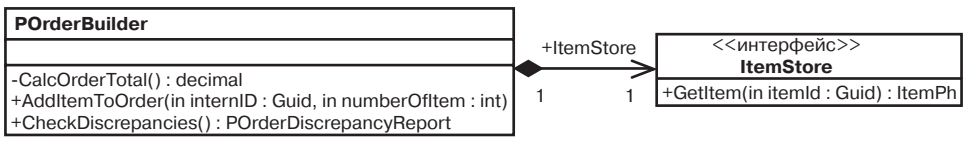


Рис. А.13. Взаимосвязь между классами `POrderBuilder` и `ItemStore`

Вариант класса `ItemStore` для клиента реализован по простому принципу "чтения всех товаров из склада и кэша". Такой принцип вполне пригоден для не очень большого ассортимента товаров. Но при большом ассортименте товаров нетрудно реализовать более логически развитый вариант для среднего яруса приложения. Если у принимающего заказы имеется привилегия работать с заказами в клиентской части приложения, в таком случае создается экземпляр объекта `POrderBuilder`, который затем назначается для реализации класса `ItemStore` на стороне клиента. В любой момент возможна проверка с помощью метода `CheckDiscrepancies()` по отношению к клиентскому варианту `ItemStore`.

Если клиент подтверждает заказ, далее следует выдача заказа в средний ярус приложения. В последовательную форму преобразуется и затем передается по каналам связи экземпляр `POrderBuilder`, а не экземпляр `ItemStore`. Ссылка на класс `ItemStore` обозначается как `[NonSerialized]`. В интерфейсном объекте устанавливается соединение и выполняется транзакция, а затем задание на прием заказа передается объекту процесса (т.е. бизнес-объекту), как показано в следующем фрагменте кода:

```
public POrderDiscrepancyReport TakeOrder(POrderBuilder order)
{
  POrder bo = new POrder();
  CnTxWrap cn =
    CnTxWrap.BeginTransaction(bo.CreateConnectionForMe() );
  try
  {
```

```
    POrderDiscrepancyReport res = bo.TakeOrder(cn, order);
    cn.Commit();
    return res;
  }
  catch
  {
    cn.Rollback();
    throw;
  }
}
```

Раз уж мы коснулись объекта процесса, представим метод `TakeOrder()` этого объекта в следующем фрагменте кода:

```
[PrincipalPermission(SecurityAction.Demand, Role = Roles.User)]
[PrincipalPermission(SecurityAction.Demand, Role =
  @"InternalUnitTesting")]
public POrderDiscrepancyReport TakeOrder(CnTxWrap cn,
  POrderBuilder order)
{
  ItemStore itmStore = new TxItemStore(cn);
  order.ItemStore = itmStore;
  POrderDiscrepancyReport res = order.CheckDiscrepancies();

  POrderPh ordr =
    ((POrderBuilder.IOrderInternal)order.POrder).POrder;
  new POrder().InsertNoRBS(cn, ordr);

  OrderLine olproc = new OrderLine();
  for(int i=0; i<order.OrderLineCount; i++)
  {
    OrderLinePh ol =
    ((POrderBuilder.IOrderLineInternal)order[i]).OrderLine;
    olproc.InsertNoRBS(cn, ol);
  }

  return res;
}
```

Таким образом, `ItemStore` фактически является экземпляром объекта `TxItemStore`, которому передается параметр `cn`, содержащий или охватывающий соединение с базой данных и текущую транзакцию. В такой реализации класса `ItemStore` чтение со склада товаров осуществляется с указанием UPDLOCK (или любым другим предпочтительным способом), в результате чего склад товаров блокируется и не может быть обновлен в данный момент другими пользователями. Корректность такого подхода весьма значительна, хотя он может оказаться критическим элементом с точки зрения производительности.

Метод `CheckDiscrepancies()` формирует исключение при серьезных нарушениях бизнес-правил, например, когда превышается глобально устанавливаемый максимальный предел. Если единственными расхождениями оказываются разница в ценах или нехватка товаров на складе, о них будет упомянуто в отчете, но заказ все равно будет размещен полностью.

Еще раз напомним о перепроверке бизнес-правил как на стороне клиента, так и на стороне сервера. Это делается из тех соображений, что клиентской части приложения никогда нельзя доверять. Такая проверка выполняется в методе `CheckDiscrepancies()`. В частности, любое манипулирование ценами на стороне клиента обычно приводит к расхождениям. В таком случае клиентская часть приложения должна работать в контролируемой среде, но бдительность все равно нельзя терять. Выполнение кода на стороне сервера оказывается намного более безопасным, чем на стороне клиента. Ведь серверу, как правило, неизвестно, что именно вызывает данный код: программа, передающая данные по каналам связи, или какая-то другая программа.

Некоторые заключительные соображения

Большинство разработчиков вряд ли назовут рассмотренный выше подход к разработке приложений объектно-ориентированным. Я не боюсь пользоваться несколько "ослабленными" объектными ссылками по сравнению с теми, что предоставляет язык программирования. Достаточно вспомнить ссылку на товарную позицию в виде идентификатора `Guid` на рис. А.13. Однако такие ссылки не намного яснее, чем "указательные". Сравните данный подход со ссылками на заменители объектов, которые дают возможность получать экземпляры объектов с запозданием или по требованию.

Этим я хотел бы подчеркнуть, что не следует глубоко вдаваться в правильность или неправильность применяемых методов проектирования. На самом деле абсолютно правильных или неправильных методов не существует (с этим, я уверен, Джимми согласится). Нужно лишь правильно наметить цели проектирования (в частности, сформулировать подходящие бизнес-правила) и просто делать свое дело, разрабатывая программное обеспечение, которое требуется в конкретной сфере деятельности.

Резюме

Итак, мне как автору этой книги остается лишь подвести итоги видам моделей и методам проектирования, описанным выше Матсом, Франком и Ингмаром. Какой из них лучше? Трудно сказать, поскольку жизнь полна компромиссов! Вместо того чтобы пускаться в очередную полемику относительно сильных и слабых сторон рассмотренных выше моделей и методов проектирования, мне бы хотелось дать следующий полезный совет: принимайте решения самостоятельно в зависимости от конкретной ситуации.

Уверен, что вы поймете меня правильно. Я попросил моих коллег поделиться своими соображениями на данную тему, рассмотреть ее под другими углами зрения, чтобы выяснить, какой подход лучше и каковы достоинства других видов моделей.

Итак, подведем краткие итоги всего сказанного выше. В частности, Матс отдает предпочтение объектно-ориентированной модели данных, уровню обслуживания с развитой логикой и обмену документами между отдельными частями приложения. Ему больше по душе размещать поведение на уровне обслуживания и рассматривать модель предметной области в качестве объектно-ориентированного представления данных. А документы служат для обмена данными между разными ярусами приложения.

Франк подчеркивает значение потенциальных возможностей базы данных, а также то, что представление модели в базе данных является единственно правильным.

Ингмар придает значение поиску подходящих абстракций, а также тому, что не следует рассчитывать на единый на все случаи жизни стандартный метод проектирования. К этому следует добавить изрядную долю прагматизма.

Приложение Б

Перечень рассмотренных шаблонов

В этом приложении представлен перечень шаблонов, рассмотренных в данной книге. Он послужит читателю не источником новой информации, а, скорее, кратким справочником по шаблонам. Каждый перечисляемый шаблон снабжен очень кратким описанием его назначения и ссылкой на источник, где чаще всего можно найти дополнительные сведения о данном шаблоне.

На заметку

Гипертекстовый вариант этого приложения, позволяющий сразу же перейти по указанной ссылке к источнику информации о шаблоне, находится по адресу: `www.jnsk.se/adddp/patterns`.

Если описание шаблона приведено в кавычках, это означает, что оно взято непосредственно из источника по указанной ссылке (или же из литературы, список которой приведен в конце данной книги, например, [GoF Design Patterns]).

