Эрик Эванс

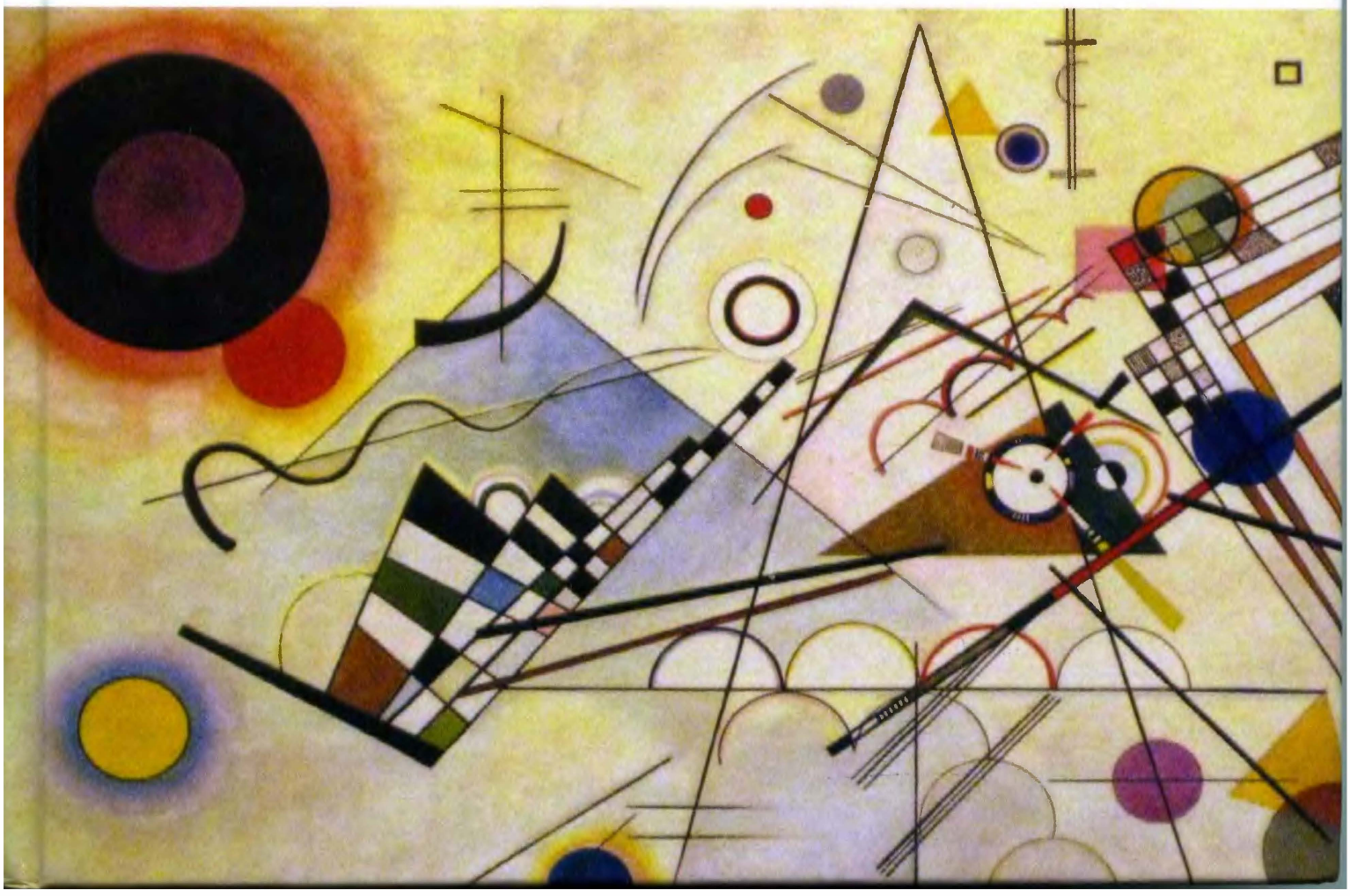
Предисловие
Мартина Фаулера

Domain-Driven

DESIGN

Предметно-ориентированное проектирование

СТРУКТУРИЗАЦИЯ СЛОЖНЫХ ПРОГРАММНЫХ СИСТЕМ



Domain-Driven Design

TACKLING COMPLEXITY IN THE HEART OF SOFTWARE

Eric Evans

Addison-Wesley

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

Предметно-ориентированное проектирование

СТРУКТУРИЗАЦИЯ СЛОЖНЫХ ПРОГРАММНЫХ СИСТЕМ

Эрик Эванс

Издательский дом "Вильямс"
Москва • Санкт-Петербург • Киев
2011

ББК 32.973.26-018.2.75
Э14
УДК 681.3.07

Издательский дом "Вильямс"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *В.Л. Бродового*

По общим вопросам обращайтесь в Издательский дом "Вильямс" по адресу:
info@williamspublishing.com, http://www.williamspublishing.com

Эванс, Эрик.

Э14 Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2011. — 448 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1597-9 (рус.)

ББК 32.973.26-018.2.75

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм.

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

Authorized translation from the English language edition published by Addison-Wesley Publishing Company, Inc, Copyright © 2004 by Eric Evans.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Russian language edition published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2011

Научно-популярное издание

Эрик Эванс

Предметно-ориентированное проектирование (DDD): структуризация сложных программных систем

Литературный редактор *Е.П. Перестюк*
Верстка *О.В. Романенко*
Художественный редактор *В.Г. Павлютин*
Корректор *Л.А. Гордиенко*

Подписано в печать 26.08.2010. Формат 70x100/16.
Гарнитура Times. Печать офсетная.
Усл. печ. л. 36,12. Уч.-изд. л. 31,4.
Тираж 1000 экз. Заказ № 23564.

Отпечатано по технологии CtP
в ОАО "Печатный двор" им. А. М. Горького
197110, Санкт-Петербург, Чкаловский пр., 15.

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

ISBN 978-5-8459-1597-9 (рус.)
ISBN 978-0-321-12521-7 (англ.)

© Издательский дом "Вильямс", 2011
© by Eric Evans, 2004

Оглавление

Содержание

Отзывы

“Эта книга должна стоять на полке у каждого мыслящего программиста”. — *Кент Бек (Kent Beck)*

“Эрик Эванс написал прекрасную книгу о том, как привести архитектуру программы в соответствие с умозрительной моделью той предметной области, для которой эта программа предназначена.

Его книга хорошо согласуется с идеями экстремального программирования. Суть книги — не в рисовании схем предметных областей, а в образе мышления, в использовании правильного языка для их описания, в такой организации программы, которая бы отражала постоянно углубляющееся понимание предмета. Эрик полагает, что узнать что-то новое о предметной области равно возможно как в начале, так и в конце работы над проектом, поэтому в его методике значительную роль играет рефакторинг.

Книгу легко читать. У Эрика в запасе много интересных историй, и языком он владеет хорошо. Я считаю эту книгу необходимым чтением для разработчиков программ — и будущей классикой.” *Ральф Джонсон (Ralph Johnson), автор книги “Design Patterns”*

“Если вам кажется, что ваша ставка на объектно-ориентированное программирование не окупает себя, то из этой книги вы узнаете, чего же вам не хватает.” — *Уорд Каннингем (Ward Cunningham)*

“Эрику удалось “ухватить” суть того, что опытные проектировщики программных объектов всегда знали, но с блеском проваливали все попытки донести это знание до своих коллег в смежных областях. Мы охотно делимся отдельными секретами... но никогда не заботились об организации и систематизации принципов построения логической структуры предметной области. Вот почему эта книга так важна.” — *Кайл Браун (Kyle Brown), автор книги “Enterprise Java Programming with IBM WebSphere”*

“Эрик Эванс убедительно доказывает важность моделирования предметной области, его определяющее значение для разработки программного проекта, а также предлагает солидную теоретическую основу и набор методов для этой цели. Это знание непреходящей ценности, и оно останется в обиходе еще долгое время после того, как отомрут все нынешние методологии-скороспелки.” *Дэйв Коллинз (Dave Collins), автор книги “Designing Object-Oriented User Interfaces”*

“Эрик обобщил реальный опыт архитектурного проектирования и реализации профессиональных приложений в виде полезной и нужной книги. Выступая с позиций опытного специалиста-практика, он обогатил нашу профессию своими описаниями всеобщего языка, аргументами в пользу совместной с пользователями разработки моделей, методами сопровождения объектов на протяжении всего их существования, физического и логического структурирования программ, процедурой и анализом результатов глубокого рефакторинга.” *Люк Хоман (Luke Hohmann), автор книги “Beyond Software Architecture”*

Посвящается маме и папе

Предисловие

В процессе разработки программного обеспечения хватает всевозможных трудностей. Главное — это естественная сложность предметной области, к которой относится решаемая задача. Всякий раз, когда при разработке программного обеспечения возникает необходимость автоматизировать созданные человеком сложные системы, избежать этой сложности нельзя — ею можно только "овладеть".

Для этого необходима хорошая предметно-ориентированная модель, проникающая значительно дальше поверхностного взгляда на проблему. Если в такой модели удастся правильно отразить внутреннюю структуру предметной области, то разработчики программного обеспечения получат именно тот инструмент, в котором они нуждаются. Хорошая модель предметной области представляет огромную ценность, но построить ее нелегко. Умеют это делать немногие, а научить других этому искусству очень трудно.

Эрик Эванс относится к тем немногим людям, которые хорошо умеют строить модели предметных областей. Я обнаружил это, непосредственно работая с ним. То был один из счастливых случаев, когда твой клиент оказывается более квалифицированным, чем ты сам. Наше сотрудничество было недолгим, но необыкновенно интересным. С тех пор мы постоянно держали связь, и я в течение долгого времени наблюдал, как вызревала эта книга. Надо признаться, ожидание оказалось не напрасным.

В результате длительной "эволюции" этой книги были достигнуты весьма амбициозные цели — дать полное справочное описание техники моделирования предметных областей, создать контекст, в котором этот непростой вид деятельности можно было бы успешно объяснять и преподавать. В процессе создания книги я почерпнул из нее множество новых идей, и будет неудивительно, если даже опытные специалисты в области концептуального моделирования также обогатят свой арсенал после ее изучения.

Эрик Эванс сумел сформулировать и закрепить многое из того, что мы постепенно осознавали годами. Прежде всего, в предметно-ориентированном моделировании нельзя отделять понятия (концепции) от их реализации. Специалист по моделированию предметных областей способен не только чертить диаграммы вместе с экспертом-аналитиком, но и программировать на Java вместе с программистом. Частично это происходит потому, что нельзя построить *полезную* концептуальную модель, не рассматривая вопросы ее реализации. Но основная причина единства понятия и ее реализации все же состоит в том, что модель предметной области приносит наибольшую пользу только тогда, когда она предоставляет специалисту в этой предметной области и инженеру-разработчику *единый язык*, на котором они могли бы разговаривать друг с другом.

Еще один урок, который можно почерпнуть из этой книги, таков: в моделях предметных областей на самом деле не разделяют архитектуру и реализацию. Как и многие другие, я пришел к отрицанию принципа поэтапности — "сначала строим модель, потом думаем о реализации". Опыт Э. Эванса учит нас, что действительно мощные модели эволюционируют со временем, и даже высококвалифицированные специалисты иногда обнаруживают, что наилучшие идеи приходят к ним уже *после* первого выпуска соответствующих программных систем.

Я думаю (и надеюсь), что эта книга сделает в своей области очень важное дело: поможет структурировать и разложить "по полочкам" весьма зыбкую и неопределенную область знания, а также научит многих людей пользоваться этим ценным инструментом. Модели предметных областей могут оказать огромное влияние на разработку программного обеспечения, какие бы среды и языки при этом не использовались.

И последнее, но важное замечание. Одна из особенностей этой книги — это способность автора откровенно говорить о своих неудачах. Многие авторы любят создавать в своих книгах стерильную атмосферу всемогущества. Эрик Эванс же дает понять, что он, как и все мы, сталкивался в своей практике как с успехами, так и с разочарованиями. Важно то, что он смог научиться чему-то как на победах, так и на поражениях, и еще важнее для нас, что он смог передать эти уроки своим читателям.

Мартин Фаулер (Martin Fowler), апрель 2003 г.

Предисловие редактора русского перевода

Переводить какую бы то ни было классическую работу, завоевавшую всеобщее признание среди специалистов, — это большая ответственность. А книга Э. Эванса обладает спецификой, которая накладывает еще большую ответственность и создает дополнительные трудности при ее переводе.

Основной этап в процессе работы над программным обеспечением, по мнению автора, — создание "единого языка" (Ubiquitous Language). Это система понятий, терминов, категорий, в рамках которой должны общаться между собой все причастные к разработке — заняты ли они построением абстрактной модели прикладной деятельности, написанием собственно программного кода или же обеспечением инфраструктуры для разработки. В целях успеха проекта этим же языком должны овладеть и заказчики, для которых пишется программа — специалисты-прикладники в той или иной предметной области.

Автор строго следит (и предлагает программистам следовать его примеру), чтобы единый язык программного проекта не запутывался, не приобретал двусмысленности, не перегружался лишним и не терял необходимого. Можно сказать, книга учит читателя-программиста, как добиться четкости и ясности не просто в управлении проектом, но и в собственном мышлении тоже, ибо язык — основа мышления. Этой цели служит и выбор имен программных объектов, и выбор терминов на диаграммах моделей или классов, и отбор необходимого минимума знаний из предметной области, и их структуризация. Большое внимание уделяется кажущимся мелочам в выражении идей, которые в реальном проекте могут сработать как на его успех, так и на полный провал.

В этом контексте добавление целого дополнительного языка к тем семантическим средствам, которыми автор доносит свои мысли до читателя, выглядит как появление слона в посудной лавке. Имеется в виду необходимость переводить весь тот храм четкого и недвусмысленного мышления, который воздвиг автор, на русский язык. Эта необходимость добавляет семантических трудностей. С одной стороны, нельзя потерять взаимосвязанность, единообразие модели и архитектуры программы, которую так тщательно создавал автор, подбирая термины и выражения. С другой стороны, нельзя и предъявить читателю схемы, описания, архитектурные образцы и шаблоны, термины, названия объектов исключительно в их первозданном виде, ибо какой же тогда смысл в переводном издании?

В связи с этим переводчику книги пришлось и самому овладевать таким способом мышления, виднейшим апологетом которого является Э. Эванс, — для того, чтобы правильно выбрать, где адаптировать выразительные средства автора к другому языку и как это сделать, а где не трогать их вообще. Нельзя было обойтись без параллельного использования переводных и оригинальных терминов — и это сделало словарь "единого языка" книги при переводе несколько объемнее, чем в оригинале. На схемах и диаграммах, напротив, сохранены большей частью английские названия, которые я попытался пояснить в тексте. Читателю переводного издания, таким образом, придется приложить несколько больше механических усилий для овладения терминологией книги, чем его американскому коллеге, — но, по-видимому, способа облегчить это бремя не существует. Впрочем, русскоязычному IT-сообществу, которое в своей работе активно пользуется и русским, и английским языками, это не должно составить большого труда.

Нелегко сказать заранее, удалось ли в русском издании книги преодолеть идейные и языковые трудности и предъявить читателю издание, сохраняющее на должном уровне хотя бы часть достоинств великолепной книги Э. Эванса. Об этом предстоит судить самому читателю.

В. Бродовой, февраль 2010 г.

Введение

Ведущие программисты считали моделирование предметных областей (*domain modeling*) и проектирование архитектуры программных объектов на этой основе (*domain design*) ключевой методологией разработки сложных программных систем на протяжении вот уже более двадцати лет. Тем не менее за это время было написано удивительно мало об основных задачах (*что делать*) и методах (*как делать*) этой области знания. Но несмотря на отсутствие четкой формализации, в профессиональной среде постепенно "вызрела" система взглядов и подходов, которую я называю *предметно-ориентированное проектирование* (*domain-driven design*, *DDD*).

За прошедшее десятилетие мне довелось участвовать в разработке нескольких сложных систем, относящихся к различным областям техники и бизнеса. В своей работе я старался применять самые лучшие приемы проектирования и разработки, созданные лидерами объектно-ориентированного программирования. Некоторые из моих проектов имели успех, другие же оказались неудачными. Все успешные проекты объединяло одно свойство: для них была построена подробная модель предметной области, которая итеративно улучшалась в ходе разработки, пока не становилась органичной частью проекта.

В этой книге предпринимается попытка построить систему понятий, в рамках которой было бы удобно принимать проектные решения, а также создать технический язык для коммуникации при разработке архитектурной модели предметной области (т.е. в том процессе, который называют *domain design*). Общепризнанные практические приемы соседствуют здесь с моими собственными выводами и результатами. С помощью такой системы коллективы разработчиков программного обеспечения, работающие со сложными предметными областями, смогут выработать строгий, формализованный подход к построению своих проектов.

Разные судьбы трех проектов

Я хорошо помню три проекта, явившиеся прекрасными примерами того, как реализация предметной области влияет на конечный результат. Хотя в итоге всех трех проектов на свет появились полезные программы, только в одном из них удалось достичь по-настоящему дерзких целей и создать такое программное обеспечение, которое продолжает успешно развиваться, удовлетворяя растущие запросы работающей с ним организации.

Один из проектов "сошел со стапелей" очень быстро, и в результате появилась простая и удобная интернет-система брокерской торговли. Ее разработка велась довольно бессистемно, но все обошлось, потому что относительно простые программы можно писать и без тщательного проектирования их архитектуры. После первоначального успеха надежды на будущее были большими. Именно тогда меня и хотели привлечь к работе над второй версией программы. Когда я поближе ознакомился с проектом, я обнаружил, что у разработчиков отсутствует не только модель предметной области, но и общий язык для всего проекта, а плохо продуманная архитектура стала настоящим бременем. Руководители проекта не согласились с моей оценкой состояния дел, и я отказался от этой работы. Но год спустя коллектив разработчиков "застрял" на месте и оказался неспособен реализовать вторую версию. Пусть у них не все было ладно и по технической части, но все-таки поражение они потерпели именно в борьбе с прикладной моделью. И вот уже самая первая версия программы преждевременно перешла в разряд дорогостоящих в обслуживании реликтов.

Реализация сложных систем требует серьезного подхода к моделированию архитектуры предметной области. В начале своей трудовой деятельности мне посчастливилось принять участие в проекте, в котором работе с предметной областью действительно придавали большое значение. Этот проект, относящийся к области по крайней мере столь же сложной, как и первый, тоже начинался с создания несложной программы для институциональных трейдеров. Но в данном случае первоначальная версия получила самое интенсивное продолжение. На каждом новом этапе разработки открывались захватывающие перспективы по объединению и усовершенствованию возможностей предыдущих версий. Группа разработчиков сумела среагировать на потребности трейдеров с должной гибкостью и эффективностью. Этим взлетом они были непосредственно обязаны модели предметной области — глубоко и последовательно проработанной, поэтапно улучшаемой, четко отраженной в коде. По мере того как разработчики вникали в предмет, совершенствовалась и модель. Улучшалось понимание не только между самими разработчиками, но и между группами экспертов и разработчиков. И вместо того чтобы накладывать на авторов тяжкое бремя поддержки и доработки программы, архитектура проекта становилась, напротив, все более удобной для модификации и расширения.

К сожалению, для столь полного успеха проекта недостаточно только серьезного подхода к моделированию. Один из проектов, которым я занимался в прошлом, имел весьма амбициозную цель — создание глобальной корпоративной системы управления на основе предметной модели. Но после нескольких лет разочарований ожидания стали самыми заурядными. У группы разработчиков были в наличии хорошие технические возможности и хорошее понимание предметной области; они также уделили должное внимание модели. Но неправильно организованное распределение труда разработчиков привело к отделению моделирования от реализации, и в результате при проектировании архитектуры не учитывалась вся глубина проделанного анализа. По крайней мере, объекты прикладной модели были спроектированы недостаточно строго, чтобы обеспечить их взаимодействие в сложных приложениях. Несколько итераций доработки не внесли в код никаких улучшений из-за разброса в уровне квалификации разработчиков — они, в целом, не владели неформальным стилистическим и техническим аппаратом для создания модельных объектов, которые бы одновременно представляли собой практичные, работоспособные программы. Шли месяцы, процесс разработки все усложнялся и запутывался, группа потеряла целостное видение системы. После нескольких лет усилий проект все-таки выдал "на-гора" небольшую полезную систему, но в итоге группа разработчиков отказалась как от своих прежних амбиций, так и от самой модели.

О сложностях

В процессе создания проекта возможны различные препятствия, — например, бюрократизм, нечеткая постановка задач, недостаток ресурсов. Но именно стратегия архитектурного проектирования главным образом определяет потенциальную сложность будущего программного продукта. Когда сложность выходит из-под контроля, разработчики перестают понимать свое изделие достаточно хорошо, для того чтобы вносить в него исправления или расширять его возможности без особого труда и опасений. В противоположность этому, качественно спроектированная архитектура создает возможность грамотно использовать сложность системы.

Некоторые влияющие на проектирование программ факторы имеют технический характер. Много труда уходит на проектирование сетей, баз данных и другие технические аспекты программного обеспечения. О решении таких проблем написаны многие тома. Целые армии программистов оттачивают свои умения и осваивают каждое новшество в этой сфере.

И все же главная сложность во многих программных продуктах — вовсе не техническая. Ее истоки — в самой предметной области, в той отрасли знания, которой занимается пользователь программы. Если сложную структуру предметной области не отразить в архитектурном проекте приложения, то не будет иметь никакого значения качественная проработка технической инфраструктуры. Для успешного проектирования программ необходимо систематически подходить к этому основному аспекту программирования.

В этой книге развиваются две основополагающие идеи.

1. В большинстве программных проектов основное внимание должно быть сосредоточено на логической структуре предметной области и взаимосвязях в ней.
2. Архитектура сложного программного обеспечения должна основываться на модели предметной области.

Предметно-ориентированное проектирование — это и образ мышления, и система приоритетов, призванная ускорить разработку программных продуктов, которые применяются в сложных областях деятельности. Для достижения этой цели в настоящей книге представлен обширный набор приемов, подходов и принципов проектирования программного обеспечения.

Процессы проектирования и разработки

Книги по проектированию и книги по методике реализации программных систем... Почему-то эти книги нечасто ссылаются друг на друга. И та, и другая область знания достаточно сложна. Эта книга — о проектировании, но, по моему мнению, проектирование и реализация неотделимы друг от друга. Архитектурные концепции следует успешно воплощать в программах, иначе они выродятся в пустую теорию.

Когда кто-нибудь изучает приемы проектирования, от возникающих возможностей у него начинает кружиться голова. Но суровая реальность практического проекта быстро излечивает от головокружения. Новые архитектурные идеи не стыкуются с технологиями, которые приходится использовать. Непонятно, когда можно отказаться от того или иного аспекта грамотного проектирования в интересах скорости разработки, а когда нужно упорно искать архитектурно и стилистически "чистое" решение. Бывает, что разработчики говорят друг с другом на абстрактном уровне о применении принципов архитектурного проектирования, но все-таки для них более естественно говорить о конкретных вопросах реализации. И хотя эта книга — о проектировании, я собираюсь в случае необходимости прорываться через искусственно созданную границу прямо в методологию разработки. Это поможет нам рассмотреть принципы проектирования в практическом контексте.

Эта книга не привязана к какой-то конкретной методологии, но в целом ориентирована на новое семейство гибких методик разработки (*agile development processes*). В частности, всегда подразумевается использование в проектах двух подходов, являющихся необходимыми предпосылками для применимости всего того, о чем говорится в книге.

1. **Итеративный характер разработки**. Итерационная разработка программ пропагандируется и практикуется уже не первое десятилетие. Этот подход лежит в основе гибкой методологии. По ней, а также по экстремальному программированию (*extreme programming, XP*) есть много хорошей литературы. Можно назвать такие книги, как *Surviving Object-Oriented Projects* (Cockburn, 1998) и *Extreme Programming Explained* (Beck, 1999).
2. **Тесное взаимодействие между разработчиками и специалистами в предметной области**. Особенность предметно-ориентированного проектирования такова, что в его

процессе в модель закладывается огромное количество знаний, отражающих глубокое понимание предметной области и концентрацию на ее ключевых концепциях. Это требует сотрудничества между теми, кто владеет предметом, и теми, кто умеет писать программы. Так как разработка носит итеративный характер, сотрудничество должно продолжаться на протяжении всего срока существования проекта.

Экстремальное программирование, авторами которого являются Кент Бек (Kent Beck), Уорд Каннингем (Ward Cunningham) и др. (см. *Extreme Programming Explained* [5]) — наиболее известная из гибких методологий разработки программ, с которой лично я работал больше всего. На протяжении этой книги для конкретизации изложения я буду использовать именно ее в качестве основы для рассмотрения взаимодействия между проектированием и разработкой. Приведенные принципы можно легко адаптировать к другим гибким методологиям разработки.

В последние годы среди разработчиков назрел "бунт" против громоздких методик разработки, перегружающих проект ненужной статичной документацией и чрезмерно детализированным планированием. В противоположность этому такие методики, как экстремальное программирование, делают акцент на способности гибко справляться с изменениями и неопределенностью.

В экстремальном программировании признается важность принятия проектных решений, но категорически отрицается жесткое заблаговременное планирование с полной детализацией проекта (что называют *upfront design*). Вместо этого прилагаются большие усилия для обеспечения коммуникации и способности быстро менять направленность проекта. Имея такую способность, разработчики могут на любом этапе проекта выбирать простейшее из работоспособных решений, а затем постоянно выполнять рефакторинг, внося много мелких проектных улучшений и постепенно приходя к архитектуре, которая будет удовлетворять действительные потребности клиента.

Такой минимализм уже послужил "лекарством" против некоторых излишеств энтузиастов архитектурного проектирования. Многие проекты в свое время утонули в море документации, толку от которой практически не было. Их подвел "мандраж" — члены группы разработчиков так боялись несовершенств архитектуры, что вообще никуда не двигались. Надо было что-то менять.

К сожалению, некоторым из этих методических идей грозит неправильная интерпретация. У каждого программиста есть свое понятие о том, что такое "простейшее" решение. Непрерывный рефакторинг — это последовательность мелких изменений проектной архитектуры. Разработчики, не имеющие четких ориентиров, могут произвести на свет базу кода, плохо понятную и не поддающуюся доработке, — т.е. прямую противоположность слову "гибкость" (*agility*), которое дало название методике. Да, страх перед непредвиденными требованиями часто ведет к излишней детализации архитектуры, но попытки избежать жесткого планирования могут перейти в другую фобию — страх вообще перед любой глубокой проработкой модели.

Фактически, методика экстремального программирования лучше всего подходит разработчикам "с обостренным чутьем" в области архитектурного проектирования. Эта методика предполагает, что архитектуру можно улучшить рефакторингом и что происходить это должно часто и быстро. Но принятые в прошлом проектные решения могут как облегчить, так и затруднить сам процесс рефакторинга. Методика экстремального программирования призвана активизировать коммуникацию в группе разработчиков, но на сам процесс коммуникации также влияют решения, принятые при моделировании и проектировании.

В этой книге проектирование архитектуры и разработка программного продукта переплетаются; на примерах показано, как предметно-ориентированное проектирование и

гибкая методология разработки помогают друг другу. Если в контексте гибкой методологии еще и принимается серьезный подход к моделированию предметной области, то это позволяет значительно ускорить разработку. Взаимосвязь между методикой разработки и моделированием предметной области делает такой подход более практичным, чем любые оторванные от жизни "чистые" архитектурные проекты.

Структура книги

Эта книга разделена на четыре большие части.

Часть I, "Модель предметной области в работе", содержит основные постановки задач предметно-ориентированного проектирования. От этих постановок будут непосредственно зависеть практические приемы в дальнейших разделах. Существует множество подходов к разработке программного обеспечения, поэтому в части I даются определения терминов, а также очерчивается тот неявный контекст, который порождается применением модели предметной области для целей коммуникации и проектирования.

Часть II, "Структурные элементы предметно-ориентированного проектирования", обобщает опыт практического объектно-ориентированного моделирования предметных областей, сводя его к набору основных структурных элементов-"кирпичиков". В этой части основное внимание уделяется преодолению пропасти между абстрактными моделями и конкретными работающими программами. Использование стандартных шаблонов привносит в проектирование упорядоченность, а члены группы разработчиков лучше понимают работу друг друга. Стандартные шаблоны также позволяют обогатить общий язык терминологией, с помощью которой все разработчики могут обсуждать модели и проектные решения.

Но главная задача этой части книги — выделить такие решения, которые позволяют модели и ее программной реализации не отставать друг от друга, а также повышать эффективность как одной, так и другой. Это требует большого внимания к проработке отдельных элементов. Тщательная проработка в мелком масштабе дает разработчику прочный фундамент, на котором можно будет реализовывать методики моделирования из частей III и IV.

Часть III, "Углубляющий рефакторинг", посвящена переходу от отдельных "кирпичиков" к задаче сборки их в практические модели, дающие в конце концов конечный результат. Вместо того чтобы сразу изложить секретные принципы проектирования, в этой части книги делается акцент на исследовательском процессе. Действительно ценные модели не возникают в один день; они требуют глубокого понимания предметной области. А такое понимание приходит не иначе, как в ходе работы: вначале архитектура реализуется в первом приближении, основанном пока на примитивной и наивной модели, а затем она снова и снова подвергается преобразованиям. Всякий раз, когда группа разработчиков делает скачок в понимании предмета, модель преобразуется для учета новых знаний, после чего код подвергается рефакторингу для отражения усовершенствований модели и раскрытия ее возможностей в приложении. Поэтапное погружение в предметную область (наподобие послойной очистки луковицы) дает результат в виде прорыва на новый уровень моделирования, после чего следует серия кардинальных изменений в архитектуре.

Творчество исследователя неформально по своей природе, но не обязательно хаотично. В части III рассматриваются принципы моделирования, которые помогают принимать решения на этом пути, а также методы, способствующие выбору направления в поиске.

Часть IV, "Стратегическое проектирование", посвящена ситуациям, возникающим в сложных системах, больших организациях, а также при взаимодействии с внешними и устаревшими системами. В этой части книги рассматривается три принципа, примени-

мых к системе в целом: соблюдение контекста, дистилляция (в смысле выделения сути, концентрации на логике приложения) и крупномасштабный подход. Стратегические проектные решения принимаются как на уровне групп разработчиков, так и на уровне взаимодействия между группами. Стратегическое проектирование делает возможной реализацию задач из части I в более крупном масштабе — масштабе системы или приложения на уровне обширной и разветвленной корпоративной сети.

На протяжении всей книги изложение иллюстрируется не упрощенными "игрушечными" задачами, а примерами из реальных проектов по разработке программного обеспечения.

Значительная часть книги написана в виде набора "архитектурных шаблонов". Читатели смогут понять материал и не задумываясь об этом, но тем из них, кто интересуется стилем и форматом таких шаблонов, рекомендуется прочитать приложение.

Дополнительные материалы можно найти на сайте `http://domaindriven-design.org`, в том числе примеры кода и дискуссии в профессиональном сообществе.

Для кого предназначена эта книга

Эта книга написана главным образом для разработчиков объектно-ориентированного программного обеспечения. Для большинства членов рабочих групп, занимающихся такими проектами, окажутся полезными те или иные части книги. Наиболее ценной она, по-видимому, будет для тех людей, которые работают над проектами прямо сейчас, пытаясь по ходу дела освоить те или иные приемы и методы, а также для уже имеющих большой опыт в такого рода проектах.

Для чтения книги требуется некоторое знакомство с объектно-ориентированным моделированием. В примерах используются диаграммы UML и код Java, так что полезно владеть хотя бы основами этих языков (хотя досконального их знание и не требуется). Рассуждения о процессе разработки будут иметь более глубокий смысл для знающих экстремальное программирование, но даже и без этого знания материал должен быть вполне понятен.

Разработчикам программ среднего уровня квалификации, которым кое-что известно об объектно-ориентированном проектировании (в том числе из пары-тройки прочитанных книг), эта книга поможет заполнить пробелы в знаниях и составить представление о реальном месте объектного моделирования в проектах по разработке программ. Такие разработчики смогут научиться применению сложных приемов моделирования и архитектурного проектирования при решении реальных проблем.

Высококлассным профессиональным разработчикам программного обеспечения будет интересна всеобъемлющая система работы с предметной областью, предложенная в этой книге. Систематический подход к проектированию поможет также техническим руководителям вести свои рабочие группы к успеху. А единая терминология, употребляемая в книге, позволит квалифицированным разработчикам эффективно общаться с коллегами.

Материал книги изложен последовательно — ее можно читать с начала до конца или же с любой главы. Читатели с различной профессиональной подготовкой могут выбрать свой порядок чтения, но я бы рекомендовал всем начать с введения к части I, а также с главы 1. Помимо этого, основными можно считать главы 2, 3, 9 и 14. Те, кто уже до некоторой степени владеет материалом, смогут определить главные моменты, читая заголовки и текст жирным шрифтом. Высококвалифицированный читатель может пропустить части I и II и перейти к, вероятно, наиболее интересным для него частям III и IV.

В дополнение к основному кругу читателей, кое-что полезное из этой книги могут почерпнуть также аналитики и руководители проектов, частично занятые в технической

их части. Аналитиков может привлечь взаимосвязь между моделированием и проектированием, что позволит им сделать более эффективный вклад в работу в контексте проектов с гибкой методологией разработки. Стратегические принципы проектирования также могут помочь им в постановке задач и организации работ.

Руководителям программных проектов стоит задуматься, как сделать работу команды более эффективной, сосредоточившись на проектировании программ, близких и понятных специалистам и пользователям в соответствующих профессиональных областях. Поскольку стратегические проектные решения тесно связаны с организацией группы и стилем работы, они неизбежно должны приниматься с участием руководства и иметь решающее влияние на ход выполнения проекта.

Предметно-ориентированная команда

Даже самостоятельному разработчику может принести немало пользы знание многих ценных приемов и общих принципов, почерпнутых из предметно-ориентированного проектирования. Но все же наибольший эффект имеет место тогда, когда предметно-ориентированный подход принимает на вооружение целая группа разработчиков, ставящая модель предметной области в центр внимания своего проекта. Таким образом члены группы приобретают общий язык, делающий более эффективным их взаимодействие и помогающий не оторваться от непосредственного создания программного продукта. В итоге это позволяет четко реализовать замысел, точно следуя модели и имея в руках надежный инструмент практической работы. Такой подход дает разработчикам схему взаимодействия рабочих групп, занятых в проектировании, и позволяет систематически сосредоточиться на функциях и возможностях, наиболее характерных и важных для организации-потребителя.

Предметно-ориентированное проектирование — это технически сложный процесс, но он окупает себя многократно, открывая широкие возможности там, где большинству программных проектов не остается ничего больше, как уйти в прошлое.

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 веб-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 координаты:

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

Адреса для писем из:

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

Благодарности

Я работал над этой книгой в той или иной форме более четырех лет, в ходе этого получая помощь и поддержку от многих людей.

Приношу свою благодарность всем тем, кто просмотрел рукопись и сделал свои замечания. Без этой работы книга просто не могла бы появиться на свет. Некоторые рецензенты удостоили мою работу особенно щедрым вниманием. Так, группа Silicon Valley Patterns Group, возглавляемая Рассом Рафером (Russ Rufer) и Трейси Биалек (Tracy Bialek) изучала мою первую законченную черновую версию книги в течение нескольких недель. Группа рецензентов из Иллинойского университета (University of Illinois) под руководством Ральфа Джонсона (Ralph Johnson) также провела несколько недель над более поздним черновиком. Слушать долгие и оживленные дискуссии этих групп оказалось чрезвычайно полезно. Кайл Браун (Kyle Brown) и Мартин Фаулер (Martin Fowler) способствовали успеху дела подробными замечаниями, ценными наблюдениями и бесценной моральной поддержкой (на совместной рыбалке). Комментарии Уорда Каннингема (Ward Cunningham) помогли укрепить самые существенные слабые места в изложении. Алистер Кокберн (Alistair Cockburn) с самого начала поощрял мою работу и помог пробиться сквозь все сложности процесса публикации. За это же следует сказать спасибо и Хилари Эванс (Hilary Evans). Благодаря Дэвиду Сигелу (David Siegel) и Юджину Уоллингфорду (Eugene Wallingford) мне удалось не опозориться в технической части изложения. Вибху Мохиндра (Vibhu Mohindra) и Владимир Гитлевич (Vladimir Gitlevich), не жалея себя, проверили все примеры кода.

Роб Ми (Rob Mee) ознакомился с самыми ранними моими опытами по данной теме и участвовал в наших совместных "мозговых штурмах" всякий раз, когда мне позарез требовалось с кем-то обсудить предлагаемый стиль архитектурного проектирования. Значительно позже он также сидел со мной над очередным черновиком книги.

Один из основных поворотных моментов в написании книги связан с именем Джоша Кериевского (Josh Kerievsky): он убедил меня испробовать формат-шаблон Александера (*Alexandrian pattern format*), в итоге занявший центральное место в организации книги. Также благодаря его помощи впервые обрел целостный вид материал, теперь содержащийся в части II. Это случилось, когда он интенсивно наставлял меня на путь истинный в ходе подготовки к конференции PLoP (*Pattern Languages of Programs*, или "*Языки описания шаблонов в разработке программ*") в 1999 году. Тогда-то и было положено начало этой книге.

Еще я хотел бы поблагодарить Авада Фаддула (Awad Faddoul) за те сотни часов, что я провел над бумагой за столом в его замечательном кафе. Этот приют спокойствия, да еще "много виндсерфинга" — вот что помогало мне держаться.

Моей признательности также заслуживают Мартин Жюссе (Martine Jusset), Ричард Паселк (Richard Paselk) и Росс Венэйблз (Ross Venables) за создание прекрасных фотографий для иллюстрации нескольких ключевых концепций (см. ссылки в разделе "Фотографии").

Прежде чем приступить к этой книге, мне пришлось выработать собственный взгляд на разработку программ и понимание этого процесса. Мое формирование как специали-

ста стало возможно во многом благодаря щедрости нескольких замечательных людей, которые были мне и неформальными наставниками, и друзьями. На мой образ мышления в области проектирования программ оказали наибольшее, пусть и различное, влияние Дэвид Сигел (David Siegel), Эрик Голд (Eric Gold) и Исульт Уайт (Iseult White). Между тем Брюс Гордон (Bruce Gordon), Ричард Фрейберг (Richard Freyberg) и д-р Джудит Сегал (Dr. Judith Segal) помогли мне, также каждый по-своему, пробить себе дорогу в мир успешных реальных проектов.

Мои собственные представления естественным образом выросли из совокупности идей, которые в то время носились в воздухе. Некоторые из них будут явно упомянуты в тексте со ссылками, а другие настолько фундаментальны, что я даже не осознаю их непосредственного влияния.

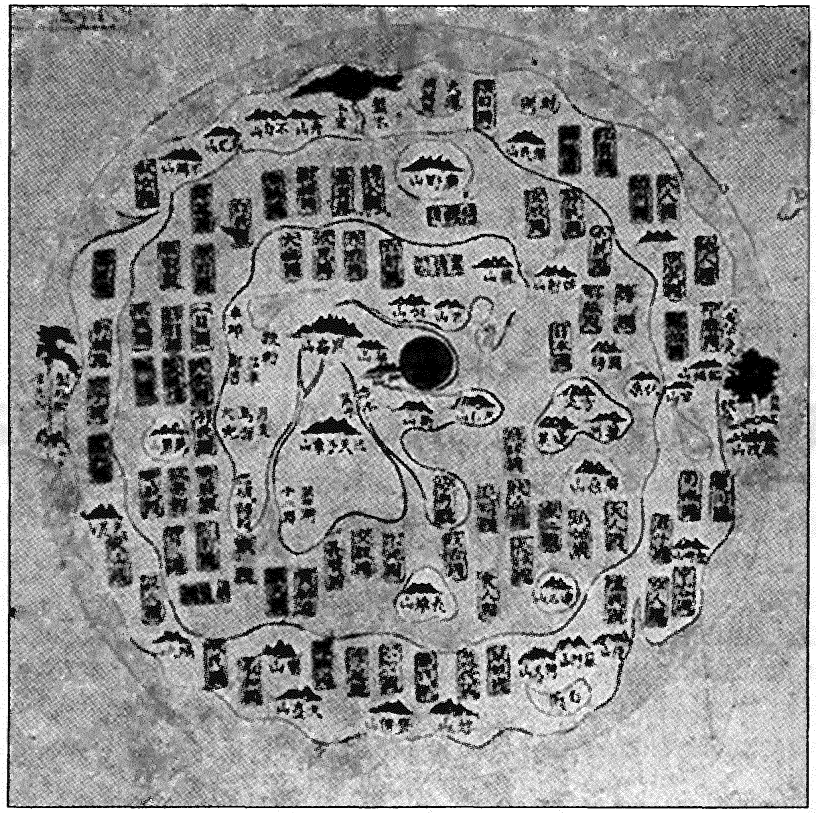
Руководитель моей магистерской диссертации, д-р Бала Субраманьюм (Dr. Bala Subramanium), приобщил меня к математическому моделированию, которое мы применяли в вопросах химической кинетики. Моделирование — всегда моделирование, так что и эта работа частично вывела меня на путь, "приведший" к этой книге.

А еще раньше мое мировоззрение формировали мои родители, Кэрол и Гэри Эванс (Carol & Gary Evans). Заложили нужные основы и пробудили во мне интерес несколько замечательных учителей, особенно учитель математики в старших классах Дейл Керриер (Dale Currier), учительница английской словесности Мэри Браун (Mary Brown) и учительница естественных наук в шестом классе Джозефина Макгламери (Josephine McGlamery).

Наконец, хочу поблагодарить своих друзей и членов семьи, а также Фернандо Де Леона (Fernando De Leon) за поддержку на протяжении всего этого времени.

I

Модель предметной области в работе



Эта китайская карта XVIII столетия изображает весь мир. В центре ее находится Китай, и он же занимает большую часть карты, так что остальным странам уделяется лишь самое поверхностное внимание. Такова была модель мира, приемлемая для того общества, которое преднамеренно замкнулось в самом себе. Но взгляд на мир, который отражала эта карта, должно быть, не слишком помогал в отношениях с иностранцами. А уж современному Китаю он никак не подошел бы. Карты — это модели, а каждая модель представляет определенный аспект действительности или какую-нибудь идею, представляющую интерес. Модель — это упрощение; это такая интерпретация реальности, при которой из явления извлекаются существенные для решения задачи аспекты, а лишние детали игнорируются.

Всякая компьютерная программа имеет применение в той или иной области деятельности или интересов своего пользователя. Область знания или деятельности, в которой пользователь использует программное обеспечение, и есть его *предметная область* (*domain*). Некоторые предметные области связаны с физической реальностью. Например, в предметной области программы онлайн-бронирования авиабилетов мы имеем дело с реальными людьми, желающими лететь на реальных самолетах. Другие предметные области трудно осязаемы физически — например, для программы бухгалтерского учета это деньги и финансовая деятельность. Предметные области программ в большинстве своем не относятся непосредственно к компьютерной технике, хотя есть и исключения. Так, предметная область системы управления исходным кодом — сам процесс разработки программного обеспечения.

Чтобы создать программу действительно ценную для пользователей из какой-либо области деятельности, группа разработчиков должна задействовать запас знаний, относящийся к этой области. Широта требуемого кругозора может оказаться пугающей, объем и сложность информации подавлять воображение. Модели это инструменты,

предназначенные как раз для того, чтобы справиться с этой нагрузкой. Модель представляет собой специально отобранный и сознательно упрощенный запас знаний в структурированной форме. Правильная модель придает информации смысл и позволяет сконцентрироваться на проблеме.

Модель предметной области — это не некая нарисованная схема, а идея, которую схема должна отражать. Это не просто знания специалиста по данному предмету; *это строго организованная выборка из такого знания.* Схема может наглядно изображать модель, передавать информацию о ней, но того же самого можно добиться и при помощи программного кода или предложения на "человеческом языке".

Моделирование предметной области не нацелено на создание максимально "реалистичной" модели. Даже в мире осязаемых, реальных вещей наша модель будет всего лишь искусственным творением. Но моделирование не состоит и в том, чтобы просто сконструировать программный механизм, который бы давал нужный результат. Процесс моделирования чем-то близок к съемке фильма — это тоже примерное изображение реальности, служащее конкретной цели. Даже в документальном фильме не показывают реальную жизнь совсем без прикрас. Как кинематографист выбирает отдельные аспекты реальной жизни и показывает их в своеобразном виде для раскрытия сюжета или передачи послания фильма, так и специалист, моделирующий предметную область, выбирает модель сообразно с ее применимостью.

Роль и выбор модели

Выбор модели в предметно-ориентированном проектировании определяется тремя фундаментальными способами ее использования при разработке программы.

1. *Модель и архитектура[1] программы взаимно определяют друг друга.* Именно тесная связь между моделью и ее программной реализацией определяет важность и нужность самой модели, а также гарантирует, что проделанный при ее разработке анализ отражен в конечном результате, т.е. в работающей программе. Связь между моделью и реализацией помогает также в процессе дальнейшей доработки программы и выпуска ее новых версий, поскольку и сам программный код можно интерпретировать, основываясь на понимании модели. (См. главу 3.)
2. *Модель лежит в основе языка, на котором говорят все члены группы разработчиков.* Ввиду связи между моделью и реализацией разработчики могут обсуждать все связанные с программой вопросы на этом языке, а также общаться со специалистами в предметной области без переводчика. Поскольку язык основывается на самой модели, возможности и средства естественного языка можно даже применить для доработки самой модели. (См. главу 2.)
3. *Модель — это дистиллированное знание[2].* Модель представляет собой согласованный между разработчиками способ структуризации знаний из предметной области, а также выделения элементов, представляющих наибольший интерес. Модель пе-

---

[1] Именно таков примерный смысл термина *design*, применяемого в книге. Это построение принципиальной схемы, объектной архитектуры программ, а также результат такого построения. Но самый полный и точный смысл этого слова передается только совокупностью понятий архитектурного проектирования, конструирования и реконструирования. — *Примеч. перев.*

[2] Автор в дальнейшем часто употребляет слово *distillation* как термин, означающий очищение совокупности информации от всего лишнего, выделение самой сути. Это процесс, издавна лежащий, например, в основе математического моделирования. — *Примеч. перев.*

редает наш способ мыслить о предмете, выраженный в выборе терминов, выделении понятий и установке связей между ними. Разработчики и специалисты в предметной области имеют возможность совместно переработать информацию в такую форму, поскольку для этого у них имеется общий язык. А связь между моделью и реализацией позволяет использовать опыт разработки ранних версий программы для корректировки самого процесса моделирования. (См. главу 1.)

В следующих трех главах последовательно рассматривается значение каждого из этих применений модели, а также взаимосвязь между ними. Правильное использование этих принципов помогает создавать программы с обширным набором функциональных возможностей, что потребовало бы намного больших усилий, если бы разработка велась не систематически, а "в текущем порядке" (*ad hoc development*).

Алгоритмическая часть программы

В алгоритмической части программы[3] сосредоточена ее способность решать для пользователя задачи из соответствующей предметной области. Все остальные части программы и ее функции, какими бы существенными они ни были, только служат этой основной цели. Если область сложна, то и решение соответствующих задач — дело непростое, требующее сосредоточенных усилий талантливых и умелых специалистов. Разработчики вынуждены сами погрузиться в предметную область, чтобы накопить достаточно знаний о ней. Им понадобятся отточенные навыки моделирования и умение строить архитектуру предметной области.

Однако в большинстве программных проектов главные приоритеты совершенно другие. Самые талантливые программисты не очень-то рвутся узнать достаточно о той прикладной области, с которой они работают, не говоря уже о том, чтобы усовершенствовать свои навыки моделирования этой области. "Технари" любят четко поставленные в количественных терминах задачи, на которых можно отточить свои технические умения. А работа с предметной областью — это что-то зыбкое и непонятное; она требует освоения новых сложных знаний, которые кажутся не очень-то нужными для повышения квалификации программиста.

В итоге технический специалист-программист приступает к работе в сложной, обширной, автоматизированной среде разработки, пытаясь решить прикладную задачу за счет развитых программных технологий. Изучение и моделирование предметной области он оставляет другим. Но сложность расчетной части программы не нужно обходить. Избегать схватки означает рисковать, что программа окажется не у дел.

В одном из интервью на телевидении комедийный актер Джон Клиз (John Cleese) рассказал об истории, которая произошла на съемках фильма "Монти Пайтон и Святой Грааль" (*Monty Python and the Holy Grail*). Одну из сцен фильма снимали много раз, но она почему-то не получалась смешной. Наконец Джон Клиз посоветовался с коллегой, Майклом Пэлином (Michael Palin), который также играл роль в этой сцене. Были внесены небольшие изменения, снят еще один дубль, и на этот раз сцена оказалась смешной — на том и порешили.

---

[3] Автор употребляет выражение *heart of software* ("сердце" или "мозговой центр" программы). Это словосочетание входит и в название книги "*Tackling Complexity at the Heart of Software*", которое можно по смыслу перевести как "систематическое упрощение кода алгоритмически сложных систем". Смысл выражения разъяснен автором — это очищенная от всего лишнего алгоритмическая суть программы, то, что она делает для пользователя, какую главную задачу решает. Поэтому избран такой перевод. — *Примеч. перев.*

На следующее утро Клиз просматривал черновой монтаж фильма, который монтажер собрал из материалов предыдущего съемочного дня. Но при просмотре сцены, отнявшей столько сил, Клиз вдруг обнаружил, что она опять не смешная — для нее был взят один из более ранних дублей. Он спросил у монтажера, почему тот не взял последний дубль, как ему было сказано. "Нет, нельзя было. Кто-то вошел в кадр", — ответил монтажер. Джон Клиз пересмотрел сцену еще несколько раз, но ничего не обнаружил. Наконец монтажер остановил фильм и показал рукав пальто, который был виден одно мгновение в кадре.

Монтажер этого фильма был сосредоточен на точном выполнении своих профессиональных обязанностей. Его заботило, что сказали бы о его работе в чисто техническом смысле другие специалисты по киномонтажу. В итоге была потеряна именно "алгоритмическая" часть процесса (передача "The Late Late Show with Craig Kilborn", CBS, сентябрь 2001 г.). К счастью, смешную сцену восстановил режиссер фильма, который понимал толк в комедии. Точно так же и руководители программных проектов, понимающие ключевую роль предметной области, могут вернуть процесс разработки в нормальное русло, если участники проекта вдруг начнут игнорировать необходимость проработки модели, которая отражала бы глубокое понимание предмета.

В этой книге будет показано, что работа с предметной областью открывает широкие возможности для развития профессиональных навыков архитектурного проектирования программ. Нечеткость, недетерминированность большинства предметных областей, для которых пишутся программы, на самом деле ставит перед разработчиками интересные технические задачи. Фактически во многих отраслях науки их "сложность" стала одной из модных тем для обсуждения и исследования, по мере того как ученые все больше пытаются иметь дело с реальным миром. Перед программистом встает аналогичная задача, когда он сталкивается со сложной предметной областью, никогда раньше не подвергавшейся формализации. Разработать четкую и ясную модель, которая не оставит от сложности предмета камня на камне, — что может быть интереснее?

Существуют систематические методы рассуждений, которые может применить программист для достижения понимания предметной области и построения ее эффективной модели. Существуют и приемы проектирования программ, позволяющие привнести порядок в хаотическое нагромождение кода. Освоение таких навыков делает программиста намного более ценным специалистом даже в тех случаях, когда он сталкивается с изначально совершенно незнакомой областью знания.

Глава 1

Переработка знаний

Несколько лет назад я взялся за создание специализированной программы проектирования электронных печатных плат (ПП). Но вот беда — я совсем ничего не знал о конструировании электронной аппаратуры. Конечно, я имел выход на нескольких проектировщиков ПП, но от их пояснений у меня минуты за три начинала кружиться голова. Откуда же мне было взять необходимые знания, чтобы написать эту программу? Назначенный для разработки программы срок никак не позволял освоить профессию инженера-схемотехника.

Мы попытались добиться от инженеров подробного технического задания на то, что именно должна делать программа. Увы, мимо. Они были замечательными инженерами, но их представление о программировании ограничивалось считыванием текстового файла данных, сортировкой, выводом его с кое-какими дополнениями, и генерированием отчета. Это было не то, что требовалось для резкого повышения производительности труда, которого они ожидали от программы.

Первые несколько рабочих совещаний оказались бесполезны. Но тут вдруг забрезжила надежда: подсказку дали сами отчеты, которые хотели получить инженеры на выходе. Там всегда присутствовала некая "цепь" (*net*) и разнообразная информация о ней. В этой области знания цепь представляет собой электропроводник, соединяющий произвольное количество компонентов на плате и подводящий к ним электрический сигнал. Так у нас появился первый элемент модели предметной области.

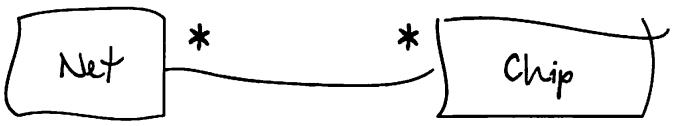


Рис. 1.1.

По мере обсуждения того, что же должна делать программа, я начал рисовать схемы сам. Для перебора возможных сценариев я использовал неформальный вариант схемы взаимодействия объектов.

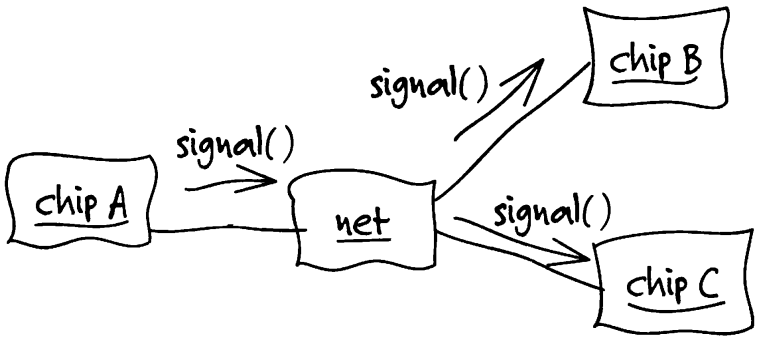


Рис. 1.2.

Специалист 1. Здесь компоненты — не обязательно микросхемы (чипы, *chips*).

Программист (я). Так что, называть их просто "компонентами"?

Специалист 1. Мы их называем "элементами"[1]. В схеме может быть много одинаковых элементов, представляющих один и тот же тип компонента.

Специалист 2. И цепь тоже нарисована как элемент.

Специалист 1. Он не пользуется нашими обозначениями. У него на схеме одни квадратики.

Программист. Увы, это так. Давайте я поясню свои обозначения.

Специалисты постоянно поправляли меня, и в процессе я начал учиться. Мы устранили нестыковки и двусмысленности в их терминологии, а также несогласие по техническим вопросам, и тут начали учиться уже они. Их объяснения стали более точными, единообразными, и нашими совместными усилиями начала рождаться модель.

Специалист 1. Недостаточно сказать, что сигнал прибывает на *Ref-Des*[2], нужно еще знать вывод (*pin*).

Программист. Какой еще *Ref-Des*?

Специалист 2. Это то же самое, что и элемент. Так он называется в одной программе, которой мы пользуемся.

Специалист 1. В общем, цепь соединяет определенный вывод одного элемента с определенным выводом другого.

Программист. Вы хотите сказать, что вывод принадлежит только одному элементу и подключается только к одной цепи?

Специалист 1. Да, именно так.

Специалист 2. А еще у каждой цепи есть топология, т.е. система, по которой соединяются друг с другом элементы цепи.

Программист. Ладно, что скажете о такой схеме?

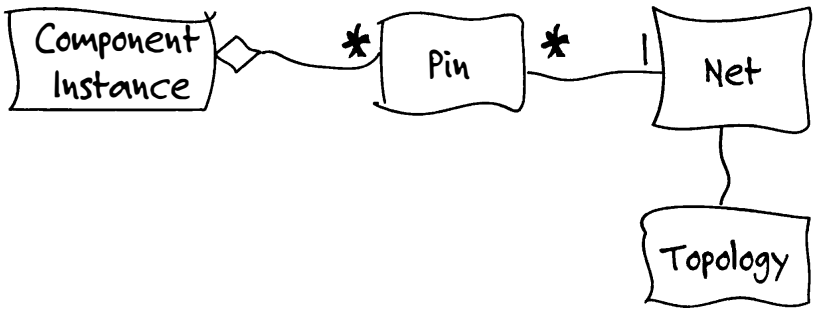


Рис. 1.3.

Чтобы сосредоточиться на главном, мы на время ограничились изучением одной конкретной функции, а именно программного прозванивания цепи (*probe simulation*) — отслеживания распространения сигнала с целью обнаружения тех или иных проблем в принципиальной схеме.

Программист. Я понимаю, как сигнал передается **Цепью** на все соединенные с ней **Выводы (Pins)** а вот идет ли он куда-нибудь дальше? Имеет ли к этому отношение **Топология (Topology)**?

Специалист 2. Нет. Сигнал пропускает сквозь себя компонент.

---

[1] Специалисты на самом деле употребляют в тексте словосочетание *component instance*, "экземпляр компонента". — *Примеч. перев.*

[2] Сокращение от *Reference Designator* — "позиционное обозначение компонента". *Примеч. перев.*

Программист. Мы, конечно же, не будем моделировать внутреннее устройство и работу микросхемы. Это слишком сложно.

Специалист 2. А нам и не нужно. Можно сделать проще: составить список сигнальных импульсов (*pushes*), передаваемых через компонент от одних **Выводов** к другим.

Программист. Что-то вроде этого?

(Методом проб и ошибок мы наконец набросали примерный сценарий.)

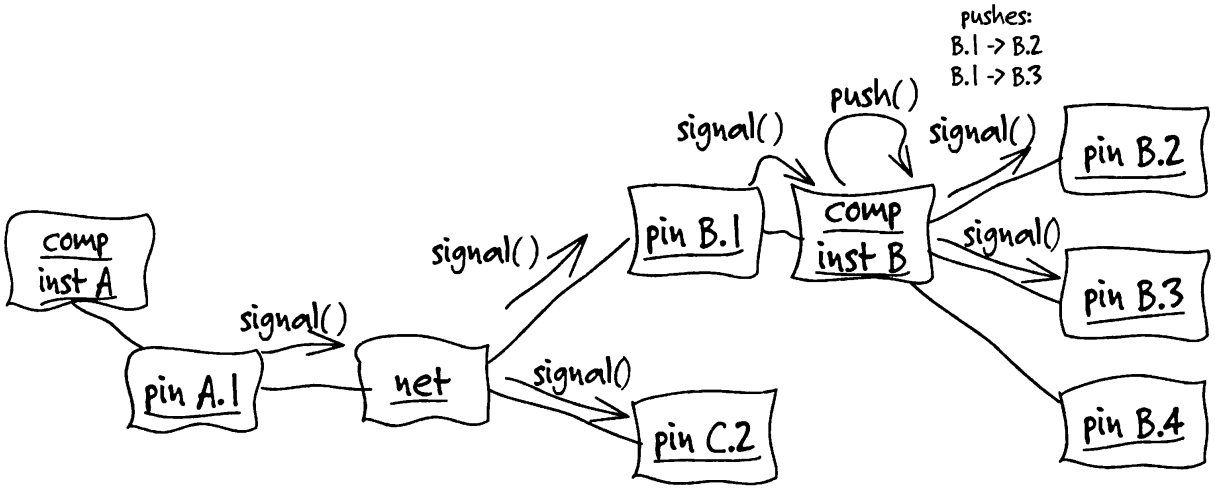


Рис. 1.4.

Программист. А что, собственно, вы хотите узнать из этого расчета?

Специалист 2. Мы ищем долгие задержки сигнала — скажем, любой путь прохождения длиннее двух или трех переприемов (*hops*). Это основное правило. Если путь прохождения сигнала слишком длинный, сигнал может не успеть поступить за один такт частоты.

Программист. Длиннее трех переприемов... То есть, нужно вычислить длину пути прохождения. А что такое "переприем"?

Специалист 2. Это один проход сигнала по **Цепи**.

Программист. То есть, мы можем передать количество переприемов, а **Цепь** увеличит его, например, так.

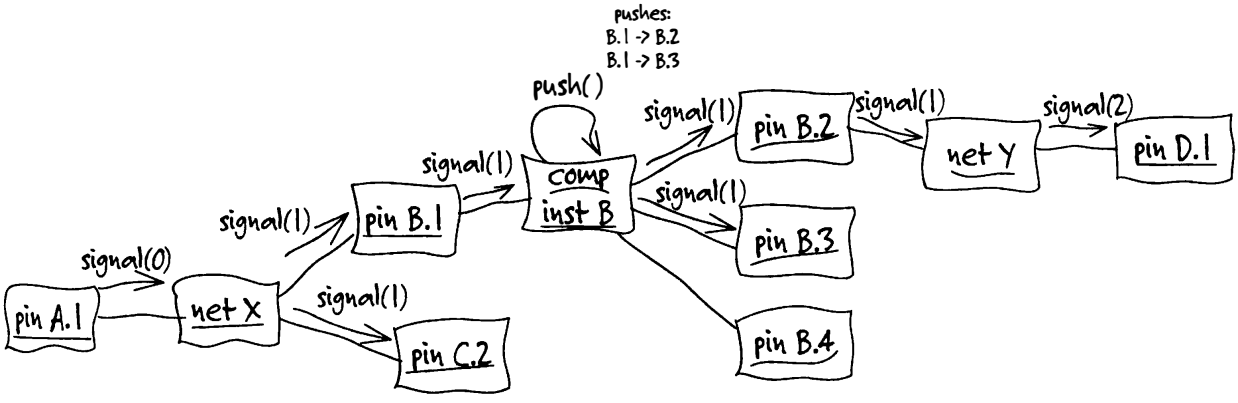


Рис. 1.5.

Программист. Единственное, что мне пока неясно это откуда возьмутся "сигнальные импульсы". Нам что же, придется хранить такие данные для каждого **Элемента**?

Специалист 2. Сигнальные импульсы будут одними и теми же для всех элементов одного компонентного типа.

Программист. Итак, тип компонента определяет набор сигналов. Они будут одними и теми же для всех элементов этого типа?

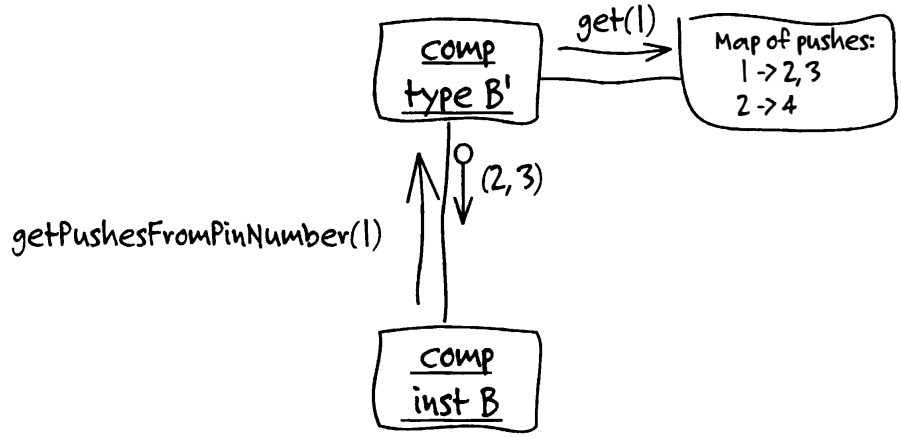


Рис. 1.6.

Специалист 2. Я не все тут до конца понял, но могу согласиться, что сохранение набора сигнальных импульсов для каждого компонента будет иметь примерно такой вид.

Программист. Прошу прощения, тут я немножко углубился в детали. Задумался и увлекся... Так где тут у нас участвует **Топология**?

Специалист 1. В прозванивании топология не используется.

Программист. Тогда я пока ее опущу, хорошо? Когда дело дойдет до следующих функций, мы вернем ее на место.

И вот дело пошло (кстати, с гораздо большим числом заминок, чем здесь описано). Мозговой штурм и усовершенствование модели, вопросы и объяснения... Модель развивалась вместе с моим пониманием предметной области, а инженеры стали лучше осознавать, как именно модель может послужить конечной цели. Диаграмма классов, представляющая эту первоначальную модель, выглядит примерно так.

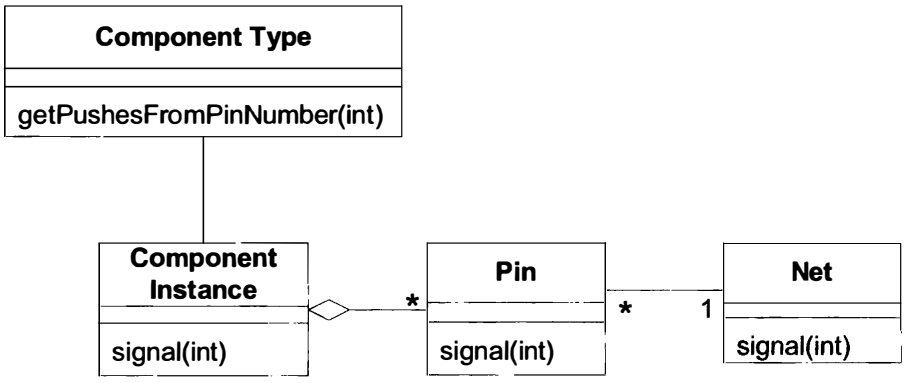


Рис. 1.7.

Проведя еще пару неполных рабочих дней за этими занятиями, я почувствовал в себе силы написать кое-какой код. Был создан очень простой прототип, помещенный в автоматизированную среду тестирования. В нем не было ни инфраструктуры, ни интерфейса пользователя, ни даже непрерывного рабочего цикла. Благодаря этому мне удалось сосредоточиться на расчетной части и продемонстрировать расчет прозванивания уже через несколько дней. Хотя данные использовались условно-тестовые, а результаты выводились на текстовую консоль, тем не менее в программе выполнялся реальный расчет

длины путей прохождения с помощью объектов Java. Сами объекты выражали собой модель, которую мы построили совместно со специалистами в предметной области.

Реальность этого прототипа показала инженерам значимость модели и ее связь с непосредственно работающим программным обеспечением. С этого момента наши обсуждения модели оживились, так как инженеры наглядно увидели, как я встраивал новоприобретенные знания в модель, а затем и в программу. Кроме того, они получили выдаваемые прототипом практические результаты и смогли оценить качество своего собственного анализа.

В описанную модель, которая разрослась до значительно больших объемов, чем было показано выше, внедрены знания о проектировании печатных плат, относящиеся непосредственно к решаемым программой задачам. В ней сконцентрировалось множество синонимичных понятий и небольших вариаций в описаниях. За пределами модели остались сотни фактов, хорошо известных инженерам, но непосредственно не относящихся к задаче — например, фактические числовые характеристики компонентов. Программист наподобие меня мог взглянуть на схемы и за несколько минут начать понимать, для чего написана эта программа. Он сразу же получал контекстную среду для организации новой информации и ускорения обучения, возможность точнее определять, что важно, а что нет, плодотворнее общаться с инженерами-схемотехниками, проектирующими ПП.

По мере того как инженеры описывали мне новые нужные им функции, я заставлял их проходить пошаговые сценарии взаимодействия объектов. Если объекты модели не справлялись с задачей прохождения того или иного алгоритма, то мы изобретали новые или вносили изменения в старые, перерабатывая новые знания в полезные функции. Мы улучшали модель, и вместе с ней эволюционировал код. Несколько месяцев спустя инженеры получили мощный программный инструмент, который превзошел их ожидания.

Составляющие эффективного моделирования

Чтобы достичь успеха, который я описал выше, мы фактически проделали следующее.

1. *Установили связь между моделью и ее реализацией.* Уже в самом грубом прототипе присутствовала существенная связь с действительностью, и после всех последующих доработок эта связь сохранилась.
2. *Ввели в обиход язык, основанный на модели.* Вначале инженерам приходилось объяснять мне элементарные понятия схемотехники, а мне — объяснять им, что такое схема взаимоотношений классов. Но по ходу проекта мы смогли вооружиться понятиями, взятыми из самой модели, составить из них предложения, согласующиеся со структурой модели, и в результате достичь полного понимания без переводчика.
3. *Разработали информоемкую модель.* У объектов есть заданные правила поведения и ограничения. Наша модель была не просто схемой хранения данных, а целостной методикой решения сложных задач. Она содержала в себе и обобщала обилие информации разного рода.
4. *Занимались дистилляцией модели.* По мере того как создание модели близилось к завершению, в нее добавлялись важные понятия. Но также важно и то, что понятия, оказавшиеся бесполезными или второстепенными, были отброшены. Если бесполезное понятие было тесно связано с необходимым, то находилась новая модель, в которой существенное понятие легко отделялось, а второстепенное отбрасывалось.
5. *Экспериментировали и проводили мозговые штурмы.* Наличие общего языка, набросков схем и готовности к интеллектуальным прорывам превратили наши дискуссии в

экспериментальную лабораторию по разработке модели, где "обкатывались" и оценивались сотни опытных вариантов. В процессе пошагового исследования рабочих сценариев даже устные высказывания участников срабатывали как быстрые тесты на пригодность предлагаемой модели, поскольку ухо человека легко отличает ясность и простоту от косноязычия.

В общем, именно творческий процесс мозгового штурма и интенсивного экспериментирования, облеченный в форму единого языка модели и направляемый обратной связью от программной реализации, сделал возможным создание информоемкой модели и ее дистилляцию. Знания, имеющиеся у коллектива разработчиков, превращаются в ценные для работы модели именно в результате такого процесса *переработки знаний* (*knowledge crunching*)[3].

Переработка знаний

Финансовые аналитики обрабатывают цифры. Они перелопачивают горы подробной документации с количественными данными, комбинируя числа так и эдак в поисках скрытого смысла, желая найти такую простую форму для представления важнейшей сути информации, чтобы она стала основой для принятия финансового решения.

Квалифицированные специалисты по моделированию предметных областей обрабатывают и перерабатывают знания. Они погружаются в поток информации и ищут в нем самую важную струю. Они испытывают один метод организации за другим, стараясь найти простое представление, которое бы придало смысл всему этому массиву данных. Пробуется, отвергается, видоизменяется множество моделей. Успех приходит в виде новой системы абстрактных понятий, учитывающей все необходимые подробности. Проделанная таким образом дистилляция дает строго сформулированное знание, наиболее важное в данной области деятельности и очищенное от всего лишнего.

Переработка знаний не осуществляется в одиночку. Для этой цели разработчики программы сотрудничают со специалистами в предметной области — как правило, руководя этим процессом. Вместе они привлекают информацию и перерабатывают ее в удобную форму. Исходное информационное сырье приходит от специалистов в предметной области, пользователей существующих систем, из опыта технических сотрудников проекта по работе с аналогичными, более ранними системами или другими проектами в той же области знания. Информация поступает в виде проектной или деловой документации, а также бесконечных устных дискуссий. Ранние версии программ или их прототипы дают группе обратную связь и помогают видоизменять ее взгляды на проблему.

В старой "каскадной методике" (*waterfall method*) специалисты по предметной области выдавали информацию аналитикам; те ее поглощали и абстрагировали, а затем передавали результат программистам, которые и писали программы. Этот подход неудачен, поскольку совершенно лишен обратной связи. Создание модели целиком возложено на аналитиков, а исходными данными им служит только информация от специалистов. Аналитики не имеют возможности научиться чему-либо у программистов или проанализировать результаты работы черновых версий программы. Знания протекают в одном направлении, но не накапливаются.

---

[3] "Переработку" тут следует понимать как в словосочетании "перерабатывающая промышленность": из набора знаний, существующего в нескольких головах и источниках, изготавливается новый интеллектуальный продукт — строгая модель, систематизирующая эти знания. В оригинальном словосочетании *knowledge crunching* имеется и несколько гастрономический оттенок пережевывания и переваривания. — *Примеч. перев.*

В некоторых других проектах используется итерационный процесс, но там знание тоже не накапливается, поскольку нет абстрагирования. Разработчики просят специалистов описать им требуемую функцию программы. Потом реализуют ее, показывают специалистам результат и спрашивают, что делать дальше. Если программисты практикуют рефакторинг, то они способны удержать свою программу в достаточно строгих стилистических рамках, чтобы успешно ее дорабатывать. Но если программисты не интересуются предметной областью как таковой, то узнают только то, что должна делать их программа, а не принципы, на которых основаны ее операции. Таким способом вполне можно написать полезную программу, но подобный проект никогда не достигнет такого уровня, на котором сами собой появляются новые мощные возможности как следствие уже реализованных.

Для хорошего программиста совершенно естественно стремление к абстрагированию и построению расширенной модели, которая способна была бы на большее, чем от нее требуется. Но если такую работу делают только разработчики, без сотрудничества со специалистами в предметной области их понятие о предмете остается наивно-поверхностным. А недостаток знаний порождает программы, которые выполняют то, что от них требуется, но которые при этом лишены глубокой логической связи с образом мышления специалиста.

Взаимодействие между членами рабочей группы меняет свой характер, по мере того как они совместно выстраивают модель. Постоянное совершенствование модели предметной области заставляет программистов осваивать фундаментальные принципы той области деятельности, для которой они пишут, а не просто механически реализовывать функции. Часто и специалисты тоже совершенствуют свое собственное понимание, так как их вынуждают делать из своих познаний краткую выжимку, оставляя самое важное. Так они приходят к пониманию того, какая строгость понятий нужна для реализации программного проекта.

Все это превращает группу разработчиков в квалифицированных “переработчиков знаний” (*knowledge crunchers*). Они отсеивают все лишнее и переделывают модель во все более полезную форму. Благодаря вкладу аналитиков и программистов модель имеет четкую организацию и должный уровень абстракции, что делает ее ценным инструментом для программной реализации. Благодаря вкладу специалистов в предметной области модель отражает глубокие знания из этой области. Абстрагированные понятия модели точно соответствуют основным положениям предметной области.

По мере улучшения модели она сама становится средством организации информации, которая продолжает поступать для проекта. Она позволяет сосредоточить анализ технических требований на главном. Модель тесно связана с процессом программирования и архитектурного проектирования. В идеальном рабочем цикле такая модель углубляет знания членов группы о предметной области, дает им более ясное видение проблемы и приводит к дальнейшему усовершенствованию самой же модели. Совершенства модели никогда не достигают, но они эволюционируют. От них требуется практическая применимость при освоении прикладной области знаний, а также достаточная строгость, благодаря которой программный код был бы прост в реализации и понимании.

Непрерывное обучение

Начиная писать программу, невозможно знать достаточно. Знания, относящиеся к проекту, всегда фрагментированы, разбросаны по документам и памяти разных людей, а также смешаны с посторонней информацией, из-за чего даже нельзя знать наверняка, что именно нам понадобится. Если какая-нибудь предметная область на первый взгляд

и кажется не слишком сложной технически, она еще может преподнести сюрприз — заранее не известно, что именно из нее мы не знаем. А невежество заставляет делать ложные предположения.

Между тем в любом проекте существующая база знаний может “дать течь”. Например, уволились сотрудники, которые уже чему-то научились. Или рабочая группа реорганизовалась, и накопленные ею знания снова стали фрагментарными. Или, скажем, критически важные подсистемы отданы для разработки субподрядчикам, и те присылают код, но не передают соответствующие знания, лежащие в его основе. Как правило, проектирование программ выполняется так, что ни код, ни документация не выражают знания, обретенные тяжкими трудами, в явной, понятной форме. Если по какой-то причине прекращается устная передача знаний в группе, это означает их потерю.

Высокопроизводительные группы разработчиков наращивают свои знания сознательно, практикуя *непрерывное обучение* [16]. Для разработчиков программного обеспечения это означает совершенствование своих технических навыков и знаний в области программирования, а также общих навыков моделирования предметных областей (например, таких, как описаны в этой книге). Но сюда же входит и серьезное изучение предметной области, с которой они работают в текущий момент.

Разработчики, склонные к самообразованию, образуют наиболее стабильное ядро группы, поэтому их ставят на самые ответственные этапы разработки. (Более подробно об этом см. главу 15.) Накопленные этим ядром группы знания дают возможность перерабатывать дальнейшую информацию более эффективно.

В этом месте стоит остановиться и задать себе вопрос: вы узнали что-нибудь о проектировании печатных плат из того, что было сказано выше? Пусть этот пример и был поверхностным, но при обсуждении модели всегда происходит обучение. Лично я узнал очень много нового. Я не стал инженером-схемотехником, но такая задача и не ставилась. Зато я научился разговаривать со специалистами, овладел основными понятиями, существенными для работы программы, и проверил на прочность то, что мы создавали.

В действительности группа разработки вскоре узнала, что функция прозванивания цепей не относится к наиболее важным, и со временем от ее реализации вообще отказались. Вместе с ней отказались от тех элементов модели, которые описывали прохождение сигнальных импульсов через компоненты и подсчет количества переприемов. Основное смысловое ядро программы, как оказалось, состояло в другом. Поэтому модель изменилась так, чтобы на первое место вышли самые нужные аспекты. Специалисты тоже кое-чему научились и более четко сформулировали цель работы программы. (Все это подробно рассматривается в главе 15.)

Даже при этом работа, проделанная на первом этапе, не пропала впустую. Сохранились ключевые элементы модели, но что еще важнее, был запущен процесс переработки знаний, который сделал всю последующую работу эффективной: накопление знаний и разработчиками, и специалистами в предметной области, и другими членами группы; создание основ единого языка; замыкание цепи обратной связи через программную реализацию. В конце концов, путешествие в неведомое должно с чего-то начинаться.

Информоемкая архитектура

Та разновидность знания, которая заключена в модели из приведенного примера с проектированием печатных плат, далеко не ограничивается набором названий и определений. Для предметной области не менее, чем объекты, важны операции и правила; во всякой области знания имеются различные категории понятий. При правильной переработке знаний возникают модели, в которых все это учитывается. Параллельно с модификацией

модели разработчики выполняют рефакторинг программной реализации для отражения этой модели, т.е. обрабатывают сформированные знания посредством программы.

Переработка знаний становится нетривиальным процессом именно тогда, когда сами знания представляют собой нечто большее, чем набор объектов и числовых показателей — ведь в заданных правилах делового регламента (алгоритмах операций, соотношениях между понятиями предметной области)[4] могут иметься противоречия. Специалисты обычно не осознают, насколько сложны происходящие у них в уме процессы: в ходе работы применяются соотношения и правила, устраняются противоречия, пробелы в знаниях заполняются с помощью интуиции и здравого смысла. Но программа ничего этого не умеет. Соотношения и правила предметной области должны проясняться, конкретизироваться, очищаться от противоречий или вообще отменяться в процессе совместной переработки знаний с участием программистов и специалистов.

Пример

Извлечение скрытого понятия

Начнем с очень простой модели предметной области, которая могла бы стать основой для программы бронирования грузомест в корабельных рейсах.

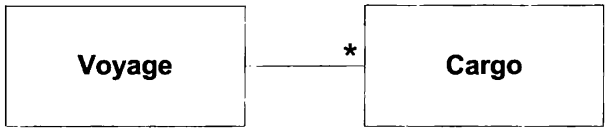


Рис. 1.8. Здесь Voyage — рейс, Cargo — груз

Можно утверждать, что в задачу программы бронирования входит ассоциировать каждый **Груз (Cargo)** с **Рейсом (Voyage)**, зафиксировать это отношение и проследить за его соблюдением. Пока все понятно. Где-то в коде программы будет присутствовать такой метод.

```
public int makeBooking(Cargo cargo, Voyage voyage) {
   int confirmation = orderConfirmationSequence.next();
   voyage.addCargo(cargo, confirmation);
   return confirmation;
}
```

Кто-то из клиентов обязательно отменит свой запрос перед самым рейсом, поэтому в индустрии грузоперевозок стало стандартным правилом принимать больше заказов на грузоместа, чем на самом деле может перевезти судно за один рейс. Такую ситуацию называют *избыточным резервированием* (*overbooking*) . Иногда для этой цели просто принимают заказы с избытком на фиксированный процент — например, на 110% от грузоподъемности судна. В других случаях применяют более сложные правила, отдавая предпочтение определенным категориям клиентов или грузов.

---

[4] Тут и далее у автора часто употребляется выражение *business rule*. Это соотношение между объектами, заданное ограничение или правило в соответствующей области знания. В теории автоматизированных систем управления (АСУ) есть термин "деловой регламент" (алгоритм реализации операций в рамках АСУ предприятия), поэтому *business rule* соответствует выражению "правило делового регламента". — *Примеч. перев.*

Если это стандартная практика в области грузоперевозок, то о ней будет известно любому работнику этой отрасли, но совсем не обязательно программистам из группы разработчиков.

Техническое задание на программу будет содержать такую строку.

Разрешить избыточное резервирование в размере 10%.

Диаграмма классов и код приобретут такой вид.

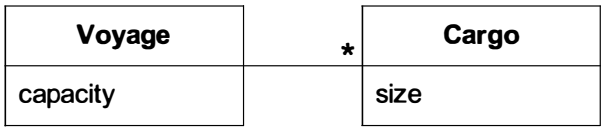


Рис. 1.9. Здесь Capacity — грузоподъемность, size — размер груза

```
public int makeBooking(Cargo cargo, Voyage voyage) {
   double maxBooking = voyage.capacity() * 1.1;
   if ((voyage.bookedCargoSize() + cargo.size()) > maxBooking)
        return -1;
   int confirmation = orderConfirmationSequence.next();
   voyage.addCargo(cargo, confirmation);
   return confirmation;
}
```

Теперь важное регламентное правило предметной области скрыто в методе приложения в виде контрольных операторов. Позже, в главе 4, мы рассмотрим принцип МНОГОУРОВНЕВОЙ АРХИТЕКТУРЫ. Согласно этому принципу, правило избыточного резервирования надо было бы переместить в объект предметной области, но пока давайте сосредоточимся на том, как придать этому знанию более явную форму и предоставить удобный доступ к нему всем участникам проекта. Итоговый результат будет примерно таким же.

1. Маловероятно, чтобы специалисту в предметной области, даже с помощью программиста, было бы удобно анализировать это правило в том виде, в каком оно сейчас внедрено в код.
2. Техническому персоналу, не связанному с этой отраслью деятельности, было бы трудно правильно выразить требование из технического задания в коде программы.

Будь правило посложнее, эти трудности увеличились бы соответственно.

Изменим архитектуру кода так, чтобы в ней лучше отображалось имеющееся знание. Правило избыточного резервирования представляет собой *стратегию*[5] (*policy*). Это и название архитектурного шаблона, известного как STRATEGY (Gamma и др., 1995). Его применение обычно мотивируют необходимостью подставлять различные правила; здесь это, насколько можно судить, не требуется. Но понятие, которое мы стараемся выразить, действительно подходит *по смыслу* под определение стратегии, а это уже достаточная мотивация в DDD. (См. главу 12.)

Код приобретет следующий вид.

```
public int makeBooking(Cargo cargo, Voyage voyage) {
   if (!overbookingPolicy.isAllowed(cargo, voyage)) return -1;
```

[5] Нельзя согласиться с прямолинейным переводом этого слова как "политика", пусть даже кое-где оно уже и прижилось. Это стратегия, методика, принятая процедура, образ действий, регламент. — *Примеч. перев.*

```
    int confirmation = orderConfirmationSequence.next();
    voyage.addCargo(cargo, confirmation);
    return confirmation;
}
```

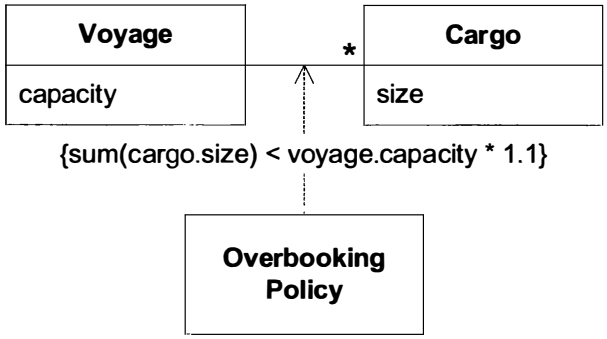


Рис. 1.10. Здесь ***Overbooking Policy — Стратегия избыточного резервирования***

Новый класс `OverbookingPolicy` будет содержать такой метод.

```
public boolean isAllowed(Cargo cargo, Voyage voyage) {
    return (cargo.size() + voyage.bookedCargoSize()) <=
           (voyage.capacity() * 1.1);
}
```

Теперь всем ясно, что избыточное резервирование — это принятая и регламентированная процедура, а реализация ее сделана явной и отдельной.

Следует отметить, что *не рекомендуется применять столь придирчивый подход к проектированию каждой мелочи в программной реализации предметной области.* В главе 15, посвященной дистилляции, будет подробно рассмотрен вопрос о том, как сосредоточиться на главном и минимизировать или отделить все остальное. А этот пример призван показать, что модель предметной области и соответствующая архитектура программы помогают зафиксировать знание в доступной форме. Такой подход имеет следующие преимущества.

1. Чтобы довести архитектуру до такого уровня, программистам и остальным участникам проекта придется осознать, что, по сути, избыточное резервирование это четко сформулированное и важное регламентное правило, а не просто некая маловразумительная вычислительная операция.
2. Программисты смогут показывать специалистам в предметной области свои технические наработки, вплоть до кода, и те поймут, о чем идет речь (конечно, при помощи и под руководством программистов). Этим самым будет замкнута цепь обратной связи.

Углубленные модели

Полезные модели редко лежат на поверхности. По мере роста понимания предметной области и необходимых функций программы в модели обнаруживаются второстепенные элементы, которые поначалу казались важными, а теперь либо отбрасываются, либо меняют статус. Часто возникают соображения, которые в начале работы вообще не приходили в голову или казались неважными, а со временем становятся “во главу угла”.

Приведенный выше пример в какой-то мере основан на одном из проектов, которым я занимался и которым воспользуюсь еще для нескольких примеров в этой книге: системе обслуживания контейнерных перевозок. Примеры в книге будут, конечно, излагаться так, чтобы их понял не только специалист по грузоперевозкам. Но в реальном проекте, в котором разработчики хорошо подготовлены благодаря непрерывному обучению, не приходится бояться сложности ни в самом предмете, ни в методах моделирования, чтобы получить мало-мальски полезную и строгую рабочую модель.

В упомянутом проекте было ясно, что операция грузоперевозки начинается с бронирования грузоместа. Вот мы и разработали модель, которая позволяла описать груз, маршрут его следования и т.п. Все это было необходимо и полезно, но специалисты по перевозкам были недовольны. Мы никак не могли посмотреть на эту работу их глазами.

Со временем, после нескольких месяцев переработки информации, мы осознали, что манипуляции с грузами, физическая разгрузка и погрузка, перемещения с места на место выполнялись то субподрядчиками, то ответственными сотрудниками компании. С точки зрения специалистов по перевозкам, этот процесс представлял собой цепочку передачи ответственности между различными юридическими лицами. Процедура перевозки представляла собой именно управление передачей юридической ответственности и технических полномочий от грузоотправителя некоему местному перевозчику, затем от одного перевозчика другому и, наконец, грузополучателю. Часто груз просто лежал на складе, а тем временем предпринимались важные шаги. В другой раз груз подвергался многочисленным сложным манипуляциям, но от компании-грузоотправителя это не требовало никаких дополнительных действий или решений. В итоге на первый план в модели вышли не физические операции на маршруте следования груза, а документация (накладные, коносаменты) и процедуры оплаты услуг.

Углубленное понимание сути грузоперевозок не привело к удалению объекта **Маршрут**, но модель изменилась очень сильно. Мы стали воспринимать перевозки не как перемещение контейнеров с места на место, а как передачу полномочий и ответственности за груз от одного юридического лица другому. Функции управления этой передачей больше не были “неуклюжими придатками” к операциям погрузки-разгрузки, а органично поддерживались в самой модели, которая и выросла из понимания существенной взаимосвязи между физическими операциями и юридической ответственностью.

Переработка и усвоение знаний — это процесс творческий и исследовательский, и куда он может завести, заранее сказать нельзя.

Глава 2

Коммуникация и язык

Модель предметной области может служить основой общего языка для коммуникации в рамках проекта по разработке программного обеспечения. Модель — это набор понятий, существующих в головах у создателей проекта, вместе с названиями (терминами), отношениями и взаимосвязями, отражающими их понимание предмета. Термины и взаимосвязи образуют семантику языка, адаптированного к предметной области, но достаточно точного и для технических нужд разработки. Это та нить, которая соединяет модель с кодом и позволяет ей органично вплетаться в процесс разработки.

Коммуникация на основе модели не ограничивается рисованием диаграмм на унифицированном языке моделирования (*Unified Modeling Language, UML*). Для наиболее эффективного применения модели она должна буквально "пронизывать" все средства коммуникации проекта. Благодаря ей повышается полезность как текстовой документации, так и неформальных диаграмм или дискуссий, важность которых подчеркивается в гибких (*agile*) методологиях разработки. Улучшается качество коммуникации как для самого кода, так и для его тестов.

Итак, роль общего языка в проекте хотя и не всегда осознается, но является основополагающей.

Единый язык

Ты должен фразу написать,
Нарезать на слова,
И как попало разбросать,
Перемешав сперва.
Порядок слов не важен тут,
И не нужна канва.

Льюис Кэрролл, "Poeta Fit, Non Nascitur"[6]

Чтобы разработать гибкую, информоемкую архитектуру программы, группе разработчиков необходимо иметь богатый возможностями общий язык и активно экспериментировать с ним, что редко происходит в программных проектах.

Специалисты в предметных областях не слишком хорошо понимают технический жаргон разработчиков, но сами при этом пользуются собственным профессиональным жаргоном — даже несколькими его "диалектами". С другой стороны, разработчики могут понимать и обсуждать систему в описательных и функциональных терминах, никак не привязываясь к тому смыслу, который в свой язык вкладывают специалисты. Могут они и создавать абстракции, полезные для проектирования программы, но непонятные специалистам-предметникам. Программисты, работающие над разными частями проекта, к тому же вырабатывают собственные архитектурные концепции и способы описания предметной области.