Abstract Factory (Абстрактная фабрика) [GoF Design Patterns]

"Предоставляет интерфейс для создания семейств зависимых объектов без указания конкретных классов".
`http://patternshare.org/default.aspx/Home.GOF.AbstractFactory`

Aggregate (Агрегат) [Evans DDD]

Группирует объекты-сущности и объекты-значения в определенных границах. Позволяет стать объектам-сущностям точкой доступа в агрегате (т.е. корнем агрегата).
`http://patternshare.org/default.aspx/Home.DDD.Aggregates`

Bounded Context (Ограниченный контекст) [Evans DDD]

Определяет границы вокруг отдельной модели.
`http://patternshare.org/default.aspx/Home.DDD.BoundedContext`

Chain of Responsibility (Цепочка ответственности) [GoF Design Patterns]

"Исключает связывание отправителя запроса с его получателем, предоставляя нескольким объектам возможность обрабатывать запрос. Выстраивает получающие объекты в цепочку и передает запрос по цепочке до тех пор, пока какой-нибудь из этих объектов не обработает его".

Class Table Inheritance (Наследование с таблицами для каждого класса) [Fowler PoEAA]

"Представляет иерархию наследования классов в виде одной таблицы для каждого класса".
`http://www.martinfowler.com/eaaCatalog/classTableInheritance.html`

Coarse-Grained Lock (Крупноструктурная блокировка) [Fowler PoEAA]

"Блокирует ряд связанных друг с другом объектов в течение одной блокировки".
`http://www.martinfowler.com/eaaCatalog/coarseGrainedLock.html`

Collecting Parameter Pattern (Шаблон накапливающего параметра) [Beck SBPP]

Объект, передающий методы для накопления информации из этих методов.

Concrete Table Inheritance (Наследование с таблицами для каждого конкретного класса) [Fowler PoEAA]

"Представляет иерархию наследования классов в виде одной таблицы для каждого конкретного класса в данной иерархии".
`http://www.martinfowler.com/eaaCatalog/concreteTableInheritance.html`

Data Mapper (Преобразователь данных) [Fowler PoEAA]

"Слой преобразователей, перемещающих данные между объектами и базой данных, а при этом сохраняется их независимость друг от друга, а также от самого преобразователя".
`http://www.martinfowler.com/eaaCatalog/dataTransferObject.html`

Data Transfer Objects (Объекты для передачи данных) [Fowler PoEAA]

"Объект, переносящий данные из одного процесса в другой с целью сократить число вызовов методов".
`http://www.martinfowler.com/eaaCatalog/dataTransferObject.html`

Decorator (Декоратор) [GoF Design Patterns]

"Динамически наделяет объект дополнительными обязанностями. Декораторы представляют собой удобную альтернативу подклассификации для расширения функциональных возможностей".
`http://patternshare.org/default.aspx/Home.GOF.Decorator`

Dependency Injection (Внесение зависимостей)

Вместо поиска экземпляром объекта его собственных зависимостей последние вносятся в сам экземпляр.

Domain Model (Модель предметной области) [Fowler PoEAA]

"Объектная модель предметной области, содержащая данные и поведение".
http://www.martinfowler.com/eaaCatalog/domainModel.html

Embedded Value (Внедренное значение) [Fowler PoEAA]

"Преобразует объект в несколько полей таблицы другого объекта".
http://www.martinfowler.com/eaaCatalog/embeddedValue.html

Entity (Объект-сущность) [Evans DDD]

"Многие объекты, по существу, определяются не по их атрибутам, а по связующей линии непрерывности и идентичности".
http://patternshare.org/default.aspx/Home.DDD.Entities

Factory (Фабрика) [Evans DDD]

"Если создание объекта или целого агрегата усложняется или же чрезмерно обнаруживает внутреннюю структуру, то необходимую инкапсуляцию обеспечивают фабрики".
http://patternshare.org/default.aspx/Home.DDD.Factories

Factory Method (Фабричный метод) [GoF Design Patterns]

"Определяет интерфейс для создания объекта, но позволяет получать в подклассах экземпляры конкретных классов. Фабричный метод дает возможность перенести получение экземпляров из класса в подклассы".

Foreign Key Mapping (Преобразование внешнего ключа) [Fowler PoEAA]

"Преобразует связь между объектами в ссылку на внешний ключ между таблицами".
http://www.martinfowler.com/eaaCatalog/foreignKeyMapping.html

Generation Gap (Разрыв в генерации кода) [Vlissides Pattern Hatching]

"Видоизменяет или расширяет сгенерированный код только один раз, независимо от того, сколько раз он регенерируется".

Identity Field (Поле идентификации) [Fowler PoEAA]

"Сохраняет содержимое поля идентификации из базы данных в объекте для поддержания тождественности объекта в оперативной памяти и строки в базе данных".
http://www.martinfowler.com/eaaCatalog/identityMap.html

Identity Map (Карта идентичности) [Fowler PoEAA]

"Обеспечивает загрузку объектов только один раз, сохраняя в карте каждый загруженный объект. Осуществляет поиск объектов, используя карту для обращения к ним".
http://www.martinfowler.com/eaaCatalog/implicitLock.html

Implicit Lock (Неявная блокировка) [Fowler PoEAA]

"Позволяет коду базовой структуры или супертипа слоя запрашивать автономную блокировку".
http://www.martinfowler.com/eaaCatalog/implicitLock.html

Layer Supertype (Супертип слоя) [Fowler PoEAA]

"Тип, выполняющий роль супертипа для всех остальных типов в данном слое".
`http://www.martinfowler.com/eaaCatalog/layerSupertype.html`

Layers (Слои) [POSA]

(См. также [Fowler PoEAA].)
"Структурирует приложения, которые могут быть разделены на группы задач, каждая из которых находится на отдельном уровне абстракции".
`http://patternshare.org/default.aspx/Home.POSA.Layers`

Lazy Load (Загрузка по требованию) [Fowler PoEAA]

"Объект, у которого отсутствуют все необходимые данные, но которому известно, как их получить".
`http://www.martinfowler.com/eaaCatalog/lazyLoad.html`

Metadata Mapping (Преобразование метаданных) [Fowler PoEAA]

"Содержит подробности объектно-реляционного преобразования в метаданные".
`http://www.martinfowler.com/eaaCatalog/metadataMapping.html`

Model-View-Controller (Модель–Представление–Контроллер) [Fowler PoEAA]

"Разделяет взаимодействие с пользовательским интерфейсом на три отдельные функции".
`http://www.martinfowler.com/eaaCatalog/modelViewController.html`

Model-View-Presenter (Модель–Представление–Презентатор) [Fowler PoEAA2]

"Отделяет поведение презентации от представления и в то же время позволяет получать в представлении события от пользователя".
`http://www.martinfowler.com/eaaDev/ModelViewPresenter.html`

Notification (Уведомление) [Fowler PoEAA2]

"Объект, накапливающий информацию об ошибках и прочие данные в слое предметной области и передающий эти данные представлению".
`http://www.martinfowler.com/eaaDev/Notification.html`

Null Object (Пустой объект) [Woolf Null Object]

Объект, предоставляющий используемые по умолчанию данные и поведение, если происходит пустая ссылка.

Optimistic Offline Lock (Оптимистическая автономная блокировка) [Fowler PoEAA]

"Предотвращает конфликты между параллельно выполняемыми транзакциями, обнаруживая конфликт и осуществляя откат транзакции".
`http://www.martinfowler.com/eaaCatalog/optimisticOfflineLock.html`

Party Archetype (Групповой архетип) [Arlow/Neustadt Archetype Patterns]

Способ представления информации о людях и организациях.

Pessimistic Offline Lock (Пессимистическая автономная блокировка) [Fowler PoEAA]

"Предотвращает конфликты между параллельно выполняемыми транзакциями, предоставляя доступ к данным каждой транзакции по очереди".
`http://www.martinfowler.com/eaaCatalog/pessimisticOfflineLock.html`

Pipes and Filters (Каналы и фильтры) [POSA]

Направляет данные по каналам и обрабатывает потоки данных с помощью фильтров.