[6] "Поэтами становятся, а не рождаются" (*лат.*). Перевод М. Бородицкой.

Учитывая все эти языковые барьеры, специалисты с трудом могут описать, что же им нужно. Программисты, пытаясь разобраться в новом для них предмете, с трудом их понимают. Пусть нескольким членам группы и удается стать двуязычными, но они оказываются в роли узких теснин в бурном потоке информации, а их перевод не всегда оказывается точным.

В проекте, в котором нет общего языка, программистам приходится служить переводчиками для специалистов из предметной области. Специалисты выполняют перевод между программистами и другими специалистами. Программисты переводят даже друг для друга. Перевод затуманивает концепции модели, и в результате коду грозит разрушительный рефакторинг. Непрямой характер коммуникации скрадывает появление "еретиков" — разные члены группы употребляют термины в разном смысле, сами того не понимая. Программа от этого становится ненадежной, а разные ее части — плохо совместимыми друг с другом (см. главу 14). Необходимость в переводе ухудшает обмен знаниями и идеями — основной источник глубокого понимания модели.

Проект сталкивается с серьезными проблемами, когда в нем отсутствует единый язык. У специалистов в предметной области есть свой жаргон, а у разработчиков — собственный язык, приспособленный для описания предметной области в терминах программной архитектуры.

Терминология повседневных дискуссий оторвана от терминологии, внедренной код (а это, в конце концов, главный продукт, создаваемый в проекте). Даже один и тот же человек пользуется двумя разными языками на письме и в разговоре, так что самые критические положения предметной области часто выражаются в промежуточной форме, не фиксируемой в коде или даже в письменной документации.

Необходимость в переводе затрудняет коммуникацию и ослабляет интенсивность переработки знаний.

Ни один из диалектов задействованных сторон не может служить общим языком, поскольку ни один из них не служит всем поставленным целям.

Объем требуемого перевода плюс риск неправильного понимания — это слишком высокая цена. Проект нуждается в общем языке, который был бы более устойчив, чем наименьший общий знаменатель. Если группа разработки приложит сознательные усилия, она может сделать основой такого языка модель предметной области, одновременно привязывая весь процесс коммуникации в группе к реализации программного продукта. Такой язык может стать единым для всех видов работ в группе.

Словарь этого ЕДИНОГО ЯЗЫКА состоит из имен классов и основных операций. Язык содержит термины для обсуждения правил, явно сформулированных в модели. Дополнением к ним служат термины из принципов высокоуровневой организации, которым подчинена модель (например, КАРТЫ КОНТЕКСТОВ и крупномасштабные структуры, рассматриваемые в главах 14 и 16). Наконец, этот язык обогащается названиями шаблонов, которые группа разработчиков применяет к модели.

Отношения в модели становятся комбинаторными правилами, содержащимися во всех языках. Значения слов и фраз отражают семантику модели.

Язык на основе модели должен использоваться среди разработчиков для описания не только объектов системы, но также задач и функциональных возможностей. Одна и та же модель должна давать разработчикам и специалистам в предметной области язык для общения друг с другом, а также позволять специалистам обсуждать между собой вопросы технических требований, планирования разработки, функциональных возможностей программы. Чем шире применяется единый язык в проекте, тем легче наладить плодотворное общение между его участниками.

По крайней мере, так должно быть. Но в самом начале модель может просто не быть готовой к тому, чтобы выполнять все эти функции. Ей может недоставать семантического богатства специализированного жаргона из предметной области. Но такие жаргоны и нельзя там использовать в чистом виде, ведь в них содержатся противоречия и двусмысленности. Профессиональному языку может не хватать гибкости и активности, вложенной программистами в код — либо потому, что они не считали эти черты частью модели, либо потому, что программирование носит процедурный характер и отражает понятия модели только в неявной форме.

Может показаться, что возникает замкнутый круг. Но процесс переработки знаний может породить и более качественную модель, если разработчики в достаточной мере привержены единому языку на основе модели. Настойчивое употребление ЕДИНОГО ЯЗЫКА обязательно выявит слабые места в модели, и группа путем эксперимента найдет альтернативу неуклюжим понятиям или соотношениям. По мере обнаружения пробелов в языке будут появляться новые слова. *Изменения в языке следует принимать как изменения в модели* — соответственно группа разработчиков должна вносить изменения в диаграммы классов, переименовывать классы и методы в исходном коде или даже изменять функции программы при изменении значения того или иного термина.

Последовательно пользуясь этим языком в контексте программной реализации, программисты обнаружат неточности и противоречия, заставив специалистов искать приемлемые альтернативы.

Разумеется, специалисты в предметной области будут общаться не только на ЕДИНОМ ЯЗЫКЕ; это нужно для целей объяснения или создания более широкого контекста. Но в пределах области действия модели они должны пользоваться этим ЯЗЫКОМ и бить тревогу, если язык оказывается неудобным или неполным, а то и неправильным. Постоянно пользуясь языком на основе модели и неустанно его совершенствуя, мы приближаемся к модели одновременно полной и доступной пониманию, составленной из простых элементов, сочетание которых способно выражать сложные идеи.

Используйте модель как основу для языка. Побуждайте разработчиков пользоваться им во всех видах внутригруппового взаимодействия, а также в коде. Пользуйтесь одним и тем же языком в диаграммах, письменной документации и особенно при разговоре.

Устраняйте трудности путем экспериментирования с альтернативными выражениями, отражающими альтернативные модели. Затем выполняйте рефакторинг кода, переименовывайте классы, методы и модули, чтобы добиться соответствия новой модели. Ликвидируйте путаницу в терминологии путем устных обсуждений — таким же образом, как мы приходим к согласию о значении обычных слов.

Осознайте, что изменения в ЕДИНОМ ЯЗЫКЕ **— суть изменения в модели.**

Специалисты в предметной области должны возражать против терминов или структур, неудобно или недостаточно передающих суть явлений из их области. А разработчикам следует отслеживать любую неоднозначность и непоследовательность, потому что из-за них пострадает архитектура программы.

При наличии ЕДИНОГО ЯЗЫКА модель перестает быть просто придатком к архитектуре, а становится естественной средой для любых совместных действий программистов и специалистов. Язык есть носитель динамической формы знания. Дискуссии на этом ЯЗЫКЕ извлекают “на поверхность” смысл, заключенный в диаграммах и коде.

* * *

В этом рассуждении о ЕДИНОМ ЯЗЫКЕ предполагается, что в работу вовлечена всего одна модель. В главе 14 рассматривается случай сосуществования нескольких разных моделей (и ЯЗЫКОВ), а также вопрос, как удержать модель от распада на части.

ЕДИНЫЙ ЯЗЫК — это основной носитель тех аспектов архитектуры, которые не проявляются в коде, а именно крупномасштабных структур, организующих всю систему (см. главу 16), ОГРАНИЧЕННЫХ КОНТЕКСТОВ, которые определяют отношения между разными системами и моделями (см. главу 14), и других шаблонов, относящихся к модели и программной архитектуре.

Пример

Разработка программы маршрутизации грузов

В следующих двух диалогах имеются небольшие, но важные различия. В каждом из приведенных ниже сценариев обратите внимание, сколько внимания собеседники уделяют роли программы в реальной деятельности по сравнению с техническими подробностями ее работы. Говорят ли пользователь и программист на одном языке? Достаточно ли этот язык богат для поддержания дискуссии о том, что должна делать программа?

Сценарий 1: минимальное абстрагирование предметной области

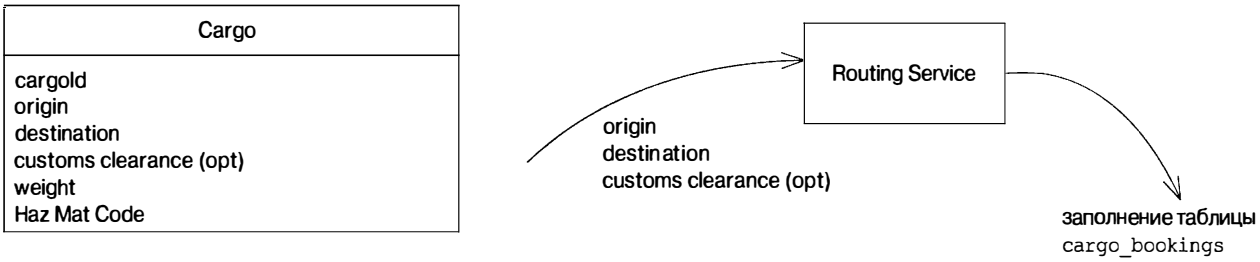


Таблица БД: `cargo_bookings`

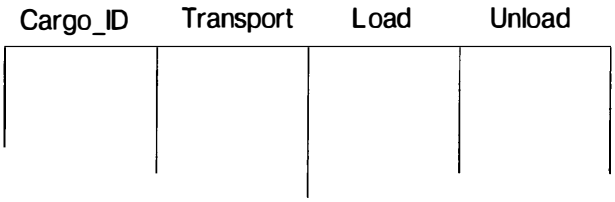


Рис. 2.1.

Пользователь. Итак, если мы меняем пункт растаможивания, то нужно менять весь план перевозки.

Программист. Да. Тогда мы удаляем все строки в таблице отправки грузов с заданным идентификатором груза, потом передаем пункт отправки, пункт назначения и новый пункт растаможивания в **`Маршрутизатор (Routing Service)`** и он заполняет таблицу заново. В объект **`Груз (Cargo)`** надо добавить логический переключатель, указывающий, есть ли данные в таблице отправки.

Пользователь. Удаляем строки? Ладно, вам виднее. Так вот, если пункта растаможивания не было вообще, то делается то же самое.

Программист. Разумеется, когда меняется пункт отправки, пункт назначения или пункт растаможивания (или же он вводится первый раз), мы проверяем, есть ли данные по отправке грузов, удаляем их и просим **`Маршрутизатор`** заново сгенерировать их.

Пользователь. Но, конечно, если пункт растаможивания не менялся, то всего этого делать не нужно.

Программист. А, это не проблема. Легче просто заставить **Маршрутизатор** генерировать список погрузок-выгрузок каждый раз заново.

Пользователь. Да, но это же лишняя работа, поэтому не надо заново составлять маршрут, если этого не требуют внесенные изменения.

Программист. М-м-м... Ладно, если вы вводите пункт растаможивания впервые, нам придется послать в таблицу запрос на поиск старого пункта растаможивания, а потом сравнить его с новым. Так мы узнаем, нужно ли все переделывать.

Пользователь. Но об этом не нужно беспокоиться при вводе пункта отправки или пункта назначения, поскольку маршрут все равно изменится.

Программист. Хорошо! Не будем!

Сценарий 2: модель предметной области, специально доработанная для удобства коммуникации

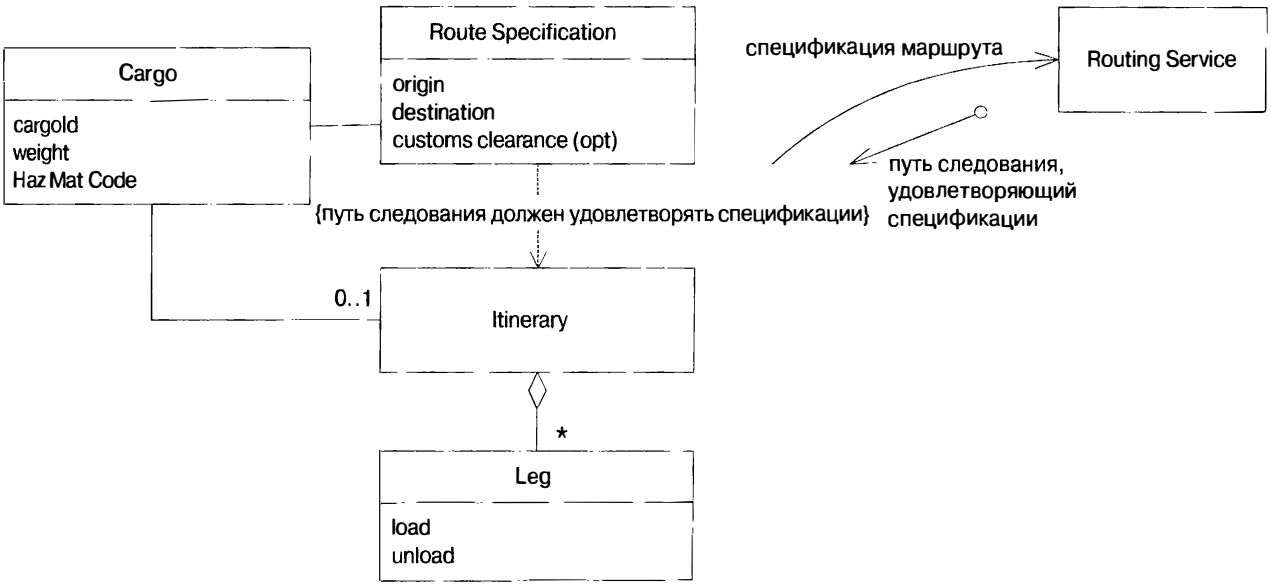


Рис. 2.2.

Пользователь. Итак, если мы меняем пункт растаможивания, то нужно менять весь план перевозки.

Программист. Да. При изменении любого из атрибутов в **Спецификации маршрута** (**Route Specification**) мы удаляем старый **Маршрут** (**Route**) и просим **Маршрутизатор** (**Routing Service**) построить новый маршрут на основе новой **Спецификации маршрута**.

Пользователь. Если пункт растаможивания ранее не был указан вообще, то это тоже нужно сделать.

Программист. Разумеется, когда что-либо меняется в **Спецификации маршрута**, **Маршрут** строится заново. Здесь учитывается и случай ввода информации первый раз.

Пользователь. Но, конечно, если старый пункт растаможивания оказался правильным, то всего этого делать не нужно.

Программист. А это не проблема. Легче просто заставить **Маршрутизатор** строить **Маршрут** каждый раз заново.

Пользователь. Да, но это же лишняя работа — составлять все нужные планы для нового **Маршрута**, поэтому не надо заново строить его, если этого не требуют внесенные изменения.

Программист. Нужно добавить кое-какие новые функции в **Спецификацию Маршрута**, тогда при внесении любых изменений в **Спецификацию** мы смотрим, соответствует ли все еще **Маршрут** этой **Спецификации**. Если не соответствует, то приказываем **Маршрутизатору** построить **Маршрут** заново.

Пользователь. Но об этом не нужно беспокоиться при вводе пункта отправки или пункта назначения, поскольку маршрут все равно изменится в этом случае.

Программист. Это так, но нам будет легче просто выполнять это сравнение каждый раз. **Маршрут** будет строиться заново только в том случае, если он больше не удовлетворяет **Спецификации**.

Второй диалог выглядит более осмысленным для специалиста по перевозкам, чем первый. Пользователь программы употреблял слово "маршрут" в обоих диалогах, но во втором случае это был объект, который они с программистом понимали одинаково и совершенно точно. Они обсуждали "спецификацию маршрута", называя ее явно, а не заменяя упоминанием всевозможных атрибутов и процедур.

Эти два диалога были специально сконструированы похожими, параллельными. В действительности первый был бы намного длиннее из-за пространных объяснений операций программы и отсутствия взаимопонимания собеседников. Терминология второй архитектуры, основанная на модели предметной области, делает второй диалог более четким и кратким.

Моделирование вслух

Отделение устной речи от других форм общения — это великая потеря, потому что мы, люди, имеем склонность к разговорам. Но, к сожалению, когда люди говорят, они не пользуются языком модели предметной области.

Это утверждение может поначалу показаться сомнительным, ведь исключения действительно бывают. Но попробуйте вслушаться в дискуссию на совещании по техническому заданию или архитектурному проектированию. Вы услышите требования к программе на профессиональном жаргоне специалистов или же подражание этому жаргону из уст неспециалистов. Еще вы услышите названия программных объектов и конкретных функций программы. Конечно, встретятся вам и термины из модели предметной области, особенно слова повседневного языка, перешедшие из профессионального жаргона в имена объектов и поэтому невольно упомянутые в дискуссии. Но услышите ли вы предложения, которые можно хотя бы приблизительно считать описаниями отношений и взаимодействий в текущей модели предметной области?

Один из лучших способов усовершенствовать модель — обкатывать ее на языке, описывая вслух разные конструкции из возможных вариантов модели. Шероховатости будут услышаны сразу же.

> *Если передать в* **Маршрутизатор** *пункт отправки, пункт назначения, время прибытия, то он найдет нужные остановки в пути следования груза, а потом, ну... запишет их в базу данных.* (Расплывчато, с техническими подробностями.)
>
> *Пункт отправки, пункт назначения и все такое... все это идет в* **Маршрутизатор**, *а оттуда получаем* **Маршрут**, *в котором записано все, что нужно.* (Уже полнее, но многословно.)
>
> **Маршрутизатор** *находит* **Маршрут**, *удовлетворяющий* **Спецификации маршрута**. (Кратко и точно.)

Жизненно важно для дела манипулировать словами и фразами, приспосабливая наши языковые способности к нуждам моделирования, точно так же как для рисования диаграмм и схем необходимо включать пространственное и визуальное воображение. В этом ряду можно назвать и аналитические способности, используемые для методичного анализа и проектирования, и загадочное "чувство" кода. Названные способы мышления дополняют друг друга, а для построения полезных моделей и ценных архитектур требуется привлекать сразу все. Из всего перечисленного экспериментирование с разговорным языком наиболее часто упускают из виду. (В части III мы погрузимся в исследовательский процесс и продемонстрируем взаимодействие на примере нескольких диалогов.)

По-видимому, наш мозг в известной степени приспособлен именно к восприятию сложного разговорного языка. Хорошо об этом для непрофессионалов вроде меня написал Стивен Пинкер (Steven Pinker) в книге *The Language Instinct* [21]. Например, если люди, происходящие из разных языковых сред, встречаются для ведения торговли, но не понимают языка друг друга, то выдумывают новый, именуемый "пиджин" (*pidgin*). Этот пиджин-язык не столь богат и всеобъемлющ, как исходные языки сторон, но свою задачу он выполняет. Когда люди беседуют, они естественным образом открывают для себя различия в интерпретациях и значениях слов и так же естественно их разрешают. Они находят нестыковки в общении и сглаживают их.

Как-то я проходил интенсивный курс испанского языка в колледже. В аудитории царило правило: во время занятий ни слова по-английски. Поначалу это раздражало, казалось неестественным и требовало самодисциплины. Но со временем мы с одногруппниками сломали языковой барьер и достигли такого уровня владения языком, какого никогда не добились бы на бумаге.

В процессе использования ЕДИНОГО ЯЗЫКА модели предметной области в дискуссиях — особенно в тех, в которых разработчики и специалисты "пережевывают" сценарии и требования — мы осваиваем язык все лучше и обучаем друг друга его нюансам. Естественным образом мы приходим к такому тесному совместному употреблению разговорного языка, какого никогда не бывает в письменной документации и диаграммах.

Внедрить ЕДИНЫЙ ЯЗЫК в проект по разработке программного обеспечения — это легче сказать, чем сделать. Для реализации этой цели приходится напрягать все способности. Точно так же, как способность человека к визуальному и пространственному мышлению позволяет быстро передавать и обрабатывать информацию, так и его природный талант к употреблению осмысленного языка со своей грамматикой можно задействовать в моделировании.

Итак, в дополнение к характеризации ЕДИНОГО ЯЗЫКА.

Говоря о системе, на словах "играйте" с моделью. Описывайте сценарии вслух с использованием элементов и взаимосвязей модели, комбинируя понятия такими способами, какие позволяет модель. Ищите более простые способы выразить то, что вы хотите выразить, и привносите найденные новые идеи в диаграммы и в код.

Одна команда — один язык

Технари-программисты часто ощущают потребность "отгородить" специалистов предметной области от ее модели, заявляя следующее.

"Она слишком абстрактна для них."
"Они не разбираются в объектах."
"Все равно техническое задание должно быть на их языке."

Это только некоторые из оправданий для наличия двух языков в группе по реализации программного проекта, которые я слышал. Забудьте их навсегда.

Разумеется, существуют такие чисто технические составляющие архитектуры программы, которые не должны волновать специалистов в предметной области, но вот смысловое ядро модели должно обязательно их интересовать. Слишком абстрактно? Тогда откуда вам знать, что эти абстракции выбраны разумно? Понимаете ли вы предмет так хорошо, как специалисты? Иногда те или иные технические требования поступают от пользователей более низкого уровня, и для них может потребоваться наличие более конкретной терминологии, но специалист на то и специалист, чтобы иметь углубленное представление о своей области деятельности. *Если квалифицированные специалисты в своей области не понимают модель, то с этой моделью что-то не так.*

В самом начале, когда пользователи обсуждают еще не смоделированные будущие возможности системы, никакой модели, которой они могли бы воспользоваться, не существует. Но как только начинается проработка основных идей совместно с программистами, начинается и движение "в сторону модели", которая была бы полезна всем сторонам. Вначале она может быть "неуклюжей" и неполной, но постепенно модель совершенствуется. По мере развития нового языка специалисты должны приложить особые усилия для овладения им, а также переработать всю старую документацию, которая еще сохранила свою ценность.

Если специалисты станут использовать этот ЯЗЫК в дискуссиях между собой или с разработчиками, они быстро обнаружат области, в которых модель неадекватна их задачам или вообще неправильна. Специалисты (при помощи программистов) также найдут и такие места, в которых благодаря точности языка, основанного на модели, обнаружатся противоречия или нечеткость в их способе мышления.

Разработчики и специалисты в предметной области могут провести неформальное тестирование модели, пройдя пошагово возможные сценарии с использованием объектов модели. Практически любая дискуссия — это возможность для разработчиков и специалистов-пользователей вместе "покрутить" модель, углубляя понимание каждой из сторон и по ходу совершенствуя понятийный аппарат.

Специалисты в предметной области могут воспользоваться языком модели для написания сценариев использования[7] (*use cases*), а могут работать с моделью и еще более непосредственно, разрабатывая приемочные тесты.

Иногда против идеи сбора требований к системе (технического задания) с использованием языка модели выдвигаются возражения. В конце концов, почему техническое задание должно зависеть от архитектуры программы, которая его реализует? Но в этих возражениях не учитывается тот факт, что любой язык основан на некоторой модели. Значения слов расплывчаты и скользки. Модель предметной области обычно вырастает из жаргона специалистов, но потом "оттачивается" до состояния четких, сосредоточенных на предмете определений. Разумеется, у специалистов есть повод протестовать, если эти определения отклоняются от принятых в их профессии значений. В гибких (*agile*) методиках разработки требования к системе эволюционируют по мере реализации проекта, поскольку редко когда заранее можно иметь настолько исчерпывающую базу знаний, чтобы составить полную и подробную спецификацию программы заблаговременно. В ходе этой эволюции должно выполняться и переопределение требований на постоянно совершенствующемся ЕДИНОМ ЯЗЫКЕ.

---

[7] Вариантов перевода термина *use case* есть множество, из которых наиболее краткий – *прецедент*, а выбранный здесь *сценарий использования* происходит от первоначального варианта *usage scenario* самого автора термина, Ивара Якобсона (Ivar Jacobson). — *Примеч. перев.*

Необходимость иметь несколько языков в проекте возникает не так уж редко, но из-за этого не должен возникать языковой барьер между специалистами и разработчиками. (Параллельное существование нескольких моделей в проекте рассматривается в главе 12.)

Конечно, программисты не обойдутся без применения технической терминологии, непонятной специалистам предметной области. Для обсуждения технических аспектов системы у них есть свой богатый жаргон. Почти наверняка и у пользователей тоже есть специализированный жаргон, далеко выходящий за узкие рамки одной программы и уровня понимания программистов. Но это только *дополнения* к единому языку. Эти диалекты не должны содержать альтернативные словари для той же области, для которой уже составлены четкие модели.

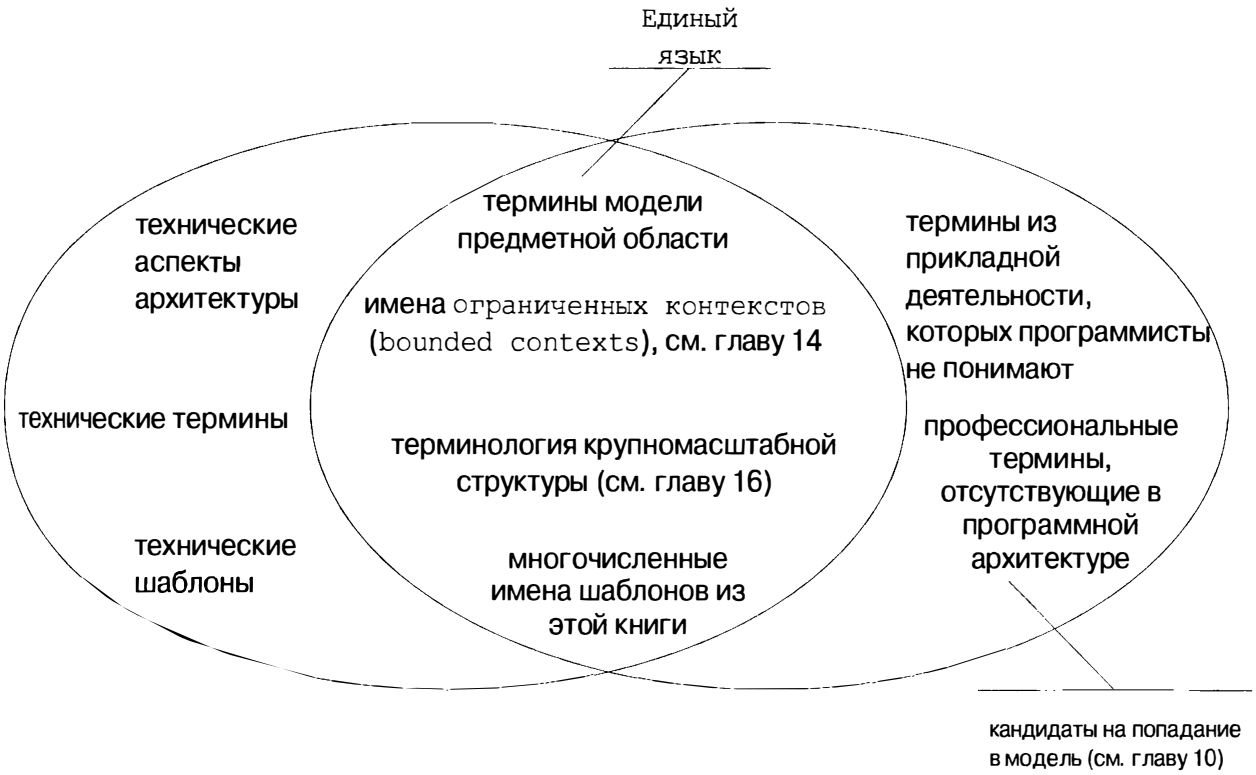


Рис. 2.3. Единый язык рождается "на пересечении" технических жаргонов

При наличии ЕДИНОГО ЯЗЫКА дискуссии в среде программистов, обсуждения в среде специалистов и выражения в самом коде будут реализовываться в одной и той же языковой среде, созданной на основе совместно используемой модели предметной области.

Документация, диаграммы, схемы

Ни на одном рабочем совещании, где обсуждается архитектура программного обеспечения, я не могу обойтись без рисования схем на доске или на бумаге. Значительную часть этих схем составляют диаграммы на языке UML — в основном диаграммы классов или взаимодействия объектов.

Для многих людей характерно преимущественно визуальное восприятие, и графические схемы хорошо помогают им в понимании разных видов информации. Диаграммы UML хорошо передают отношения между объектами и неплохо показывают взаимодействия. Но они не содержат концептуальные определения объектов. На совещании я обычно передаю их устной речью в ходе рисования диаграммы или же они возникают в диалоге с другими участниками.

Простые, неформальные UML-диаграммы вполне могут оказаться "объединяющим центром" дискуссии. Набросайте схему из трех-пяти основных объектов, относящихся к обсуждаемому вопросу, и никто из присутствующих не останется в стороне. Всем будет доступна картина взаимоотношений между объектами и, что важно, имена или названия этих объектов. Устное обсуждение много выиграет от такой поддержки. Диаграмму можно изменять по ходу "мысленных экспериментов" участников, так что графическая схема в какой-то мере даже приобретет динамичность устной речи и займет органичное место в дискуссии. В конце концов, UML ведь означает унифицированный *язык* моделирования (Unified Modeling *Language*).

Неприятности возникают, когда участники процесса считают себя обязанными передавать всю модель или архитектуру программы средствами UML. Если объектных диаграмм слишком много, то одновременно возникает и избыток, и недостаток информации. Избыток — оттого, что все до единого объекты для будущей программы помещаются в модель. А недостаток — оттого, что из-за обилия деталей за деревьями можно не заметить леса.

Даже если указаны все возможные детали, атрибуты и взаимоотношения — это только полдела в описании объектной модели. Поведение объектов и наложенные на них связи-ограничения проиллюстрировать не так-то легко. С помощью диаграмм взаимодействия объектов можно наглядно показать некоторые сложные, критические места архитектуры, но весь объем взаимодействий этим способом отобразить не удастся — слишком много усилий отнимает как составление таких диаграмм, так и их чтение. Да и диаграмма взаимодействия отражает саму цель построения модели только лишь неявно. Для включения в модель ограничений и утверждений язык UML дополняется текстом в скобках прямо на диаграммах.

Обязанности, возложенные на тот или иной объект, можно косвенно обозначить именами операций и проиллюстрировать диаграммами взаимодействия объектов, но их нельзя *сформулировать* таким способом. Эта задача возлагается на текстовое описание или устную дискуссию. Другими словами, диаграмма UML не может передать два самых существенных аспекта модели: смысл представляемых ею концепций и практическое назначение ее объектов. Впрочем, это не должно нас беспокоить — этой цели успешно служит правильно употребляемый английский (или русский[8], или любой другой) язык.

UML не слишком хорош также и в роли языка программирования. Всякая известная мне попытка воспользоваться средствами генерирования кода, имеющимися в этом инструменте моделирования, оканчивалась плачевно. Если ограничить себя рамками UML, часто приходится опускать самую важную часть модели просто потому, что выражаемое ею правило нельзя описать одними лишь средствами блок-схемы. А генератор кода, конечно, не может воспользоваться текстовыми аннотациями. Если у вас даже и есть технология, позволяющая писать выполняемые программы на схематическом языке наподобие UML, то диаграмма UML сводится просто к другому способу изображения самой программы, и теряется сам смысл понятия "модель". Если использовать UML в качестве языка непосредственной реализации, то все равно не обойтись без других средств изображения и обсуждения модели, не "загроможденной подробностями".

Диаграммы — это средства коммуникации и наглядного объяснения, которые облегчают "мозговой штурм" проблемы. Лучше всего они служат этим целям, когда содержат минимум элементов. От обширных схем целой объектной модели нет никакой пользы при обсуждении или объяснениях; они утомляют читателя деталями и лишены ясного смысла. Поэтому мы не будем связываться с полными диаграммами объектных моделей

---

[8] В оригинале у автора "испанский", но ведь мы сейчас читаем книгу на-русском, не так ли? — *Примеч. перев.*

и даже хранилищами данных UML. Наш инструмент — упрощенные схемы концептуально важных частей объектной модели, существенных для понимания архитектуры программы. Именно такие схемы и диаграммы, как показано в этой книге, я использую в реальных проектах. Они призваны упрощать и пояснять, и даже содержат кое-какие нестандартные обозначения, если это нужно для облегчения понимания задачи. В них показаны ограничения и связи, но это не детализированные проектные спецификации. Это только идейный костяк проекта.

Все существенные подробности архитектуры проекта заключены в его исходном коде. Хорошо написанная программная реализация должна быть "прозрачной" и отражать модель, лежащую в ее основе. (Как этого добиться, обсуждается в следующей главе и в значительной части прочих глав книги.) Вспомогательные диаграммы и документы способны привлечь внимание людей к проблеме, а дискуссия на естественном языке — раскрыть смысловые нюансы. Вот почему я предпочитаю поступать прямо противоположно типичной идеологии UML-диаграмм. Вместо того чтобы составлять схему, аннотированную текстовыми пояснениями, я пишу текстовый документ, избирательно иллюстрированный упрощенными схемами.

Всегда помните, что *модель — это не схема.* Назначение схемы или диаграммы состоит в том, чтобы помочь в общении и объяснении модели. А вот код способен хранить подробности реализации архитектурного проекта. Хорошо написанный код на Java в своем роде так же выразителен, как диаграмма UML. Тщательно отобранные и построенные схемы помогают сосредоточить внимание и следить за мыслью, если только навязчивая идея автора выразить в них всю целиком модель и архитектуру не лишила их всякой полезности.

Письменная проектная документация

Устное общение помогает дополнить ясным смыслом строгость и детализированность кода программы. Но хотя устные дискуссии и важны для того, чтобы все участники были в курсе устройства модели, для стабильного обмена информацией группе любой численности все-таки будет полезно иметь под рукой и некоторое количество письменных документов. Но составление такой документации, которая бы действительно помогала в разработке качественного программного продукта, — не такое уж простое дело.

Как только документ приобретает фиксированную форму, он часто теряет связь с динамично изменяющимися обстоятельствами проекта. Его оставляет позади эволюция кода или же эволюция языка, употребляемого в проекте.

Здесь может помочь несколько подходов. Позже, в части IV, будет предложено несколько конкретных форм документов для тех или иных нужд, но я не хотел бы давать предписаний, какую документацию использовать в проекте. Вместо этого я предложу две общие рекомендации по оценке качества документа.

Документация должна дополнять код и устные обсуждения

В каждой методологии гибкой разработки программ — своя стратегия работы с документацией. В экстремальном программировании предлагается вообще исключить проектную документацию — пускай код говорит сам за себя. Дескать, вся правда — в выполняемом коде, а документы могут врать. В поведении выполняемой программы нет места двусмысленности и неоднозначности.

В экстремальном программировании акцент сделан исключительно на *активных* элементах программы и выполняемых тестах. Даже комментарии, добавляемые в код, не влияют на работу программы, так что им никогда не удается "идти в ногу" с активным кодом и стоящей за ним моделью. Внешняя документация и диаграммы не влияют на работу

программы, так что и они тоже не синхронизированы с кодом. С другой стороны, устные обсуждения и схемы, нарисованные на доске, не фиксируются достаточно надолго, чтобы создать путаницу. Разработчики полагаются на код как на среду коммуникации в проекте, и это побуждает их поддерживать код стилистически чистым и прозрачным по смыслу.

Но у кода, как у вида проектной документации, все-таки есть свои пределы. Его читатель вполне может утонуть в деталях. Работа программы однозначна и недвусмысленна, но это не делает ее код очевидным. Да и смысл выполняемой работы не обязательно легко понятен. Другими словами, документирование только посредством самого кода имеет примерно те же недостатки, что использование всеобъемлющих UML-диаграмм. Конечно, интенсивные устные дискуссии в пределах группы разработчиков задают контекст и помогают ориентироваться в коде, но все это временно и слишком локально. К тому же разработчики — не единственные люди, которым необходимо понимать модель.

На документацию не надо возлагать те задачи, с которыми хорошо справляется сам код. В коде уже есть все детали. Это точная спецификация работы программы.

В документации необходимо отразить смысл, помочь в понимании крупномасштабных структур, сконцентрировать внимание на ключевых элементах. Документация может прояснить смысл архитектуры программы там, где в языке программирования не хватает средств для непосредственной и ясной реализации той или иной архитектурной концепции. Итак, письменная документация должна дополнять код и устную дискуссию.

Документация должна активно работать и не отставать от процесса.

Когда я документирую модель в письменном виде, я изображаю небольшие, тщательно отобранные подмножества модели в виде схем и окружаю их текстом. Я даю определения классов и их функций словами, заключая их в смысловой контекст так, как это можно сделать только на естественном языке. Но зато графические схемы наглядно показывают тот или иной выбор, сделанный в ходе формализации и усечения понятий при их включении в объектную модель. Такие схемы вполне могут быть неформальными — даже набросками от руки. Кроме экономии труда, нарисованные от руки схемы имеют еще одно преимущество — они *создают ощущение* временности и неформальности. А создавать именно такое ощущение — это правильно, поскольку и сами идеи нашей модели обладают теми же свойствами.

Самая большая ценность проектной документации состоит в том, что она разъясняет понятийный аппарат модели, помогает ориентироваться в деталях кода, а также может частично давать понятие о том, как именно должна использоваться модель. В зависимости от подхода, принятого группой разработчиков, проектный документ может быть самым простым, наподобие серии черновых набросков на стене, или существенно проработанным.

Документ должен активно участвовать в работе проекта. Самый простой способ проверить это — проследить за взаимосвязью документа и ЕДИНОГО ЯЗЫКА. Написан ли документ на языке, на котором в настоящее время говорят разработчики? Написан ли он на языке, внедренном в код?

Вслушайтесь в ЕДИНЫЙ ЯЗЫК и процесс его изменения. Если термины, определенные в проектном документе, не начинают появляться в дискуссиях и в коде, значит, документ не выполняет своей задачи. Может быть, он слишком обширный или сложный. Может быть, он не сосредоточен на достаточно важной теме. Либо этот документ не читают, либо не находят убедительным. Если документация никак не влияет на ЕДИНЫЙ ЯЗЫК, значит, что-то здесь неправильно.

И наоборот, в проекте можно услышать, как видоизменяется естественным образом ЕДИНЫЙ ЯЗЫК, а документация остается “далеко позади”. Очевидно, такой документ либо не кажется разработчикам полезным для дела, либо ему не придают достаточного зна-

чения, чтобы вовремя обновлять. Его можно положить в архив для истории, но в активном состоянии он может вызвать путаницу и навредить проекту. А если документ не играет реальной роли, то применять дисциплинарные меры, чтобы заставить дорабатывать и обновлять его — это пустая трата сил.

Наличие ЕДИНОГО ЯЗЫКА дает возможность сократить и сделать более строгими другие документы, такие как технические задания (спецификации требований). По мере того как модель предметной области приходит к отражению наиболее существенных знаний об этой области, технические требования к программе превращаются в сценарии в пределах модели, и описание таких сценариев можно давать на ЕДИНОМ ЯЗЫКЕ в таких терминах, которые непосредственно связаны с ПРОЕКТИРОВАНИЕМ АРХИТЕКТУРЫ ПО МОДЕЛИ[9] (см. главу 3). В результате технические задания можно будет писать в более простой форме, поскольку они не обязаны передавать все профессиональные знания, лежащие в основе модели.

Сохраняя в документации лишь минимально необходимое и ставя перед ней задачу *дополнения* кода и устных дискуссий, можно сохранить тесную связь документации с проектом. Пусть ЕДИНЫЙ ЯЗЫК и его эволюция станут теми факторами, которые помогут отобрать наиболее активные и необходимые для реализации проекта документы.

Выполняемый код решает все

Рассмотрим подробнее выбор, сделанный сторонниками "экстремального" программирования (и не только) — полагаться в работе практически целиком на выполняемый код и его тесты. Значительная часть этой книги посвящена тому, как сделать код передающим смысл посредством ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ (см. главу 3). Хорошо написанный код может быть очень информативным, но сама информация, которую он несет, не обязательно безупречно точна. О, разумеется, *работа* той или иной части кода не оставляет места для двусмысленностей. Но вот имя метода может быть двусмысленным, дезориентирующим или устаревшим по сравнению с его внутренним устройством. Результаты тестов — вещь строгая и однозначная, но информация, которую можно почерпнуть из имен переменных и организации кода — вовсе нет. Хороший стиль программирования предполагает максимально понятную связь между организацией кода и его работой, но все-таки это требует известной самодисциплины. Нужно быть достаточно привередливым и педантичным, чтобы писать код, который не только *делает* то, что нужно, но и *сообщает в письменном виде* то, что нужно.

Устранение подобных противоречий — это главный козырь таких методик, как декларативное программирование (см. главу 10), в котором объявление о предназначении того или иного элемента программы определяет его фактическое поведение в ходе работы программы. Частично этим стремлением объясняются и попытки генерировать код из диаграмм UML, хотя пока из них ничего хорошего не вышло.

И все же, хотя даже код способен вводить в заблуждение, он намного ближе к грешной земле, чем прочие виды документации. Привести в соответствие фактическое поведение, смысловое содержание и внешнюю форму кода с помощью предписанной в данный момент стандартной технологии — все это требует дисциплины и определенного взгляда на архитектурное проектирование программы (см. часть III). Для успешной передачи смысла код должен основываться на том же языке, на котором написаны технические задания — т.е. на том же, на котором между собой общаются разработчики и специалисты в предметной области.

---

[9] В оригинале *model-driven design*. — Примеч. перев.

Пояснительные модели

Основной акцент в нашей книге сделан на то, что в основе программной реализации, архитектурного проектирования и взаимодействия в группе разработчиков должна лежать одна и та же модель. Иметь несколько моделей для нескольких разных целей означает подвергать проект риску.

Модели также могут быть ценным наглядным учебным пособием для передачи знаний о предметной области. Модель, по которой выполняется проектирование программы, — это один взгляд на предметную область, но для целей обучения полезно иметь и другие точки зрения, исключительно вспомогательные, для лучшей передачи общих знаний. Для этой цели могут применяться схемы или тексты, описывающие другие виды моделей — те, которые непосредственно не связаны с разработкой программного продукта.

Одна из причин, по которой могут понадобиться другие модели, — это широта охваченной тематики. Техническая модель, используемая для проектирования программной архитектуры, неизбежно урезается до самого необходимого минимума, чтобы хорошо исполнить свое назначение. А вот пояснительная модель вполне может включать такие аспекты предметной области, которые расширяют контекст рабочей модели и тем самым обеспечивают более глубокое ее понимание.

В расширенных рамках пояснительных моделей есть возможность обмениваться информацией в более удобном виде, адаптированном к той или иной теме. Визуальные метафорические образы, используемые специалистами в предметной области, могут существенно помочь в объяснениях, отчего разработчики приобретают лучшее знание предмета, а сами специалисты становятся более спокойными на их счет. Пояснительные модели также способны представить предметную область в совершенно другом свете, и разнообразие возможных объяснений дает возможность разработчикам лучше понять предмет.

Пояснительным моделям вовсе не обязательно быть объектными — даже лучше, когда они таковыми не являются. В этих моделях полезно избегать как языка UML, так и ложного ощущения сходства с архитектурой программы. Пусть даже пояснительная модель и рабочая модель разработчиков в чем-то соответствуют друг другу, все же их сходство довольно приблизительно. Чтобы избежать путаницы, все должны четко понимать различие между ними.

Пример

Операции и маршруты грузоперевозок

Рассмотрим прикладную программу для управления перемещением грузов, предназначенную для использования в компании по водным грузоперевозкам. Модель включает в себя подробную схему объединения портовых операций и судовых рейсов в оперативный план перемещения груза, или его “маршрут” (*route*). Но для непосвященных в таинства перевозок диаграмма классов этой программы не слишком проясняет дело.

В этом случае пояснительная модель может помочь членам рабочей группы понять, о чем действительно идет речь в диаграмме классов. Вот другая точка зрения на те же понятия и процессы.

Каждая линия на рис. 2.5 представляет либо портовую операцию (погрузку или разгрузку), либо пребывание груза на складе на суше, либо пребывание груза на судне, находящемся в рейсе. Здесь нет полного соответствия с диаграммой классов, но зато подчеркнуты ключевые моменты из предметной области.

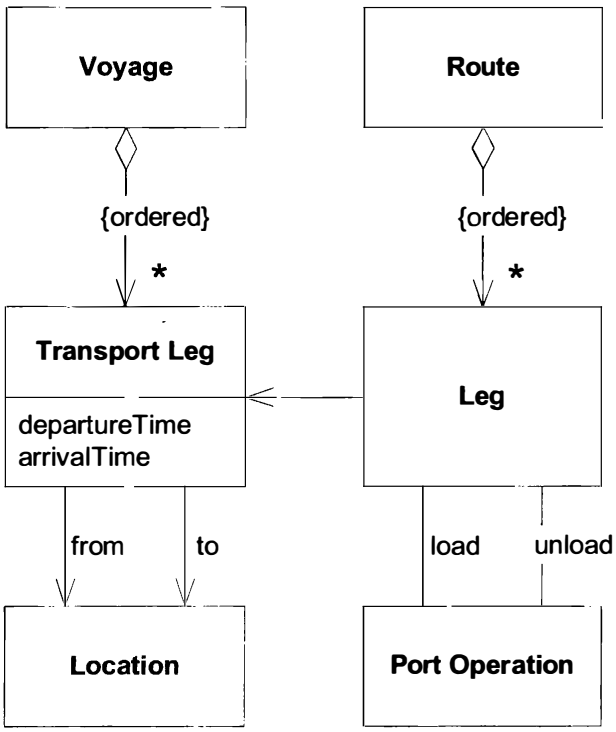


Рис. 2.4. Диаграмма классов для маршрута перевозки груза

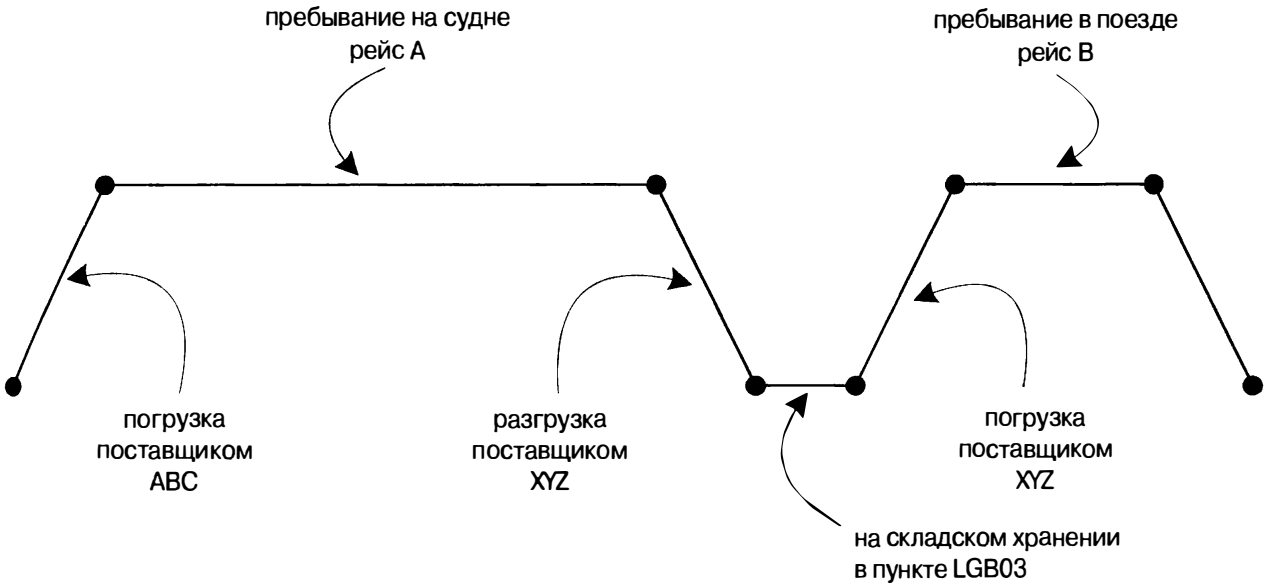


Рис. 2.5. Пояснительная модель маршрута перевозки груза

Схемы такого рода, наряду с объяснениями представляемой ими модели на естественном языке, могут помочь как разработчикам, так и специалистам в понимании более строгих диаграмм, представляющих программную архитектуру приложения. Понимать такие схемы и диаграммы совместно легче, чем каждую по отдельности.

Глава 3

Связь между моделью и реализацией

Первое, что я увидел, войдя в дверь, — полную диаграмму классов, напечатанную на покрывавших стену больших листах бумаги. Это был первый день моего участия в проекте, в котором очень неглупые люди в течение нескольких месяцев тщательно исследовали и разрабатывали подробную модель предметной области. Типичные объекты модели имели сложные связи с еще тремя-четырьмя другими объектами, и у всей этой паутины связей практически не было естественных границ. В этом отношении аналитики хранили верность самой сути предметной области.

Пусть модель размером со стену и была пугающе велика, ей нельзя было отказать в содержательности. Через какое-то время я узнал о предмете довольно много (хотя обучение шло бессистемно — что-то вроде случайного блуждания по веб-страницам). Но больше меня беспокоило то, что в результате этих занятий я не получил никакого представления о коде и архитектуре приложения.

Когда программисты начали программную реализацию, они обнаружили, что переплетение ассоциативных связей, пусть и более-менее обозримое для аналитика-человека, никак не получается превратить в хранимые и вызываемые модули, которыми можно было бы манипулировать с соблюдением транзакционной целостности. Прошу обратить внимание, что в проекте использовалась объектно-ориентированная база данных, так что разработчикам даже не пришлось решать задачу преобразования объектов в реляционные таблицы. Но на самом фундаментальном уровне модель была неспособна обеспечить управление программной реализацией.

Поскольку модель была "правильной", т.е. результатом интенсивного сотрудничества между техническими аналитиками и специалистами в предметной области, разработчики пришли к выводу, что на основе концептуально проработанных объектов архитектуру приложения спроектировать невозможно, и перешли к несистематической (*ad hoc*) разработке. В их архитектуре еще использовались некоторые из ранее принятых имен классов и атрибутов для целей хранения данных, но программа больше не была основана ни на старой, ни на какой-либо другой модели.

Итак, в проекте модель была, но что толку от ее наличия на бумаге, если она не влияет напрямую на процесс разработки реального кода?

Несколько лет спустя мне довелось наблюдать такой же конечный результат в совершенно ином процессе. Проект состоял в замене существующей программы на языке C++ новой архитектурой, реализованной на Java. Старое приложение было "склепано" без какого бы то ни было объектного моделирования. Архитектуру этой старой программы, если ее вообще можно так назвать, строили, громоздя одну функциональную возможность на другую без какого-либо заметного глазу обобщения или абстрагирования.

Поразительно, что конечные результаты в обоих случаях были очень похожи по своему характеру! Обе программы выполняли свои функции, но были неестественно разду-

ты, плохо понятны для чтения, и в конечном счете не поддавались дальнейшему развитию и доработке. Хотя программные реализации в отдельных местах демонстрировали некую прямую связь с задачей, назначение системы нельзя было толком выяснить, читая код. Ни в одном из проектов фактически не были использованы преимущества объектной парадигмы (если не считать хитроумных структур данных), предоставляемой имеющимися средствами разработки.

Модели имеют множество разновидностей и могут играть множество ролей даже в ограниченном контексте проекта по разработке программного обеспечения. Предметно-ориентированное проектирование программ (DDD) требует применения модели, которая не просто помогает в предварительном анализе проблемы, но является прочным фундаментом для архитектуры. Такой подход оказывает существенное влияние на структуру получающегося в итоге кода. Но менее очевидно, что предметно-ориентированное проектирование требует также и другого подхода к самому моделированию.

Проектирование по модели

Астролябия, прибор для вычисления координат звезд, представляет собой механическую реализацию модели звездного неба.

Средневековый астрономический калькулятор

Астролябию изобрели древнегреческие астрономы и усовершенствовали ученые средневекового мусульманского Востока. Вращающаяся решетка (или *сетка*) представляла положения неподвижных звезд на небесной сфере. Сменные пластины с выгравированной на них местной сферической системой координат представляли виды неба с различных широт. Вращая сетку относительно пластины, можно было вычислить положение звездного неба в любое время и день года. И наоборот, зная положение звезд или Солнца, можно было вычислить время. Астролябия была механической реализацией объектно-ориентированной модели неба.

Тесная привязка кода к модели, лежащей в основе системы, придает коду смысл, а модели — практическую ценность.

* * *

В проектах, в которых вообще нет модели предметной области, а приложения разрабатываются путем последовательной реализации одной функции за другой, полностью отсутствуют преимущества переработки знаний и коммуникации в группе, рассмотренные в предыдущих двух главах. В любой сложной предметной области такой проект просто утонет, как в трясине.

С другой стороны, во многих сложных проектах предпринималась попытка создания неких предметных моделей, но отсутствовала тесная связь между моделью и кодом. Развиваемая в проекте модель, скорее всего, вначале полезная как исследовательский инструмент, со временем теряла связь с ходом работ и даже могла ввести в заблуждение. Все усилия, с которыми разрабатывалась такая модель, не могли гарантировать правильности архитектуры, поскольку связь между ними отсутствовала.

Эта связь может порваться по разным причинам, но довольно часто ее разрывают сознательно. Во многих методологиях проектирования предполагается использование *аналитической модели* (*analysis model*), не имеющей ничего общего с архитектурой программы и разрабатывающейся, как правило, совершенно другими людьми. Ее называют аналитической моделью потому, что она является результатом анализа и систематизации прикладной предметной области, безо всякого учета той роли, которую она будет играть в программной системе. Аналитическая модель задумывается исключительно как средство для понимания предметной области; считается, что примешивать к ней вопросы реализации программы — это мутить воду и вносить путаницу. Позже разрабатывается программная архитектура, которая может иметь лишь самое отдаленное сходство с аналитической моделью. Аналитическую модель разрабатывают, не учитывая потребностей проектирования приложения, и поэтому она, скорее всего, будет малопригодна для этой цели.

В процессе такого анализа, конечно, выполняется некоторая переработка и усвоение знаний, но с началом написания кода все это теряется, потому что разработчикам приходится изобретать новые абстракции для архитектуры приложения. Также нет гарантии, что выводы, полученные аналитиками и включенные в модель, сохранятся или хотя бы будут получены заново. В такой ситуации невыгодно даже поддерживать соответствие между архитектурой и слабо связанной с нею моделью.

Чисто аналитическая модель нередко не решает даже свою основную задачу — помочь в понимании предметной области, потому что ключевые открытия всегда делаются в ходе работ по проектированию и реализации приложения. Неизбежно возникают специфические проблемы, которые нельзя было предвидеть заранее. Модель, которую полностью проработали заблаговременно, нередко тонет в несущественных деталях, тогда как важные моменты в ней могут оказаться пропущенными. А кое-что окажется представлено таким способом, от которого нет никакой пользы для программной реализации. В результате работа над чисто аналитическими моделями прекращается вскоре после начала написания кода, и большую часть вопросов приходится решать заново. Но повторюсь еще раз: если разработчики воспринимают анализ как отдельную процедуру, тогда и моделирование выполняется менее строго. А если руководство воспринимает анализ как отдельную процедуру, то группа разработчиков может не получить нужного ей доступа к специалистам в предметной области.

Независимо от причины, программа, в основании архитектуры которой не лежит четкая концепция, в лучшем случае будет механизмом, который делает что-то полезное, но ничего не объясняет.

Если архитектура программы или хотя бы некая ее центральная часть, не соответствует структуре модели предметной области, то такая модель практически бесполезна, и правильность работы программы тоже нужно поставить под подозрение. В то же время слишком сложные взаимосвязи между моделями и функциями в программной архитектуре трудно поддаются пониманию, и на практике их сложно сохранить по мере изменения архитектуры. Страшнее всего — отделить анализ проблемы от проектирования архитектуры; при этом между двумя процессами не происходит обмен полезными знаниями и выводами, полученными в результате выполнения каждого из них.

В ходе анализа нужно извлечь из предметной области ее фундаментальные понятия и представить их понятным и выразительным способом. А в ходе проектирования архитектуры следует задать набор компонентов, которые конструируются с помощью средств разработки, имеющихся в проекте, и призваны эффективно работать в той выполняющейся среде, для которой предназначена программа, правильно решая поставленные перед ней задачи.

ПРОЕКТИРОВАНИЕ ПО МОДЕЛИ предметной области (MODEL-DRIVEN DESIGN) ликвидирует разрыв между аналитической моделью и архитектурой, поскольку в ходе него ищется такая модель, которая бы служила обеим целям. Если не вдаваться в чисто технические подробности, каждый объект в программной архитектуре играет концептуальную роль, определенную в модели. Это требует от нас большей придирчивости в выборе модели, поскольку она должна выполнять две совершенно различные задачи.

Всегда существует много способов абстрагирования предметной области, и всегда есть несколько программных архитектур, пригодных для решения прикладной задачи. Именно это обстоятельство говорит в пользу тесной связи между моделью и архитектурой. Но связь эта не должна достигаться за счет снижения качества анализа, на который пагубно влияет учет чисто технических соображений. Не можем мы принять и топорно сработанную архитектуру, в которой отражены идеи предметной области, но не соблюдены принципы проектирования программного обеспечения. Наш подход требует модели, которая в равной степени работала бы и на анализ, и на проектирование. Если модель кажется непригодной для практической реализации, то следует поискать другую. Если модель не выражает органичным образом ключевых понятий предметной области, то и в этом случае нужно поискать другую. Процессы моделирования и проектирования таким образом становятся единым итерационным циклом.

Императив тесной связи между моделью предметной области и архитектурой порождает дополнительный критерий выбора наиболее полезных моделей из всех возможных. Он требует напряженной работы мысли, а также, как правило, множества итераций и интенсивного рефакторинга. Но в результате возникает модель, являющаяся *неотъемлемой* частью проекта.

Спроектируйте часть программной системы так, чтобы она отражала модель предметной области самым буквальным образом — соответствие между ними должно быть очевидным. Вернитесь к модели и измените ее так, чтобы она наиболее естественно реализовалась в программе, даже если вы хотите еще и как можно глубже отразить в ней суть предметной области. Добивайтесь построения одной модели, хорошо служащей обеим целям, и к тому же поддерживающей надежный ЕДИНЫЙ ЯЗЫК**.**

“Выводите” из модели терминологию, используемую в проектировании программы, и примерное распределение обязанностей. Код при таком подходе является выражением модели, так что изменения в коде будут и изменениями модели, а влияние таких изменений должно расходиться, как круги по воде, по всем видам работ в проекте.

Чтобы плотно привязать программную реализацию к модели, обычно требуются средства разработки и языки программирования, поддерживающие парадигму моделирования, — например, объектно-ориентированное программирование.

Иногда для разных подсистем используются разные модели (см. главу 14). Но и в этом случае в основе конкретной части системы должна лежать одна и та же модель. Она должна распространяться на все аспекты разработки, от анализа технических требований до написания кода.

Использование одной модели снижает вероятность ошибки, поскольку архитектура проектируется так, что непосредственно вырастает из тщательно проработанной модели. Как архитектура, так и сам код имеют те же свойства информативности, что и сама модель.

* * *

Разработать одну модель, которая бы охватывала проблему целиком и порождала качественную программную архитектуру, не так просто. Нельзя просто взять какую-нибудь модель и превратить ее в работоспособную архитектуру. Чтобы модель оказалась пригодной для целей практической реализации, ее нужно выстроить очень тщательно. Следует применять такие средства проектирования и реализации, чтобы получающийся код эффективно выражал сущность модели (см. часть II). Для этого и нужны переработчики знаний (*knowledge crunchers*), чтобы изучить возможности и варианты моделирования, а затем "выплавить" из этого "сырья" эффективную программную реализацию. Разработка становится итерационным процессом пошагового усовершенствования модели, архитектуры и кода как единого целого (см. часть III).

Парадигмы моделирования и средства программирования

Чтобы ПРОЕКТИРОВАНИЕ ПО МОДЕЛИ имело смысл, соответствие архитектуры и модели должно быть буквальным и точным в пределах естественной человеческой погрешности. Для того чтобы столь близкое соответствие стало возможным, приходится фактически ограничивать себя границами той парадигмы моделирования, которую поддерживают имеющиеся средства программирования. Это необходимо для создания прямых программных аналогов объектам и концепциям модели.

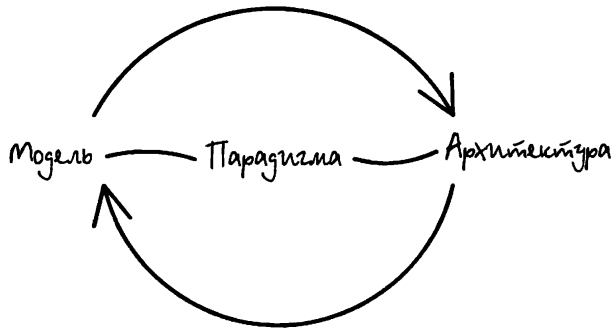


Рис. 3.1.

Объектно-ориентированное программирование является мощным инструментом, поскольку основывается на парадигме моделирования и предлагает конкретные реализации построения моделей. Если же смотреть под тем углом зрения, который интересует программиста, объекты действительно существуют в памяти, связаны с другими объектами, организованы в классы и выполняют операции, обмениваясь данными. Хотя многие разработчики пользуются лишь техническими возможностями объектов для организации кода, настоящая сила объектного подхода проявляется тогда, когда код выражает концепции модели. Язык Java и многие другие инструментальные средства программи-

рования допускают создание объектов и взаимосвязей между ними, имеющих прямые аналогии в концептуальных объектных моделях.

Язык Prolog, пусть и не достигший такой же массовой популярности, как объектно-ориентированные языки, естественным образом приспособлен к ПРОЕКТИРОВАНИЮ ПО МОДЕЛИ. В его случае парадигмой является логика, а моделью — набор логических правил и фактов, которыми они оперируют.

Возможности ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ ограничены, если используется такой язык, как C, поскольку не существует парадигмы моделирования, соответствующей чисто *процедурному* языку. Процедурные языки отличаются тем, что в них программист приказывает компьютеру выполнить последовательность шагов. Хотя программист может при этом держать в уме понятия некоторой предметной области, программа, тем не менее, представляет собой последовательность технических операций над данными. В результате может получиться нечто полезное, но вот сама суть, смысл в программе практически не отражается. В процедурных языках часто поддерживаются сложные типы данных, которые начинают даже соответствовать естественным понятиям предметной области, но эти типы — все-таки не более чем организованные данные, и активные операционные аспекты предметной области в них не отражены. В итоге программы, написанные на процедурных языках, состоят из функций, соединенных в одно целое на основе предполагаемых цепочек выполнения операций, а не концептуальных связей в модели предметной области.

Прежде чем вообще узнать о существовании объектно-ориентированного программирования, я писал реализации математических моделей на языке FORTRAN — именно в этой области данный язык превосходно зарекомендовал себя. Математические функции являются главными концептуальными компонентами таких моделей, и они вполне четко выражаются средствами этого языка. Однако в нем не существует средств для выражения смысла более высокого уровня, стоящего за такими функциями. Большинство предметных областей, не относящихся к области математики, не поддается ПРОЕКТИРОВАНИЮ ПО МОДЕЛИ на процедурных языках, потому что эти области нельзя свести ни к совокупности математических функций, ни к последовательности шагов процедуры.

В этой книге рассматривается почти исключительно объектно-ориентированная архитектура программного обеспечения — именно эта парадигма доминирует в настоящее время в подавляющем большинстве крупных проектов.

Пример

От процедурного подхода к модельно-ориентированному

Как рассказывалось в главе 1, электронную печатную плату (ПП) можно представить в виде совокупности электрических проводников (*цепей*, *nets*), соединяющих выводы или контакты (*pins*) различных компонентов. Нередко количество таких цепей достигает десятков тысяч. Специальные программы автоматической трассировки печатных плат находят способ физического размещения всех цепей, чтобы они не пересекались друг с другом и не мешали работе друг друга. Выполняется оптимизация дорожек на плате с одновременным удовлетворением огромного количества условий, наложенных проектировщиками и ограничивающих способы расположения цепей. Хотя программы автотрассировки ПП реализуют весьма сложные алгоритмы, у них все-таки есть недостатки.

Одна из проблем состоит в том, что у каждой из тысяч цепей имеется собственный набор правил трассировки. Инженеры-схемотехники группируют многие цепи по естественному критерию удовлетворения одних и тех же правил. Например, некоторые цепи в совокупности образуют *шины* (*buses*).

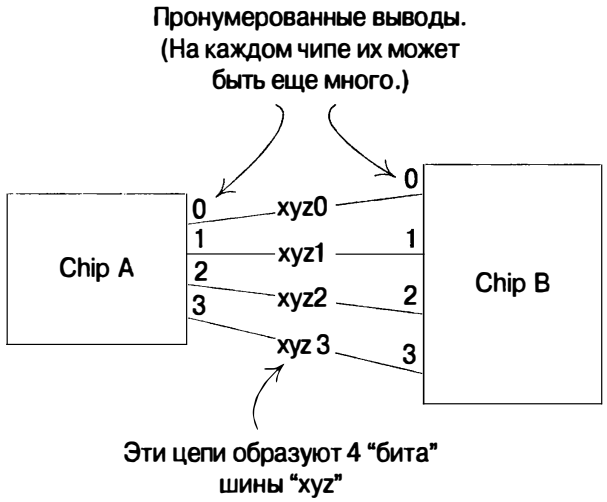


Рис. 3.2. Пояснительная схема цепей и шин

Группируя цепи в шины, например, по 8, 16 или 256 штук, инженер уменьшает объем работы до более обозримого, повышая производительность труда и снижая вероятность ошибки. Но беда в том, что в программе трассировки нет такого понятия, как шина. Правила приходится ассоциировать поочередно с каждой из десятков тысяч цепей.

Механистический подход

Инженеры-схемотехники, которых такое положение не устраивает, пытаются обойти это ограничение в программе автотрассировки. Для этого они разрабатывают сценарии, анализирующие файлы данных автотрассировщика и вставляющие правила непосредственно в файл, что позволяет применять их сразу к целой шине.

Автотрассировщик хранит все электрические соединения в файле *списка цепей*, который выглядит примерно так.

```
Цепь      Компонент.Вывод
----      ---------------
Xyz0      A.0, B.0
Xyz1      A.1, B.1
Xyz2      A.2, B.2
. . .
```

Правила трассировки хранятся в файле примерно такого формата.

```
Цепь      Правило          Параметры
----      -------          ---------
Xyz1      min_linewidth    5
Xyz1      max_delay        15
Xyz2      min_linewidth    5
Xyz2      max_delay        15
. . .
```

Инженеры тщательно выбирают правила именования цепей, чтобы при алфавитной сортировке файла данных все цепи одной шины оказывались вместе в отсортированном файле. После этого выполняемый сценарий анализирует файл и модифицирует каждую цепь по данным ее шины. Код для синтаксического анализа, манипулирования данными и записи файла слишком запутан и неочевиден, чтобы служить наглядным примером, поэтому ниже просто перечислены шаги, входящие в процедуру.

1. Отсортировать файл списка цепей по именам цепей.
2. Считать каждую строку из файла, разыскивая первую, которая начинается с заданного шаблона имени шины.
3. Для каждой строки с подходящим к шаблону именем извлечь из строки имя цепи.
4. Добавить имя цепи с текстом правила в конец файла правил.
5. Повторить, начиная с п. 3, пока левая часть строки не перестанет соответствовать имени шины.

Пусть на вход поступает следующее правило для шины.

```
Шина      Правило     Параметры
----      -------     ---------
Xyz       max_vias    3
```

Тогда в выходной файл будут добавлены следующие правила для цепей.

```
Цепь      Правило     Параметры
----      -------     ---------
. . .
Xyz0      max_vias    3
Xyz1      max_vias    3
Xyz2      max_vias    3
. . .
```

Могу себе представить, что перед автором первоначального сценария стояла только эта несложная задача, и если бы это требование так и осталось единственным, то от этого сценария было бы много пользы. Но на практике теперь используются десятки сценариев. Их, конечно, можно подвергнуть рефакторингу хотя бы для совместного использования функций сортировки и сравнения строк. А если бы в языке сценариев еще и поддерживались функциональные вызовы, в которые можно было бы "спрятать" детали реализации, то сценарии уже читались бы практически как приведенная выше процедура. Но все равно еще остаются манипуляции с файлами. Стоит изменить формат файла (а их существует несколько), и потребуется писать все заново, пусть даже концепции группировки шин и применения правил к ним останутся неизменными. Если бы понадобились более обширные функциональные возможности или лучшая интерактивность, их реализация потребовала бы непомерных трудозатрат.

Авторы выполняемых сценариев пытались дополнить модель предметной области понятием "шины". В их реализации неявно предполагается существование такого понятия, для чего и выполняются сортировка и сравнение строк. Однако в явном виде это понятие не фигурирует.

Проектирование по модели

Выше уже описаны понятия, в которых специалисты думают о проблемах из их области деятельности. Теперь необходимо организовать эти понятия явным образом в виде модели, на основе которой можно написать программу.

Если эти объекты определить на каком-нибудь объектно-ориентированном языке, то основные функции программы становятся практически тривиальными в реализации.

Метод `assignRule()` можно реализовать в классе **Абстрактная Цепь (Abstract Net)**. Метод `assignedRules()` класса **Цепь (Net)** считывает правила своего собственного класса и правила соответствующей **Шины (Bus)**[1].

```
abstract class AbstractNet {
    private Set rules;

    void assignRule(LayoutRule rule) {
        rules.add(rule);
    }

    Set assignedRules() {
        return rules;
    }
}

class Net extends AbstractNet {
    private Bus bus;

    Set assignedRules() {
        Set result = new HashSet();
        result.addAll(super.assignedRules());
        result.addAll(bus.assignedRules());
        return result;
    }
}
```

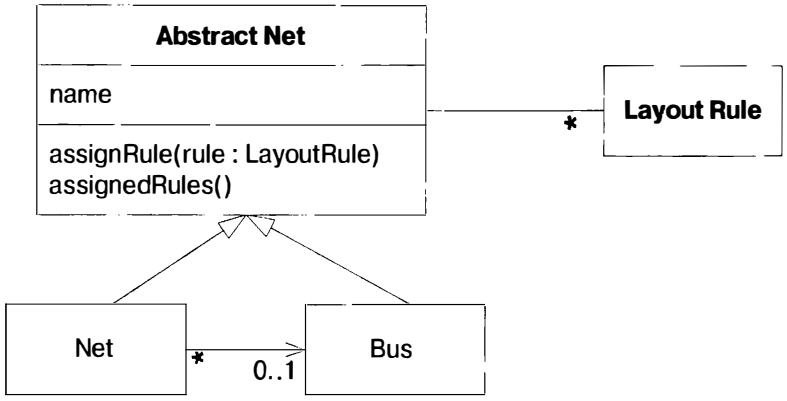


Рис. 3.3. Диаграмма классов, ориентированная на эффективное назначение правил трассировки

Разумеется, понадобится еще много вспомогательного кода, но основные функции сценария уже реализованы здесь.

Для приложения понадобится реализация системы импортирования-экспортирования, которую мы инкапсулируем в нескольких простых службах.

---

[1] В переводе сохранены и оригинальные названия (потому что от них происходят имена классов) и их переводы на русский язык (потому что они должны естественно вписываться в словесное описание модели). Все это совершенно в духе предлагаемого в книге подхода "единого языка". Правда, в скобках оказывается то одно, то другое в зависимости от близости предложения либо к естественному языку, либо к техническому определению. — *Примеч. перев.*

Служба	Выполняемые функции
Импортирование **Списка Цепей (Net List)**	Считать файл **Списка Цепей (Net List)**, создать экземпляры объекта **Цепь (Net)** для каждого пункта списка
Экспортирование **Правил цепей (Net Rules)**	Имея коллекцию **Цепей (Nets)**, записать все ассоциированные с ними правила в **Файл Правил (Rules File)**

Понадобится нам также и ряд вспомогательных средств.

Класс	Выполняемые функции
Хранилище цепей **(Net Repository)**	Обеспечить доступ к **Цепям** по именам
Фабрика определений шин **(Inferred Bus Factory)**	Имея коллекцию **Цепей**, с помощью правил именования построить определения **Шин (Buses)**, создать объекты
Хранилище шин **(Bus Repository)**	Обеспечить доступ к **Шинам** по именам

Теперь запуск приложения сводится к инициализации хранилищ импортированными данными.

```
Collection nets = NetListImportService.read(aFile);
NetRepository.addAll(nets);
Collection buses = InferredBusFactory.groupIntoBuses(nets);
BusRepository.addAll(buses);
```

Все службы и хранилища можно подвергнуть модульному тестированию (*unit testing*). Что еще важнее, можно протестировать реализованную здесь логическую структуру предметной области. Вот модульный тест ключевых функций (с использованием среды тестирования JUnit).

```
public void testBusRuleAssignment() {
    Net a0 = new Net("a0");
    Net a1 = new Net("a1");
    Bus a = new Bus("a"); // Шина (Bus) не зависит концептуально
    a.addNet(a0);         // от распознавания по имени, поэтому
    a.addNet(a1);         // ее тесты тоже не должны зависеть.

    NetRule minWidth4 = NetRule.create(MIN_WIDTH, 4);
    a.assignRule(minWidth4);

    assertTrue(a0.assignedRules().contains(minWidth4));
    assertEquals(minWidth4, a0.getRule(MIN_WIDTH));
    assertEquals(minWidth4, a1.getRule(MIN_WIDTH));
}
```

В интерактивном пользовательском интерфейсе можно показать список шин, чтобы пользователь мог каждой из них назначить правила. Или же, в целях совместимости с предыдущими версиями, можно считывать эти правила из файла. Фасадный метод легко обеспечивает доступ к данным для любых типов интерфейса. Его реализация отражает структуру теста.

```
public void assignBusRule(String busName, String ruleType,
      double parameter){
   Bus bus = BusRepository.getByName(busName);
   bus.assignRule(NetRule.create(ruleType, parameter));
}
```

И наконец:

```
NetRuleExport.write(aFileName, NetRepository.allNets());
```

Здесь служба запрашивает у каждой **Цепи (Net)** назначенные ей правила (`assignedRules()`) и записывает их в полностью развернутом виде.

Конечно, если бы требовалось выполнять только одну операцию (как в примере), то и сценария было бы достаточно. Но на самом деле необходимо реализовать более двадцати операций. При ПРОЕКТИРОВАНИИ ПО МОДЕЛИ масштабирование выполняется легко и позволяет также вводить ограничения на комбинирование правил.

Во второй архитектуре также учитывается тестирование. У ее компонентов имеются хорошо определенные интерфейсы, которые можно подвергать модульному тестированию (*unit testing*). Единственный же способ протестировать сценарий — это прогнать через него тестовый файл с заранее известным выходным результатом.

Следует помнить, что подобные архитектуры не проектируются быстро. Чтобы выделить (дистиллировать) важнейшие концепции предметной области в простую и точную модель, требуется много итераций рефакторинга и переработки знаний.

Анатомия модели: зачем модель нужна пользователю

Теоретически пользователю можно предъявить любую наглядную модель системы, не заботясь о том, что на самом деле находится в глубине. Но на практике несоответствие представления и сути вызывает в лучшем случае путаницу, а в худшем — реальные ошибки. Рассмотрим очень простой пример того, как пользователей вводят в заблуждение поверхностные теоретические модели: закладки веб-сайтов в современных версиях браузера Microsoft Internet Explorer[2].

Пользователь Internet Explorer считает "Избранное" ("Favorites") списком имен веб-сайтов, который сохраняется между сеансами работы. Но в программной реализации закладка из "Избранного" является файлом, содержащим URL-адрес. Имя этого файла помещается в папку с именем `Favorites` (`Избранное`). Вот тут-то и возникает проблема, если заголовок веб-страницы содержит символы, недопустимые в именах файлов системы Windows. Предположим, пользователь хочет сохранить закладку и вводит для нее следующее имя: `"Лень: секрет счастья"`. Появится сообщение об ошибке: "Имя файла не может содержать следующие символы: \ / : * ? " < > | ". Какое еще имя файла? Причем тут имя файла? С другой стороны, если заголовок веб-страницы уже содержит недопустимые символы, то Internet Explorer просто уберет их оттуда, вот и все. Такая потеря данных в этом случае вполне терпима, но это не совсем то, чего хотел пользователь. Автоматическое изменение данных "исподтишка" совершенно недопустимо в большинстве программ.

ПРОЕКТИРОВАНИЕ ПО МОДЕЛИ требует работы только с одной моделью (внутри одного контекста, см. главу 14). Большая часть изложенной теории и примеров относится к проблемам, связанным с наличием различных моделей для анализа предметной области

[2] Об этом примере сообщил мне Брайан Марик (Brian Marick).

и для проектирования архитектуры. А здесь у нас проблема возникла из-за наличия другой пары разных моделей: модели представлений пользователя и модели для проектирования/реализации программы.

Конечно, голая модель предметной области в большинстве случаев удобной для пользователя не будет. Но попытка создания в пользовательском интерфейсе некоей иллюзорной модели, отличающейся от модели предметной области, приведет к путанице — если только иллюзия не доведена до совершенства. Если список закладок веб-сайтов на самом деле представляет собой совокупность файлов с адресами, надо сообщить об этом факте пользователю и "убрать с глаз долой" альтернативную модель, которая только мешает. Это не только сделает данную функцию программы более понятной, но еще и даст пользователю возможность применить свои знания файловой системы при манипуляциях с "Избранным". Например, пользователь сможет реорганизовать закладки через файловый менеджер, а не через неуклюжие встроенные средства браузера. Хорошо информированные пользователи смогут активнее использовать такую гибкую возможность программы, как расположение файлов веб-ссылок в любом месте файловой системы. Итак, одно только удаление лишней "обманчивой" модели сразу добавило удобства и ясности операциям приложения. Так зачем навязывать пользователям новую модель, если программистам хватает и старой?

Вместо этого можно изменить способ хранения для списка избранных веб-страниц, — например, поместить их в файл данных, чтобы они подчинялись своим собственным правилам. Система этих правил будет, по-видимому, соответствовать правилам именования веб-страниц. И это снова будет одна-единственная рабочая модель, которая сообщит пользователю, что к закладкам в "Избранном" применимы все правила, которые он знает об именах веб-страниц.

Если архитектура программы проектируется на основе модели, отражающей основные потребности и проблемы как пользователей, так и специалистов в предметной области, то ее "анатомию" можно смело показывать пользователю в гораздо большей степени, чем в других методологиях программирования. Показ "изнанки" модели дает пользователю возможность лучше понять потенциал программы, ведет к более единообразному и управляемому ее использованию.

Моделировщики-практики

Разработку программного обеспечения принято метафорически сравнивать с процессом фабричного производства. В этой метафоре скрыта одна неявная посылка: высококвалифицированные инженеры проектируют и конструируют, а менее квалифицированные рабочие собирают продукцию. Эта метафора загубила множество программных проектов по одной простой причине: разработка программ *целиком состоит* из проектирования/конструирования. Во всех группах разработчиков есть распределение по специализированным ролям, но слишком резкое разделение ответственности за анализ, моделирование, проектирование и программирование вступает в противоречие с принципом ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN).

В одном из проектов мои обязанности состояли в координировании нескольких рабочих групп и помощи им в разработке модели предметной области, которая бы лежала в основе проектирования архитектуры. Но руководство решило, что те, кто моделирует, должны только моделировать, и что заставить их программировать означает пустую трату сил. В итоге мне фактически запретили писать код или прорабатывать детали вместе с программистами.

Некоторое время все шло как будто нормально. Работая со специалистами и руководителями разных групп разработчиков, мы осуществили переработку знаний и выработали неплохую предметную модель. Но она так и не заработала по двум причинам.

Во-первых, часть достоинств модели потерялась из-за эффекта "испорченного телефона". Общая эффективность модели сильно зависит от деталей (эта тема будет рассмотрена в частях II и III), и эти детали не всегда легко уловить в UML-диаграммах или общей дискуссии. Если бы я мог, закатав рукава, трудиться непосредственно рядом с программистами, писать кое-какой код в качестве примеров и обеспечивать прямую поддержку на месте, то группа разработчиков овладела бы абстракциями модели и следовала бы им в программной реализации.

Во-вторых, имела место лишь косвенная обратная связь от взаимодействия модели с программной реализацией и технологиями программирования. Например, некоторые аспекты модели оказались совершенно неэффективными на нашей технологической платформе, но полная информация о последствиях этого "просочилась" ко мне только через многие месяцы. Вначале проблему могли бы решить сравнительно небольшие изменения, но к тому времени это было уже неважно. Программисты уже изрядно продвинулись в написании программы, которая бы просто работала, — притом безо всякой модели. Вернее, там, где эта модель вообще использовалась, она свелась просто к структуре данных. Программисты выплеснули с водой ребенка, но что еще им было делать? Они не могли дальше рисковать, подставляя шею под ярмо мудрствующего архитектора-теоретика.

Первоначальные обстоятельства проекта располагали к привлечению моделировщика-теоретика точно так же, как это бывает в большинстве случаев. Я уже имел обширный опыт работы с большинством используемых в проекте технологий. Я даже возглавлял небольшую рабочую группу в том же проекте, пока моя роль не изменилась, так что я был знаком с процедурой разработки и средой программирования. Но даже этих факторов оказалось недостаточно для моей эффективной работы, раз моделировщик оказался отделенным от программной реализации.

Если сотрудники, пишущие код, не чувствуют своей ответственности за модель или не понимают, зачем она нужна при написании программы, то модель не имеет никакого отношения к программе. Если программисты не понимают, что изменение кода должно приводить к изменению модели, то их рефакторинг будет ухудшать модель, а не улучшать ее. Помимо этого, если моделировщик отстранен от процесса программной реализации, то он не приобретает или быстро теряет интуитивное понимание требований и особенностей реализации. Основное требование ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ **— помощь со стороны модели в эффективной программной реализации и абстрагирование ею ключевых знаний предметной области — выполняется лишь частично, и в итоге модель теряет практическую ценность. В конце концов знания и навыки опытных проектировщиков перестают передаваться другим разработчикам, если разделение труда мешает тому виду сотрудничества, который сохраняет особенности программирования на основе** ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ**.**

Потребность в моделировщиках-практиках не означает отсутствия специализации в группе разработчиков. Во всякой методологии гибкой разработки программ (*agile*-методологии), в том числе и в экстремальном программировании, между членами группы распределяются роли, и естественным образом возникают неформальные специализации. Проблема возникает тогда, когда разделяются две задачи, которые согласно принципу ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ должны "идти в связке" — моделирование и программная реализация.

Эффективность всего процесса проектирования сильно зависит от качества и согласованности множества мелких проектных и технических решений. В ПРОЕКТИРОВАНИИ

ПО МОДЕЛИ фрагмент кода есть выражение модели; если меняется код — надо менять модель. Программисты являются одновременно и моделировщиками, нравится это кому-то или нет. Поэтому лучше организовать проект так, чтобы программисты имели возможность заниматься моделированием.

Технический сотрудник, делающий свой вклад в модель, должен некоторое время работать над кодом, какую бы главенствующую роль он не играл в проекте. Любой, кому позволено вносить изменения в код, должен научиться выражать модель в этом коде. Каждый разработчик должен участвовать на каком-то уровне в дискуссиях о модели и иметь контакты со специалистами в предметной области. Те, кто работает над проектом в любом другом качестве, должны сознательно привлекать программистов, напрямую работающих с кодом, к динамичному обмену идеями модели посредством ЕДИНОГО ЯЗЫКА**.**

* * *

Резкое разграничение моделирования и программирования неэффективно, однако в больших проектах не обойтись без технических руководителей, которые бы координировали проектирование и моделирование высокого уровня, помогая в выработке наиболее трудных, критических решений. В части IV, “Стратегическое проектирование”, рассматриваются такие решения и поощряется выработка идей насчет наиболее продуктивных способов распределения ролей и ответственности в верхнем эшелоне технических сотрудников программных проектов.

Предметно-ориентированное проектирование (DDD) ставит своей задачей применение модели для решения задач, стоящих перед приложением. Посредством переработки знаний группа разработчиков фильтрует (*дистиллирует*) поток хаотической информации и делает из него работоспособную модель. ПРОЕКТИРОВАНИЕ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN) создает тесную связь между моделью и реализацией. А ЕДИНЫЙ ЯЗЫК является каналом, по которому вся нужная информация расходится между программистами, специалистами в предметной области и непосредственно программным продуктом.

В результате возникает программа, богатые функциональные возможности которой базируются на фундаментальном понимании сути предметной области, в которой она работает.

Как уже говорилось, успех ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ зависит от многочисленных деталей проектных решений, о чем и пойдет речь в нескольких следующих главах.

II

Структурные элементы предметно-ориентированного проектирования

Чтобы реализация программной системы велась четко и шла “в ногу” с моделью, несмотря на суровую хаотическую реальность, необходимо применять наиболее надежные и проверенные методы моделирования и проектирования. Эта книга не представляет собой введение в объектно-ориентированное проектирование программ, не предлагает она и какой-то радикальной теории. В *предметно-ориентированном проектировании* (*domain-driven design*) просто смещаются акценты некоторых общепринятых идей.

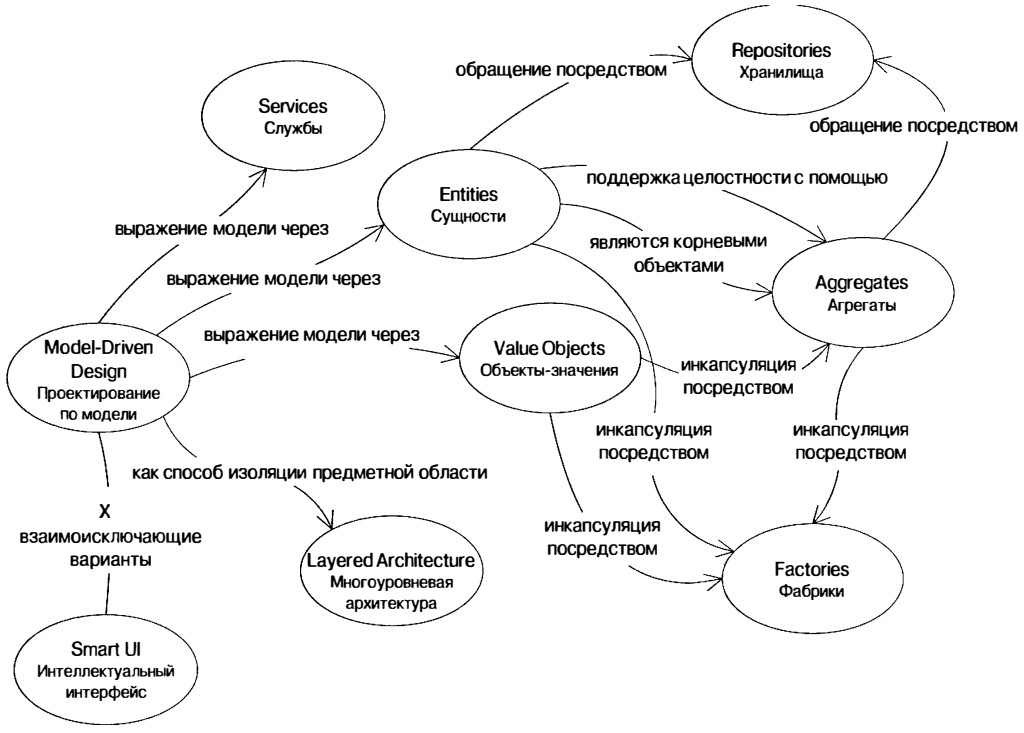
Чтобы модель и ее реализация не отставали одна от другой, усиливая эффективность друг друга, необходимо принимать определенные виды решений. Синхронизация модели и программной реализации требует внимания к деталям отдельных структурных элементов программы. Тщательность проработки в мелком масштабе дает разработчикам твердую основу, на которой можно применять стратегии и приемы моделирования из частей III и IV.

Стиль архитектурного проектирования программ в этой книге во многом следует принципу “проектирование по ответственности” (*responsibility-driven design*), выдвинутом в [24] и доработанном в [25]. Очень сильное влияние, особенно в части III, оказали также идеи “контрактного проектирования” (*design by contract*), изложенные в [19]. Все это согласуется с общими принципами других распространенных методологий объектно-ориентированного проектирования, описанными в таких книгах, как [17].

Если проект сталкивается с какими-либо препятствиями, большими или малыми, программистам начинает казаться, что упомянутые принципы неприменимы в тех или иных ситуациях. Чтобы сделать процесс предметно-ориентированного проектирования достаточно гибким, разработчикам необходимо понять, *как именно* общеизвестные фундаментальные принципы реализуются в ПРОЕКТИРОВАНИИ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN) — тогда можно будет идти на компромисс, не отходя от основ.

Материал в следующих трех главах организован в форме “языка описания шаблонов” (см. приложение), что позволяет показать, как специфические особенности моделей и проектных решений влияют на процесс предметно-ориентированного проектирования.

Схема, приведенная далее, — это карта-путеводитель. Она демонстрирует шаблоны-образцы, которые будут представлены в этом разделе, и некоторые разновидности взаимосвязей между ними.



Следование этим стандартным шаблонам упорядочивает процесс проектирования и облегчает членам группы разработчиков понимание работы друг друга. Использование стандартных шаблонов также обогащает и дополняет ЕДИНЫЙ ЯЗЫК, на котором все члены группы могут обсуждать модель и проектные решения.

Разработка хорошей модели предметной области — это искусство, творчество. Но на практике проектирование и реализацию отдельных элементов модели вполне можно выполнять по определенной системе. Отделение работ по моделированию структуры предметной области от великого множества других работ, необходимых для разработки программы, сильно облегчает привязку программной архитектуры к модели. Определение элементов модели по установленным правилам и критериям яснее очерчивает их смысл. Следование проверенным образцам для отдельных элементов помогает создать модель, удобную для реализации на практике.

Модели обширных предметных областей способны не утонуть в сложности только в том случае, если уделяется должное внимание самим основам — т.е. если детально прорабатываются элементы, которые группа разработчиков сможет затем уверенно объединить в одно целое.