Presentation Model (Модель представления) [Fowler PoEAA2]

"Обозначает состояние и поведение представления независимо от элементов графического пользовательского интерфейса, используемых в интерфейсе".
`http://www.martinfowler.com/eaaDev/PresentationModel.html`

Proxy (Заменитель) [GoF Design Patterns]

"Предоставляет заполнитель или заменитель другого объекта для управления доступом к нему".
`http://patternshare.org/default.aspx/Home.GOF.Proxy`

Query Object (Объект запроса) [Fowler PoEAA]

"Объект, обозначающий запрос базы данных".
`http://www.martinfowler.com/eaaCatalog/recordSet.html`

Recordset (Набор записей) [Fowler PoEAA]

"Представление табличных данных в оперативной памяти".
`http://www.martinfowler.com/eaaCatalog/recordSet.html`

Reflection (Рефлексия) [POSA]

Позволяет программно проверять тип экземпляра объекта, используя метаданные, и взаимодействовать с этим экземпляром, не зная заранее его тип.

Registry (Реестр) [Fowler PoEAA]

"Известный объект, который может использоваться другими объектами для поиска общих объектов и служб".
`http://www.martinfowler.com/eaaCatalog/registry.html`

Remote Façade (Интерфейс удаленного доступа) [Fowler PoEAA]

"Предоставляет крупноструктурный интерфейс для мелкоструктурных объектов с целью повысить эффективность работы в сети".
`http://www.martinfowler.com/eaaCatalog/remoteFacade.html`

Repository (Хранилище) [Evans DDD]

(См. также [Fowler PoEAA].)
Объект для размещения определенной сущности (или ряда сущностей) посредине их жизненного цикла.
http://patternshare.org/default.aspx/Home.DDD.Repositories

Separated Presentation (Раздельное представление) [Fowler PoEAA2]

"Гарантирует, что любой код для манипулирования представлением манипулирует только представлением, выделяя все данные предметной области и логику доступа к источнику данных в совершенно раздельные области программы".
http://www.martinfowler.com/eaaDev/SeparatedPresentation.html

Service Layer (Уровень обслуживания) [Fowler PoEAA]

"Устанавливает границы приложения на уровне обслуживания, где определяется ряд доступных операций и координируется реакция приложения на каждую операцию".
http://www.martinfowler.com/eaaCatalog/serviceLayer.html

Service Locator (Указатель службы) [Alur/Crupi/Malks Core J2EE Patterns]

"Служит для реализации и инкапсуляции службы, а также поиска ее компонентов. Скрывает подробности реализации механизма поиска и инкапсулирует связанные с этим зависимости".
http://www.corej2eepatterns.com/Patterns2ndEd/ServiceLocator.htm

Services (Службы) [Evans DDD]

Обязанности, которыми обычно не наделяются объекты-сущности или объекты-значения, могут быть переданы службам.
http://patternshare.org/default.aspx/Home.DDD.Services

Single Table Inheritance (Наследование с одной таблицей) [Fowler PoEAA]

"Представляет иерархию наследования классов в виде одной таблицы со столбцами для всех полей разных классов".
http://www.martinfowler.com/eaaCatalog/singleTableInheritance.html

Singleton (Одиночка) [GoF Design Patterns]

"Одиночка обеспечивает лишь единственное наследование для класса и глобальную точку доступа к нему".
http://patternshare.org/default.aspx/Home.GOF.Singleton

Specification (Описание) [Evans DDD]

(См. также [Fowler Analysis Patterns].)
Фиксирует и раскрывает предикаты с помощью имен, описывающих концепции.

State (Состояние) [GoF Design Patterns]

"Позволяет объекту изменять свое поведение при изменении его внутреннего состояния. При этом объект может также изменить свой класс".

Table Module (Модуль таблицы) [Fowler PoEAA]

"Единственный экземпляр объекта, предоставляющий бизнес-логику для обработки всех строк в таблице базы данных или в представлении".
http://www.martinfowler.com/eaaCatalog/tableModule.html

Template Method (Шаблонный метод) [GoF Design Patterns]

"Определяет структуру алгоритма в отдельной операции, передавая некоторые этапы ее выполнения подклассам. Позволяет подклассам переопределить некоторые этапы выполнения операции в таком алгоритме, не меняя его структуру".
http://patternshare.org/default.aspx/Home.GOF.TemplateMethod

Transaction Script (Сценарий транзакции) [Fowler PoEAA]

"Организует бизнес-логику по процедурам, каждая из которых обрабатывает один запрос из представления".
http://www.martinfowler.com/eaaCatalog/transactionScript.html

Unit of Work (Единица работы) [Fowler PoEAA]

"Сохраняет список объектов, задействованных в транзакции, а также координирует регистрацию изменений и разрешение конфликтов параллельной обработки".
http://www.martinfowler.com/eaaCatalog/unitOfWork.html

Value Object (Объект-значение) [Evans DDD]

(См. также [Fowler PoEAA].)
"У многих объектов отсутствует концептуальная идентичность. Такие объекты описывают лишь некоторые свойства предметов".
http://patternshare.org/default.aspx/Home.DDD.ValueObjects

Список литературы

[Alexander Pattern Language]
Alexander, Christopher, Sara Ishikawa, and Murray Silverstein with Max Jacobson, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press, 1977.

[Alur/Crupi/Malks Core J2EE Patterns]
Alur, Deepak, John Crupi, and Dan Malks. *Core J2EE Patterns, Second Edition: Best Practices and Design Strategies*. New Jersey: Prentice Hall, 2003.

[Arlow/Neustadt Archetype Patterns]
Arlow, Jim, and Ila Neustadt. *Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML*. Boston, MA: Addison-Wesley, 2004.

[AspectJ]
`http://eclipse.org/aspectj/`

[AspectSharp]
`http://www.castleproject.org/index.php/AspectSharp`

[Astels TDD]
Astels, David. *Test Driven Development. A Practical Guide*. New Jersey: Prentice Hall, 2004.

[Bauer/King HiA]
Bauer, Christian, and Gavin King. *Hibernate in Action*. Greenwich, CT: Manning, 2005.

[Beck SBPP]
Beck, Kent. *Smalltalk Best Practice Patterns*. New Jersey: Prentice Hall, 1997.

[Beck TDD]
Beck, Kent. *Test-Driven Development: By Example*. Boston, MA: Addison-Wesley, 2003.

[Beck XP]
Beck, Kent. *Extreme Programming Explained: Embrace Change*. Boston, MA: Addison-Wesley, 2000.

[Bloch Effective Java]
Bloch, Joshua. *Effective Java Programming Language Guide*. Boston, MA: Addison-Wesley, 2001.

[Booch OOAD]
Booch, Grady. *Object-Oriented Analysis and Design with Applications, Second Edition*. Reading, MA: Addison-Wesley, 1994.

[Bosch Product Line]
Bosch, Jan. *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Boston, MA: Addison-Wesley, 2000.

[POSA]
Buschmann, Frank, Regine Meunier, Hans Rohnert, and Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture. A System of Patterns*. New York: Wiley, 1996.

[Castle]
`http://www.castleproject.org/`

[Cattell ODM]
Cattell, R.G.G. *Object Data Management: Object-Oriented and Extended Relational Database Systems*. Reading, MA: Addison-Wesley, 1994.

[CC.NET]
`http://confluence.public.thoughtworks.org/display/CCNET`

[Chen ER]
Chen, P. "The entity-relationship model—toward a unified view of data." *ACM Transactions on database systems*, vol.1 no.1, 1976.

[Cockburn Agile]
Cockburn, Alistair. *Agile Software Development*. Boston, MA: Addison-Wesley, 2002.

[Codd Relational Model]
Codd, E. F. "A Relational Model of Data for Large Shared Data Banks." *Communications of the ACM*, vol. 13 #6, 1970.

[Connolly/Begg DB Systems]
Connolly, Thomas M. and Carolyn E. Begg. *Database Systems, Fourth Edition: A Practical Approach to Design, Implementation, and Management*. Boston, MA: Addison-Wesley, 2004.