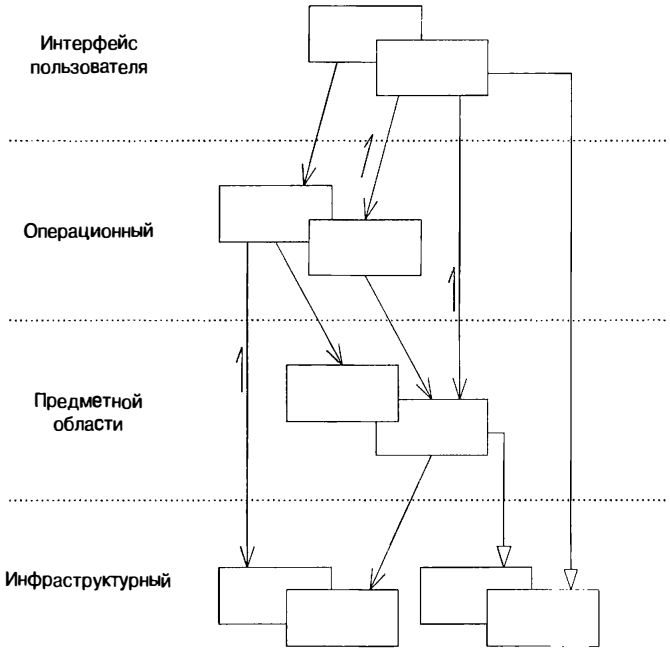
Глава 4

Изоляция предметной области

Та часть программы, которая собственно и решает задачи из предметной области, — это, как правило, лишь небольшой фрагмент всей программной системы, хотя его значение далеко превосходит размеры. Чтобы применить наиболее эффективные подходы, необходимо посмотреть на элементы модели как на систему. Не следует поддаваться тенденции просто выдергивать их из значительно большей совокупности объектов, наподобие выделения созвездий на ночном небе. Необходимо отграничить объекты предметной области от остальных функций системы и тем самым избежать путаницы между понятиями этой области и программными технологиями, а то и опасности вовсе потерять видение предмета в общей массе системы.

Существуют сложные и изощренные приемы такой изоляции предметной области. Эти вопросы хорошо изучены, но они настолько важны для успешного применения принципов моделирования на основе структуры предметной области, что осветить их хотя бы кратко необходимо, — конечно, с предметно-ориентированной точки зрения.

Многоуровневая архитектура



Для реализации в программе обслуживания грузоперевозок простого действия пользователя — выбора места назначения груза из списка городов — необходим программный код, который бы (1) выводил на экран некий интерфейс для диалога с пользователем,

(2) запрашивал в базе данных список городов, (3) интерпретировал и проверял информацию, введенную пользователем, (4) ассоциировал выбранный город с грузом и (5) отражал сделанное изменение в базе данных. Весь этот код принадлежит одной программе, но к собственно грузоперевозкам относится лишь небольшая часть его.

Структурные элементы и код программных систем предназначены для выполнения самых разных задач. В них обрабатываются данные и команды, введенные пользователем, выполняются прикладные операции, происходит обращение к базам данных, пересылаются данные по сетям, отображается информация для пользователя и т.д. Так что для реализации каждой функции программы требуется внушительный объем кода.

В объектно-ориентированной программе код для реализации интерфейса пользователя, обращений к базе данных и других технических задач нередко вписывают напрямую в объекты прикладной модели (*business objects*). Кроме того, часть прикладной логики (*business logic*), т.е. алгоритмической части программы, часто реализуется прямо в элементах интерфейса пользователя и сценариях баз данных. Это происходит потому, что так легче всего работать на ближнюю перспективу.

Когда код, относящийся к операциям предметной области, размазан по огромным объемам другого кода, его становится очень трудно разыскивать и анализировать. Поверхностные изменения в интерфейсе пользователя могут случайно затронуть и операции алгоритмической части. А чтобы изменить какие-либо правила делового регламента (*business rules*), потребуется тщательная трассировка кода интерфейса, обращений к базе данных и других элементов программы. Реализация логически последовательных, основанных на модели объектов становится непрактичной. Автоматизированное тестирование оказывается неудобным. Учитывая объем технологий и операций в каждом из этих видов работ, программу следует "удерживать" в максимально упрощенном виде, иначе понять ее станет невозможно.

Разработка программ, которые могут выполнять сложные задачи, требует разделения обязанностей, которое бы позволило сосредоточиться на принципиальных частях архитектуры в отдельности, в изоляции от других. В то же время, сложные взаимосвязи в пределах системы необходимо поддерживать, несмотря на разделение.

Существует много способов членения программной системы, но по накопленному в программной индустрии опыту и неписаным соглашениям преимущество отдается МНОГОУРОВНЕВОЙ АРХИТЕКТУРЕ (LAYERED ARCHITECTURE), состоящей из нескольких достаточно стандартных уровней. Метафора многоуровневости настолько широко распространена, что большинство программистов воспринимают ее естественно, интуитивно. В литературе о многоуровневой архитектуре написано достаточно много и хорошо, иногда даже в формате шаблона (например, [7]). Важнейший принцип многоуровневости состоит в том, что любой элемент какого-нибудь уровня зависит от работы только других элементов того же уровня или элементов более низких уровней. Передача информации наверх должна выполняться косвенными способами, о чем еще будет говориться.

Ценность многоуровневости состоит в том, что каждый уровень специализируется на конкретном аспекте программы. Такая специализация позволяет выполнять более связное проектирование каждого аспекта, и получающуюся архитектуру намного легче интерпретировать. Конечно, жизненно важно выбирать уровни так, чтобы в них изолировались наиболее важные аспекты связной архитектуры. Здесь опыт также привел к определенным соглашениям в отрасли. В наиболее успешных архитектурах используются, в том или ином виде, следующие четыре концептуальных уровня (со многими возможными вариациями).

Интерфейс пользователя (*User Interface*) или **Уровень представления** (*Presentation Layer*)	Отвечает за вывод информации пользователю и интерпретирование его команд. Внешним действующим субъектом может быть не человек, а другая компьютерная система
Операционный уровень или **Уровень прикладных операций** (*Application Layer*)	Определяет задачи, которые должна решить задача, и распределяет их между объектами, выражающими суть предметной области. Задания, выполняемые этим уровнем, имеют смысл для пользователя-специалиста или же необходимы для интерактивного взаимодействия с операционными уровнями других систем. Этот уровень не нужно "раздувать" в размерах. В нем не содержатся ни знания, ни деловые регламенты (*business rules*), а только выполняется координирование задач и распределение работы между совокупностями объектов предметной области на следующем, более низком уровне. В нем не отражается состояние объектов прикладной модели, но зато он может содержать состояние, информирующее пользователя о степени выполнения задачи для информирования пользователя
Уровень предметной области (*Domain Layer*) или **Уровень модели** (*Model Layer*)	Отвечает за представление понятий прикладной предметной области, рабочие состояния, деловые регламенты. Именно здесь контролируется и используется текущее состояние прикладной модели, пусть даже технические подробности манипуляций данными делегируются инфраструктуре. *Этот уровень является главной, алгоритмической частью программы*
Инфраструктурный уровень (*Infrastructure Layer*)	Обеспечивает непосредственную техническую поддержку для верхних уровней: передачу сообщений на операционном уровне, непрерывность существования объектов на уровне модели, вывод элементов управления на уровне пользовательского интерфейса и т.д. Инфраструктурный уровень может также брать на себя поддержку схемы взаимосвязей между четырьмя уровнями через архитектурную среду приложения

В некоторых проектах не делают четкого разграничения между уровнями пользовательского интерфейса и прикладных операций. В других есть несколько инфраструктурных уровней. Но ПРОЕКТИРОВАНИЕ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN) становится возможным только в том случае, если четко отделен *уровень предметной области.*

Разбейте сложную программу на уровни. Внутри каждого уровня разработайте связную структуру, полагающуюся только на нижние уровни и зависящую только от них. Чтобы обеспечить связь с верхними уровнями, используйте стандартные архитектурные шаблоны. Сосредоточьте весь код, относящийся к предметной модели, в одном уровне, и изолируйте его от кода интерфейса пользователя, прикладных операций и инфраструктуры. Объекты предметной области, избавленные от необходимости выводить самих себя на экран, хранить в базах данных, распределять задачи и т.п., можно сосредоточить на выражении модели этой области. Это позволяет модели раз-

виться до достаточно сложной и в то же время достаточно понятной, чтобы заключить в себе существенные знания о предмете и эффективно их применить.

Отделение уровня предметной области от уровней инфраструктуры и пользовательского интерфейса дает возможность гораздо тщательней спроектировать каждый из них. Изолированные уровни значительно легче дорабатывать и поддерживать, поскольку они имеют тенденцию развиваться разными темпами и обслуживать различные потребности. Изоляция способствует также гибкому и эффективному размещению модулей программы в распределенной системе — разные уровни можно разместить на различных серверах или клиентских станциях, чтобы минимизировать обмен данными и улучшить быстродействие [11].

Пример

Разбиение сетевой системы банковского обслуживания на уровни

Некая программа реализует различные средства и функции для управления банковскими счетами. Одна из ее функций — перевод денежных средств, для выполнения которого пользователь вводит или выбирает два номера банковских счетов и требуемую сумму, а затем инициирует перевод.

Чтобы не перегружать этот пример, я опустил все основные технические подробности, в частности вопросы безопасности. Структура предметной области тоже изрядно упрощена. (В условиях реальной сложности потребность в многоуровневой архитектуре только возросла бы.) Более того, и подразумеваемая в примере инфраструктура специально задумана как простая и ясная, чтобы не усложнять пример, — в действительности для такой программы требуется совершенно другая структура. Обязанности же оставшихся частей программы будут распределены по уровням, как показано на рис. 4.1.

Обратите внимание, что за реализацию деловых регламентов (*business rules*) отвечает не уровень прикладных операций (*application*), а уровень предметной области (*domain*). В данном случае правило регламента выражается так: “Для всякого кредита найдется соответствующий дебет”.

На операционном уровне также не делается никаких предположений об источнике запроса на перевод. Можно считать, что программа содержит интерфейс пользователя с полями ввода для номеров счетов и сумм, а также с кнопками для команд. Но этот интерфейс можно и заменить сетевым XML-запросом, никак не влияя на операционный или любой другой уровень, расположенный ниже. Это разделение важно не только потому, что в программных проектах часто бывает необходимо заменить интерфейс пользователя на обработку сетевых запросов, но и потому, что оно помогает распределять обязанности и поддерживать архитектуру уровней в удобном для понимания и доработки виде.

Фактически рис. 4.1 в какой-то мере иллюстрирует проблемы, которые возникают из-за *неизолированности* предметной области. Так как на рисунке нужно было показать все операции от запроса до управления транзакцией, уровень предметной области пришлось упростить, чтобы вся работа системы в целом осталась обозримой. А вот если сосредоточиться на проектировании только изолированного предметного уровня, то и на странице, и в головах у нас хватило бы места для модели, которая лучше представляет логику модели — даже содержит объекты главной книги, дебета и кредита, валютных транзакций.

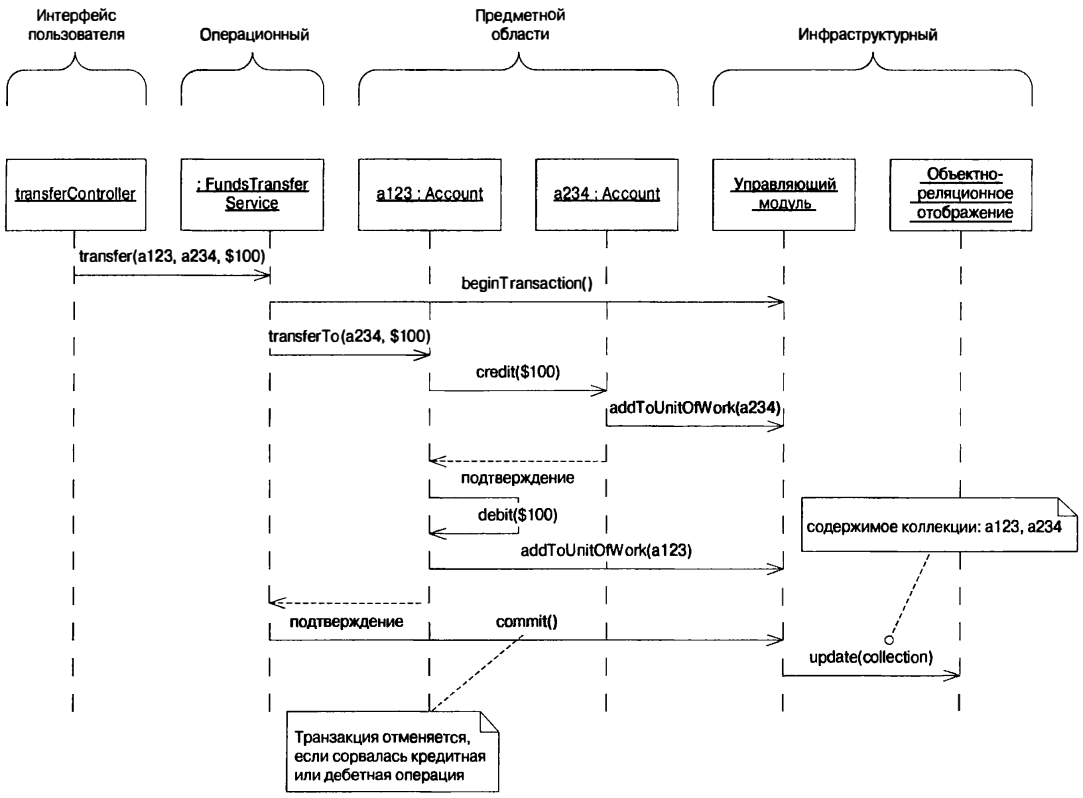


Рис. 4.1. Объекты наделены обязанностями, соответствующими их уровню, и связаны с другими объектами своего уровня

Связь между уровнями

До сих пор речь у нас шла о разделении уровней и усовершенствовании архитектуры всех частей и аспектов программы, в частности уровня предметной области (*domain layer*), благодаря этому разделению. Но, разумеется, между уровнями должна также существовать связь. Как установить такую связь и не потерять преимуществ разделения — этой проблемой задавались создатели многочисленных архитектурных шаблонов.

Уровни должны быть связаны нежестко, и зависимость проектирования одного уровня от другого должна быть направлена только в одну сторону. Верхние уровни могут использовать элементы нижних или манипулировать ими напрямую: путем вызова их общедоступных (*public*) интерфейсов, хранения ссылок на них (хотя бы временного) или каким-нибудь другим обычным для программ способом. Но если объекту нижнего уровня требуется связь с верхним (причем связь, не вписывающаяся в рамки простого ответа на прямой запрос), то для этого нужен другой механизм, основанный на таких архитектурных шаблонах взаимодействующих слоев, как *уведомление* (*callback*) или НАБЛЮДАТЕЛЬ (OBSERVER) [14].

"Дедушка всех шаблонов", связывающих интерфейс пользователя с уровнями прикладных операций и предметной области, — это *модель-представление-контроллер* (*model-view-controller*, *MVC*). Его впервые ввели в мире Smalltalk еще в 1970-е, и с тех пор на его основе было создано множество архитектур интерфейсов. В [13] рассматривается этот шаблон и несколько полезных вариаций на тему. В [17] эти проблемы нашли освещение в ШАБЛОНЕ РАЗДЕЛЕНИЯ МОДЕЛИ И ПРЕДСТАВЛЕНИЯ (MODEL-VIEW SEPARATION PATTERN); там же предлагается один из вариантов связи с операционным уровнем, КООРДИНАТОР ПРИКЛАДНЫХ ОПЕРАЦИЙ (APPLICATION COORDINATOR).

Существуют и другие способы организации связи между интерфейсом и операционным уровнем. Для наших целей подходят все эти способы, если они обеспечивают изоляцию предметной области и позволяют проектировать ее объекты, не думая о пользовательском интерфейсе и его возможных обращениях к этим объектам.

Инфраструктурный уровень обычно не инициирует никаких операций на уровне предметной области. Находясь "ниже" предметной области, он не обязан знать ничего конкретного об объектах, которые он обслуживает. Такие технические возможности чаще всего предоставляются в виде СЛУЖБ (SERVICES). Например, если приложению нужно послать письмо по электронной почте, то на инфраструктурном уровне найдется для этих целей некий почтовый интерфейс, а элементы операционного уровня смогут запросить передачу сообщения. Такое разграничение дает дополнительную гибкость. Почтовый интерфейс может подключаться к модулю отправки электронной почты, модулю отправки факсов или любому другому аналогичному средству. Но главное преимущество заключается в упрощении операционного уровня, сосредоточении его на выполнении узкого круга обязанностей: он знает, *когда* послать сообщение, но ему не нужно знать, *как* это сделать.

Итак, уровни прикладных операций (*application*) и предметной области (*domain*) обращаются к СЛУЖБАМ (SERVICES), предоставляемым инфраструктурным (*infrastructure*) уровнем. Если область действия СЛУЖБЫ выбрана удачно, а ее интерфейс хорошо спроектирован, то вызывающая сторона привязана к ней очень слабо и не подвержена влиянию той сложной функциональности, которая может стоять за интерфейсом СЛУЖБЫ.

Но не вся инфраструктура сводится к СЛУЖБАМ, вызываемым с верхних уровней. Некоторые технические компоненты проектируются так, чтобы напрямую поддерживать функционирование других уровней (например, предоставлять абстрактный базовый класс для всех объектов предметной области) и механизм, через который они взаимодействуют (например, реализации MVC и т.п.). Такая *архитектурная среда* (*architectural framework*) имеет значительно большее влияние на структуру других частей программы, чем службы.

Архитектурные среды

Если инфраструктура реализована в форме СЛУЖБ (SERVICES), вызываемых через интерфейсы, то многоуровневость и разделение уровней достигается вполне естественно, без принципиальных затруднений. Но некоторые технические проблемы требуют более "навязчивых" форм инфраструктуры. В подобных инфраструктурных *средах* (*frameworks*) реализуются в интегрированной форме нужные функции инфраструктуры, и одновременно эти среды навязывают другим уровням определенные способы реализации — например, в виде подклассов одного из классов среды или с заранее заданными сигнатурами методов. (Может показаться неестественным, что подкласс находится на уровне выше, чем родительский класс, но следует учесть, в каком из классов содержится больше знания о другом.) Лучшие архитектурные среды решают сложные технические задачи и при этом позволяют разработчику уровня предметной области сосредоточиться на выражении модели. Но среды могут и помешать работе: либо накладывая слишком много ограничений, сужающих выбор проектных решений в предметной области, либо делая реализацию столь тяжеловесной, что это тормозит разработчиков.

Обычно для работы требуется архитектурная среда в каком-то виде (кстати, группа разработчиков вполне может выбрать и такую среду, которая сослужит ей плохую службу). Используя среду, разработчики не должны забывать о своей цели: строить программную реализацию, выражающую модель предметной области, и решать с ее помощью важную прикладную задачу. Группа должна применять среду в той мере, в какой

решается главная задача, пусть при этом задействуются и не все имеющиеся возможности. Например, в ранних приложениях на основе J2EE все объекты предметной области часто реализовывались в виде хранимых объектов *Entity Beans*. Эта методика снижала как быстродействие, так и темпы разработки. В настоящее время хорошим стилем считается использовать среду J2EE для более крупномасштабных объектов, реализуя большую часть прикладной модели в виде оригинальных объектов Java. Большинства недостатков архитектурных сред удается избежать, если применять их избирательно для преодоления трудных мест, а не искать одно общее готовое решение. Скрупулезный отбор только наиболее ценных в каждом случае функций среды снижает зависимость программной реализации от этой среды, давая большую гибкость в выборе будущих проектных решений. Что еще важнее, учитывая высокую сложность работы со многими современными средами, такой минимализм помогает сохранять объекты прикладной модели выразительными и доступными для чтения.

Архитектурным средам и другим средствам разработки, конечно, предстоит еще развиваться. В более новых средах все больше технических аспектов приложений будет реализовываться автоматически. Если эту автоматизацию применять правильно, разработчики смогут все лучше концентрироваться на моделировании поведения именно прикладного уровня задачи, чем повысят производительность и качество своего труда. Но двигаясь в этом направлении, следует сдерживать энтузиазм в отношении технических решений; сложные архитектурные среды вполне способны стеснять "свободу движений" разработчиков прикладных систем.

Уровень предметной области — вместилище модели

МНОГОУРОВНЕВАЯ АРХИТЕКТУРА (LAYERED ARCHITECTURE) используется сейчас в большинстве систем с различными вариациями на тему многоуровневости. Во многих стилях программирования также в той или иной мере пользуются разделением на уровни. А вот в предметно-ориентированном проектировании программ требуется обязательное наличие только одного конкретного уровня.

Модель предметной области (*domain model*) — это набор понятий (концепций). Уровень предметной области (*domain layer*) — это выражение данной модели и все элементы программной архитектуры, непосредственно имеющие к ней отношение. Уровень предметной области образуется путем проектирования и реализации всей совокупности понятий и связей в предметной области (*business logic*). В ПРОЕКТИРОВАНИИ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN) программные конструкции уровня предметной области непосредственно отражают концепции модели.

Столь полного соответствия нет смысла добиваться, если структуру предметной области смешивают с другими функциями программы. Изоляция реализации предметной области — это обязательная предпосылка для применения такой методологии, как предметно-ориентированное проектирование.

"Антишаблон" интеллектуального интерфейса пользователя

Подытожим наш разговор о шаблоне многоуровневой архитектуры в объектных приложениях. Разделение интерфейса пользователя, уровня прикладных операций и предметной области так часто пытаются реализовать и так редко достигают этого, что его полная противоположность тоже заслуживает отдельного рассмотрения.

Во многих программных проектах принимается и должен приниматься в дальнейшем намного менее структурированный подход к проектированию, который я называю ИНТЕЛЛЕКТУАЛЬНЫЙ ИНТЕРФЕЙС ПОЛЬЗОВАТЕЛЯ (SMART UI). Это подход, альтернативный предметно-ориентированному проектированию (DDD) и категорически с ним несовместимый. Если принимается именно он, большую часть написанного в этой книге можно выбросить. Меня интересуют ситуации, в которых ИНТЕЛЛЕКТУАЛЬНЫЙ ИНТЕРФЕЙС ПОЛЬЗОВАТЕЛЯ (SMART UI) неприменим, поэтому я иронично называю его "антишаблоном". Но обсуждение этого подхода создает полезный контраст и помогает прояснить, как и почему выбирается более трудный путь, которому посвящена вся остальная часть книги.

* * *

Пусть в программном проекте должны быть реализованы простые функции, сводящиеся в основном к вводу и отображению данных, практически без сложной прикладной логики. В состав группы разработчиков не входят высококвалифицированные моделировщики объектов.

Если низкоквалифицированная группа разработчиков, работая над простым проектом, решает применить ПРОЕКТИРОВАНИЕ ПО МОДЕЛИ **(**MODEL-DRIVEN DESIGN**) с** МНОГОУРОВНЕВОЙ АРХИТЕКТУРОЙ**, то ей предстоит трудный процесс обучения. Разработчикам придется осваивать новые сложные технологии, на каждом шагу спотыкаясь при изучении объектного моделирования (а это дело сложное, даже с помощью нашей книги!). Лишние затраты на управление инфраструктурой и уровнями значительно затягивают реализацию простых задач. Простым проектам ставятся сжатые сроки реализации, и многого от них не требуется. Проект могут закрыть значительно раньше, чем разработчики закончат задачу, не говоря уже о том, чтобы продемонстрировать блестящие преимущества своей методологии.**

Но даже если группе дадут больше времени, разработчики могут не суметь овладеть необходимыми приемами без помощи специалистов. И в конце концов, даже если эти трудности будут преодолены, в итоге все равно получится простая программа. Сложные функциональные возможности от нее не требовались с самого начала.

Будь квалификация разработчиков повыше, на такие жертвы идти не пришлось бы. Закаленные программисты освоились бы быстрее и сэкономили бы время, необходимое для реализации многоуровневой архитектуры. Предметно-ориентированное проектирование дает наибольший эффект в масштабных проектах, но и требует от участников высокой квалификации. Не все проекты имеют такой размах. И не все группы разработчиков способны собрать нужные силы.

Итак, как действовать, когда этого требуют обстоятельства.

Поместите реализацию прикладной модели (*business logic*) прямо в интерфейс пользователя. Разбейте приложение на небольшие функции и реализуйте их в виде отдельных пользовательских интерфейсов, помещая правила деловых регламентов (*business rules*) непосредственно в них. Для хранения данных используйте общую реляционную базу. Пользуйтесь наиболее автоматизированными средствами построения интерфейса и визуального программирования.

Ересь! Истинное учение гласит (повсеместно, в том числе и в остальных частях этой книги), что предметная область и интерфейс должны быть отделены друг от друга! Фактически без такого разделения трудно применить какие бы то ни было методы, рассматриваемые в нашей книге далее. Поэтому ИНТЕЛЛЕКТУАЛЬНЫЙ ИНТЕРФЕЙС ПОЛЬЗОВАТЕЛЯ (SMART UI) можно считать "анти-шаблоном" в контексте предметно-ориентированного проектирования программ. Но в некоторых других контекстах он вполне допустим. По

правде говоря, он даже имеет некоторые преимущества, а в некоторых ситуациях просто незаменим — в частности, поэтому он столь широко распространен. Здесь мы рассмотрели его для того, чтобы понять, почему нужно отделять уровень прикладных операций от предметной области, а также, что немаловажно, в каких случаях этого делать не нужно.

Преимущества.

- Высокая производительность труда и мгновенный результат в простых приложениях.
- Менее квалифицированным разработчикам практически не требуется дополнительное обучение, чтобы приступить к работе.
- Даже недостатки в анализе технического задания можно преодолеть, выпустив прототип для пользователей и затем быстро внеся нужные изменения.
- Функции и операции отделены друг от другой, так что можно более-менее точно спланировать сроки готовности небольших модулей. Расширение системы за счет дополнительных простых функций выполняется легко.
- Реляционные базы данных справляются с задачей и обеспечивают интеграцию на уровне данных.
- Применимы средства 4GL.
- При передаче приложения в другие руки программисты-доработчики смогут быстро переделать фрагменты кода, которых они не понимают, потому что эффект внесенных изменений должен локализоваться в каждом отдельном интерфейсе.

Недостатки.

- Сложно осуществить интеграцию приложений — разве что через базу данных.
- Отсутствует переносимость функциональных модулей и абстракция прикладных моделей. Правила прикладной модели (*business rules*) приходится дублировать в каждой операции, связанной с ними.
- Быстрое создание опытных образцов (прототипов) с последующим итерированием затруднено, поскольку недостаток абстрагирования ограничивает возможность рефакторинга.
- Сложность такой системы быстро похоронит все надежды, так что развитие возможно только в сторону дополнительных простых приложений. Невозможно "изящно" реализовать сложные функции и сложное поведение системы.

Если этот шаблон применить сознательно, группе разработчиков удастся избежать больших дополнительных трудозатрат, с которыми связано использование других методологий. Широко распространена такая ошибка, как обращение к сложной методологии проектирования без намерения хранить ей "верность до конца". Еще одна распространенная и дорогостоящая ошибка — это построить сложную инфраструктуру и воспользоваться мощными профессиональными средствами для реализации проекта, в котором они вовсе не нужны.

Большинство гибких языков программирования (таких как Java) для подобных проектов слишком изощренны, и их применение дорого обойдется. Следует пользоваться чем-то в стиле 4GL.

Помните: одно из закономерных следствий описываемого шаблона состоит в том, что к другой методологии проектирования невозможно перейти, не заменяя всю программу целиком. Простой переход на язык общего назначения типа Java сам по себе не даст возможности позже легко уйти от шаблона интеллектуального интерфейса, так что если вы

выбрали этот путь, для него следует подобрать и средства программирования. Не надо перестраховываться без нужды. Одним только применением гибкого языка не сделаешь всю систему гибкой, а вот слишком дорогой — запросто.

Аналогично, если разработчики привержены принципам ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN), то им следует работать по ним с самого начала. Разумеется, даже опытные коллективы с большими амбициями должны начинать с самых простых функций и двигаться вперед в режиме последовательных приближений. Но все эти приближения, пусть предварительные, будут построены на модели с изолированным уровнем предметной области, иначе проект так и застрянет на стадии ИНТЕЛЛЕКТУАЛЬНОГО ИНТЕРФЕЙСА (SMART UI).

* * *

Шаблон ИНТЕЛЛЕКТУАЛЬНОГО ИНТЕРФЕЙСА (SMART UI) рассматривается здесь только с целью прояснить, почему и когда для изоляции предметной области необходимо применять такой шаблон, как МНОГОУРОВНЕВАЯ АРХИТЕКТУРА (LAYERED ARCHITECTURE).

Существуют и другие архитектурные решения в промежутке между ИНТЕЛЛЕКТУАЛЬНЫМ ИНТЕРФЕЙСОМ (SMART UI) и МНОГОУРОВНЕВОЙ АРХИТЕКТУРОЙ (LAYERED ARCHITECTURE). Например, в [13] описан ТРАНЗАКЦИОННЫЙ СЦЕНАРИЙ (TRANSACTION SCRIPT), который отделяет интерфейс от уровня прикладных операций, но не предназначен для реализации объектной модели. Так что, подводя итог, надо сказать следующее. *Если та или иная архитектура изолирует код предметной области таким образом, что внутренне связную реализацию предметной области можно без труда выделить из остальной части системы, то такая архитектура, вероятно, сможет поддерживать предметно-ориентированное проектирование.*

Другие стили программирования тоже имеют свое место под солнцем, и нужно правильно выбирать подход, балансируя на грани сложности и гибкости. Отказ от четкого выделения предметной области при определенных условиях может обернуться катастрофой. Если задача предстоит сложная и принято решение следовать принципам ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN), то стисните зубы, пригласите нужных специалистов и избегайте ИНТЕЛЛЕКТУАЛЬНОГО ИНТЕРФЕЙСА (SMART UI).

Другие виды изоляции

К сожалению, повредить хрупкому организму модели предметной области могут не только инфраструктура и интерфейс пользователя. Приходится иметь дело с другими компонентами предметной области, которые не вполне интегрированы в модель. Приходится работать с другими группами разработчиков, которые используют другую модель той же предметной области. Эти и другие факторы могут испортить модель и лишить ее полезности. В главе 14 рассматривается именно этот вопрос; там вводятся такие шаблоны, как ОГРАНИЧЕННЫЙ КОНТЕКСТ (BOUNDED CONTEXT) и ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ (ANTICORRUPTION LAYER). Очень сложная модель предметной области сама по себе может стать громоздкой и неповоротливой. В главе 15 рассматривается проблема, как проводить различение и разграничение в пределах уровня предметной модели, чтобы не путать наиболее существенные понятия с второстепенными подробностями.

Но все это будет позже. А в ближайших главах мы разберем в деталях, как же совместно развивать эффективную модель предметной области и выразительную ее реализацию. В конце концов, изоляция предметной области на то и нужна, чтобы убрать все постороннее с дороги и сосредоточиться на моделировании прикладной проблемы.

Глава 5

Модель, выраженная в программе

Чтобы суметь найти компромиссы в программной реализации и при этом не потерять преимуществ ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN), нужно начинать с пересмотра азов. Связь между моделью и реализацией необходимо прорабатывать на уровне деталей. В этой главе мы сосредоточимся на отдельных элементах модели и придадим им форму, для того чтобы иметь возможность двигаться дальше в следующих главах.

Начнем с вопросов проектирования и рационализации ассоциаций. Ассоциации между объектами легко себе представить и нетрудно нарисовать на схеме, но их реализация может завести Бог знаст куда. Именно ассоциации показывают, насколько важно в ПРОЕКТИРОВАНИИ ПО МОДЕЛИ принимать детализированные проектные решения.

Переходя к самим объектам, но при этом продолжая "препарировать" соотношения между детализированными решениями в модели и вопросами программной реализации, мы сосредоточимся на различиях между тремя шаблонами элементов модели, которые ее выражают: СУЩНОСТЯМИ (ENTITIES), ЗНАЧЕНИЯМИ (VALUE OBJECTS) и СЛУЖБАМИ (SERVICES).

Определение объектов, заключающих в себе понятия модели, поначалу кажется интуитивно очевидным, но оттенки смысловых значений могут заставить всерьез задуматься. Существуют определенные нюансы, которые проясняют смысл элементов модели и в совокупности образуют методы проектирования специфических типов объектов.

Представляет ли объект собою нечто, имеющее протяженность и обособленность — то, что можно проследить в разных его состояниях или даже разных реализациях? Или же это атрибут, который описывает состояние какого-то другого объекта? В этом и состоит основное различие между СУЩНОСТЬЮ (ENTITY) и ОБЪЕКТОМ-ЗНАЧЕНИЕМ (VALUE OBJECT). Объявление объектов в четком соответствии с одним или другим шаблоном устраняет двусмысленность и многозначность, прокладывая путь к надежным проектным решениям.

Есть и такие аспекты предметной области, которые выражаются более удобно в виде действий или операций, а не объектов. Несмотря на некоторый отход от традиций объектно-ориентированного моделирования, такие аспекты лучше выражать в виде СЛУЖБ (SERVICES), а не навязывать ответственность за эти операции сущностям или объектам-значениям. СЛУЖБА (SERVICE) — это нечто, срабатывающее в ответ на запрос клиента. На технических уровнях программного обеспечения существует множество служб. Появляются они и на уровне предметной области при моделировании некоей деятельности, которая соответствует операциям программы, но не ассоциируется ни с каким ее состоянием.

Неизбежно возникают ситуации, в которых приходится идти на компромисс и отходить от "чистой" объектной модели — например, для хранения данных в реляционной базе данных. В этой главе будут даны некоторые рекомендации, как не свернуть с правильного пути в обстоятельствах, когда приходится иметь дело с суровой реальностью.

Наконец, в разговоре о МОДУЛЯХ (MODULES) мы поймем, почему каждое проектное решение должно основываться на умозаключении, выведенном из модели. Идеи высокой внутренней связности (*high cohesion*) и низкой внешней зависимости (*low coupling*), которые часто относят исключительно к техническим аспектам программирования, можно применять и к самим понятиям модели. В методологии проектирования по модели МОДУЛИ составляют часть модели, поэтому должны отражать понятия предметной области.

В этой главе все упомянутые структурные элементы собираются в одно целое — в модель, используемую для разработки программного обеспечения. Эти идеи общеизвестны, а об особенностях моделирования и проектирования, которые из них следуют, уже много писали раньше. Но встраивание их в наш контекст поможет разработчикам создавать детально проработанные компоненты, реализующие принципы DDD при анализе проблем в больших моделях и архитектурах. И, конечно, приверженность базовым принципам поможет разработчикам не свернуть с пути при неизбежных компромиссах.

Ассоциации

Когда требуется соблюсти однозначное соответствие между моделью и программной реализацией, труднее всего приходится при создании ассоциаций между объектами.

Для всякой прослеживаемой в модели ассоциации должен существовать механизм в программе, обладающий теми же свойствами.

Модель, содержащая ассоциацию между клиентом-покупателем и торговым представителем, соответствует двум аспектам. Во-первых, она в абстрактной форме представляет те отношения, которые разработчики считают существенными между этими двумя реальными людьми. Во-вторых, она соответствует объектному указателю между двумя объектами Java, или же инкапсуляции поиска по базе данных, или какому-то аналогичному способу программной реализации.

Например, ассоциацию "один ко многим" можно реализовать в виде коллекции в переменной-экземпляре. Но вовсе не обязательно делать это так прямолинейно. Коллекции может и не быть; вместо этого метод доступа к атрибуту может запрашивать в базе данных поиск соответствующих записей и создавать экземпляры объектов на их основе. Оба варианта отражают одну и ту же модель. Но в архитектуре программы следует реализовать конкретный механизм, поведение которого в точности соответствует ассоциации в модели.

В реальной жизни существует множество ассоциаций "многие ко многим", и значительная часть из них — естественно двунаправленного характера. То же самое справедливо и в отношении ранних версий моделей, создаваемых в ходе мозговых штурмов и исследований предметной области. Но слишком общие ассоциации усложняют реализацию и доработку программы. Более того, они несут слишком расплывчатую информацию о природе тех или иных отношений.

Существует как минимум три способа сделать ассоциации более удобными и управляемыми.

1. Свести отношения к однонаправленным, прослеживаемым в одном направлении.
2. Добавить квалификаторы, тем самым снижая кратность.
3. Устранить несущественные ассоциации.

Важно ограничить взаимосвязи до минимума. Двунаправленные ассоциации означают, что два объекта можно понять только совместно, как одно целое. Если техническое задание непосредственно не требует двунаправленных отношений в том или ином слу-

чае, то сведение их к однонаправленным снижает взаимозависимость и упрощает структуру программы. Исследование предметной области часто выявляет естественные асимметричные связи.

В США, как и в других странах, было много президентов. Это двунаправленное отношение "один ко многим". И все-таки мы редко начинаем с имени "Джордж Вашингтон" и задаем вопрос "Президентом какой страны он был?". С практической точки зрения эту связь можно считать однонаправленной ассоциацией, прослеживаемой от страны к президенту. Это усовершенствование отражает как лучшее понимание модели, так и практичность архитектуры. В нем заключено утверждение, что одно из направлений ассоциации более важно и значимо, чем другое. При этом класс **Человек (Person)** оказывается независимым от значительно менее фундаментального понятия **Президент (President)**.

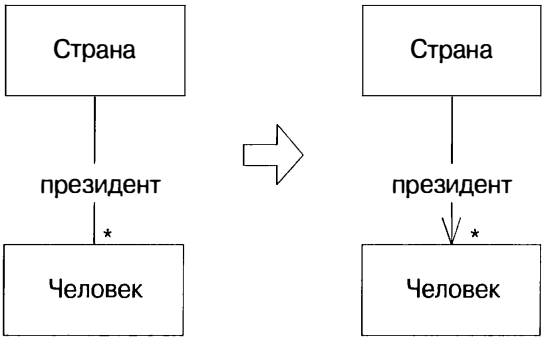


Рис. 5.1. Естественная асимметрия в предметной области отражается в виде однонаправленной ассоциации

Очень часто углубленное понимание модели порождает квалификатор к той или иной ассоциации. Если глубже задуматься над примером с президентами, то можно сделать вывод, что в стране может быть одновременно не более одного президента (если не считать времена гражданской войны). Этот квалификатор уменьшает кратность до отношения "один к одному" и внедряет это правило в модель явным образом. Теперь вопрос: кто был президентом США в 1790 году? Ответ: Джордж Вашингтон.

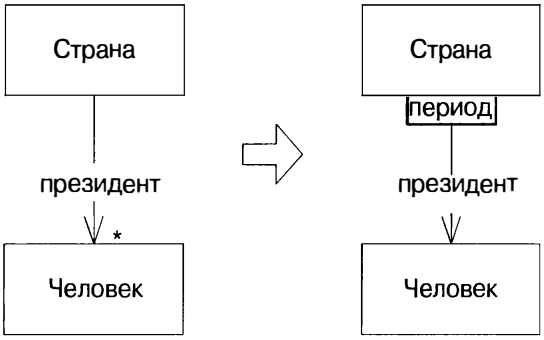


Рис. 5.2. Ограниченные ассоциации передают больше знаний и порождают более практичную архитектуру

Выбор главного направления в ассоциации "многие ко многим" фактически сужает ее реализацию до ассоциации "один к одному", а это уже гораздо более простая структура.

Если последовательно ограничить ассоциации с тем, чтобы выразить естественную асимметрию предметной области, это не только упрощает программную реализацию и делает их более информативными, но и придает большую значимость оставшимся дву-

направленным ассоциациям. Если двунаправленность ассоциации является семантической характеристикой явления, если она существенна для функционирования программы, то такую ассоциацию стоит сохранить.

Самым "сильным" упрощением, конечно, будет полное устранение ассоциации, если она несущественна для работы или фундаментального понимания объектов модели.

Пример

Ассоциации в модели брокерского счета

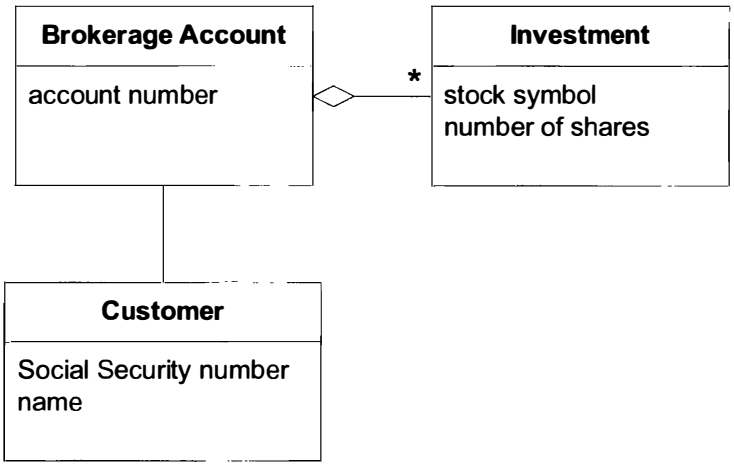


Рис. 5.3.

Одной из реализаций брокерского счета (**Brokerage Account**) в этой модели на языке Java будет следующая.

```
public class BrokerageAccount {
    String accountNumber;
    Customer customer;
    Set investments;
  // Конструкторы и т.п. опущены

  public Customer getCustomer() {
    return customer;
  }
  public Set getInvestments() {
    return investments;
  }
}
```

Но если необходимо загружать данные из реляционной базы данных, то лучше применить другую реализацию, в той же мере согласующуюся с моделью.

Таблица: BROKERAGE_ACCOUNT

ACCOUNT_NUMBER	CUSTOMER_SS_NUMBER

Таблица: CUSTOMER

SS_NUMBER	NAME

Таблица: INVESTMENT

ACCOUNT_NUMBER	STOCK_SYMBOL	AMOUNT

```
public class BrokerageAccount {
  String accountNumber;
  String customerSocialSecurityNumber;

  // Опущены конструкторы и т.п.

  public Customer getCustomer() {
    String sqlQuery =
      "SELECT * FROM CUSTOMER WHERE" +
      "SS_NUMBER='"+customerSocialSecurityNumber+"'";
    return QueryService.findSingleCustomerFor(sqlQuery);
  }
  public Set getInvestments() {
    String sqlQuery =
      "SELECT * FROM INVESTMENT WHERE" +
      "BROKERAGE_ACCOUNT='"+accountNumber+"'";
    return QueryService.findInvestmentsFor(sqlQuery);
  }
}
```

(*Примечание.* Утилита `QueryService`, используемая для извлечения строк из базы данных и создания объектов, удобна в пояснительных примерах, но не всегда — в реальном проекте.)

Давайте усовершенствуем модель, добавив квалификатор к ассоциации между **Брокерским счетом (Brokerage Account)** и **Инвестицией (Investment)** с целью уменьшить ее кратность. Квалификатор гласит: разрешается только одна инвестиция на одну акцию.

Это не будет справедливо во всех рабочих ситуациях (например, при работе с лотами — партиями ценных бумаг). Но независимо от конкретных правил, по мере обнаружения ограничений на ассоциации их следует добавлять в модель и в реализацию. От этого модель становится более точной, а программная реализация — более удобной в работе.

Реализация на Java могла бы выглядеть так.

```
public class BrokerageAccount {
  String accountNumber;
  Customer customer;
  Map investments;

  // Опущены конструкторы и т.п.
  public Customer getCustomer() {
```

```
    return customer;
  }
  public Investment getInvestment(String stockSymbol) {
    return (Investment)investments.get(stockSymbol);
  }
}
```

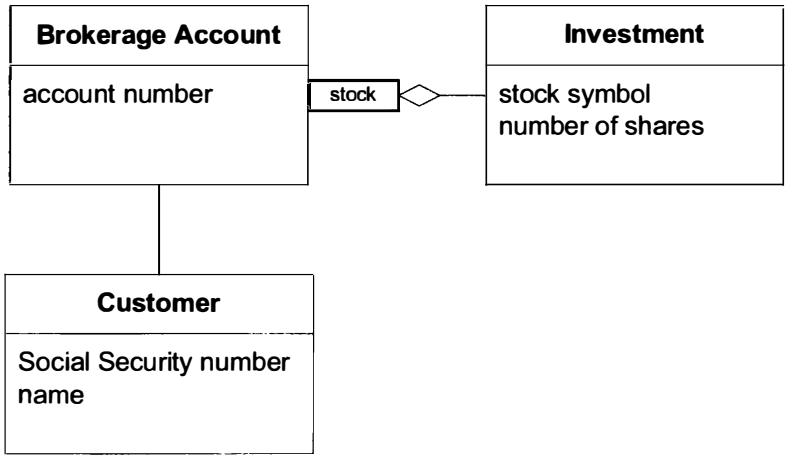


Рис. 5.4.

Реализация с применением SQL будет такой.

```
public class BrokerageAccount {
  String accountNumber;
  String customerSocialSecurityNumber;

  // Опущены конструкторы и т.п.
  public Customer getCustomer() {
    String sqlQuery = "SELECT * FROM CUSTOMER WHERE SS_NUMBER='"
      + customerSocialSecurityNumber + "'";
    return QueryService.findSingleCustomerFor(sqlQuery);
  }
  public Investment getInvestment(String stockSymbol) {
    String sqlQuery = "SELECT * FROM INVESTMENT "
      + "WHERE BROKERAGE_ACCOUNT='" + accountNumber + "'"
      + "AND STOCK_SYMBOL='" + stockSymbol +"'";
    return QueryService.findInvestmentFor(sqlQuery);
  }
}
```

Тщательная дистилляция и накладывание ограничений на ассоциации модели — это большой шаг к ПРОЕКТИРОВАНИЮ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN). Теперь обратимся к самим объектам. Существуют нюансы, которые делают модель более понятной и одновременно ведут к более практичной реализации.