[Demeyer/Ducasse/Nierstrasz OORP]
Demeyer, Serge, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. San Francisco: Morgan Kaufmann, 2002

[Enterprise Library Logging]
`http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag2/html/entlib.asp`

[Evans DDD]
Evans, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston, MA: Addison-Wesley, 2004.

[Ewald TxCOM+]
Ewald, Tim. *Transactional COM+: Building Scalable Applications*. Boston, MA: Addison-Wesley, 2001.

[Feathers Humble Dialog Box]
Feathers, Michael. "The Humble Dialog Box."
http://www.objectmentor.com/resources/articles/TheHumbleDialogBox.pdf

[Feathers Self-Shunt]
Feathers, Michael. "The 'Self'-Shunt Unit Testing Pattern."
http://www.objectmentor.com/resources/articles/SelfShunPtrn.pdf

[Fowler Analysis Patterns]
Fowler, Martin. *Analysis Patterns: Reusable Object Models*. Reading, MA: Addison-Wesley, 1997.

[Fowler FixedLengthString]
Fowler, Martin. http://martinfowler.com/bliki/FixedLengthString.html

[Fowler FluentInterface]
http://martinfowler.com/bliki/FluentInterface.html

[Fowler HarvestedFramework]
Fowler, Martin. http://www.martinfowler.com/bliki/HarvestedFramework.html

[Fowler InversionOfControl]
Fowler, Martin. http://www.martinfowler.com/bliki/InversionOfControl.html

[Fowler LW]
Fowler, Martin. "Language Workbenches: The Killer-App for Domain Specific Languages?"
http://www.martinfowler.com/articles/languageWorkbench.html

[Fowler Mocks Aren't Stubs]
Fowler, Martin. "Mocks Aren't Stubs."
http://martinfowler.com/articles/mocksArentStubs.html

[Fowler PoEAA]
Fowler, Martin. *Patterns of Enterprise Application Architecture*. Boston, MA: Addison-Wesley, 2003. (Мартин Фаулер, *Архитектура корпоративных программных приложений,* пер. с англ., ИД "Вильямс", 2004 г.)

[Fowler PoEAA2]
Fowler, Martin. http://martinfowler.com/eaaDev/

[Fowler R]
Fowler, Martin. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley, 1999.

[Fowler Snapshot]
http://www.martinfowler.com/ap2/snapshot.html

[Fowler UML Distilled]
Fowler, Martin. *UML Distilled, Third Edition: A Brief Guide to the Standard Object Modeling Language*. Boston, MA: Addison-Wesley, 2004.

[Fowler/Foemmel CI]
Fowler, Martin, and Foemmel, Matt.
`http://www.martinfowler.com/articles/continuousIntegration.html`

[GoF Design Patterns]
Gamma, Erich, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.

[Greenfield/Short SF]
Greenfield, Jack, and Keith Short. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Indianapolis, IN: Wiley, 2004.

[Gregor Kiczales]
`http://www.cs.ubc.ca/~gregor/`

[Halpin IMRD]
Halpin, Terry. *Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design*. San Francisco: Morgan Kaufmann, 2001.

[Halpin/Nijssen Conceptual Schema]
Halpin, T.A. and G.M. Nijssen. *Conceptual Schema and Relational Database Design: A Fact Oriented Approach*. New Jersey: Prentice Hall, 1989.

[Hay Data Model Patterns]
Hay, David. *Data Model Patterns: Conventions of thought*. NewYork: Dorset House, 1996.

[Hibernate]
`http://hibernate.org`

[Hohpe/Woolf EIP]
Hohpe, Gregor and Bobby Woolf. *Enterprise Integration Patterns. Designing, Building, and Deploying Messaging Solutions*. Boston, MA: Addison-Wesley, 2004.

[iBATIS]
`http://ibatis.apache.org/`

[Jacobson OOSE]
Jacobson, Ivar, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, MA: Addison-Wesley, 1992.

[Johnson J2EE Development without EJB]
Johnson, Rod with Juergen Hoeller. *Expert one-on-one J2EE Development without EJB*. Indianapolis, IN: Wiley, 2004.

[Jordan/Russell JDO]
Jordan, David, and Craig Russel. *Java Data Objects*. Sebastopol, CA: O'Reilly, 2003.

[Kerievsky R2P]
Kerievsky, Joshua. *Refactoring to Patterns*. Boston, MA: Addison-Wesley, 2005. (Джошуа Кериевски, *Рефакторинг с использованием шаблонов*, пер. с англ., ИД "Вильямс", 2006 г.)

[Lhotka BO]
Lhotka, Rocky. *Expert C# Business Objects*. Berkeley, CA: APress, 2004.

[Log4Net]
`http://logging.apache.org/log4net/`

[Loom.NET]
`http://www.dcl.hpi.uni-potsdam.de/research/loom/`

[Löwy Programming .NET Components]
Löwy, Juval. *Programming .NET Components*. Second Edition. Cambridge, MA: O'Reilly, 2005.

[Martin PPP]
Martin, Robert C. *Agile Software Development: Principles, Patterns, and Practices*. New Jersey: Prentice Hall, 2002. (Роберт Мартин, *Быстрая разработка программ: принципы, примеры, практика*, пер. с англ., ИД "Вильямс", 2003 г.)

[Meszaros XUnit]
Meszaros, Gerard. `http://tap.testautomationpatterns.com:8080/index.html`

[Meyer OOSC]
Meyer, Bertrand. *Object-Oriented Software Construction, Second Edition*. New Jersey: Prentice Hall, 2000.

[MockObjects]
`http://sourceforge.net/projects/dotnetmock`

[MVC History]
`http://c2.com/cgi/wiki?ModelViewControllerHistory`

[NanoContainer]
`http://www.nanocontainer.org/`

[NHibernate]
`http://nhibernate.sourceforge.net`

[Nicola et al. SOM]
Nicola, Jill, Mark Mayfield, and Mike Abney. *Streamlined Object Modeling: Patterns, Rules, and Implementation*. New Jersey: Prentice Hall, 2002.

[Nilsson COMB]
Nilsson, Jimmy. *The Cost of GUIDs as Primary Keys*.
http://www.informit.com/articles/article.asp?p=25862

[Nilsson NED]
Nilsson, Jimmy. *.NET Enterprise Design with Visual Basic .NET and SQL Server 2000*. Indianapolis, IN: Sams Publishing, 2001.

[Nilsson NWorkspace]
Nilsson, Jimmy. http://www.jnsk.se/nworkspace

[NMock]
http://www.nmock.org

[NUnit]
http://www.nunit.org

[OMG MDA]
http://www.omg.org/mda/

[Pawson/Matthews Naked Objects]
Pawson, Richard, and Robert Matthews. *Naked Objects*. New York: Wiley, 2003.

[Pfister Wolfpack]
Gregory F. Pfister. *In Search of Clusters*. Upper Saddle River, NJ: Prentice Hall, 1998.

[PicoContainer]
http://www.picocontainer.org/

[POCMock]
http://www.prettyobjects.com/pocmock.aspx

[Prevayler]
http://www.prevayler.org/wiki.jsp

[Refactor!]
http://www.devexpress.com

[ReSharper]
http://www.jetbrains.com

[Richter .NET Framework]
Richter, Jeffrey. *Applied Microsoft .NET Framework Programming*. Redmond, Wash: Microsoft Press, 2002.

[Ross BRB]
Ross, Ronald. *The Business Rules Book: Classifying, Defining and Modeling Rules. Second edition*. Houston, TX: Business Rule Solutions, 1997.

[Rumbaugh OMT]
Rumbaugh, James R., Michael R. Blaha, William Lorensen, Frederick Eddy, and William Premerlani. *Object-Oriented Modeling and Design*. New Jersey: Prentice Hall, 1990.

[SnapDAL]
http://sourceforge.net/projects/snapdal

[Spolsky Leaky Abstractions]
Spolsky, Joel. *The Law of Leaky Abstractions*.
http://www.joelonsoftware.com/articles/LeakyAbstractions.html

[Spring]
http://www.springframework.org/

[Spring.NET]
http://www.springframework.net/

[Szyperski Component Software]
Szyperski, Clemens. *Component Software, Second Edition*. Boston, MA: Addison-Wesley, 2003.

[Testdriven.net]
http://www.testdriven.net

[Valhalla]
http://www.jnsk.se/weblog/posts/part7.htm

[Vlissides Pattern Hatching]
Vlissides, John M. *Pattern Hatching: Design Patterns Applied*. Reading, MA: Addison-Wesley, 1998.

[von Halle BRA]
von Halle, Barbara. *Business Rules Applied: Building Better Systems Using the Business Rules Approach*. New York: Wiley, 2001.

[Windsor]
http://www.castleproject.org/index.php/Container

[Woolf Null Object]
Woolf, Bobby. "The Null Object Pattern," in *Pattern Languages of Program Design III*. By Robert C. Martin, Dirk Riehle, and Frank Buschmann (Eds.). Reading, MA: Addison-Wesley, 1998.

[XtUnit]
http://weblogs.asp.net/rosherove/archive/2004/10/05/238201.aspx

Предметный указатель

А

Абстрактная модель NIAM/ORM, *511*
Автоматические блочные тесты, *74*
Автоматическое блочное тестирование, *73*
Автосвязывание
  как свойство, *423*
  типизированное, *424*
Агрегаты, *240*; *267*; *287*; *295*
Адаптеры, *459*
Администраторы ресурсов, *329*
Анонимные методы, *316*
Архитектура
  используемая по умолчанию, новая, *151*
  клиент-сервер и АОС, *401*
  ориентированная на службы, АОС
    и объектная ориентация, *401*
    назначение, *395*
    особенности, *397*
    руководящие принципы, *399*
  основанная на модели, *70*
  приложений, построение, *515*
  семейства программных продуктов, *69*
Асинхронные сообщения, *66*
Аспектно-ориентированное
  программирование, АОП
  перенос бизнес-правил в аспекты, *443*
  предпосылки, *429*
  применение, *434*
  разделение на аспекты, *432*
  реализации, *429*
  терминология, *433*

Б

Библиотеки доступа к данным, *259*
Бизнес-объекты, *515*
Быстрая разработка приложений, *449*

В

Визуальная модель, *42*
Внесение зависимостей
  в Spring.NET, *420*
  в конструктор, *415*
  в механизм установки, *418*
  как механизм, *135*
Внешний интерфейс API
  варианты представления, *338*
  расширение, *296*
  создание, *274*
  уточнение реализации, *297*
Встроенная в шаблон MVC модель, *451*

Г

Генерация заказного кода, *331*
Гибридные базы данных, *327*
Голые объекты, *50*
Графы загрузки, *267*

Д

Динамическая подклассификация, *226*
Домен, *61*

Ж

Жизненный цикл экземпляра объекта, *214*

З

Заглушки
  назначение, *123*
  определение, *202*
  тестовые, *130*; *203*
Загрузка по требованию, *349*; *383*
Заключение объектов в оболочку, *475*
Заполнители бизнес-объектов, *517*

Запросы
отчетов, *373*
скалярные, *375*

И

Идентификаторы
Guid, *346*
комбинированные, COMB, *378*
Идентичность
виды, *345*
отображение, *482*
Имитации
динамически создаваемые, *258*
методом самошунтирования, *465*
назначение, *123*
определение, *202*
предположения, *257*
средствами NMock, *472*
Имитирующие объекты, *133*; *203*
Инварианты, *278*
Инверсия управления, *135*; *410*
Инструменты
BODef, *515*
NUnit, *72*; *116*; *117*
Refactor!, *143*
ReSharper, *143*
Testdriven.net, *116*
рефакторинга кода, *143*
тестирования кода, *116*
Интеграция
непрерывная, *79*
пошаговая, *79*
причины затруднений, *79*
ручная, *78*
Интерфейс, пользовательский
абстракция, *462*
блочное тестирование, *461*
код, *472*
место в приложении, *450*
преобразование, *474*
элементы и адаптеры, *459*
Инфраструктура
место для хранения данных, *324*
назначение, *321*
требования для сохраняемости, *322*; *330*; *366*

К

Карты идентичности, *241*; *348*; *383*
Категории бизнес-логики, *512*
Контейнеры
вложенные, *425*
запросов, *426*
инверсии управления, *420*
сеансовые, *425*
Контроллер
в шаблоне MVC, *452*
развязка от представления, *459*
сочетание с представлением, *460*
Концептуальные описания в качестве запросов, *269*
Крупное проектирование наперед, КПН, *76*

Л

Логическая единица работы, *240*; *349*; *383*

М

Метаданные
преобразование, *333*
типы, *343*
Метод
NIAM, *511*
ORM, *511*
аспектно-ориентированного программирования, АОП, *429*
заключения в оболочку, *477*
моделирования объектов, MMO, *42*
неведения сохраняемости, НС, *223*
обратного вызова, *461*
прецедентов использования, *43*
разработки посредством тестирования, РПТ, *115*
разрешения зависимостей, *410*
рефакторинга кода, *137*
структурно-рефлексивный, *337*
фабричный, *110*

Механизмы эксплуатации, *81*; *82*
Модель предметной области
заключение в оболочку модели представления, *476*
и РПТ, *78*
иерархии наследование, *347*
история применения, *49*
как корневая структура, *52*
особенности применения, *152*
преобразование из модели представления, *477*
привязка к пользовательскому интерфейсу, *173*
реализация, *178*
связывание с уровнем представления, *474*
специальная настройка, *310*
уровень исполнения, *177*
уточнение, *181*; *188*
шаблон, *44*; *52*; *68*
Модуль таблицы, *44*

Н

Набор объектов
как модель представления, *475*
преобразование, *475*
Наведение мостов, *31*
Написание заказного кода вручную, *330*
Направление POJO и POCO, *222*
Неведение сохраняемости, НС
во время выполнения или компиляции, *228*
дополнительные издержки, *229*
назначение, *223*
применение для хранилищ, *231*
соблюдение, *228*
требования, *223*
Недоброкачественный код, *137*
Неявная загрузка по требованию, *226*

О

Обмен сообщениями, *65*
Обобщения, *315*
Обозначение
абстрактных классов, *465*
интерфейсов, *465*
Объектно-объектные преобразователи, *478*
Объектно-ориентированная модель, *59*
Объектно-ориентированный язык запросов, OQL, *263*
Объектно-реляционное преобразование
виды, *336*
как гимнастика Тайцзи, *333*
назначение, *508*
определение, *57*
Объектно-реляционные преобразователи
другие функциональные возможности, *341*
классификация
исходная, *335*
по шаблонам инфраструктуры, *342*
недостатки, *333*
определенение, *333*
отправные точки, *337*
поддержка
базы данных, *340*
иерархий наследования, *347*
модели предметной области, *336*
формирование запросов, *339*
Объектные базы данных, *327*
Объекты-значения, *200*
Объекты-сущности
модель сущность-связь, *506*
описание в полях идентификации, *344*
определение, *506*
отслеживание, *200*
постоянные, жизненный цикл, *322*; *367*
связь в реляционной модели, *510*
состояния, *508*
экземпляры, *507*
Объекты процесса, *518*
Обыкновенные утверждения, *191*
Ограниченные контексты, *394*
Одиночки, *147*
Односторонний асинхронный обмен сообщениями, *403*
Операторы защиты, *142*
Оптимистический параллелизм, *502*

Отношения между объектами, *480*
Отслеживание изменения состояния объектов, *231*