Сущности (указуемые объекты)

Многие объекты[1] не сводятся к набору атрибутов, а характеризуются непрерывностью и индивидуальностью существования[2].

* * *

На меня как-то подала в суд домовладелица, желая взыскать возмещение ущерба, причиненного ее имуществу. Предоставленные мне документы описывали квартиру с дырами в стенах, с пятнами на ковре, с какой-то ядовитой жидкостью в раковине, от едких испарений которой на кухне облезли обои. В судебных документах меня именовали квартиросъемщиком, несущим ответственность за этот ущерб, и обозначали по имени и тогдашнему адресу. Это было непонятно, так как я никогда в жизни не бывал в пострадавшей квартире.

Но почти сразу же я догадался, что меня, должно быть, просто приняли за другого человека. Я позвонил истице и объяснил ей это, но она не поверила. Бывший жилец несколько месяцев скрывался от нее. Как я мог доказать, что я не тот человек, из-за которого она понесла такие убытки? Ведь я был единственным Эриком Эвансом в телефонном справочнике.

Что ж, именно телефонная книга и стала моим спасением. Я жил в одной и той же квартире уже два года, поэтому спросил истицу, есть ли у нее справочник прошлого года. После того как она его нашла и убедилась, что мои данные в нем были теми же, что и сейчас (причем рядом с данными моего тезки), она поняла, что я не тот человек, на которого следует подавать в суд, извинилась и пообещала снять с меня обвинение.

Но компьютеры такой изобретательностью не отличаются. Подобная путаница в программной системе чревата повреждением данных и ошибками выполнения.

Здесь имеются некоторые проблемы технического характера, о которых будет сказано позже, а пока рассмотрим фундаментальную теоретическую проблему: многие вещи определяются не частными атрибутами, а индивидуальным, неповторимым существованием. В нашем обычном понимании человек (если продолжить наш нетехнический пример) имеет индивидуальное существование, которое продолжается от рождения до смерти

---

[1] В заголовке употребляются термины *entity* (сущность, самостоятельная логическая единица) и *reference object* (называемый, обозначаемый, указуемый объект). — *Примеч. перев.*

[2] Автор употребляет здесь слово *identity*, которое означает и набор идентификационных данных типа имени, фамилии, адреса, и индивидуальное, уникальное существование некоего объекта или субъекта. Это дает ему право пояснять свои мысли на приведенном далее примере. В русском языке приходится использовать целый комплекс терминов: индивидуальность, индивидуальное существование, идентичность, идентификация и т.п. — *Примеч. перев.*

и даже после нее. Физические атрибуты человека изменяются, а со временем и вовсе исчезают. Может измениться его имя. Завязываются и обрываются финансовые взаимоотношения. У человеческой личности нет ни единого атрибута, который бы не мог измениться, а вот его индивидуальность сохраняется. Тот ли я человек, которым был в пятилетнем возрасте? Подобный метафизический вопрос немаловажен при поиске эффективных моделей предметных областей. Если слегка перефразировать: имеет ли *для пользователя программы* значение, был или не был я тем же человеком в свои пять лет?

В программной системе для отслеживания задолженностей по счетам скромный объект "клиент" может иметь более колоритные черты. Он приобретает кредитную репутацию своевременной оплатой или же перенаправляется в коллекторское агентство для взимания долговых обязательств. Может вообще оказаться, что он ведет двойную жизнь в совершенно другой системе, когда банковский агент извлечет его данные в свою программу управления контактами. В любом случае данные клиента бесцеремонно "штампуют", подгоняя под нужный формат для хранения в таблице базы данных. Как только из этого источника перестают поступать новые запросы и распоряжения, объект клиента переводится в архив, где и сохраняется тень его прежней индивидуальности.

Каждая из этих форм объекта-клиента реализуется по-разному, с применением разных языков и технологий программирования. Но когда нужно поднять трубку и предъявить денежную претензию, необходимо знать: тот ли это клиент, на счету которого зафиксированы нарушения? Тот ли это клиент, о котором Джек (конкретный агент) собирает информацию уже несколько недель? Или же это совершенно новый клиент?

Индивидуального субъекта приходится опознавать, сравнивая различные реализации объектов, их постоянно хранимые формы, а также внешних агентов наподобие телефонных абонентов. Атрибуты при этом могут различаться. Агент мог ввести новый адрес в программе управления контактами, которая только начинает работать с должниками. У двух клиентов может быть одно и то же имя. В распределенных программных системах данные могут вводиться несколькими пользователями из разных источников, из-за чего транзакции обновления "ходят" по всей системе и согласуются в разных базах данных асинхронно.

В объектном моделировании главное внимание обычно фокусируется на атрибутах объекта, но для фундаментального понятия СУЩНОСТИ главное — не атрибуты, а абстрактное непрерывное существование в течение всего жизненного цикла, даже с переходом в различные формы.

Для некоторых объектов определение через атрибуты не является главным. Они представляют собой индивидуально существующие логические единицы (*сущности*), протяженные во времени и часто проходящие через несколько различных представлений. Иногда такой объект приходится ставить в соответствие другому объекту, хотя их атрибуты отличаются. Или же объект требуется отличать от другого объекта, хотя их атрибуты могут быть одинаковыми. Путаница в объектах может привести к искажению данных.

Логически целостный объект, определяемый совокупностью индивидуальных черт, называется СУЩНОСТЬЮ (ENTITY)[3]. Для таких объектов существуют особые принципы моделирования и проектирования. На протяжении их цикла существования у них может радикально меняться и форма, и содержание, но непрерывность этого существования

---

[3] Модель объекта ENTITY — это совсем не то, что объект *entity bean* в Java. Последний задумывался как основа для реализации объектов-единиц, но так и не стал таковой. Большинство сущностей-*entity* реализуются в виде обыкновенных объектов. Но независимо от способа реализации, такие сущнос ги являются принципиально важными в модели предметной области.

обязана поддерживаться. Они должны идентифицироваться таким образом, чтобы их можно было однозначно отследить. Определения классов, обязанностей, атрибутов и ассоциаций для таких объектов следует строить вокруг их смыслового значения, а не присвоенных им атрибутов. Но даже если некоторые объекты-СУЩНОСТИ не подвергаются таким радикальным изменениям или не имеют такого сложного цикла существования, их все равно полезно отнести к этой семантической категории, потому что такие модели будут более ясными, а их реализации — более надежными.

Большинство логических "сущностей" в программной системе не представляют собой физические или юридические лица[4]. СУЩНОСТЬ — это все то, что сохраняет свое индивидуальное существование и отличие на протяжении срока "жизни", независимо от атрибутов, важных для пользователя приложения. Это может быть человек, город, автомобиль, лотерейный билет или банковская транзакция.

С другой стороны, отчетливо индивидуальным существованием бывают наделены далеко не все объекты модели. Этот важный момент "затуманивается" тем фактом, что в объектно-ориентированных языках операции проверки идентичности (например, "==" в Java) автоматически встраиваются в каждый объект. Эти операции определяют, указывают ли две ссылки на один и тот же объект, сравнивая их положение в памяти или используя какой-нибудь другой механизм. В этом смысле любой экземпляр объекта является индивидуальной логической единицей, сущностью. Если предметная область — это, скажем, создание исполняемой среды Java или технических средств для поддержки локального кэширования удаленных объектов, то каждый экземпляр объектов в них действительно будет СУЩНОСТЬЮ. Но в других прикладных областях такой механизм индивидуальной идентификации значит весьма мало. Индивидуальное существование — это тонкий смысловой атрибут СУЩНОСТЕЙ, который невозможно превратить в стандартное автоматизированное средство языка.

Рассмотрим транзакции в банковских операциях. Два депозита на одну и ту же сумму в один и тот же день являются различными транзакциями, у них есть индивидуальность, и поэтому они — СУЩНОСТИ. С другой стороны, атрибуты "сумма" в этих двух транзакциях, скорее всего, представляют собой экземпляры некоего объекта "деньги". У этих значений никакого индивидуального существования нет, потому что различать их никому не надо. Фактически два объекта могут не иметь индивидуальных отличий друг от друга, даже если у них нет одинаковых атрибутов; более того, они могут даже не принадлежать одному классу. Когда клиент банка сверяет транзакции из банковского баланса с расчетными операциями, зафиксированными в его собственных учетных документах, перед ним стоит задача найти одинаковые транзакции, пусть даже они фиксировались разными людьми в разные дни (конечно, при условии, что баланс подсчитан после всех взаимных расчетов). Уникальным идентификатором для этой цели призван служить номер чека — неважно, выполняется обработка данных вручную или программно. Снятие наличных и помещение средств на депозит при отсутствии такого идентификационного номера представляют большие трудности, но принцип здесь тот же: каждая транзакция это СУЩНОСТЬ, проявляющаяся как минимум в двух формах.

Индивидуальность объекта-СУЩНОСТИ, как правило, является весомым фактором и за пределами компьютерных систем, — например, в приведенных примерах банковских транзакций и квартиросъемщиков. Но иногда индивидуальное существование возможно только в контексте программной системы, — например, в виде вычислительного процесса.

---

[4] Это пояснение автора стало необходимым только потому, что одно из значений слова *entity* в английском языке — предприятие, фирма, юридическое лицо. *Примеч. перев.*

Если объект определяется уникальным индивидуальным существованием, а не набором атрибутов, это свойство следует считать главным при определении объекта в модели. Определение класса должно быть простым и строиться вокруг непрерывности и уникальности цикла существования объекта. Найдите способ различать каждый объект независимо от его формы или истории существования. С особым вниманием отнеситесь к техническим требованиям, связанным с сопоставлением объектов по их атрибутам. Задайте операцию, которая бы обязательно давала неповторимый результат для каждого такого объекта, — возможно, для этого с объектом придется ассоциировать некий символ с гарантированной уникальностью. Такое средство идентификации может иметь внешнее происхождение, но это может быть и произвольный идентификатор, сгенерированный системой для ее собственного удобства. Однако такое средство должно соответствовать правилам различения объектов в модели. В модели должно даваться точное определение, *что такое одинаковые объекты*.

Индивидуальность, индивидуальное существование — это не имманентное свойство вещей в природе, а качество, всего лишь приписываемое им из-за его полезности. На самом деле одна и та же реально существующая в природе вещь может как представляться, так и не представляться объектом-СУЩНОСТЬЮ в модели предметной области.

Например, в программе резервирования мест на стадионе как зрители, так и места могут считаться СУЩНОСТЯМИ. В случае билетов *с местами*, где на каждом билете напечатан номер строго определенного места, это место действительно является сущностью. Идентификатор этой сущности — номер места, и в рамках данного стадиона он уникален. Место может иметь много других атрибутов, таких как точное местоположение, качество вида на игровое поле, цену, но только номер места (или пара "ряд-место") может использоваться для точного определения и различения мест.

С другой стороны, если билеты продаются *без места*, т.е. купившие их зрители садятся на любые еще не занятые места, различать отдельные места нет никакой необходимости. Значение имеет только общее количество мест. Хотя номера мест по-прежнему физически нанесены на сиденья, программа не обязана за ними следить. Вообще-то, в этом случае ассоциировать номера мест с билетами в модели было бы прямой ошибкой, потому что при продаже билетов "без мест" такое требование отсутствует. Здесь места не являются СУЩНОСТЯМИ, и идентификатор для них не нужен.

* * *

Моделирование СУЩНОСТЕЙ

При моделировании объекта совершенно естественно думать о его атрибутах, и очень важно думать о его поведении, рабочих функциях. Но основная функция СУЩНОСТИ — поддерживать непрерывность своего существования таким образом, чтобы ее поведение было понятным и предсказуемым. Лучший способ добиться этого — умеренность и экономия. Вместо того чтобы сосредоточиться на атрибутах или даже на рабочих функциях объекта, следует ограничить определение объекта-СУЩНОСТИ наиболее неотъемлемыми его характеристиками, т.е. теми, которые однозначно выделяют его среди других и обычно используются для его поиска и сравнения. Задавайте только те рабочие функции, которые существенны для создания понятия об объекте, и только те атрибуты, которых требуют эти функции. Все атрибуты и функции, которые выходят за эти рамки, старайтесь выносить в другие объекты, ассоциированные с главной СУЩНОСТЬЮ. Некоторые из них тоже будут СУЩНОСТЯМИ, а некоторые — ОБЪЕКТАМИ-ЗНАЧЕНИЯМИ (VALUE OBJECTS); это следующий шаблон, который мы рассматриваем в этой главе. Кроме индивидуальности и непрерывности существования, для СУЩНОСТЕЙ часто характерно то, что они выполняют свои функции, координируя операции объектов, которые им принадлежат.

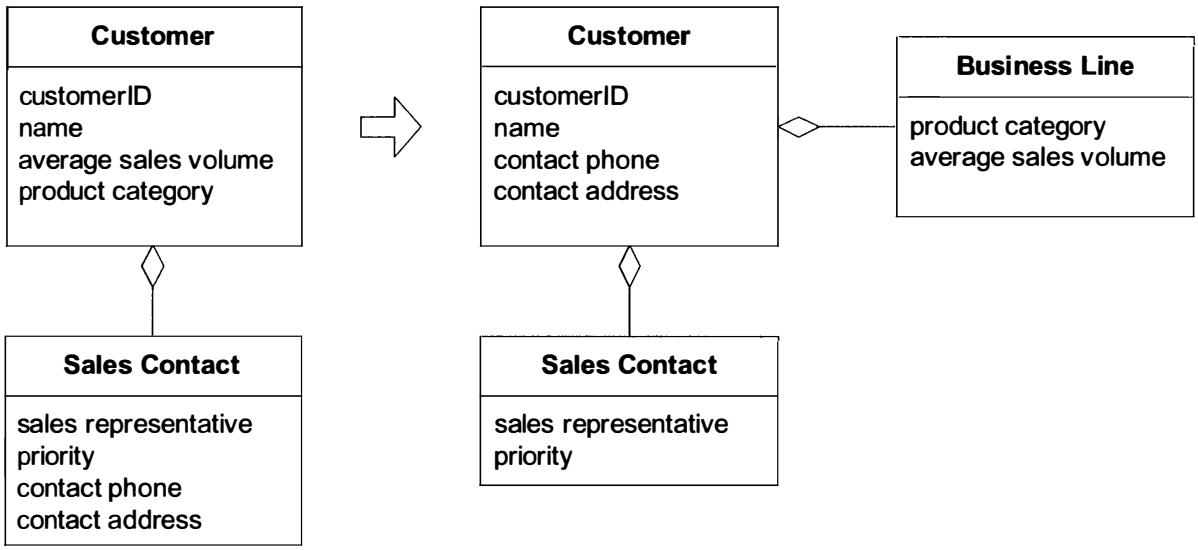


Рис. 5.5. Атрибуты, неотъемлемые от индивидуально существующего объекта, приписываются СУЩНОСТИ

Атрибут `customerID` — это единственный реальный идентификатор объекта **Клиент** (**Customer**) на рис. 5.5, но для поиска или сопоставления личности **Клиента** может также использоваться номер телефона или адрес. Имя *не определяет* индивидуальность человека, но входит в число средств, позволяющих проверить ее подлинность. В данном примере атрибуты номера телефона и адреса помещены в объект **Клиент**, но в реальном проекте для такого решения были бы приняты во внимание типичные способы различения или сопоставления клиентов. Например, если у **Клиента** есть много номеров телефонов для разных целей, то номер нельзя ассоциировать с индивидуальной личностью клиента, и его следует вынести в **Деловые контакты** (**Sales Contact**).

Проектирование операций идентификации

Для каждого объекта-СУЩНОСТИ должен быть задан способ сопоставления его с другим объектом — такой, который бы позволял различать их даже в случае совпадения описательных атрибутов. Идентифицирующий атрибут должен гарантированно оставаться уникальным в пределах системы, независимо от способа ее определения — даже в распределенной системе, даже в архиве.

Как уже говорилось, в объектно-ориентированных языках имеются операции проверки идентичности, которые помогают определить, указывают ли две ссылки на один и тот же объект, путем сравнения положений объектов в памяти. Этот способ для наших целей слишком ненадежен. В большинстве технологий длительного хранения объектов при каждом извлечении объекта из базы данных создается новый экземпляр, и первоначальная его индивидуальность теряется. Проблема может еще усугубиться, если в системе одновременно сосуществуют несколько версий одного и того же объекта — например, при распространении обновлений по распределенной базе данных.

Даже если у нас есть среда программирования, упрощающая решение технических задач, остается принципиальный вопрос: как узнать, представляют ли два объекта одну и ту же концептуальную СУЩНОСТЬ? Определение идентичности возникает из модели. Чтобы построить такое определение, нужно понимать предметную область.

Иногда некоторые атрибуты данных или комбинации атрибутов гарантированно являются уникальными в пределах системы, — например, в силу наложенных на них тре-

бований. При таком подходе у СУЩНОСТИ есть неповторимый ключ, который ее однозначно идентифицирует. Например, ежедневную газету можно идентифицировать по названию, городу и дате выпуска. (Если не считать внеочередных выпусков и возможных изменений названия.)

Если из атрибутов объекта нельзя составить действительно уникальный ключ, то есть другое типовое решение — каждому экземпляру присвоить символ-идентификатор (число или текстовую строку), не повторяющийся в пределах класса. После создания идентификатора и помещения его в атрибут СУЩНОСТИ его следует сделать неизменяемым. Он никогда не должен изменяться, даже если среда программирования не может напрямую принудить программу к выполнению этого требования. Например, такой идентификатор сохраняется, когда объект упаковывают в базу данных, а затем извлекают оттуда. Иногда технические средства программирования могут помочь в этом, но в других случаях приходится полагаться на дисциплинированность разработчика.

Идентификаторы часто генерируются системой автоматически. Алгоритм генерирования должен гарантировать уникальность в пределах системы, что для параллельных вычислений и распределенных систем может оказаться непростой задачей. Генерирование таких идентификаторов может потребовать приемов и технологий, далеко выходящих за пределы тематики нашей книги. Здесь хотелось бы только указать, когда именно возникает необходимость рассмотрения таких вопросов, чтобы разработчики были в курсе существования проблемы и могли сузить поиск решения до самой критической области. Главное здесь — осознать, что вопросы идентификации связаны со специфическими аспектами самой модели. Создание средств идентификации часто требует тщательного изучения предметной области.

При автоматическом генерировании идентификатора у пользователя может даже не возникнуть потребность его видеть. Идентификатор может понадобиться только для внутренних целей, — например, в программе управления контактами, позволяющей пользователю найти запись по имени человека. Программе нужно уметь различать два контакта с одним и тем же именем простым и однозначным способом. Именно это позволяют делать уникальные внутренние идентификаторы. Извлекая из базы два различных элемента, система показывает пользователю два разных контакта, а вот идентификаторы может и не показать. Пользователь сам различит их по названию фирмы, адресу и т.п.

Наконец, бывают и случаи, когда сгенерированный идентификатор все-таки представляет интерес для пользователя. Если я отправляю посылку через почтовую службу доставки, мне выдают контрольный номер, сгенерированный программным обеспечением фирмы-доставителя, и с этим номером я могу сопровождать и находить свою посылку. Если я заказываю авиабилеты или номер в гостинице, мне выдают числовые коды подтверждения, которые и являются уникальными идентификаторами в соответствующей транзакции.

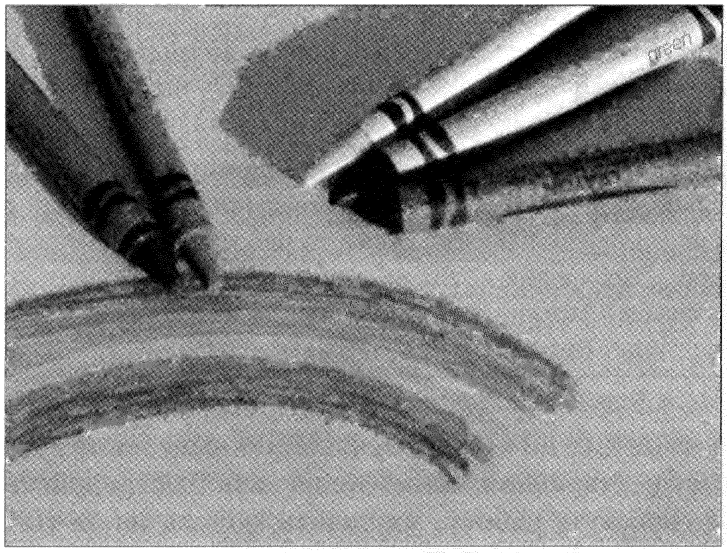
В некоторых случаях уникальность идентификатора не должна ограничиваться пределами компьютерной системы. Например, если между двумя больницами, имеющими отдельные компьютерные системы управления, идет обмен медицинскими картами, то в идеале каждая система должна была бы обозначать пациента одним и тем же идентификационным кодом. Но это сложно сделать, если они сами генерируют свои идентификаторы. В таких системах часто используются идентификаторы, выпущенные какой-нибудь другой организацией — как правило, государственной. В США в качестве идентификаторов для пациентов больниц часто применяются номера социального страхования (*Social Security*). Впрочем, такие методы не дают полной гарантии. Например, не у всех есть номера социального страхования (в частности, их нет у детей и у иностранных граждан), и многие граждане протестуют против использования этих номеров по соображениям сохранения личной тайны.

В менее формальных ситуациях (например, в видеопрокате) идентификаторами могут служить номера телефонов. Но один телефон может использоваться несколькими людьми. К тому же номер может измениться или его могут даже передать другому абоненту.

В силу всех этих причин часто используются специальные идентификаторы (например, номера отрывных квитанций), а другие атрибуты, такие как номера телефонов и социального страхования, служат для сопоставления и проверки. В любом случае, если система запрашивает внешний идентификатор, за его ввод и уникальность отвечают пользователи системы, а сама система должна предоставить технические средства для обработки возможных исключений.

Встречаясь с таким обилием технических проблем, легко потерять из виду концептуальные задачи, лежащие на более глубоком уровне. Что означает по сути "одинаковость" двух объектов? Легко пометить каждый объект идентификатором или написать операцию сравнения двух экземпляров, но если эти идентификаторы или операции не соответствуют никакому смысловому аспекту в модели, они только приведут к путанице. Вот почему операции назначения индивидуальных идентификационных данных часто требуют обязательного ввода пользователем каких-либо данных. Например, программы учета личных финансов могут найти и предложить приблизительные соответствия между платежами, но окончательное решение они ожидают от пользователя.

Объекты-значения



У многих объектов в принципе нет индивидуального существования. Они описывают характеристики того или иного предмета.

* * *

Когда ребенок рисует, его интересует цвет карандаша или фломастера, а также, возможно, острота его грифеля. Но если есть два карандаша одинакового цвета и формы, ему, вероятно, будет все равно, каким рисовать. Если карандаш потерялся и его пришлось заменить таким же из новой коробки, малыш может продолжить работу, нисколько не отвлекаясь на сам факт замены.

Спросите ребенка о рисунках на холодильнике, и он быстро отличит, какие рисовал он сам, а какие его сестра. Как он, так и сестра обладают индивидуальным существованием. Это же свойство характерно и для их рисунков. Но представьте себе, сколько бы потребовалось труда, если бы пришлось отслеживать, какие линии и каким карандашом сделаны. Рисование сразу перестало бы быть детской игрой.

Самые важные объекты в модели обычно представляют собой СУЩНОСТИ. Каждую такую сущность очень важно правильно идентифицировать и отслеживать. Естественным образом возникает желание сделать все объекты области индивидуальными. Действительно, в некоторых средах программирования каждому объекту присваивается уникальный идентификатор.

Системе приходится справляться с отслеживанием всех этих индивидуальных объектов, и это часто мешает применить различные способы оптимизации быстродействия. Требуется немалая аналитическая работа, чтобы грамотно определить способы идентификации и выработать надежные методы отслеживания объектов в распределенных системах или при хранении в базах данных. Искусственно же сконструированная идентификация только вводит в заблуждение, портит модель, перемешивает все объекты в одну хаотическую "кашу".

Является ли адрес объектом-значением?

Смотря для кого. В программе для фирмы, торгующей по почте, адрес необходим для подтверждения кредитной карточки и отправки посылки. Но если там же заказывает товар сосед по квартире, фирме совершенно безразлично, что он находится по тому же адресу. Здесь адрес — это ОБЪЕКТ-ЗНАЧЕНИЕ.

В программе обслуживания почтовых отправлений, предназначенной для организации маршрутов доставки почты, страна может быть представлена в виде иерархии областей, городов, индексных зон и жилищных массивов, которая заканчивается отдельными адресами. Такие адресные объекты наследуют свой почтовый индекс от родительского объекта в иерархии, и если почтовое управление решит изменить индексные зоны, все адреса в их пределах "подстроятся" под это изменение. Здесь адрес является СУЩНОСТЬЮ.

В программе для коммунальной электрической компании адрес соответствует месту, к которому подведены кабельные линии и где оказываются услуги. Если два соседа по квартире позвонят и по отдельности закажут какие-то работы, фирме нужно знать, что они находятся в одном месте. Здесь адрес — также СУЩНОСТЬ. Но в модели коммунальные услуги могут ассоциироваться с "жилищем", т.е. объектом-сущностью, у которого есть атрибут "адрес". Тогда "адрес" будет ОБЪЕКТОМ-ЗНАЧЕНИЕМ.

Идентификация СУЩНОСТЕЙ **— дело нужное, но выделение других видов объектов как индивидуально существующих может повредить быстродействию системы, добавить лишнюю аналитическую работу, ухудшить модель из-за того, что все объекты в ней приобретут одинаковый характер.**

Проектирование программного обеспечения — это постоянная битва за простоту. Нужно различать, когда что использовать, чтобы специальные приемы работы с объектами использовались только тогда, когда нужно.

Однако если думать об этой категории объектов только как о безличности, отсутствии индивидуальности, то мы не добавим ничего нового в свой рабочий словарь или арсенал. На самом деле у таких объектов есть собственные характеристики, и они также значимы для модели. *Это объекты, которые используются для описаний предметов и явлений*.

ОБЪЕКТОМ-ЗНАЧЕНИЕМ (VALUE OBJECT) называется объект, который представляет описательный аспект предметной области и не имеет индивидуального существования, собственной идентичности. Такие объекты создаются в программе для представления тех элементов проекта, о которых достаточно знать только, *что* они собой представляют, но не *кем именно* они являются.

В базовых библиотеках многих современных систем разработки программ представлены такие ОБЪЕКТЫ-ЗНАЧЕНИЯ, как цвета, а также строки и числа. (Кому нужно различать отдельные четверки или буквы Q?) Это совсем простые примеры, но реальные объекты-показатели не обязательно бывают такими простыми. Например, программа анализа и генерирования цветов может иметь в своей основе сложную модель, в которой объекты-цвета можно смешивать для получения новых цветов. Для получения нового объекта-значения таким образом могут применяться сложные алгоритмы.

ОБЪЕКТ-ЗНАЧЕНИЕ может представлять собой совокупность других объектов. В программах архитектурного проектирования жилых домов для любого стиля окон можно создать отдельный объект. Этот "стиль окна" можно инкорпорировать в объект "окно" наряду с атрибутами ширины и высоты, а также правилами изменения и комбинирования. Такие окна — сложные ОБЪЕКТЫ-ЗНАЧЕНИЯ, составленные из других ОБЪЕКТОВ-ЗНАЧЕНИЙ. Они, в свою очередь, могут инкорпорироваться в еще большие составные элементы плана здания — например, объекты-"стены".

ОБЪЕКТЫ-ЗНАЧЕНИЯ могут даже ссылаться на СУЩНОСТИ. Например, если я запрошу картографическую онлайн-службу проложить мне живописный маршрут из Сан-Франциско в Лос-Анджелес, то она может сгенерировать объект **Маршрут (Route)**, соединяющий эти два города по шоссе Pacific Coast Highway. Этот объект будет ЗНАЧЕНИЕМ, пусть даже все три объекта, на которые он ссылается (два города и шоссе), являются СУЩНОСТЯМИ.

ОБЪЕКТЫ-ЗНАЧЕНИЯ часто передаются в качестве параметров в сообщениях между объектами. Нередко они носят временный характер — создаются для конкретной операции и тут же уничтожаются. ОБЪЕКТЫ-ЗНАЧЕНИЯ могут использоваться в виде атрибутов СУЩНОСТЕЙ (и других ЗНАЧЕНИЙ). Так, человек, в целом, может представляться индивидуальной СУЩНОСТЬЮ, но его имя будет ОБЪЕКТОМ-ЗНАЧЕНИЕМ.

Если элемент модели полностью определяется своими атрибутами, то его следует считать ОБЪЕКТОМ-ЗНАЧЕНИЕМ**. Сделайте так, чтобы он отражал смысл заложенных в него атрибутов и придайте ему соответствующую функциональность. Считайте такой объект неизменяющимся. Не давайте ему индивидуальности, вообще избегайте любых сложностей, неизбежных при программном управлении** СУЩНОСТЯМИ**.**

Атрибуты, образующие в совокупности ОБЪЕКТ-ЗНАЧЕНИЕ, должны быть единым концептуальным целым[5]. Например, улица, город и почтовый индекс не должны быть отдельными атрибутами объекта **Человек (Person)**. Они входят как составные части в адрес, что делает проще устройство объекта **Человек** и больше соответствует концепции ОБЪЕКТА-ЗНАЧЕНИЯ.

* * *

Проектирование ОБЪЕКТОВ-ЗНАЧЕНИЙ

Нам все равно, какой именно экземпляр ОБЪЕКТА-ЗНАЧЕНИЯ (VALUE OBJECT) у нас имеется. Такая вольность дает проектировщику достаточно места для маневров по упрощению архитектуры и оптимизации быстродействия. При этом приходится принимать решения по части копирования, совместного использования и неизменяемости.

Если двое людей имеют одно и то же имя, они от этого не становятся одним и тем же человеком; не становятся они и взаимозаменяемыми. Но вот объект, представляющий имя, — вполне заменяем, лишь бы имя было написано правильно. Объект **Имя (Name)** можно смело *копировать* из первого объекта **Человек (Person)** во второй.

---

[5] Шаблон WHOLE VALUE (*целостный объект-значение*), автор Уорд Каннингем (Ward Cunningham).

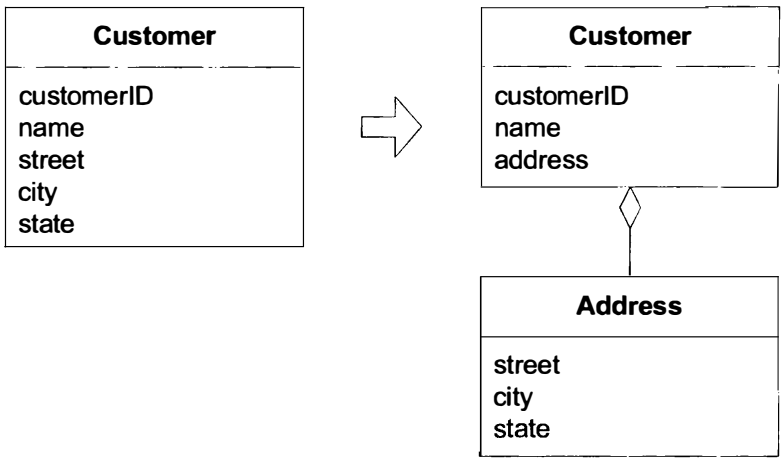


Рис. 5.6. ОБЪЕКТ-ЗНАЧЕНИЕ предоставляет информацию об объекте-СУЩНОСТИ. Он должен быть единым концептуальным целым

В принципе два объекта **Человек** могут и не нуждаться в собственных экземплярах объекта-имени. Один и тот же объект **Имя** может совместно использоваться двумя объектами **Человек** (в каждом будет содержаться указатель на один и тот же экземпляр **Имени**), и при этом никаких изменений в их поведении или способе идентификации не потребуется. То есть, они будут работать правильно, пока у одного человека не изменится имя. Тогда так же изменится имя и другого человека! Чтобы этого избежать и сделать совместное использование объекта безопасным, его нужно сделать *неизменяемым* запрещенным для любых изменений иначе как путем полной замены.

Та же проблема возникает, когда объект передает один из своих атрибутов другому объекту в качестве аргумента или возвращаемого значения. Со странствующим объектом может случиться все, что угодно, за то время, пока он не находится под контролем владельца. ЗНАЧЕНИЕ (VALUE) может измениться таким образом, что будет поврежден и объект-владелец путем искажения его инвариантов. Чтобы избежать этого, передаваемый объект делают неизменяемым или же передают его копию.

Бывает полезно создать несколько потенциальных возможностей для оптимизации быстродействия, так как ОБЪЕКТЫ-ЗНАЧЕНИЯ имеют тенденцию множиться. Примером может служить программа для проектирования жилого дома. Если каждый электрический вывод является отдельным ОБЪЕКТОМ-ЗНАЧЕНИЕМ, то всего в одной версии одного проекта дома их может быть более сотни. Но если все выводы посчитать взаимозаменяемыми, то можно совместно использовать всего один объект, указывая на него сотню раз (это пример шаблона FLYWEIGHT, *мелкий объект* [14]). В больших системах объектам подобного рода идет счет на тысячи, а оптимизация сразу показывает разницу между нормальной работоспособной системой и еле ползающим бесполезным монстром, задыхающимся от миллионов лишних объектов. И это всего лишь один пример такой оптимизации, которая недоступна объектам-СУЩНОСТЯМ.

Сравнительная экономичность копирования и совместного использования зависит от среды реализации. Копирование может "затопить" систему огромным количеством объектов, зато совместное использование способно замедлить работу распределенной сетевой системы. Если между двумя рабочими станциями передается копия объекта, то посылается одно-единственное сообщение, и копия затем существует независимо от оригинала. Но если совместно используется один и тот же экземпляр объекта, то передается только ссылка на него, отчего при каждом обращении необходимо пересылать сообщение объекту.

Совместное использование объектов стоит ограничить теми случаями, когда оно приносит наибольшую пользу и доставляет меньше всего неприятностей:

- если занимаемый объем памяти или количество объектов в базе данных являются критическими параметрами;
- если дополнительные затраты на пересылку по сети невелики (например, на централизованном сервере);
- если совместно используемый объект категорически неизменяем.

Особые случаи: когда разрешать изменяемость

Неизменяемость объектов сильно упрощает программную реализацию, гарантируя безопасность совместного использования и передачи ссылок. Она также следует в русле идеи объектов как *значений*. Если изменяется значение некоторого атрибута, то вместо модификации существующего ОБЪЕКТА-ЗНАЧЕНИЯ просто используется новый. Но бывают случаи, когда из соображений быстродействия лучше разрешить вносить изменения в ОБЪЕКТЫ-ЗНАЧЕНИЯ. В пользу этого обычно говорят следующие факторы:

- частые изменения значения объекта (т.е. собственно ОБЪЕКТА-ЗНАЧЕНИЯ);
- затратность создания и уничтожения объекта;
- опасность замены (вместо модификации) при группировании объектов, как было показано в предыдущем примере;
- незначительное совместное использование или же полный отказ от него с целью улучшения группирования объектов или из каких-то других технических соображений.

Нелишне будет еще раз напомнить: если реализация ОБЪЕКТА-ЗНАЧЕНИЯ предполагается изменяемой, такой объект *не должен* совместно использоваться с другим объектом. Но независимо от того, будут они совместно использоваться или нет, при любой возможности не изменяйте ОБЪЕКТЫ-ЗНАЧЕНИЯ.

Атрибут или объект в некоторых языках и средах программирования можно объявить неизменяемым, а в других — нет. Если такая возможность есть, это помогает выразить принятое проектное решение в явной форме, но если ее нет — ничего страшного. Многие особенности, присутствующие в модели, невозможно определить явно в программной реализации с помощью существующих средств и языков программирования. Например, нельзя так объявить объект-СУЩНОСТЬ, чтобы ему при этом автоматически назначалась операция проверки идентичности. Но если для концептуальной особенности модели не существует прямой поддержки в языке, это еще не значит, что сама особенность бесполезна. Если существуют только неявные способы для ее реализации, это всего лишь потребует от разработчика большей "стилистической дисциплины". Здесь должны пойти в ход соглашения об именах, тщательность подготовки документации и *постоянное обсуждение проблемы*.

Если ОБЪЕКТ-ЗНАЧЕНИЕ объявлен неизменяемым, то управлять его изменениями очень просто. Единственный способ изменения — полная замена. Неизменяемые объекты можно предоставлять в совместный доступ, как в примере с электрическим выводом. Если "сборка мусора" (*garbage collection*) работает надежно, то удаление — всего лишь вопрос сброса всех ссылок на объект. Если в архитектуре программы ОБЪЕКТ-ЗНАЧЕНИЕ сделан неизменяемым, то разработчики вольны принимать любые решения по таким вопросам, как копирование и совместный доступ, из чисто технических соображений — они твердо знают, что работа программы не зависит критически от конкретного экземпляра такого объекта.

Определение ОБЪЕКТОВ-ЗНАЧЕНИЙ и объявление их неизменяемыми — это частный случай общего правила, которое состоит в следующем. Чтобы иметь возможность технически оптимизировать быстродействие, разработчики должны избегать лишних связей-ограничений в модели. А если явно определить все существенные ограничения, это дает возможность оптимизировать архитектуру, не боясь нарушить работу важнейших алгоритмических частей программы. Такая оптимизация архитектуры часто зависит от конкретной технологии, используемой в проекте.

Пример

Оптимизация базы данных с помощью объектов-значений

На самом нижнем уровне данные в базах физически хранятся в определенных местах на жестком диске, и для чтения данных приходится механически перемещать детали дискового механизма. Высококлассные системы управления базами данных пытаются так группировать физические адреса данных, чтобы связанные по смыслу данные можно было перенести с диска за одну физическую операцию считывания.

Если на объект ссылается много других объектов, то не все из них обязательно располагаются близко от него (на той же странице), и поэтому для получения данных требуется дополнительная физическая операция. Но если экземпляр ОБЪЕКТА-ЗНАЧЕНИЯ, который служит атрибутом нескольких СУЩНОСТЕЙ, физически скопировать, а не просто поставить ссылку на него, то такой объект можно будет разместить на той же странице, что и любую из использующих его СУЩНОСТЕЙ. Этот прием размножения копий одних и тех же данных называется *денормализацией* и часто используется тогда, когда время обращения к данным важнее, чем экономия дискового пространства или удобство обслуживания базы.

В реляционной базе данных можно поместить ОБЪЕКТ-ЗНАЧЕНИЕ в таблицу объекта-СУЩНОСТИ, который им владеет, а не создавать ассоциативную связь с отдельной таблицей. Если в распределенной системе хранить ссылку на ОБЪЕКТ-ЗНАЧЕНИЕ на другом сервере, от этого наверняка замедлится передача сообщений. Поэтому лучше передать на другой сервер полную копию объекта. Таких копий можно сделать сколько угодно, поскольку мы имеем дело с ОБЪЕКТАМИ-ЗНАЧЕНИЯМИ.

Проектирование ассоциаций с помощью ОБЪЕКТОВ-ЗНАЧЕНИЙ

Многое, что было раньше сказано об ассоциациях, относится как к СУЩНОСТЯМ (ENTITIES), так и к ОБЪЕКТАМ-ЗНАЧЕНИЯМ (VALUE OBJECTS). Чем меньше в модели ассоциаций и чем они проще, тем лучше.

В то время как двунаправленные ассоциации между СУЩНОСТЯМИ бывает трудно поддерживать, двунаправленные ассоциации между ОБЪЕКТАМИ-ЗНАЧЕНИЯМИ вообще не имеют смысла. При отсутствии индивидуальности у объектов бессмысленно говорить, что тот или иной объект ссылается на тот же ОБЪЕКТ-ЗНАЧЕНИЕ, который ссылается на него. Можно разве что сказать так: первый объект ссылается на другой, *эквивалентный* тому, который сам ссылается на первый. Но и этот инвариант надо как-то принудительно задать в другом месте. Хотя можно было бы теоретически сделать указатели в обе стороны, трудно придумать примеры, в которых этот прием был бы уместен. Поэтому старайтесь избегать двунаправленных ассоциаций между ОБЪЕКТАМИ-ЗНАЧЕНИЯМИ. Если же такие ассоциации кажутся необходимыми в модели, пересмотрите само решение сделать эти объекты значениями. Может быть, у них есть своеобразная индивидуальность, которую вы до сих пор не заметили.

СУЩНОСТИ (ENTITIES) и ОБЪЕКТЫ-ЗНАЧЕНИЯ (VALUE OBJECTS) — это главные элементы самых распространенных объектных моделей, но проектировщики программ на своем опыте пришли к выводу о необходимости еще и третьего вида элементов — СЛУЖБ (SERVICES).

Службы

Не все на свете сводится к вещам и предметам.

В некоторых случаях самые четкие и практичные программные архитектуры содержат операции, которые по своей сути не принадлежат ни одному конкретному объекту. Чем выкручиваться из этого положения искусственными методами, можно последовать естественному ходу событий и включить в модель так называемые СЛУЖБЫ (SERVICES).

* * *

В прикладной предметной области бывают такие операции, которым нельзя найти естественное место в объекте типа СУЩНОСТИ (ENTITY) или ЗНАЧЕНИЯ (VALUE OBJECT). Они по своей сути являются не предметами, а видами деятельности. Но поскольку в основе нашей парадигмы моделирования лежит объектный подход, мы попробуем превратить их в объекты.

В этом месте легко совершить распространенную ошибку: отказаться от попытки поместить операцию в подходящий для нее объект, и таким образом прийти к процедурному программированию. Но если насильно поместить операцию в объект с чуждым ей определением, от этого сам объект утратит чистоту замысла, станет труднее для понимания и рефакторинга. Если в простом объекте реализовать много сложных операций, он может превратиться в непонятно что, занятое непонятно чем. В таких операциях часто участвуют другие объекты предметной области, и между ними выполняется согласование для выполнения совместной задачи. Дополнительная ответственность создает цепочки зависимости между объектами, смешивая понятия, которые можно было бы рассматривать независимо.

Часто службы выступают под личиной объектов модели, не наполненных никаким особым смыслом, кроме выполнения определенной операции. Эти "исполнители" в конце концов получают имена наподобие "диспетчер (*manager*) чего-нибудь". У них нет ни собственного характерного состояния, ни другой роли в предметной области, кроме реализации конкретной операции. Но, по крайней мере, при таком решении характерные действия, выполняемые в модели, куда-то помещены и не загромождают настоящие, смысловые объекты модели.

Некоторые понятия модели неестественны в роли объектов. Если принудительно реализовать нужные функции модели в объекте-сущности или объекте-значении, это либо исказит определение объекта из модели, либо добавит в модель лишних, искусственно сконструированных объектов.

СЛУЖБА (SERVICE) — это операция, предлагаемая в модели в виде обособленного интерфейса, который, в отличие от СУЩНОСТЕЙ и ОБЪЕКТОВ-ЗНАЧЕНИЙ, не инкапсулирует никакого состояния. СЛУЖБА — это обычный шаблон в технических средах программирования, но службы применимы и на уровне предметной области.

Само название *служба* подчеркивает положение, подчиненное нуждам других объектов. В отличие от СУЩНОСТЕЙ и ОБЪЕКТОВ-ЗНАЧЕНИЙ, она определяется только тем, что умеет делать для клиента. Соответственно, и имя СЛУЖБЕ обычно дают по ее функции, а не сути — это глагол, а не существительное. Определение у нее, тем не менее, может быть достаточно абстрактным, но с другим оттенком по сравнению с объектом. На СЛУЖБУ должны возлагаться конкретные обязанности, и они вместе с интерфейсом должны определяться в составе модели. Имена операций следует взять из ЕДИНОГО ЯЗЫКА (UBIQUITOUS LANGUAGE) или ввести в него. Параметрами и результатами работы СЛУЖБЫ должны быть объекты модели.

Не стоит, однако, и злоупотреблять СЛУЖБАМИ, лишая СУЩНОСТИ и ОБЪЕКТЫ-ЗНАЧЕНИЯ всякой функциональной нагрузки. В то время как операция — это важное понятие предметной области, СЛУЖБА является естественным элементом в рамках ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ. Если операция объявлена в модели СЛУЖБОЙ, а не "липовым" объектом, ничего реального не представляющим, то она никого не введет в заблуждение.

Хорошая СЛУЖБА обладает тремя свойствами.

1. Выполняемая ею операция соответствует понятию модели, не являющемуся естественной частью объекта-сущности или объекта-значения.
2. Интерфейс службы определен через другие элементы модели предметной области.
3. Операция не имеет собственного состояния.