П

Пакетная обработка, *66*
Параметризированные конструкторы, *110*
Переоценка ценностей, *41*
Перечень задач и возможностей приложения, *156*
Плавный интерфейс, *219*
Платформно-зависимая модель, *70*
Платформно-независимая модель, *70*
Плоские объекты, *483*
Подготовка к инфраструктуре, *222*
Подделки
  воздействие на блочные тесты, *247*
  механизм, *243*
  определение, *202*
  реализация, *246*
Поддельные объекты, *203*
Поля
  закрытые, *228*
  идентификации, *344*
  общедоступные, *121*
  специальные, *227*
  только для чтения, *227*
Понятие модели, *45*
Постоянное хранение данных, *508*
Потеря соответствия, *58*
Правила
  агрегирование, *295*
  внедряемые, *513*
  возможность внесения, *297*
  категории, *271*
  классы, *303*
  нарушенные, список, *282*; *307*
  обслуживания, *289*
  переходы из одних состояний в другие, *275*
  применение в пользовательском интерфейсе, *296*
  принципы применения, *272*
  проверка в реальном масштабе времени, *314*
  проверки достоверности, *271*; *290*
  сложные, *311*
  соблюдение
    в базе данных, *276*
    в контексте, *275*
    в модели предметной области, *278*; *294*
  сохраняемости, *281*; *310*
Предварительное проектирование архитектуры, *68*
Предметная область
  вопросы, решаемые логикой, *407*
  изучение, *46*
  универсальный язык, *47*
Представление
  в шаблоне MVC, *451*
  развязка от контроллера, *459*
  сложное и простое, *460*
  сочетание с контроллером, *460*
  тестирование, *460*
Преобразователь объектно-реляционный NHibernate
  в контексте ППО, *385*
  введение, *354*
  классификация
    исходная, *371*
    по шаблонам инфраструктуры, *377*
  контроль транзакций, *365*
  метаданные для преобразования, *356*
  операции интерфейса API, *362*
  перехватчики проверки достоверности, *384*
  подготовка к применению, *355*
  причины для выбора, *353*
  сборки, *385*
  хранилища, *387*
  язык запросов HQL, *364*
Прецеденты использования, *43*
Принципы
  KISS, *41*
  YAGNI, *41*; *82*

атомарности, непротиворечивости, изоляции и долговечности, АНИД, *325*
единственной ответственности, ПЕО, *87*
инверсии управления, *419*
объектно-ориентированного проектирования, *87*
объектно-ориентированные, *42*
открытости-закрытости, ПОЗ, *87*
подстановки Лискова, ППЛ, *87*
Проверка изменения состояния объектов, *225*
Проектирование на основе предметной области, ППО
в сочетании с РПТ, *160*
главный принцип, *152*
назначение, *45*
прикладное, *71*
разделение на слои, *154*
Прозрачность местоположения, *66*
Процессы
водопадные, *67*
составляющие, *67*
функциональные, *513*
экстремального программирования, *67*

Р

Разбиение на разделы, *393*
Разделение на слои, *99*; *154*; *392*
Размещение экземпляров запросов, *266*
Разработка
на основе модели, *70*
посредством тестирования, РПТ
недостатки, *126*
основания для применения, *73*
основная формула, *116*
порядок выполнения, *75*; *116*
следствия для проектирования, *123*
приложений в стиле ППО, *486*
Раннее архитектурное решение, *69*
Распределение
достоинства, *64*
недостатки, *63*
Регистрация, *244*
Регулярная чистка кода, *138*
Реляционная модель, *59*
Реляционные базы данных, *328*
Рефакторинг кода
встраивание одиночки, *147*
замена вложенного условного выражения оператором защиты, *121*
инкапсуляция поля, *140*
инкапсуляция совокупности объектов, *140*
как термин, *136*
метод извлечения, *122*; *142*
назначение, *75*; *136*
объединение условного выражения, *122*
основания для применения, *76*
формирование шаблонного метода, *146*
Рефлексия
при проверке правил, *305*
при чтении и записи в поля, *229*
применение к частным полям, *195*

С

Самошунтирование, *461*
Свойства
общедоступные, *121*
ссылки, *483*
Система
интеллектуального управления, *123*
непрерывной интеграции, *80*
Слои
и разделы, *392*
карт идентичности, *246*
назначение, *99*
Службы
взаимодействие, *404*
масштабирование, *408*
назначение, *397*
представления, *451*; *474*; *486*
принципы ориентации, *397*
составляющие, *398*
структура, *403*
управление версиями, *400*
функции, *400*

Состояние
пользователя, *507*
приложения, *507*
Сохранение данных, *205*
Специальные типы данных, *226*
Стили архитектуры, *42*
Схема
клиентской базы данных, *498*
серверной базы данных, *498*
сообщений, *398*
Сценарии
написание с применением агрегатов, *349*
сохранения, *238*
тестирования базы данных, изменяемые, *256*
транзакции, *44*

Т

Тестирование
базы данных
автоматическое, *252*
методы, *253*
рекомендации по имитации, *259*
кода, взаимодействующего с базами данных, *257*
мелкомасштабной интеграции, *232*
поведения объекта, *128*
по взаимодействию, *128*
по состоянию, *128*
хранилищ, *232*
Технология как средство реализации модели, *52*
Транзакции
абстрактное представление, *291*
обработка, *244*
откат, *245*
Трассировка
на месте, *82*
приложения, *81*
эпизодическая, *82*

У

Удобство сопровождения программ, *57*
Указатель службы, *135*; *414*
Универсальный язык, *47*
Унифицированный язык моделирования, UML, *43*; *49*
Управление параллельным выполнением операций, *350*; *384*
Уровень
абстракции сохраняемости, *235*
обслуживания
контроль транзакций, *505*
статические методы, *500*
структура, *499*
представления, *474*

Ф

Фабрики
заказов, *182*; *188*
как термин, *113*
применение в виде шаблонов, *112*
разработки программ, *69*
сеансов работы, *355*
Формирование запросов, *261*
Функциональные процессы, *513*
Функциональный анализ, *511*

Х

Хранение данных
в объектных базах данных, *327*
в оперативной памяти, *324*
в реляционной базе данных, *328*
в файловой системе, *327*
Хранилища
вспомогательные, *197*
единственного ряда, *237*
заказов, *188*
компоновки, *235*
назначение, *231*
поддельные варианты, *232*
преимущества, *232*

структура тестов, *248*
тестирование, *232*
Хранимые процедуры, *56*

Ш

Шаблоны
Adapter/Wrapper, *459*
Aggregate, *164*; *287*; *295*
Bounded Context, *393*
Chain of Responsibility, *425*
Class Table Inheritance, *347*
Collecting Parameter, *303*
Concrete Table Inheritance, *347*
Constructor Dependency Injection, *415*
Contextualized Dependency Lookup, *420*
Data Mapper, *62*; *329*; *330*
Decorator, *310*; *431*
Dependency Injection, *415*; *426*
Domain Model, *491*
Embedded Value, *346*; *380*
Entity, *200*
Factory, *109*; *111*; *412*
Foreign Key Mapping, *346*
Generation Gap, *515*
Identity Field, *167*
Identity Map, *199*; *289*; *331*; *348*
Interface Driven Setter Dependency Injection, *419*
Inversion of Control, *419*
Layers, *99*
Lazy Load, *162*; *226*; *331*; *349*
Metadata Mapping, *333*; *343*
Model-View-Controller, *451*
Notification, *282*; *307*
Null Object, *112*; *170*; *175*
Pipes and Filters, *99*
Presentation Model, *178*; *450*
Proxy, *65*; *195*
Query Object, *101*; *105*
Recordset, *44*; *69*
Reflection, *99*
Registry, *413*
Repository, *112*; *159*; *428*
Separated Presentation, *175*
Service, *164*; *288*
Service Layer, *474*
Service Locator, *413*; *426*
Setter Dependency Injection, *418*
Single Table Inheritance, *347*
Singleton, *147*
Specification, *108*; *269*; *311*
State, *90*; *95*
Strategy, *107*
Template Method, *144*; *248*
Transaction Script, *69*
Unit of Work, *170*; *226*; *240*; *331*; *349*
Value Object, *200*
архитектурные, *98*
категории, *85*
модели предметной области, *48*; *85*; *100*; *152*
назначение, *86*
основания для изучения, *86*
перечень и краткое описание, *529*
поэтапное усвоение, *89*
предметной области, *107*
предостережения, *88*
проектирования, *90*
составляющие, *86*

Э

Экстремальное программирование, ЭП, *41*; *67*
Энергичная загрузка, *349*; *383*
Эффективность программ и базы данных, *53*

Я

Языки
запросов, SQL, *269*
описания данных, DDL, *357*
программирования предметной области, DSL, *70*

Научно-популярное издание

Джимми Нильссон

Применение DDD и шаблонов проектирования:
проблемно-ориентированное проектирование приложений с примерами на C# и .NET

Литературный редактор *П.Н. Мачуга*
Верстка *Т.А. Корзун*
Художественный редактор *В.Г. Павлютин*
Корректор *Л.А. Гордиенко*

Издательский дом "Вильямс"
127055, г. Москва, ул. Лесная, д. 43, стр. 1

Подписано в печать 23.08.2007. Формат 70×100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 45,15. Уч.-изд. л. 31,32.
Тираж 2000 экз. Заказ № 0000.

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

Proekt_book.indb 550 27.08.2007 21:33:45

АРХИТЕКТУРА КОРПОРАТИВНЫХ ПРОГРАММНЫХ ПРИЛОЖЕНИЙ

Мартин Фаулер

www.williamspublishing.com

ISBN 5-8459-0604-0

в продаже

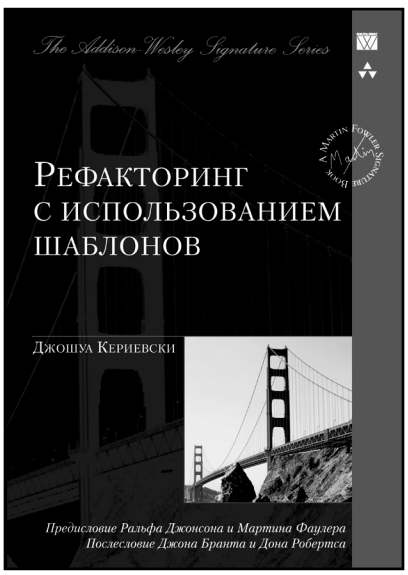
Создание компьютерных систем — дело далеко не простое. По мере того как возрастает их сложность, процессы конструирования соответствующего программного обеспечения становятся все более трудоемкими, причем затраты труда растут экспоненциально. Как и в любой профессии, прогресс в программировании достигается исключительно путем обучения, причем не только на ошибках, но и на удачах — ак своих, так и чужих.

Книга дает ответы на трудные вопросы, с которыми приходится сталкиваться всем разработчикам корпоративных систем. Автор, известный специалист в области объектно-ориентированного программирования, заметил, что с развитием технологий базовые принципы проектирования и решения общих проблем остаются неизменными, и выделил более 40 наиболее употребительных подходов, оформив их в виде типовых решений. Результат перед вами — незаменимое руководство по архитектуре программных систем для любой корпоративной платформы. Это своеобразное учебное пособие поможет вам не только усвоить информацию, но и передать полученные знания окружающим значительно быстрее и эффективнее, чем это удавалось автору

Книга предназначена для программистов, проектировщиков и архитекторов, которые занимаются созданием корпоративных приложений и стремятся повысить качество принимаемых стратегических решений.

РЕФАКТОРИНГ С ИСПОЛЬЗОВАНИЕМ ШАБЛОНОВ

Джошуа Кериевски



www.williamspublishing.com

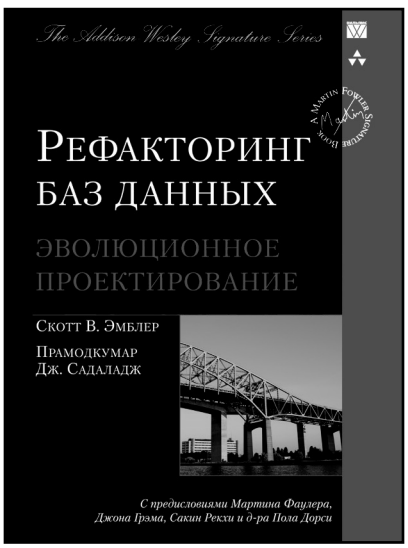
Данная книга представляет собой результат многолетнего опыта профессионального программиста по применению шаблонов проектирования. Авторский подход к проектированию состоит в том, что следует избегать как недостаточного, так и избыточного проектирования, постоянно анализируя готовый работоспособный код и реорганизуя его только в том случае, когда это приведет к повышению его эффективности, упрощению его понимания и сопровождения. Автор на основании как собственного, так и чужого опыта детально рассматривает различные признаки кода, требующего рефакторинга, описывает, какой именно рефакторинг наилучшим образом подходит для той или иной ситуации, и описывает его механику, подробно разбирая ее на конкретных примерах из реальных задач. Книга может рассматриваться и как учебник по рефакторингу для программиста среднего уровня, и как справочное пособие для профессионала.

ISBN 5-8459-1087-0

в продаже

РЕФАКТОРИНГ БАЗ ДАННЫХ
ЭВОЛЮЦИОННОЕ ПРОЕКТИРОВАНИЕ

Скотт В. Эмблер,
Прамодкумар
Дж. Садаладж



www.williamspublishing.com

Книга посвящена описанию процедур проектирования базы данных с точки зрения архитектора объектно-ориентированного программного обеспечения, поэтому представляет интерес и для разработчиков прикладного кода, и для специалистов в области реляционных баз данных. Значительное место уделено описанию того, как действовать в тех практических ситуациях, когда база данных уже существует, но плохо спроектирована, или когда реализация первоначального проекта базы данных не позволила получить качественную модель. Прежде всего книгу можно использовать в качестве технического руководства для разработчиков, непосредственно занятых на производстве. С другой стороны, она представляет собой теоретическую работу, стимулирующую дальнейшие исследования в направлении объединения объектно-ориентированного и реляционного подходов.

ISBN 978-5-8459-1157-5

в продаже

БЫСТРАЯ РАЗРАБОТКА ПРОГРАММ: ПРИНЦИПЫ, ПРИМЕРЫ, ПРАКТИКА

Роберт С. Мартин

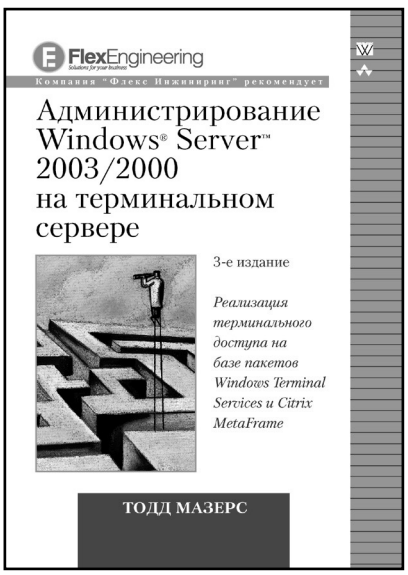
www.williamspublishing.com

в продаже

Имя Роберта Мартина, одного из "отцов" методик, заложенных в основу быстрого и экстремального программирования, не нуждается в представлении. Вниманию читателей предлагается его новая книга, написанная в соавторстве с Джеймсом Ньюкирком и Робертом Коссом. На сегодняшний день уже появились десятки публикаций, в которых рассматриваются современные методики разработки ПО, взятые на "вооружение" программистами. Но именно в этой книге особенно удачно продемонстрировано то, что методы экстремального программирования носят универсальный характер и не зависят от выбранной платформы и средств разработки программ. Изложение начинается с обзора основных понятий экстремального программирования и завершается готовыми программами, применяемыми на практике. В каждой главе приведены примеры кода на языках программирования Java и C++. Книга будет полезной руководителям групп программистов, нацеленных на быструю разработку коммерческих программных проектов, характеризующихся высоким уровнем качества и минимальной себестоимостью.