Последний пункт — отсутствие состояния — означает, что любой клиент может воспользоваться любым экземпляром нужной ему СЛУЖБЫ, не обращая внимания на индивидуальную предысторию ее работы. При работе СЛУЖБЫ может использоваться (и даже подвергаться изменениям) информация, доступная глобально, поэтому возможны побочные эффекты. Но СЛУЖБА не хранит данных о своем собственном состоянии, которое влияло бы на ее работу, в отличие от большинства объектов модели.

Если существенно важный процесс или преобразование в модели не относится к естественным обязанностям объекта-СУЩНОСТИ **или** ЗНАЧЕНИЯ**, добавьте в модель эту операцию с отдельным интерфейсом и назовите ее** СЛУЖБОЙ**. Определите интерфейс на языке модели и сделайте имя операции элементом** ЕДИНОГО ЯЗЫКА**. У** СЛУЖБЫ **не должно быть собственного состояния.**

* * *

Службы и изоляция уровня предметной области

В основе этого шаблона лежат такие СЛУЖБЫ, которые сами по себе много значат для предметной области. Но, конечно, их применение не ограничивается одним этим уровнем. Не так уж легко отличить СЛУЖБЫ, принадлежащие к уровню предметной области, от СЛУЖБ других уровней, и соответственно разделить обязанности, чтобы четко обозначить это различие.

Большинство описанных в литературе СЛУЖБ имеет чисто технический характер и принадлежит инфраструктурному уровню. СЛУЖБЫ уровней модели и прикладных операций взаимодействуют с инфраструктурными СЛУЖБАМИ. Например, в банке может работать программа, которая отсылает клиенту письмо по электронной почте всякий раз, когда баланс на его счету падает ниже определенного порога. Интерфейс, инкапсулирующий почтовую систему, а с нею, возможно, и другие способы извещения клиента, образует СЛУЖБУ на инфраструктурном уровне.

А вот отличить СЛУЖБЫ операционного уровня от уровня предметной области (модели) бывает труднее. Операционный уровень отвечает за то, чтобы отдать приказ об извещении клиента. А уровень модели определяет, достигнут ли порог — правда, эта задача не требует отдельной СЛУЖБЫ, а скорее, входит в полномочия объекта "банковский счет". Та же самая банковская программа, возможно, занимается и переводом денежных средств. Если сконструировать СЛУЖБУ, отвечающую за подведение дебета и кредита в процессе перевода, то она будет принадлежать уровню предметной области. Перевод денег — это смысловой элемент банковского дела, а его реализация — это фундаментальная операция предметной области. В технических же службах никакие понятия прикладной модели вообще не должны упоминаться.

Многие СЛУЖБЫ уровней модели и прикладных операций строятся на основе совокупностей ОБЪЕКТОВ-СУЩНОСТЕЙ и ОБЪЕКТОВ-ЗНАЧЕНИЙ. Они ведут себя как сценарии, которые мобилизуют потенциал предметной области на выполнение чего-то полезного. Сами СУЩНОСТИ и ЗНАЧЕНИЯ часто бывают слишком мелкомасштабными, чтобы дать пользователю удобный доступ к возможностям уровня предметной области. Можно заметить, что грань между уровнями предметной области и прикладных операций очень тонка. Например, если банковская программа умеет преобразовывать и экспортировать все наши финансовые транзакции в файл электронной таблицы (чтобы мы могли его читать и анализировать), то СЛУЖБА такого экспорта относится к уровню прикладных операций. В предметной области банковского дела никаких "форматов файлов" нет, да и логика прикладной модели здесь тоже не используется.

С другой стороны, функция перевода денег с одного счета на другой — это СЛУЖБА предметной области, потому что в ней реализуются ее алгоритмы, деловые регламенты (например, ведение дебета и кредита по счетам), а также потому, что "перевод денежных средств" — это термин банковского дела. В этом случае сама СЛУЖБА не делает ничего особенного, а просит два объекта "банковский счет" выполнить большую часть работы. Но поместить операцию "перевод" в объект "счет" было бы неверно, поскольку в операции участвуют два счета и ряд глобальных регламентов.

Теоретически можно было бы создать объект **Перевод (Funds Transfer)**, представляющий два банковских счета плюс регламентные правила и история перевода. Но в межбанковских сетях все равно никуда не деться от вызова СЛУЖБ. Более того, в большинстве сред разработки создавать прямой интерфейс между объектом предметной области и внешними ресурсами — это нарушение стиля программирования. Внешние СЛУЖБЫ можно "задрапировать" ФАСАДНЫМ МЕТОДОМ (FACADE), на вход которого поступают объекты модели, а возвращается в качестве результата, например объект **Перевод**. Но какие бы у нас ни были посредники (пусть даже не из нашей системы), эти СЛУЖБЫ все равно выполняют обязанности из предметной области по переводу денежных средств.

Распределение служб по уровням

Операционный	*Операционная служба перевода средств:* • принимает входные данные (например, XML-запрос); • посылает сообщение в службу модели для выполнения; • ожидает подтверждения; • принимает решение об отправке извещения через службу инфраструктурного уровня
Предметной области	*Служба перевода средств в предметной модели:* • взаимодействует с нужными объектами **`Счет (Account)`** и **`Книга (Ledger)`**, выполняя операции дебита и кредита; • посылает подтверждение результата (разрешен перевод или нет и т.п.)
Инфраструктурный	*Служба рассылки извещений:* • рассылает бумажные и электронные письма, осуществляет связь другого рода по указаниям операционного уровня.

Степень модульности

Хотя этот раздел в основном посвящен тому, чтобы подчеркнуть, как выразительно и удобно некоторые понятия модели реализуются в виде СЛУЖБ, сам по себе этот шаблон ценен тем, что помогает управлять крупностью разбиения (степенью модульности) в интерфейсах уровня предметной области, а также изоляцией клиентов от СУЩНОСТЕЙ и ОБЪЕКТОВ-ЗНАЧЕНИЙ.

Средней крупности СЛУЖБЫ, не имеющие собственного состояния, легче переносить в большие системы, поскольку в них за простым интерфейсом инкапсулируются большие функциональные возможности. Если же объекты имеют слишком мелкий масштаб, в распределенной системе это может привести к неэффективному обмену сообщениями.

Как говорилось ранее, из-за слишком мелких объектов предметной области может произойти утечка знания на операционный уровень, где координируется работа объектов модели. Сложность взаимодействия, слишком загроможденного мелкими деталями, в конце концов, передается на операционный уровень. Знание из модели потихоньку "растекается" по коду прикладных операций или пользовательского интерфейса, уходя с положенного ему уровня. Разумное же применение служб уровня предметной области позволяет удерживать четкую границу между уровнями.

В этом архитектурном шаблоне простоте интерфейса отдается предпочтение перед разнообразием функций и управлением со стороны клиента. Разбиение (модульность) функциональных возможностей поддерживается на среднем уровне, как раз удобном для инкапсулирования компонентов в больших или распределенных системах. К тому же, иногда СЛУЖБА — это самый естественный способ выразить понятие из модели предметной области.

Доступ к службам

В распределенных системных архитектурах, таких как J2EE и CORBA, имеется специальный механизм объявления СЛУЖБ вместе с соглашениями по их использованию, а также функциями распределения и доступа. Но такие среды не всегда используются в программных проектах, а даже если и используются, то в таком простом деле, как логическое разделение обязанностей, это будет стрельба из пушки по воробьям.

Не так важны конкретные средства для обеспечения доступа к службе, как базовое проектное решение разделить те или иные обязанности. Для реализации интерфейса какой-нибудь СЛУЖБЫ вполне может подойти и объект-"агент". Например, для обеспечения доступа можно легко написать простой объект типа SINGLETON [14]. С помощью стилевых соглашений легко выделить эти объекты так, чтобы стало ясно — это просто механизмы для создания интерфейса СЛУЖБ, а не важные объекты модели. Сложные архитектуры следует применять только тогда, когда есть реальная потребность в сетевом распределении системы или другом использовании мощных средств архитектурной среды.

Модули (пакеты)

МОДУЛЬ (MODULE) — это устоявшийся элемент архитектуры программ. Кроме технических соображений, при разбиении на модули в основном руководствуются стремлением уменьшить смысловую сложность той или иной части программы. МОДУЛИ дают возможность посмотреть на модель с разных сторон: во-первых, можно изучить подробности устройства модуля, не вникая в сложное целое; во-вторых, удобно рассматривать взаимоотношения между модулями, не вдаваясь в детали их внутреннего устройства.

На уровне предметной области МОДУЛИ должны соответствовать смысловым частям модели, выражая суть и структуру модели в крупном масштабе.

* * *

МОДУЛИ **используют все, но мало кто воспринимает их как полноценные структурные составляющие модели. Код подразделяют по разным критериям на всевозможные виды структурных элементов: от отдельных аспектов технической архитектуры до рабочих заданий конкретных разработчиков. Но даже те разработчики, которые много занимаются рефакторингом, склонны не нарушать границ** МОДУЛЕЙ**, установленных на ранних этапах проекта.**

То, что при делении на модули должна соблюдаться низкая внешняя зависимость (*low coupling*) при высокой внутренней связности (*high cohesion*) — это общие слова. Определения зависимости и связности грешат уклоном в чисто технические, количественные критерии, по которым их якобы можно измерить, подсчитав количество ассоциаций и взаимодействий. Но это не просто механические характеристики подразделения кода на модули, а идейные концепции. Человек не может одновременно удерживать в уме слишком много предметов (отсюда низкая внешняя зависимость). А плохо связанные между собой фрагменты информации так же трудно понять, как неструктурированную "кашу" из идей (отсюда высокая внутренняя связность).

Низкая внешняя зависимость и высокая внутренняя связность — это общие принципы программной архитектуры, применяемые как к отдельным объектам, так и к МОДУЛЯМ. Но особенную важность они приобретают в крупном масштабе моделирования архитектуры. Эти термины существуют уже долгое время; изложение на эту тему в стиле описания архитектурных шаблонов можно найти в [17].

Когда два элемента модели разводятся по разным модулям, взаимосвязь между ними становится не такой прямой, как раньше. Из-за этого бывает труднее понять, какое место они занимают в архитектуре программы. Если минимизировать зависимость между МОДУЛЯМИ, задача облегчается, и становится возможным анализировать содержимое одного МОДУЛЯ, почти не обращаясь к тем другим, с которыми он взаимодействует.

В то же время элементы хорошей модели склонны к синергизму, и если подразделение на МОДУЛИ выбрано удачно, то взаимоотношения между элементами модели выходят на новый концептуальный уровень. Высокая связность между объектами родственн-

ного назначения позволяет сосредоточить работу по моделированию и архитектурному проектированию в пределах одного МОДУЛЯ, т.е. на таком уровне сложности, который человеческому уму вполне по плечу.

МОДУЛИ и меньшие структурные элементы должны эволюционировать вместе, но на практике так обычно не получается. МОДУЛИ выбираются на ранней стадии работы, чтобы придать организованность первоначальным версиям объектов. После этого объекты имеют тенденцию развиваться так, чтобы не нарушать границ уже установленного определения МОДУЛЯ. Рефакторинг модулей требует больших затрат труда и более опасен, чем рефакторинг классов, да и делать его так же часто, скорее всего, не удастся. Но как объекты модели вначале выглядят наивно и слишком конкретно, а со временем преображаются и начинают выражать более глубокое понимание предмета, так и МОДУЛИ могут выйти на более высокий уровень точности и абстракции. Если дать МОДУЛЯМ возможность отражать изменяющееся понимание предмета, то и у объектов внутри них появится возможность более гибкой эволюции.

Как и все остальное в искусстве DDD, МОДУЛИ являются *механизмом коммуникации*. Выбор разбиения на МОДУЛИ должен основываться на *смысле* объектов, подвергающихся разделению. Если вы помещаете несколько классов в один МОДУЛЬ, вы тем самым говорите вашему преемнику, которому предстоит изучать эту архитектуру: думай о них вместе, как о целом. Если ваша модель рассказывает историю, то МОДУЛИ — это ее главы. Имя МОДУЛЯ должно передавать его содержание и входить в ЕДИНЫЙ ЯЗЫК модели. Можно сказать специалисту в предметной области: "а сейчас давайте поговорим о модуле **`Клиент`**", и для разговора сразу же создается контекст.

Выберите такие МОДУЛИ**, которые бы рассказывали историю системы и содержали связные наборы понятий. От этого часто сама собой возникает низкая зависимость** МОДУЛЕЙ **друг от друга. Но если это не так, найдите способ изменить модель таким образом, чтобы отделить понятия друг от друга, или же поищите пропущенное в модели понятие, которое могло бы стать основой для** МОДУЛЯ **и тем самым свести элементы модели вместе естественным, осмысленным способом. Добивайтесь низкой зависимости модулей друг от друга в том смысле, чтобы понятия в разных модулях можно было анализировать и воспринимать независимо друг от друга. Дорабатывайте модель до тех пор, пока в ней не возникнут естественные границы в соответствии с высокоуровневыми концепциями предметной области, а соответствующий код не разделится соответствующим образом.**

Дайте модулям такие имена, которые войдут в ЕДИНЫЙ ЯЗЫК**. Как сами** МОДУЛИ**, так и их имена должны отражать знание и понимание предметной области.**

Анализ концептуальных взаимосвязей не заменяет технической работы. Это разные аспекты одной и той же проблемы, и заниматься придется ими всеми. Но мышление на основе модели порождает более глубокое и качественное решение, чем сиюминутные мероприятия. Если возникает проблема выбора, лучше придерживаться концептуальной ясности, пусть даже от этого между МОДУЛЯМИ будет больше внешних ссылок или же при внесении изменений проявятся какие-нибудь мелкие побочные эффекты. Разработчики справятся с этими мелочами, если они понимают, что именно им рассказывает модель.

* * *

Гибкая модульность

МОДУЛЯМ необходимо эволюционировать вместе с остальными компонентами модели. Это означает, что над МОДУЛЯМИ, как и над моделью с кодом, тоже нужно выполнять рефакторинг. Но такой рефакторинг часто не делают вовсе. Чтобы изменить МОДУЛИ, как правило, нужно вносить в код широкомасштабные изменения. Такие изменения мо-

гут оказаться пагубными для коммуникации в группе разработчиков, и даже нарушить работу средств разработки, например, систем управления исходным кодом. В результате структура и имена МОДУЛЕЙ часто отражают гораздо более ранние "очертания" модели, чем это делают классы.

Неизбежные ошибки, сделанные на ранних этапах разработки, приводят к высокой взаимозависимости МОДУЛЕЙ, что затрудняет рефакторинг. А недостаточный рефакторинг — это способ оттягивать решение проблемы. Решить ее можно, только стиснув зубы и реорганизовав модули. При этом надо основываться на опыте, который подсказывает, где в коде находятся проблемные места.

Некоторые средства разработки и системы программирования только усугубляют проблему. Но какая бы технология разработки не применялась для реализации программы, необходимо найти способ минимизировать работу по рефакторингу МОДУЛЕЙ и устранить препятствия, мешающие коммуникации в группе.

Пример

Правила программирования пакетов в Java

В языке Java импорт, т.е. подключение других модулей, нужно объявлять в отдельных классах. Моделировщик может себе позволить думать о взаимосвязях между пакетами (*packages*), но объявить это непосредственно в Java нельзя. Согласно распространенным правилам, рекомендуется импортировать отдельные классы, так что получается примерно следующее.

```
ClassA1
import packageB.ClassB1;
import packageB.ClassB2;
import packageB.ClassB3;
import packageC.ClassC1;
import packageC.ClassC2;
import packageC.ClassC3;
. . .
```

К сожалению, в Java никуда не деться от импортирования *в* отдельные классы. Но можно хотя бы импортировать целые пакеты за раз, предполагая, что они являются высокосвязными модулями, и одновременно уменьшая потенциальный объем работы по переименованию пакетов.

```
ClassA1
import packageB.*;
import packageC.*;
. . .
```

Правда, в этом приеме смешиваются два разных масштаба модульности (классы импортируют пакеты), но и информации сообщается больше, чем в предыдущем объемистом списке классов — здесь декларируется намерение установить связь (создать зависимость) с некоторым МОДУЛЕМ.

Если же конкретный класс действительно зависит от отдельного класса из другого пакета, а его МОДУЛЬ не показывает ярко выраженной концептуальной зависимости от того, другого МОДУЛЯ, то, возможно, следует или переместить класс, или даже пересмотреть всю структуру этих МОДУЛЕЙ.

Ловушки инфраструктуры

Сильное влияние на проектные решения по организации пакетов оказывают технические среды программирования. Иногда это влияние благотворно, а иногда его нужно остерегаться.

Пример очень полезного стандарта, задаваемого средой — принудительное введение МНОГОУРОВНЕВОЙ АРХИТЕКТУРЫ путем помещения кода инфраструктуры и пользовательского интерфейса в разные группы пакетов. Вследствие этого уровень предметной области (модели) тоже физически выделяется в свой, отдельный набор пакетов.

С другой стороны, многоярусные архитектуры могут привести к слишком фрагментированной реализации объектов модели. В некоторых средах многоуровневость создается таким образом, что обязанности одного объекта модели распределяются между многими объектами программы, которые еще и помещаются в разные пакеты. Например, обычная практика в J2EE — поместить данные и средства доступа к ним в объект Java Bean (*entity bean*), а связанные с ними деловые регламенты и прикладные операции — в сеансовый объект Java Bean (*session bean*). Не считая дополнительных сложностей реализации каждого компонента, это разделение еще и лишает объектную модель связности и согласованности. Один из фундаментальных принципов объектно-ориентированного программирования состоит в том, что в объекте должны инкапсулироваться и данные, и операции над этими данными.

В этой многоуровневой реализации, казалось бы, еще нет ничего страшного, поскольку оба компонента можно рассматривать вместе как выражение одного элемента модели. Но как будто этого мало, сеансовый объект Java Bean часто еще и выносится в другой пакет. На этом этапе мысленно собирать разные объекты в одну концептуальную СУЩНОСТЬ (ENTITY) уже отнимает слишком много труда. Теряется связь между моделью и архитектурой программы. Хорошим стилем тут будет применение более крупномасштабных компонентов EJB, чем объекты-СУЩНОСТИ, чтобы уменьшить тем самым побочные эффекты разделения уровней. Но и мелкомасштабные объекты тоже часто разбиваются на уровни.

С такими проблемами я столкнулся, например, в одном довольно неплохо управляемом проекте, в котором каждый концептуальный объект разбивался на четыре уровня. И каждое из таких разбиений имело веское обоснование. Первый уровень управлял хранением и отображением данных, а также доступом к реляционной базе. Затем шел уровень, управлявший характерными, универсальными операциями объектов. Далее находился уровень, накладывавший на все это специфические для приложения операции. И наконец, четвертый уровень задумывался как общедоступный интерфейс, отделенный от всей лежащей ниже операционной части. Эта схема была, пожалуй, сложновата, однако уровни были хорошо определены, и обязанности разделены довольно аккуратно. Мысленно объединить все эти программные объекты в один концептуальный объект модели было не так уж сложно, а разделение обязанностей даже иногда помогало. В частности, многое упростило отделение уровня данных.

Но в дополнение ко всему этому архитектурная среда программирования требовала, чтобы каждый уровень находился в отдельном наборе пакетов с именами, соответствующими правилам идентификации для этого уровня. Теперь разделение уже не помещалось в голове. В результате разработчики предметной области старались завести поменьше МОДУЛЕЙ (учитывая, что их количество надо было умножать на четыре), и почти никогда не изменяли ни одного из них, поскольку рефакторинг любого МОДУЛЯ требовал слишком много усилий. Что еще хуже, отследить все данные и операции, относящиеся к одному концептуальному объекту, стало так трудно (учитывая еще и непрямой характер деления на уровни), что разработчикам некогда было думать

о каких-то там моделях. Программа все-таки была написана и сдана заказчику. Но ее "чахлая" модель предметной области свелась к тому, чтобы удовлетворить требования приложения к работе с базой данных и распределить все операции между несколькими СЛУЖБАМИ. Почти никак не проявились преимущества ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN), потому что модель не настолько прямо выражалась в коде, чтобы разработчик мог с нею работать.

Программирование в рамках архитектурных сред преследует две вполне законные цели. Во-первых, это логическое разделение обязанностей: один объект отвечает за доступ к базе данных, другой — за операции прикладной модели и т.д. Такое разделение позволяет легче понимать работу каждого уровня (в техническом смысле) и свободнее переключаться между ними. Беда в том, что при этом не осознается, какую цену приходится платить разработчикам. В этой книге вопросы программирования в архитектурных средах не рассматриваются, поэтому альтернативных решений мы искать не будем (хотя они есть). Но даже если бы других вариантов не было, лучше было бы отбросить все эти преимущества ради более связного уровня предметной области.

Движущим мотивом для применения таких схем разбиения на пакеты еще называют сетевую распределенность уровней. Это сильный довод в тех случаях, когда код действительно устанавливается на разных серверах. Но обычно это не так. А гибкости в этом смысле стоит добиваться только тогда, когда это действительно необходимо. В проекте, где есть надежда воспользоваться преимуществами ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ, такая жертва слишком велика, если только она не решает наболевшую, не терпящую отлагательства проблему.

Сложные схемы разбиения на пакеты, базирующиеся на технических соображениях, имеют две особенности:

- если требования архитектурной среды к распределению обязанностей таковы, что элементы, реализующие концептуальные объекты, оказываются физически разделенными, то код больше не выражает модель;
- нельзя разделять до бесконечности — у человеческого ума есть свои пределы, до которых он еще способен соединять разделенное; если среда выходит за эти пределы, разработчики предметной области теряют способность расчленять модель на осмысленные фрагменты.

Нужно стремиться к простоте. Сформируйте минимальный набор технических правил разделения — таких, без которых в данной среде обойтись нельзя, или которые действительно помогают в работе. Например, отделение сложного кода для хранения и обмена данными от операционных аспектов объектов может облегчить рефакторинг.

Если нет реальной необходимости распределять код между разными серверами, храните весь код, реализующий один концептуальный объект, в одном МОДУЛЕ**, а то и классе.**

К тому же самому заключению можно было бы придти, следуя старому стандарту "высокая внутренняя связность + низкая внешняя зависимость". Связи между подобъектом, реализующим операции модели, и другим подобъектом, отвечающим за доступ к базе данных, настолько обширны, что высокая внешняя зависимость тут налицо.

Существуют и другие ловушки, грозящие принципу ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ из-за правил и соглашений, принятых в архитектурных средах программирования, компаниях или отдельных проектах. Из-за них тоже может пострадать естественная связность объектов предметной области, и вывод будет таким же. Наличие строгих правил или чрезмерное количество обязательных пакетов часто сводит на нет возможность реализации других схем модульного разбиения, специально построенных для нужд модели предметной области.

Используйте разбиение на пакеты только для того, чтобы отделить уровень предметной области от остального кода. Оставляйте побольше свободы разработчикам модели предметной области, чтобы они могли объединить ее объекты в пакеты таким образом, какой лучше всего подходит для выражения модели и проектных архитектурных решений.

Исключение можно сделать в том случае, если код генерируется на основе декларативной архитектуры (см. главу 10). В этом случае разработчику нет нужды читать код, и его лучше поместить в отдельный пакет, чтобы он не мешал разработчикам выстраивать архитектурные элементы программы.

Вопрос модульного строения программы встает все острее по мере того, как архитектура программы усложняется и разрастается. В этом разделе представлены только элементарные соображения по этому поводу. В части IV "Стратегическое проектирование" много говорится о том, как правильно разбивать программу на модули, на какие части и по каким принципам делить большие модели и архитектуры, а также как привлечь внимание к их ключевым элементам для лучшего понимания процесса.

Каждое понятие модели должно найти свое отражение в элементе реализации. СУЩНОСТИ, ОБЪЕКТЫ-ЗНАЧЕНИЯ, ассоциации между ними, а также несколько СЛУЖБ уровня предметной области и МОДУЛИ, служащие для организации всего этого, — в них устанавливается прямое соответствие между программной реализацией и моделью. Объекты, указатели и механизмы извлечения данных в реализации должны соответствовать модели напрямую и самым очевидным образом. Если это не получается, приведите в порядок код, вернитесь на несколько шагов назад, измените модель — или сделайте все это вместе.

Боритесь с искушением нагрузить объекты предметной области чем-нибудь таким, что непосредственно не относится к понятиям, которые они представляют. Помните, что у этих элементов программной архитектуры уже есть обязанности — они выражают модель. Существуют и другие относящиеся к предметной области обязанности, и их надо выполнять; существуют и другие данные, и их нужно обрабатывать для слаженной работы всей системы. Но объекты модели не обязаны тащить все это на себе. В главе 6 мы рассмотрим некоторые вспомогательные объекты, выполняющие технические обязанности уровня предметной модели — например, задание поиска по базе данных и инкапсуляция создания сложных объектов.

Четыре архитектурных шаблона, представленных в этой главе, — это структурные единицы, "кирпичики" объектной модели. Но в ПРОЕКТИРОВАНИИ ПО МОДЕЛИ не обязательно насильно впихивать все на свете в формат объектов. Существуют и другие парадигмы моделирования, поддерживаемые средами программирования — например, *сервер приложений* (*business rule engine*). В проектах приходится идти на прагматические компромиссы между такими парадигмами. Но все эти среды, архитектуры и приемы — только средства для ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ, а не замена ему.

Парадигмы моделирования

ПРОЕКТИРОВАНИЕ ПО МОДЕЛИ требует такой технологии реализации, которая согласовывалась бы с конкретной парадигмой моделирования, применяемой в проекте. Эксперименты проводились со многими парадигмами, но всего несколько из них остались в широком употреблении. В настоящее время основная парадигма — объектно-ориентированная. В основе разработки наиболее сложных проектов лежит именно работа с объектами. Объектно-ориентированный подход стал доминировать по ряду причин. Частично сыграли свою роль естественные свойства объектов, частично так

сложились обстоятельства, а некоторые преимущества вытекают уже просто из широкой распространенности этой парадигмы.

Причины доминирования объектной парадигмы

Разработчики не обязательно выбирают объектную парадигму по техническим причинам или ради идейных преимуществ объектов. Но нам важно не разнообразие мотивов, а то, что объектное моделирование действительно предлагает неплохой баланс простоты в работе и возможности сложной организации систем.

Если парадигма моделирования слишком трудна для восприятия, ее не сможет освоить достаточное количество разработчиков, и использоваться она будет неправильно. Если члены группы, непосредственно не связанные с кодом, не будут владеть хотя бы основами парадигмы, они не поймут модель, и в проекте потеряется ЕДИНЫЙ ЯЗЫК. Принципы объектно-ориентированного программирования большинству людей кажутся естественными. Хотя некоторые программисты теряются в тонкостях моделирования, даже "не-технари" способны понять графическую схему объектной модели.

Несмотря на простоту концепции объектного моделирования, она оказалась достаточно емкой для передачи важных знаний о предметной области. С самого своего возникновения она поддерживалась набором средств программирования, с помощью которого модель можно было выразить в программе.

В настоящее время объектная парадигма также имеет ряд значительных ситуационных преимуществ, которые проистекают из ее развитости и широкой распространенности. Без развитой инфраструктуры и инструментальной поддержки проект может стать похожим на научно-исследовательскую работу по новым технологиям программирования. Во-первых, на это уйдут время и ресурсы, которые надо было бы употребить на основную разработку, а во-вторых, появится лишний технический риск. Некоторые технологии между собой плохо совместимы, и может оказаться, что интегрировать их в одно целое не удастся с помощью стандартных профессиональных решений, отчего придется заново изобретать многие нужные вещи. Но за долгие годы многие из этих проблем для объектно-ориентированных сред либо уже решены, либо благодаря широкому распространению этого подхода стали неактуальны. (Теперь уже другие парадигмы вынуждены подстраиваться, чтобы иметь возможность интегрироваться с объектной.) Большинство новых технологий изначально включает в себя средства интеграции с популярными объектно-ориентированными платформами. Это не только облегчает интеграцию, но и часто дает дополнительную возможность смешивания в одной системе нескольких подсистем, построенных на разных парадигмах моделирования (о чем мы поговорим позже в этой главе).

Не менее важна также достаточная степень развития *сообщества разработчиков и культуры проектирования программных архитектур.* Для проекта, в котором принимается новая парадигма, иногда не удается найти разработчиков с достаточными знаниями нужных технологий или опытом построения эффективных моделей для выбранной парадигмы. Не исключено, что будет невозможно обучить программистов за разумное время, потому что еще не устоялись типовые архитектурные шаблоны для построения большей части парадигмы и технологии. Новаторы в этой области, может, и могли бы справиться с задачей, но они еще не опубликовали свои наработки в доступной форме.

А вот объектно-ориентированный подход уже известен сообществу из многих тысяч программистов, руководителей проектов и всевозможных специалистов, участвующих в разработке программного обеспечения.

Проиллюстрировать рискованность работы с неразвитой парадигмой можно на примере одного проекта, разрабатывавшегося около полутора десятилетий тому назад. В начале

90-х годов для этого проекта были приняты на вооружение несколько самых передовых технологий, в том числе широкое использование объектно-ориентированной базы данных. Это было очень увлекательно. Гостям проекта с гордостью рассказывали, что создается и разворачивается самая большая база данных, когда-либо поддерживавшаяся в рамках этой технологии. Когда я присоединился к проекту, несколько разных групп уже мастерили объектно-ориентированные архитектуры и безо всякого труда помещали свои объекты в базу. Но постепенно на нас снизошло понимание того, что мы уже заняли значительную часть допустимого объема базы — и это еще только тестовыми данными! А реальная база должна была быть в десятки раз объемнее, и объем транзакций, проходящих через нее, — в десятки раз больше. Так возможно ли применить данную технологию в этой разработке? Использовали ли мы ее неправильно? Это было выше нашего понимания.

К счастью, нам удалось привлечь к работе одного из немногих в мире специалистов, имевших достаточную квалификацию, чтобы решить нашу проблему. Он назвал свою цену, и мы ее заплатили. Проблема имела три источника. Во-первых, готовая инфраструктура, которая прилагалась к базе данных, не масштабировалась под наши нужды. Во-вторых, хранение мелкомасштабных объектов отнимало гораздо больше ресурсов, чем мы думали. В-третьих, отдельные части объектной модели были связаны такой паутиной взаимосвязей, что даже при относительно небольшом числе одновременных транзакций возникали конфликты.

С помощью привлеченного специалиста мы усовершенствовали инфраструктуру. Группа, уже будучи в курсе недостатков мелкомасштабных объектов, стала искать такие модели, которые бы работали с этой технологией. Все мы осознали, как важно ограничить количество взаимосвязей в модели, и в новых моделях старались добиться большей независимости отдельных, пусть и родственных, конструкций.

На все это ушло несколько месяцев, не считая еще нескольких, в течение которых проект “шел неверной дорогой”. И это была не первая неудача группы разработчиков, вызванная несовершенством выбранных технологий и недостатком опыта в освоении таких вопросов. К сожалению, впоследствии проект урезали, и он практически законсервировался. В наше время в нем еще используются эти экзотические технологии, но так тщательно ограничивают сферу их применения, что это, вероятно, не дает никаких преимуществ.

За прошедшие с тех пор годы объектно-ориентированные технологии достигли сравнительно высокого уровня развития. Большинство требуемых инфраструктурных решений доступны в готовом виде, обкатанном в индустрии. Критически важные для проекта инструментальные средства можно купить у крупных поставщиков, часто у нескольких, или взять из надежных проектов с открытым кодом. Многие из инфраструктурных решений сами по себе применяются настолько широко, что уже существует и достаточный круг знающих людей, и книги с описаниями, и т.п. Ограничения широко применяемых технологий также хорошо известны, поэтому квалифицированный коллектив имеет мало шансов “съехать” в сторону.

Но другие интересные парадигмы моделирования еще не достигли такой степени зрелости. Некоторые слишком трудны в освоении, поэтому никогда не выйдут за пределы узкой специализации. У других есть потенциал, но их инфраструктура еще недоработана или ненадежна, и лишь немногие специалисты владеют тонкостями моделирования для таких систем. Может быть, их час еще настанет, но пока они “сыроваты” для большинства проектов.

Вот почему в настоящее время в большинстве проектов, где планируется применять ПРОЕКТИРОВАНИЕ ПО МОДЕЛИ, разумно будет выбрать объектно-ориентированную технологию в качестве ядра системы. При этом чисто объектной системой ограничиваться

нет нужды — объектная парадигма главенствует в отрасли, так что существуют средства интеграции с практически любой существующей технологией.

Впрочем, это не значит, что мы всегда обязаны выбирать объектно-ориентированный подход. Идти вместе с толпой, может, и безопаснее, но не всегда это правильный выбор. Объектные модели позволяют адекватно решать огромное количество задач практического программирования, но существуют такие области, которые неестественно моделировать дискретными смысловыми единицами с инкапсулированными в них операциями. Например, предметные области со сложной математикой или преобладанием глобальных логических построений не всегда хорошо вписываются в объектно-ориентированную парадигму.

Не-объекты в объектном мире

Модель предметной области не обязана быть объектной. Например, существуют архитектуры, реализованные на языке Prolog, на основе моделей, составленных из логических правил и фактов. В парадигмах моделирования формализуются способы, которыми люди думают о тех или иных предметных областях. Потом модели приобретают конкретную форму под влиянием созданных парадигм. В результате получаются модели, соответствующие парадигмам и пригодные для эффективной реализации теми средствами и в тех средах, которые поддерживают данный стиль моделирования.

Какова бы ни была основная парадигма моделирования в проекте, в предметной области непременно будут такие аспекты, которые легче выразить в рамках другой парадигмы. Если в предметной области совсем немного элементов, которые не вписываются в принятую (и подходящую) парадигму, то разработчики вполне могут смириться с наличием нескольких "неуклюжих" объектов в модели, которая в остальном выглядит стройно и согласованно. (В случае другой крайности — если большая часть модели более естественно описывается в рамках другой парадигмы — может быть, имеет смысл перейти на эту другую парадигму и сменить платформу реализации?) Но если достаточно крупные фрагменты модели относятся к другим парадигмам, разумно будет в каждом частном случае принять на вооружение подходящие парадигмы, а программную реализацию делать с применением смешанного набора средств разработки. Если взаимосвязи и зависимости не слишком тесные, можно ограничиться инкапсуляцией подсистемы, построенной на другой парадигме, — например, просто вызвать из объекта процедуру сложных математических расчетов. Но бывает и так, что разные аспекты модели связаны более тесно, — например, если взаимодействие между объектами зависит от неких математических соотношений.

Именно поэтому в объектные системы часто приходится интегрировать такие необъектные компоненты, как *сервер приложений* (*business rules engine*) и *сервер делопроизводства* (*workflow engine*). Смешивание парадигм позволяет разработчикам моделировать те или иные понятия таким способом, который лучше всего для них подходит. К тому же, в большинстве систем все равно приходится использовать необъектные элементы технической инфраструктуры, — как правило, реляционные базы данных. Но построить единообразную модель, подчиняющуюся нескольким парадигмам, нелегко, и обеспечить сосуществование технических средств программирования тоже задача непростая. Если разработчики не видят четкого выражения модели в программе, ПРОЕКТИРОВАНИЕ ПО МОДЕЛИ "приказывает долго жить" — пусть даже смешение парадигм и инфраструктурных средств требует именно его.

ПРОЕКТИРОВАНИЕ ПО МОДЕЛИ в условиях смешения парадигм

Здесь в качестве примера технологии, иногда "примешиваемой" к объектно-ориентированным приложениям, будет служить сервер приложений (*rules engine*), реализую-

щий деловые регламенты из определенной области. Информоемкая модель предметной области вполне может содержать явно прописанные регламентные правила, вот только объектная парадигма лишена нужных семантических средств для формулировки правил и связей между ними. Конечно, регламенты можно моделировать объектами, и это часто делается не без успеха, но их инкапсуляция в объектах мешает применять глобальные правила, распространяющиеся на всю систему. Технология серверов приложений не лишена привлекательности, поскольку обещает более естественный и декларативный способ определения правил, при котором парадигма регламентов может смешиваться с объектной парадигмой. Парадигма логики (регламентированных правил) хорошо разработана и высокоэффективна, и есть основания полагать, что она может послужить хорошим дополнением к сильным и слабым местам объектов.

Но не всегда от серверов приложений удается получить то, на что надеешься. Многие программные продукты не хотят работать как следует, и все тут. Некоторым из них не хватает внутренней согласованности — когда понятия модели, разделенные между двумя средами реализации, сохраняют родственность и взаимосвязанность. В результате часто получается так, что приложение как бы разбивается на две части: статическую систему хранения данных с использованием объектов и регламентно-алгоритмическую подсистему, которая утратила почти всякую связь с объектной моделью.

Работая с деловыми регламентами, важно не прекращать думать в терминах модели. Разработчикам необходимо найти единую модель, которая годилась бы для обеих парадигм реализации. Это нелегко, но возможно, если сервер приложений допускает выразительную смысловую реализацию. В противном случае данные и регламенты оказываются несвязанными. Регламентные правила в составе сервера начинают напоминать, скорее, маленькие программки, чем концептуальные правила модели предметной области. Но если между регламентами и объектами установлены тесные и четкие отношения, то смысл обеих частей системы сохраняется.

Если нет согласованной единообразной среды, то вся тяжесть построения модели из четких и фундаментальных концепций ложится на разработчиков. Именно от них зависит, удастся ли выдержать всю архитектуру как более-менее единое целое.

Наиболее эффективный инструмент сбора частей в одно целое — это надежный ЕДИНЫЙ ЯЗЫК, стоящий за неоднородной моделью. Последовательное применение имен и терминов в двух разных средах и введение этих имен в ЕДИНЫЙ ЯЗЫК помогает заполнить смысловой пробел.

Эта тема сама по себе заслуживает отдельной книги. Цель же этого раздела состоит в том, чтобы показать: нет причин отказываться от ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ, и стоит приложить все усилия, чтобы удержаться на этом пути.

Хотя ПРОЕКТИРОВАНИЕ ПО МОДЕЛИ не обязано быть объектно-ориентированным, в нем все-таки нельзя обойтись без выразительной реализации конструкций модели — будь то объекты, регламенты или процедуры делопроизводства. Если выбранные для работы программные средства не обладают нужной выразительностью, пересмотрите этот выбор. Невыразительная программная реализация сводит на нет преимущества дополнительной парадигмы.

Ниже даны четыре основных правила для ввода необъектных элементов в преимущественно объектно-ориентированную систему.

- *Не противостоять парадигме реализации.* Всегда есть другой способ посмотреть на предметную область. Найдите концепции, которые бы соответствовали парадигме.

- *Полагаться на единый язык.* Даже если между частями среды реализации нет отчетливой связи, применение единообразного языка может удержать разные части модели от “расползания” в стороны.
- *Не зацикливаться на UML.* Иногда чрезмерное увлечение каким-нибудь техническим средством, таким как UML-диаграммы, заставляет разработчиков искажать модель в угоду удобству вычерчивания схем. В UML есть кое-какие средства для представления, например, связей-ограничений, но их не всегда достаточно. Лучше прибегнуть к графическим схемам другого вида (может быть, вполне обычным для другой парадигмы) или описаниям на обычном человеческом языке, чем мучительно адаптироваться к схемам одного частного вида, передающим только один из взглядов на объектные модели.
- *Все подвергать сомнению.* Действительно ли то или иное инструментальное средство работает на полную мощность? Пусть в проекте и применяется несколько регламентов — это еще не значит, что для них нужно заводить целый сервер приложений. Регламентные правила можно выразить в виде объектов, пусть и не так аккуратно, а вот одновременное применение нескольких парадигм все усложняет многократно.

Прежде чем взваливать на себя груз нескольких смешанных парадигм, следует исчерпать все возможности в пределах доминирующей парадигмы. Пусть некоторые понятия модели и не представляют собой очевидных объектов — все равно их часто можно промоделировать в пределах парадигмы. В главе 9 будет рассматриваться моделирование необычных видов понятий моделей с использованием объектной технологии.

Особый случай смешения парадигм представляет собой реляционная парадигма. Будучи самой распространенной необъектной технологией, реляционная база данных в то же время более тесно связана с объектной моделью, чем другие компоненты, потому что служит постоянным хранилищем данных, образующих в совокупности сами объекты. Хранение объектных данных в реляционных базах, как и многие другие проблемы цикла существования объектов, будет рассматриваться в главе 6.

Глава 6

Цикл существования объектов модели

Каждый объект имеет свой цикл существования. Объект “рождается”, затем, как правило, проходит ряд сменяющихся состояний, а потом “умирает” — либо удаляется, либо помещается в архив. Многие из объектов — простые и временные; они создаются после вызова конструктора, используются в вычислительных операциях, после чего их оставляют сборщику мусора. Усложнять такие объекты ни к чему. Но у некоторых объектов существование может продлиться намного дольше и частично пройти не в оперативной памяти. Они имеют сложные связи и отношения с другими объектами и проходят через изменения состояния, подчиняющиеся определенным инвариантам. Управление такими объектами не обходится без трудностей, которые легко могут привести к нарушению принципов ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN).

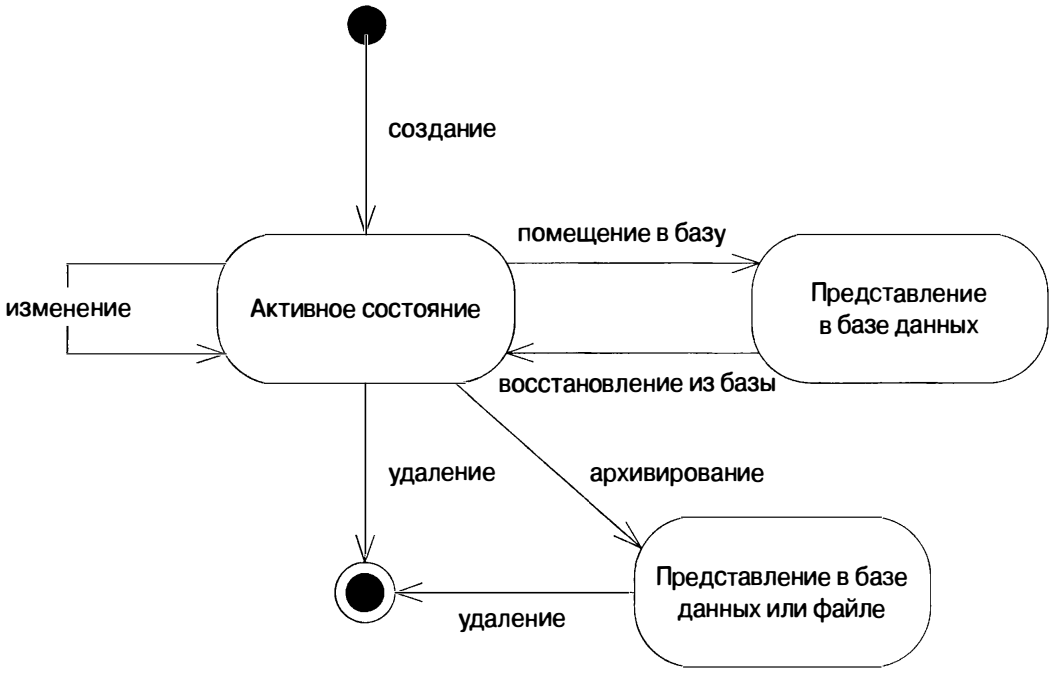


Рис. 6.1. Цикл существования объекта из модели предметной области

Эти трудности делятся на две категории.

1. Поддержание целостности объекта на этапе его существования.
2. Предотвращение излишней сложности в управлении циклом существования объектов.

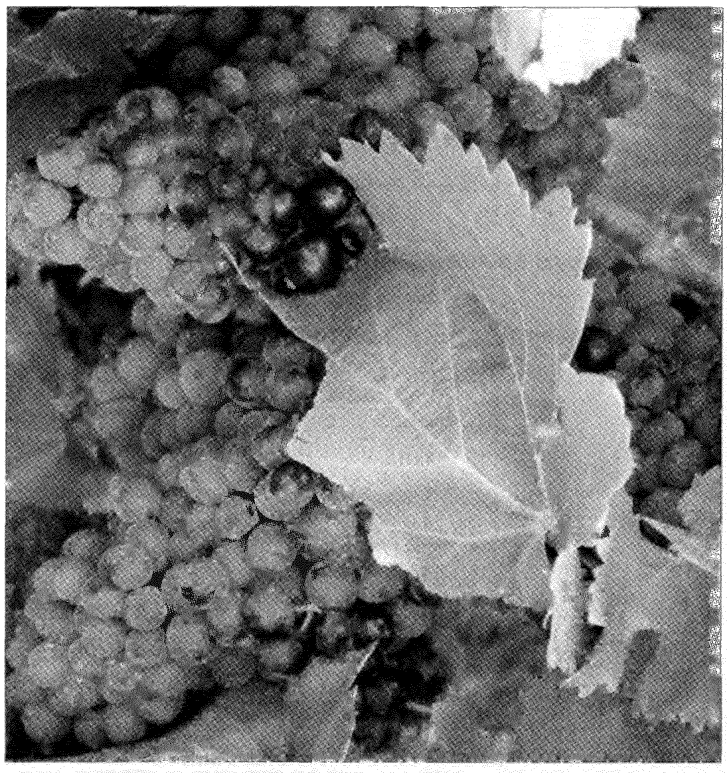
В этой главе указанные проблемы будут решаться на основе трех архитектурных шаблонов. Во-первых, АГРЕГАТЫ (AGGREGATES) помогут "подтянуть" саму модель и избавиться от хаотической путаницы разнообразных объектов, четко определив права собственности и границы. Этот шаблон играет решающую роль в поддержании целостности объектов на всех этапах их существования.