АДМИНИСТРИРОВАНИЕ WINDOWS® SERVER™ 2003/2000 НА ТЕРМИНАЛЬНОМ СЕРВЕРЕ
3-Е ИЗДАНИЕ

Тодд Мазерс



www.williamspublishing.com

В этой книге содержатся все необходимые сведения для создания быстрого, масштабируемого и стабильного терминального решения на базе пакетов Microsoft Windows Terminal Services и Citrix MetaFrame. Имея за плечами десятилетний опыт работы, автор описал практически полный жизненный цикл проекта, от оценки исходной ситуации и планирования вплоть до тестирования, развертывания и обслуживания. Масштаб рассматриваемых систем — от терминальных решений для малого бизнеса до серверных ферм Citrix уровня предприятия. В книге приводятся проверенные рецепты обхода различных трудностей, начиная от обеспечения безопасности сервера и заканчивая групповыми политиками, управлением исправлениями и автоматизированной установкой клиентов.

ISBN 978-5-8459-1266-4

в продаже

C# 2005
СПРАВОЧНИК ПРОГРАММИСТА

Эдриан Кингсли-Хьюджес, Кэти Кингсли-Хьюджес

www.dialektika.com

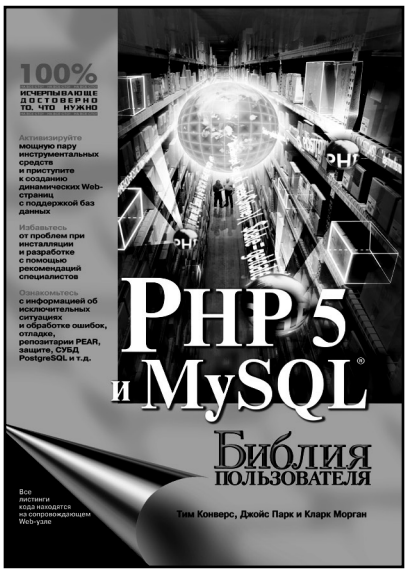
Язык C# позволяет создавать надежные и устойчивые приложения производственного уровня, выполняемые в рамках среды .NET Framework, включая Web-службы в формате XML, приложения клиента и сервера, баз данных и т.д. С появлением C# 2005 в распоряжение программистов предоставлен целый набор новых возможностей для создания еще более мощных приложений, если вы знаете, как эти возможности использовать. Наша книга научит вас этому. Объединяя в себе комплексный учебник и справочник, эта книга раскрывает самые последние возможности языка, начиная с его основ. В части I объясняются основные понятия программирования и обеспечивается приобретение необходимых навыков в деле создания собственных приложений. В части II предлагается подробная информацию о специфических особенностях языка, чтобы вы могли немедленно воспользоваться преимуществами всех его инструментов.

ISBN 978-5-8459-1288-6

в продаже

PHP 5 И MYSQL
БИБЛИЯ ПОЛЬЗОВАТЕЛЯ

Тим Конверс,
Джойс Парк
и Кларк Морган



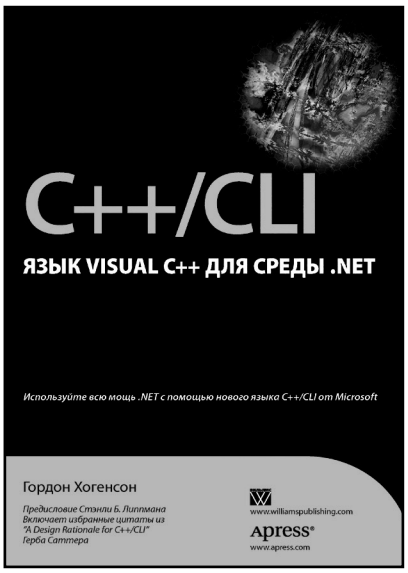
www.dialektika.com

В книге приведены исчерпывающие сведения по созданию динамических Web-узлов на основе программных средств с открытым исходным кодом: языка PHP, сервера Apache и СУБД MySQL, а также показано, как обеспечить бесперебойную эксплуатацию таких узлов под управлением операционных систем Windows или Linux. Многочисленные сценарии и готовые программы, представленные в книге, подробно описаны, тщательно прокомментированы и составляют основу практически значимых приложений. Книга дополняет электронную документацию, содержит все необходимые справочные данные и рассчитана на широкий круг читателей.

ISBN 5-8459-1022-6 **в продаже**

C++/CLI
ЯЗЫК VISUAL C++ ДЛЯ СРЕДЫ .NET

Гордон Хогенсон



www.williamspublishing.com

Книга Гордона Хогенсона представляет собой описание нового языка C++/CLI, разработанного фирмой Microsoft для платформы .NET. В ней достаточно подробно описаны все отличия данного диалекта от базового языка Visual C++ и для каждого нового средства языка приведены короткие выразительные, законченные примеры, которые читатель может самостоятельно скомпилировать и выполнить. В некоторых случаях приведены более объемные реалистичные проекты. Книга может служить для первоначального знакомства с C++/CLI и будет полезна широкому кругу программистов, желающих ознакомиться с программированием для платформы .NET.

ISBN 978-5-8459-1279-4

в продаже

ПОЛНЫЙ СПРАВОЧНИК ПО JAVA™ 7-Е ИЗДАНИЕ

Герберт Шилдт

www.williamspublishing.com

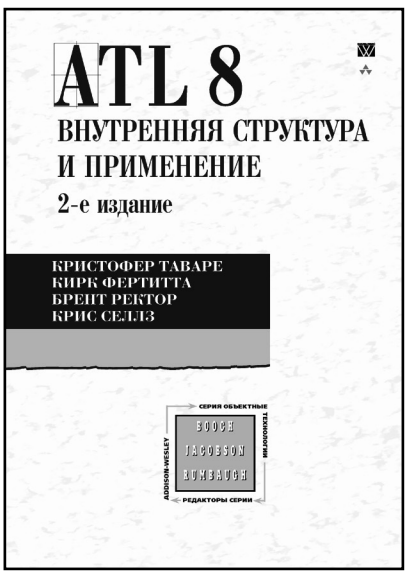
Книга известного “гуру” в области программирования посвящена новой версии одного из наиболее популярных и совершенных языков – Java. Построенная в виде учебного и справочного пособия, она является превосходным источником исчерпывающей информации по последней версии платформы Java, Java SE 6, и позволяет практически с нуля научиться разрабатывать приложения и аплеты производственного качества. Помимо синтаксиса самого языка и фундаментальных принципов программирования, в книге подробно рассматриваются такие сложные вопросы, как ключевые библиотеки Java API, каркас коллекций, создание аплетов и сервлетов, AWT, Swing и Java Beans. Немалое внимание уделяется вводу-выводу, работе в сети, регулярным выражениям и обработке строк.

ISBN 978-5-8459-1168-1

в продаже

ATL 8
ВНУТРЕННЯЯ СТРУКТУРА И ПРИМЕНЕНИЕ
2-Е ИЗДАНИЕ

Кристофер Таваре
Кирк Фертитта
Брент Ректор
Крис Селлз



www.williamspublishing.com

Эта книга адресована программистам, работающим в Visual Studio 2005, использующим C++/COM и собирающимся перейти на ATL 8. Эти разработчики создают приложения, которые должны работать на любых системах, начиная с Windows 95, со слишком старыми процессорами и малыми объемами памяти для .NET. Они не могут позволить себе роскошь тратить драгоценные ресурсы на сбор мусора, JIT-компиляцию или на огромные библиотеки, набитые ненужными им классами. Для них важнее всего скорость загрузки программ, их быстродействие и способность работать в любых условиях. Чтобы они стали использовать какую-то платформу, она должна быть свободна от накладных расходов на ненужные им возможности, обладать гибкостью и высокой производительностью. Именно для таких разработчиков и предназначена ATL 8, новейшая платформа для разработки Windows-приложений.

ISBN 978-5-8459-1169-8

в продаже