Далее мы сместим акцент на начало цикла существования и воспользуемся ФАБРИКАМИ (FACTORIES) для создания и восстановления сложных объектов и АГРЕГАТОВ с сохранением инкапсуляции их внутренней структуры. Наконец, середина и конец жизненного цикла объектов — это компетенция ХРАНИЛИЩ (REPOSITORIES), которые позволяют находить и извлекать постоянно хранимые объекты, при этом инкапсулируя гигантскую инфраструктуру, стоящую за этими операциями.

Хотя ХРАНИЛИЩА и ФАБРИКИ непосредственно не происходят из предметной области, они играют важную смысловую роль в ее архитектуре. Эти конструкции — ценное подспорье в ПРОЕКТИРОВАНИИ ПО МОДЕЛИ, поскольку они дают нам в руки удобные средства обращения с объектами модели.

Моделирование АГРЕГАТОВ наряду с добавлением в программу ФАБРИК и ХРАНИЛИЩ дает нам возможность манипулировать объектами систематически, в естественных смысловых границах, на протяжении всего цикла их существования. АГРЕГАТЫ обозначают область действия, в пределах которой на каждом этапе жизненного цикла должны удовлетворяться определенные инварианты. ФАБРИКИ и ХРАНИЛИЩА оперируют АГРЕГАТАМИ, инкапсулируя сложности специфических переходных этапов их существования.

Агрегаты



Если снизить до минимума количество вводимых в модель ассоциаций, становится легче проследить взаимосвязи между понятиями и хотя бы немного ограничить их лавинообразный рост. Но большинство практических предметных областей обладает такой богатой внутренней связностью, что в конце концов все равно приходится трассировать

длинные цепочки ссылок одних объектов на другие. В каком-то смысле это отражение богатства реального мира, который нечасто балует нас четкими и нерушимыми границами. Но это-то и создает проблему при разработке программного обеспечения.

Предположим, мы удаляем объект **Человек (Person)** из базы данных. Вместе с ним удаляется имя, дата рождения, описание его работы. А как насчет адреса? По тому же адресу могут проживать и другие люди. Если удалить адрес, соответствующие объекты **Человек** будут содержать ссылки на удаленный объект. А если оставить, в базе будут накапливаться отработанные адреса. Автоматическая сборка мусора могла бы ликвидировать эти адреса, но даже если такая возможность есть в СУБД, это чисто техническая мера, — а принципиальный вопрос моделирования остается нерешенным.

Даже для отдельной транзакции трудно предсказать, до каких пределов может простираться ее влияние, учитывая переплетение взаимосвязей в типичной объектной модели. А обновлять каждый объект в системе на случай наличия какой-то связи — не слишком практично.

Проблема особенно обостряется в системах с параллельным доступом к одним и тем же объектам со стороны многочисленных клиентов. В случае, когда сразу много пользователей могут просматривать и изменять различные объекты в системе, приходится думать, как предотвратить одновременные модификации взаимозависимых объектов. Если неправильно оценить области определения и действия, можно оказаться в беде.

При внесении изменений в объекты модели, обладающей сложной системой ассоциаций, трудно гарантировать согласованность этих изменений. Необходимо соблюдать инварианты, относящиеся к тесно связанным группам объектов, а не отдельным объектам. Но если применить слишком осторожные схемы блокирования, то параллельные пользователи станут невольно мешать друг другу, и это сделает всю систему неработоспособной.

Сформулируем задачу по-другому. Откуда мы знаем, где начинается и где заканчивается объект, составленный из других объектов? В любой системе с постоянным хранением данных у каждой транзакции, которая вносит в данные изменения, должна быть своя ограниченная область действия. Кроме того, должен быть способ поддерживать согласованность данных (путем соблюдения их инвариантов). В базах данных можно применять разные схемы блокирования, а также программировать тесты. Но эти половинчатые решения отвлекают нас от модели, и очень скоро положение только усугубится.

На самом деле, чтобы найти разумное решение проблем такого рода, требуется углубленное понимание предметной области, а именно таких ее параметров, как частота смены экземпляров того или иного класса. Необходимо построить модель, которая давала бы больше свободы в местах интенсивной передачи конкурирующих данных, но заставляла четко соблюдать строгие инварианты.

Эта проблема, на первый взгляд, кажется технической, связанной с транзакциями баз данных, но корни ее лежат в модели — в недостаточно определенных границах. Если вывести решение из модели, она сама станет понятнее, и выразить ее в программной архитектуре станет легче. По мере пересмотра модели соответствующие изменения будут вноситься и в программную реализацию.

Существуют проработанные схемы для определения прав собственности/владения в модели. Описанная ниже простая, но строгая система, выведенная из них, предлагает набор правил для реализации транзакций, которые вносят изменения и в первичные объекты, и в те, которые ими владеют[1].

---

[1] Эту систему разработал и реализовал Дэвид Сигел (David Siegel) в проектах 90-х годов, но не опубликовал ее.

Прежде всего нам потребуется абстракция для инкапсуляции ссылок в пределах модели. Совокупность взаимосвязанных объектов, которые мы воспринимаем как единое целое с точки зрения изменения данных, называется АГРЕГАТОМ (AGGREGATE). У каждого АГРЕГАТА есть *корневой объект* и есть *граница*. Граница определяет, что находится внутри АГРЕГАТА. Корневой объект — это один конкретный объект-СУЩНОСТЬ (ENTITY), содержащийся в АГРЕГАТЕ. Корневой объект — единственный член АГРЕГАТА, на который могут ссылаться внешние объекты, в то время как объекты, заключенные внутри границы, могут ссылаться друг на друга как угодно. СУЩНОСТИ, отличные от корневого объекта, локально индивидуальны, но различаться они должны только в пределах АГРЕГАТА, поскольку никакие внешние объекты все равно не могут их видеть вне контекста корневой СУЩНОСТИ.

Пусть, например, в авторемонтной мастерской используется программа, в которой построена модель автомобиля. Автомобиль представляет собой сущность с глобальной идентичностью — его необходимо отличать от всех остальных машин в мире, даже очень похожих. Для этой цели подходит номерной знак, поскольку это индивидуальный идентификатор, присвоенный каждому автомобилю. Нам может понадобиться проследить историю использования шин в их четырех возможных положениях на колесах, узнать их пробег и степень износа. Чтобы точно знать, где какая шина, их тоже следует сделать СУЩНОСТЯМИ. Но очень маловероятно, что их индивидуальность будет нас интересовать вне контекста конкретного автомобиля. Если заменить шины и отослать старые на переработку, то либо наша программа вообще перестанет их отслеживать, либо они станут анонимными членами кучи списанных шин. Их пробег уже никого не будет волновать. Но более уместно к теме будет заметить, что даже если шины установлены на машине, никто не попытается ввести в систему запрос на поиск конкретной шины, чтобы потом посмотреть, на какой машине она стоит. Наоборот, в базу данных будет послан запрос на поиск машины, а потом запрос на временную ссылку, ведущую к шинам. Поэтому автомобиль является корневой СУЩНОСТЬЮ в АГРЕГАТЕ, граница которого охватывает также и шины. С другой стороны, двигатели имеют собственные, выбитые на них серийные номера, по которым их иногда разыскивают независимо от автомобилей. В некоторых приложениях двигатели могут быть корневыми объектами своих собственных АГРЕГАТОВ.

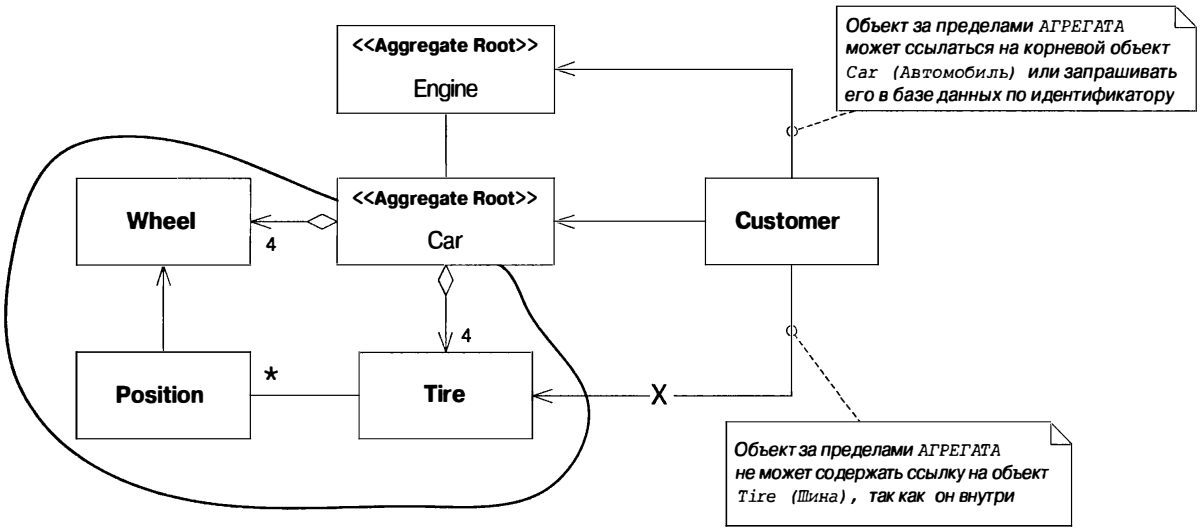


Рис. 6.2. Локальная/глобальная идентичность и ссылки на объекты

Из взаимосвязей между объектами АГРЕГАТА можно составить так называемые *инварианты*, т.е. правила совместности, которые должны соблюдаться при любых изменени-

ях данных. Не всякое правило, распространяющееся на АГРЕГАТ, обязано выполняться непрерывно. Восстановить нужные взаимосвязи за определенное время можно с помощью обработки событий, пакетной обработки и других механизмов обновления системы. Но соблюдение инвариантов, имеющих силу внутри агрегата, должно контролироваться немедленно по завершении любой транзакции.

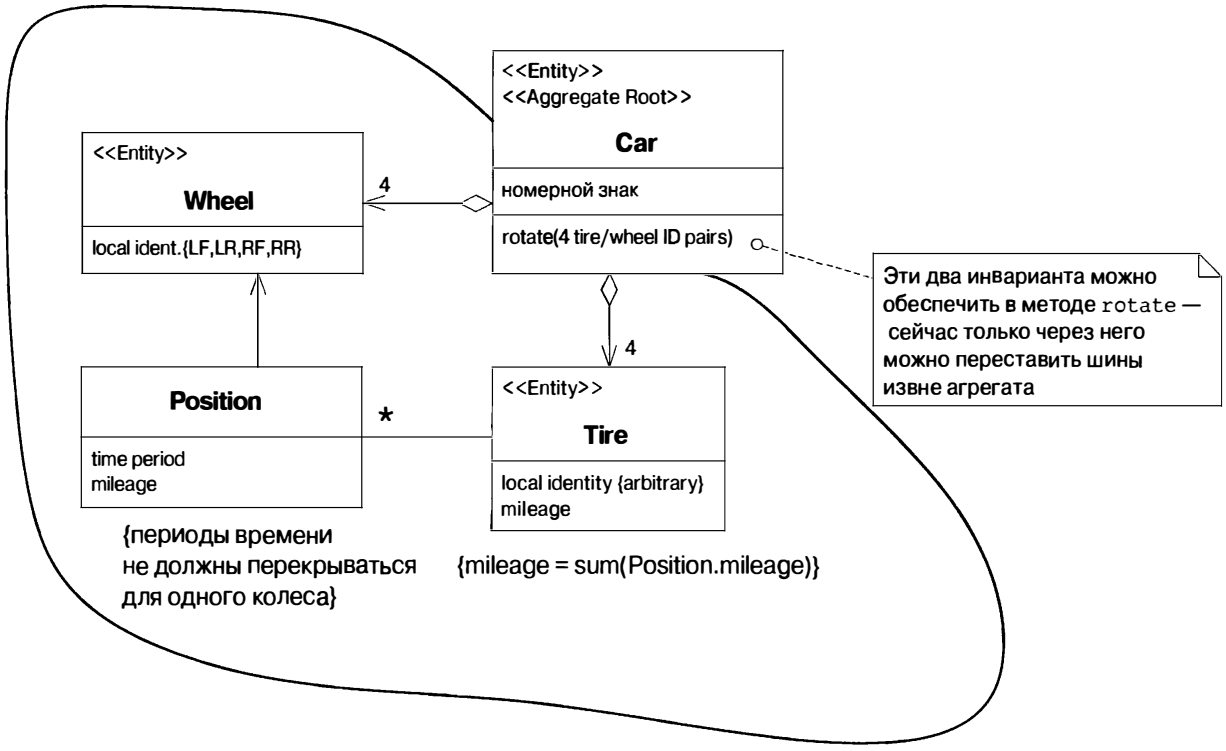


Рис. 6.3. Инварианты АГРЕГАТОВ

Чтобы реализовать концепцию АГРЕГАТА в виде практической программной конструкции, необходимо иметь набор правил, выполняющихся для любой транзакции.

- Корневой объект-СУЩНОСТЬ имеет глобальную идентичность и несет полную ответственность за проверку инвариантов.
- Некорневые объекты-СУЩНОСТИ имеют локальную идентичность — они уникальны только в границах АГРЕГАТА.
- Нигде за пределами агрегата не может постоянно храниться ссылка на что-либо внутри него, кроме его корневого объекта. Корневой объект-СУЩНОСТЬ может передавать ссылки на внутренние объекты-СУЩНОСТИ другим объектам. Но эти другие объекты могут использовать их только временно, и не имеют права хранить их или как-то фиксировать. Корневой объект может передать копию ОБЪЕКТА-ЗНАЧЕНИЯ другому объекту, и что с ней потом случится не играет роли, поскольку это не более чем значение, и оно не имеет никаких связей с АГРЕГАТОМ.
- Как следствие из предыдущего правила, только корневые объекты АГРЕГАТОВ можно непосредственно получать по запросам из базы данных. Все остальные объекты разрешается извлекать только по цепочке связей.
- Объекты внутри АГРЕГАТА могут хранить ссылки на корневые объекты других АГРЕГАТОВ.

- Операция удаления должна одновременно ликвидировать все, что находится в границах АГРЕГАТА. (При наличии сборки мусора это просто. Поскольку внешних ссылок ни на что, кроме корневого объекта, не существует, достаточно удалить корневой объект, а остальное будет подчищено при сборке.)
- Как только вносится изменение в любой объект внутри границ АГРЕГАТА, следует сразу удовлетворить все инварианты этого АГРЕГАТА.

Группируйте СУЩНОСТИ **и** ОБЪЕКТЫ-ЗНАЧЕНИЯ **в** АГРЕГАТЫ **и определяйте границы каждого из них. Выберите один объект-**СУЩНОСТЬ **и сделайте его корневым. Осуществляйте все обращения к объектам в границах** АГРЕГАТА **только через его корневой объект. Разрешайте внешним объектам хранить ссылки только на корневой объект. Ссылки на внутренние объекты** АГРЕГАТА **следует передавать только во временное пользование, на время одной операции. Поскольку доступ к объектам** АГРЕГАТА **контролируется через корневой объект, неожиданные изменения внутренних объектов невозможны. В такой схеме разумно требовать удовлетворения всех инвариантов для объектов в** АГРЕГАТЕ **и для всего** АГРЕГАТА **в целом при любом изменении состояния.**

Было бы очень удобно иметь такую среду программирования, которая бы позволяла объявлять АГРЕГАТЫ, автоматически реализуя схемы блокирования и т.п. Не имея такой поддержки, группа разработчиков должна иметь достаточно самодисциплины, чтобы выработать соглашение об АГРЕГАТАХ и программировать в четком соответствии с ним.

Пример

Целостность товарного заказа

Рассмотрим потенциальные осложнения, которые могут возникнуть в упрощенной системе приема товарных заказов.

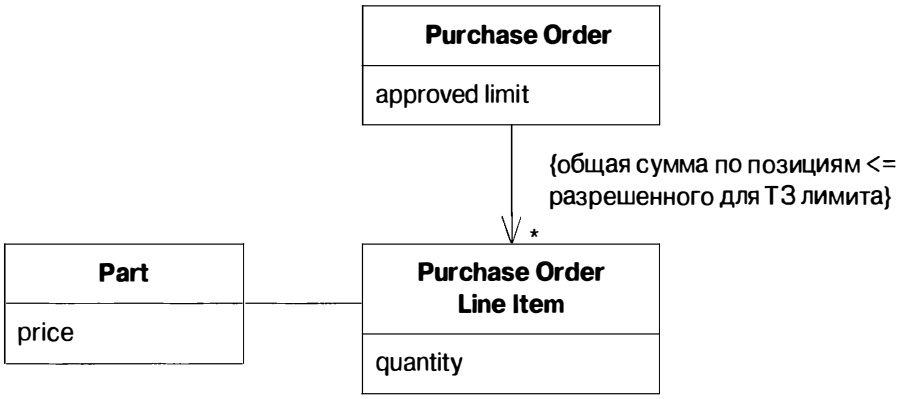


Рис. 6.4. Модель системы по обработке товарных заказов

На рисунке показана довольно обычная схема товарного заказа (ТЗ), разбитого на позиции, для которого введено правило-инвариант: суммарная стоимость всех позиций не может превышать предел, установленный для одного ТЗ в целом. В существующей реализации имеется три взаимосвязанные проблемы.

1. *Удовлетворение инварианта.* При добавлении новой позиции объект-заказ проверяет сумму и помечает себя как недопустимый, если превышен лимит. Как мы увидим далее, этой защитной меры недостаточно.

2. *Управление изменениями.* Если ТЗ перемещается в архив или удаляется, позиции уходят вместе с ним, но модель не дает никаких указаний, где нужно остановиться в следовании по цепочке взаимосвязей. Непонятно также, как именно на процесс влияет изменение цены того или иного товара в разное время.
3. *Параллельное обращение к базе данных.* Обращения разных пользователей к базе данных могут привести к конфликту данных.

Сразу несколько пользователей будут одновременно вводить и обновлять несколько ТЗ, и нам необходимо не дать им помешать друг другу. Начнем с очень простой схемы, в которой мы блокируем любой объект, который начал редактировать пользователь, пока он не закончит свою транзакцию. Таким образом, пока Джордж правит позицию 001, Аманда не имеет к ней доступа. Она может редактировать любую другую позицию в любом другом ТЗ (включая и другие позиции из ТЗ, над которым работает Джордж).

PO #0012946 Approved Limit: $1,000.00

Item #	Quantity	Part	Price	Amount
001	3	Guitars	@100.00	300.00
002	2	Trombones	@200.004	00.00
			Total :7	00.00

Рис. 6.5. Начальное состояние объекта ТЗ, хранящегося в базе данных

Объекты считываются из базы данных и помещаются в виде экземпляров в область памяти каждого пользователя. Там их можно просматривать и редактировать. Блокировка базы запрашивается только тогда, когда начинается редактирование. Итак, и Джордж, и Аманда могут работать параллельно, пока они "не трогают" товарные позиции друг друга. И все идет хорошо... пока Джордж и Аманда не приступают к работе над разными строками одного и того же товарного заказа.

Джордж добавляет гитары через свой интерфейс

PO #0012946 Approved Limit: $1000.00

Item #	Quantity	Part	Price	Amount
001	**5**	Guitars	@100.00	**500.00**
002	2	Trombones	@200.00	400.00
			Total:	900.00

Аманда добавляет тромбон через свой интерфейс

PO #0012946 Approved Limit: $1,000.00

Item #	Quantity	Part	Price	Amount
001	3	Guitars	@100.00	300.00
002	**3**	Trombones	@200.00	600.00
			Total:	900.00

Рис. 6.6. Одновременное редактирование, разные транзакции

Пользователям и их программам кажется, что все хорошо, потому что они игнорируют изменения, внесенные в другие части базы данных во время их транзакций — ни одна из заблокированных одним пользователем позиций не влияет на работу другого пользователя.

PO #0012946 Approved Limit: **$1,000.00**

Item #	Quantity	Part	Price	Amount
001	5	Guitars	@100.00	500.00
002	3	Trombones	@200.00	600.00
			Total:	**1,100.00**

Рис. 6.7. В итоговом ТЗ нарушается инвариант — лимит общей стоимости

После того как оба пользователя сохраняют свои изменения, в базу данных записывается ТЗ, в котором нарушается инвариант модели предметной области. Не соблюдено важное правило делового регламента. И никто даже не догадывается об этом.

Очевидно, блокирование одной позиции в заказе не является достаточной защитной мерой. Если же блокировать весь заказ сразу, этой проблемы, очевидно, не возникнет.

Джордж редактирует заказ у себя

PO #0012946 Approved Limit: $1000.00

Item #	Quantity	Part	Price	Amount
001	5	Guitars	@100.00	500.00
002	2	Trombones	@200.00	400.00
			Total:	900.00

Для Аманды ТЗ 0012946 заблокирован

Изменения Джорджа зафиксированы

Аманда получает доступ: видны изменения Джорджа

PO #0012946 Approved Limit: $1000.00

Item #	Quantity	Part	Price	Amount
001	5	Guitars	@100.00	500.00
002	3	Trombones	@200.00	600.00
		Limit exceeded →	Total:	1,100.00

Рис. 6.8. Соблюдение инварианта при блокировании целого ТЗ

Программа не позволит сохранить результат, пока Аманда не устранит проблему — например, повысит лимит или уберет из заказа одну гитару. Итак, проблема предотвращена. Это решение может быть приемлемым, если работа в основном ведется одновременно над множеством разных заказов. Но если обычно вместе редактируются разные позиции одного большого ТЗ, такая блокировка создаст неудобства.

Но даже в случае множества мелких заказов есть и другие способы нарушить инвариант. Зайдем со стороны товара. Если кто-нибудь изменит цену на тромбоны, пока Аманда добавляет их в свой заказ, разве это не вызовет нарушения инварианта?

Попробуем заблокировать и товар в дополнение к целому ТЗ. И вот что получится, если Джордж, Аманда и Сэм работают над *разными* ТЗ.

Джордж редактирует ТЗ

Гитары и тромбоны заблокированы

PO #0012946 Approved Limit: $1,000.00

Item #	Quantity	Part	Price	Amount
001	**2**	Guitars	@100.00	**200.00**
002	2	Trombones	@200.00	400.00
			Total:	600.00

Аманда добавляет тромбоны; должна ждать Джорджа

Скрипки заблокированы

PO #0012932 Approved Limit: $1,850.00

Item #	Quantity	Part	Price	Amount
001	3	Violins	@400.00	1,200.00
002	**2**	**Trombones**	**@200.00**	**400.00**
			Total:	1,600.00

Сэм добавляет тромбоны; должен ждать Джорджа

PO #0013003 Approved Limit: $15,000.00

001	1	Piano	@1,000.00	1,000.00
002	**2**	**Trombones**	**@200.00**	**400.00**
			Total:	1,400.00

Рис. 6.9. Как блокировка-перестраховка мешает людям работать

Неудобства накапливаются, потому что данные о музыкальных инструментах (товарах) конкурируют за очередность изменений и происходит вот что.

Джордж добавляет скрипки; должен ждать Аманду (!)

PO #0012946 Approved Limit: $1,000.00

Item #	Quantity	Part	Price	Amount
001	**2**	Guitars	@100.00	**200.**00
002	2	Trombones	@200.00	400.00
003	**1**	**Violins**	**@400.00**	**400.00**
			Total:	1,000.00

Рис. 6.10. Затор

Итак, всем троим придется подождать.

В этот момент можно начать усовершенствование модели, добавляя в нее следующие знания из предметной области.

1. Товары используются во многих ТЗ (высокая конфликтность данных).

2. Изменения в данные о товарах вносятся реже, чем в товарные заказы.

3. Изменения в ценах товаров не обязательно распространяются на уже заполненные ТЗ. Это зависит от момента времени, когда было внесено изменение, по отношению к состоянию ТЗ.

Пункт 3 особенно очевиден, когда дело касается архивных, уже выполненных ТЗ. В них, конечно, должны указываться цены на момент заполнения заказов, а не текущие.

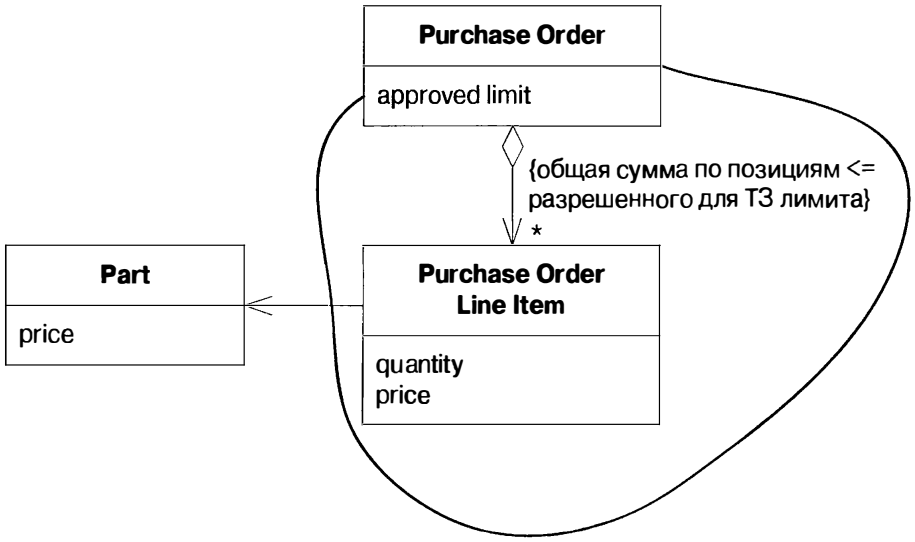


*Рис. 6.11. Цена копируется в **Позицию (Line Item)**. Теперь можно проконтролировать инвариант АГРЕГАТА*

Если привести программную реализацию в соответствие с этой моделью, то инвариант для ТЗ и его позиций будет выполняться гарантированно, а изменения в цене на товар не обязаны немедленно влиять на позиции, которые ссылаются на этот товар. Более широкие правила совместности можно учесть другими способами. Например, система могла бы каждый день предлагать пользователю список устаревших цен, чтобы их можно было обновить или удалить. Но это не инвариант, за выполнением которого нужен строгий контроль. Ослабляя взаимосвязь позиций в заказе и цен на товары, мы тем самым избегаем конфликтов данных и лучше отражаем реальные способы ведения дел. В то же время усиление взаимосвязи ТЗ и его позиций гарантирует выполнение важного делового регламента.

АГРЕГАТ вводит четкие права собственности на ТЗ и его позиции таким способом, который согласуется с реальной деловой практикой. Создание и удаление ТЗ и его позиций естественным образом объединены, тогда как создание и удаление товаров реализовано независимо.

* * *

АГРЕГАТЫ обозначают области действия, внутри которых на каждом этапе существования должны соблюдаться определенные инварианты. Описанные далее конструкции, а именно ФАБРИКИ (FACTORIES) и ХРАНИЛИЩА (REPOSITORIES), оперируют АГРЕГАТАМИ, инкапсулируя в себе некоторые сложные преобразования, выполняемые в жизненном цикле объектов.

Фабрики

Если создание объекта или целого АГРЕГАТА представляет большую сложность или открывает постороннему глазу слишком много внутренней структуры, то нужную инкапсуляцию обеспечивают ФАБРИКИ (FACTORIES).

* * *

В значительной мере сила такого инструмента, как объекты, заключается в его сложном внутреннем устройстве, включая и ассоциации между его элементами. Объект следует "дистиллировать" (т.е. очищать от всего лишнего), пока в нем не останется ничего, что не имеет отношения к самой его сути и его роли в транзакциях. С объекта довольно и этих обязанностей, свойственных середине цикла его существования. Если на сложный объект возложить еще и ответственность за создание самого себя, то начинают возникать проблемы.

Двигатель автомобиля — весьма непростое техническое устройство, в котором десятки частей взаимодействуют друг с другом для выполнения главной задачи — вращения вала. Теоретически можно спроектировать такой двигатель, который сам бы брал поршни и вставлял их в цилиндры, а свечи зажигания в нем сами находили бы свои гнезда и ввинчивались в них. Но маловероятно, чтобы такая сложная машина была такой же надежной и работоспособной, как обычный двигатель. Вместо этого мы считаем нормальным, чтобы двигатель из частей собирал кто-то посторонний — это может быть автомеханик-человек или промышленный робот. Как робот, так и человек устроены значительно сложнее, чем двигатель, который они собирают. Работа по сборке из деталей не имеет ничего общего с работой по вращению вала. Сборщики работают только в процессе создания автомобиля — когда вы ведете его по дороге, вам не нужен ни механик, ни робот. Не бывает так, чтобы автомобиль собирали и вели одновременно, поэтому нет нужды в механизме, который бы объединял в себе обе эти функции. Аналогично, сборка сложного составного объекта — это такая работа, которую лучше отделить от обязанностей, которые объект будет выполнять впоследствии, в ходе своего существования.

Но перекладывание ответственности на другую заинтересованную сторону, объект-клиент, приводит к еще худшим последствиям. Клиент знает, какую работу нужно сде-

лать, и поручает необходимые вычислительные операции объектам предметной области. Если от клиента требуется самому собирать те объекты модели, которые для этого нужны, то он должен знать достаточно об их внутреннем устройстве. А чтобы соблюсти все инварианты, касающиеся взаимоотношений между отдельными частями объекта модели, клиент должен знать еще и некоторые внутренние правила (деловые регламенты) этого объекта. Даже простой вызов конструктора привязывает объект-клиент к конкретным классам объекта, который он создает. Никакие изменения в реализацию объектов модели теперь невозможно внести, не изменяя и объект-клиент, отчего становится труднее выполнять рефакторинг.

Когда создание объекта модели перекладывается на объект-клиент, это приводит к лишним сложностям и делает менее четким распределение обязанностей. К тому же, образуется брешь в инкапсуляции создаваемых объектов прикладной модели и АГРЕГАТОВ. Что еще хуже, если клиент находится на операционном уровне, то происходит "утечка" части обязанностей с уровня предметной области. Тесная привязка операционного уровня к подробностям реализации модели сводит на нет большинство преимуществ абстрагирования на уровне предметной области и сильно усложняет внесение дальнейших изменений.

Создание объекта само по себе может быть очень важной операцией. Но сборка объекта — это совершенно не та работа, которую потом придется делать этому же объекту в ходе своего существования. Объединение этих обязанностей в одном объекте порождает неуклюжие, трудные для понимания программные конструкции. Если дать клиенту возможность управлять созданием объектов, это искажает архитектуру самого клиента, нарушает инкапсуляцию собранного объекта или АГРЕГАТА**, а также слишком сильно привязывает клиента к конкретной реализации создаваемого им объекта.**

Создание сложных объектов — это обязанность уровня предметной области, но не объектов, которые непосредственно выражают модель. Бывают случаи, когда создание и сборка объекта соответствуют существенному событию в предметной области — например, "открытию банковского счета". Но сами по себе операции создания и сборки объекта обычно не имеют смысла в модели; это уже вопрос программной реализации. Чтобы решить эту проблему, приходится добавлять в модель конструкции, не являющиеся ни сущностями (ENTITIES), ни объектами-значениями (VALUE OBJECTS), ни службами (SERVICES). Здесь мы отходим от принципов предыдущей главы, поэтому важно подчеркнуть еще раз: мы добавляем в модель элементы, которые ничему в ней непосредственно не соответствуют, но, тем не менее, выполняют обязанности, относящиеся к уровню модели.

В любом объектно-ориентированном языке есть механизм создания объектов (например, конструкторы в Java и C++, класс создания экземпляров в Smalltalk). Но существует потребность и в более абстрактном механизме создания объектов, который не зависел бы от других объектов. Элемент программы, обязанность которого — создавать объекты, называется ФАБРИКОЙ (FACTORY).

Рис. 6.12. Работа с ФАБРИКОЙ

Так же, как интерфейс объекта должен инкапсулировать его реализацию, т.е. давать возможность использовать операции объекта, не зная, как они устроены, ФАБРИКА инкапсулирует знания, необходимые для создания сложного объекта или АГРЕГАТА. Она предоставляет клиенту интерфейс, который соответствует его потребностям, и абстрактное представление созданного объекта.

Передайте обязанности по созданию экземпляров сложных объектов и АГРЕГАТОВ **отдельному объекту, который сам по себе может не выполнять никаких функций в модели предметной области, но, тем не менее, является элементом ее архитектуры. Обеспечьте интерфейс, который бы инкапсулировал все сложные операции сборки объекта и не требовал от клиента ссылаться на конкретные классы создаваемого объекта. Создавайте** АГРЕГАТЫ **как единое целое, контролируя выполнение инвариантов.**

* * *

ФАБРИКИ можно проектировать по-разному. В [14] подробно разобрано несколько специальных архитектурных шаблонов для создания объектов — FACTORY METHOD (МЕТОД-ФАБРИКА), ABSTRACT FACTORY (АБСТРАКТНАЯ ФАБРИКА) и BUILDER (КОМПОНОВЩИК). Изложение в упомянутой книге касается в основном самых трудных вопросов создания объектов. Наша же цель — не углубиться в подробности проектирования ФАБРИК, а показать, что они занимают важное место в архитектуре предметной области. Правильное пользование ФАБРИКАМИ — залог успешного ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN).

Любая хорошая фабрика должна отвечать двум фундаментальным требованиям.

1. Каждый метод создания объекта должен быть един и неделим; он должен гарантировать соблюдение всех инвариантов создаваемого объекта или АГРЕГАТА. ФАБРИКА должна уметь создавать только объект целиком в корректном состоянии. Для объекта-СУЩНОСТИ это означает создание сразу целого АГРЕГАТА с соблюдением всех инвариантов; только необязательные второстепенные элементы разрешается добавить позже. Для неизменяемого ОБЪЕКТА-ЗНАЧЕНИЯ это означает, что все атрибуты инициализируются окончательными корректными значениями. Если интерфейс позволяет клиенту запросить создание некорректного объекта, то должна инициироваться исключительная ситуация или запуститься какой-то другой механизм, не позволяющий возвратить некорректное значение.
2. Абстрагировать ФАБРИКУ следует к желаемому типу, а не к конкретному классу. В этом могут помочь оригинальные шаблоны фабрик, предлагаемые в [14].

Выбор фабрик и их местонахождения

Грубо говоря, ФАБРИКА вводится для того, чтобы создавать некие объекты, скрывая подробности, и помещается туда, где ею будет удобно пользоваться. Принимаемые по этому поводу решения обычно касаются тех или иных АГРЕГАТОВ.

Например, если есть потребность добавлять элементы внутрь уже существующего АГРЕГАТА, можно создать МЕТОД-ФАБРИКУ в корневом объекте агрегата. Реализация внутренней структуры АГРЕГАТА таким образом скрывается от всех внешних клиентов, а поддержание целостности АГРЕГАТА при добавлении в него элементов поручается корневому объекту, как показано далее на рис. 6.13.

А вот еще один пример. МЕТОД-ФАБРИКУ можно поместить в объект, который непосредственно участвует в порождении другого объекта, хотя и не владеет им после создания. В том случае, когда при создании объекта преимущественно используются данные и, возможно, деловые регламенты другого объекта, такой подход позволяет сэкономить

на операции по извлечению информации из порождающего объекта с целью послать ее куда-то для создания объекта. Также при этом отражаются особые отношения между порождающим объектом и его продуктом.

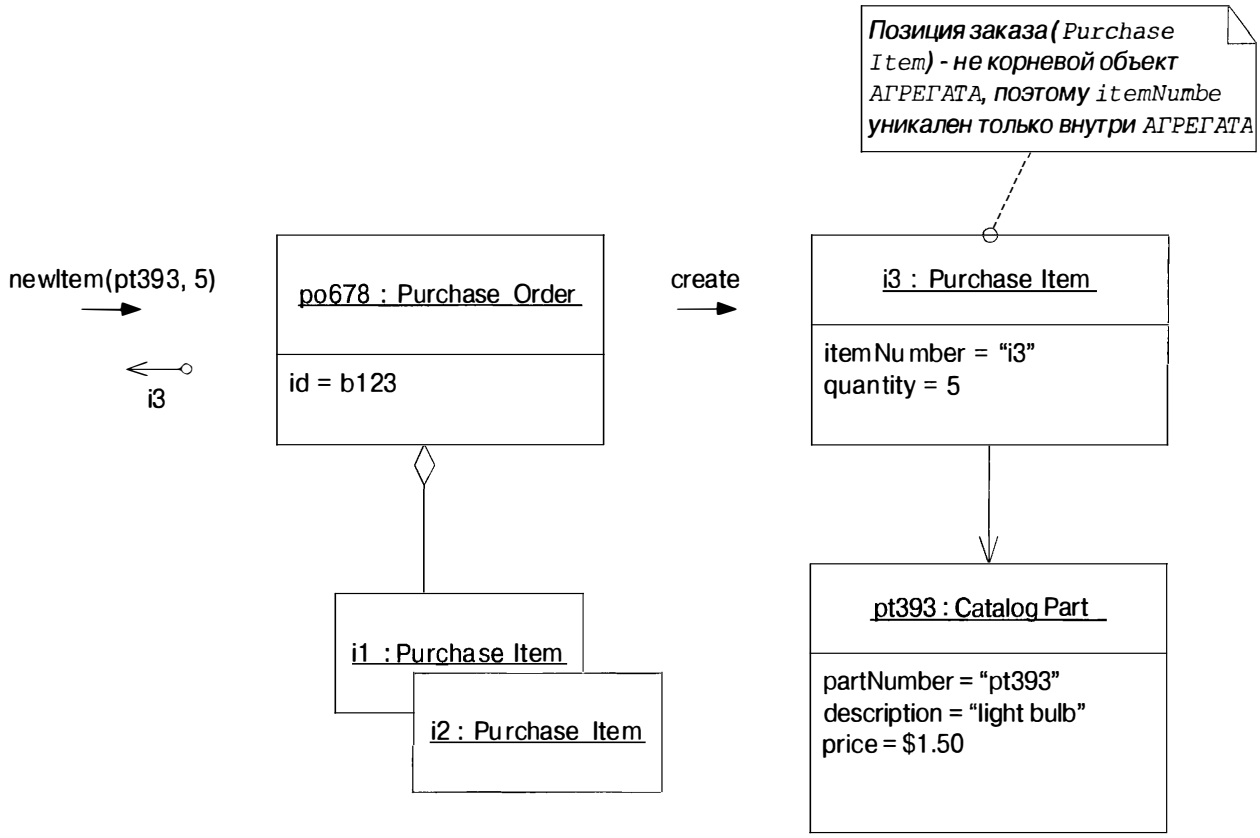


Рис. 6.13. МЕТОД-ФАБРИКА инкапсулирует расширение АГРЕГАТА

На рис. 6.14 объект **Торговая заявка** (**Trade Order**) не входит в тот же агрегат, что и **Брокерский счет** (**Brokerage Account**), потому что дальше он будет взаимодействовать с программой выполнения заявки, где **Брокерский счет** будет только мешать. Но даже с учетом этого кажется естественным дать **Брокерскому счету** право контроля над созданием **Торговых заявок**. **Брокерский счет** содержит информацию, которая должна помещаться в **Торговую заявку** (начиная с ее собственных идентификационных данных), а также правила, которые определяют, какие заявки являются правильными и разрешенными. Полезно было бы также скрыть реализацию **Торговой заявки**. Например, она может подвергнуться рефакторингу в иерархическую структуру, с отдельными подклассами для **Заявки на продажу** (**Buy Order**) и **Заявки на покупку** (**Sell Order**). Наличие же ФАБРИКИ избавляет клиента от привязки к конкретным классам.

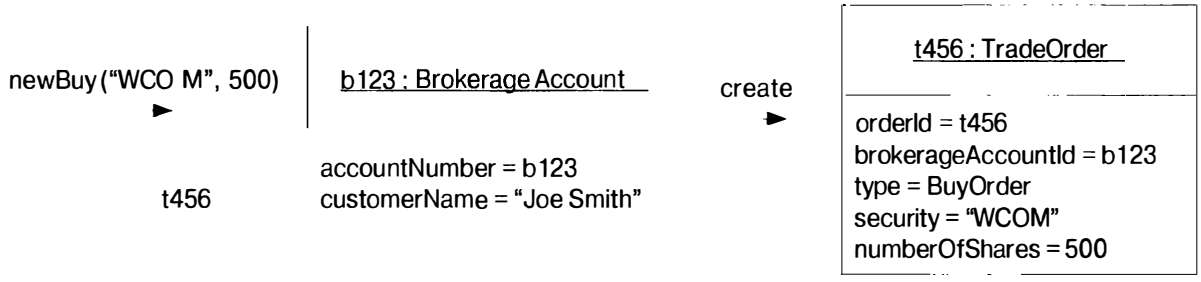


Рис. 6.14. МЕТОД-ФАБРИКА порождает СУЩНОСТЬ, не входящую в тот же АГРЕГАТ

ФАБРИКА очень тесно связана со своими "изделиями", поэтому и помещать ее нужно только в объект, который имеет сильную естественную связь с порождаемым объектом. Если нужно скрыть что-то — то ли конкретную программную реализацию, то ли просто сложность процесса создания объекта, — но не находится естественного объекта для инкапсуляции, нужно создать специальную ФАБРИКУ или СЛУЖБУ. Автономная ФАБРИКА обычно порождает сразу целый АГРЕГАТ, выдавая ссылку на его корневой объект и контролируя соблюдение инвариантов созданного АГРЕГАТА. Если нужна ФАБРИКА для объекта, внутреннего по отношению к некоторому агрегату, а корневой объект агрегата по каким-то причинам не подходит, смело создавайте для него отдельную ФАБРИКУ. Однако при этом надо уважать правила доступа внутри АГРЕГАТА и следить, чтобы на порождаемый объект были возможны только временные ссылки извне этого АГРЕГАТА.

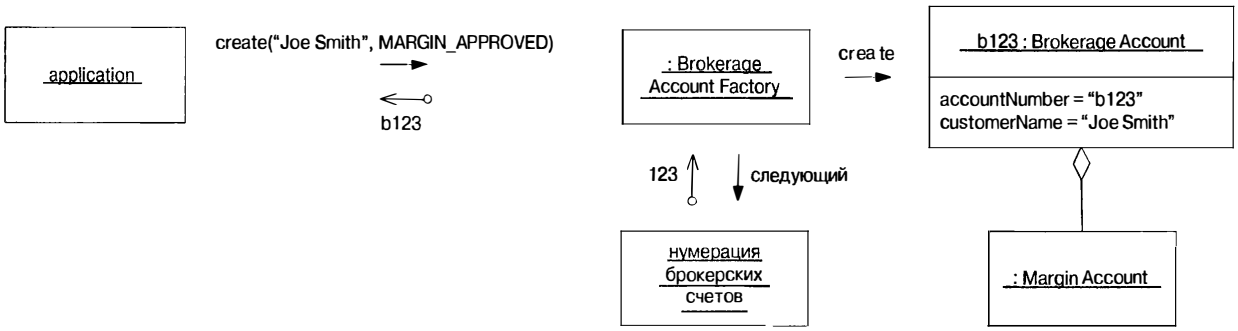


Рис. 6.15. Построение АГРЕГАТА автономной ФАБРИКОЙ

Когда достаточно конструктора

Я видел чересчур много программ, в которых *все* экземпляры объектов создавались прямыми вызовами конструкторов классов или подобного же первичного механизма порождения объектов, имевшегося в языке программирования. Применение ФАБРИК имеет большие преимущества, но, в целом, используется реже, чем хотелось бы. И все же время от времени приходится делать выбор в пользу конструктора именно за его прямоту, потому что ФАБРИКИ могут усложнить работу с простыми объектами без полиморфизма.

Взвешивая "за" и "против", обычному общедоступному (*public*) конструктору нужно отдать предпочтение в следующих обстоятельствах.

- Класс является типом. Он не входит ни в какую интересную иерархию и не используется полиморфически для реализации интерфейса.
- Клиенту нужно знать реализацию объекта — возможно, с точки зрения выбора СТРАТЕГИИ.
- Все атрибуты объекта доступны клиенту, так что в конструкторе, предоставляемом клиенту, не создаются никакие новые объекты.
- Создание объекта не является сложным процессом.
- Общедоступный конструктор должен следовать тем же правилам, что и ФАБРИКА: это должна быть единая, неделимая операция, которая подходит для всех инвариантов создаваемого объекта.

Избегайте вызывать конструкторы в конструкторах других классов. Конструкторы должны быть проще простого. Сложную сборку, особенно АГРЕГАТОВ, поручите ФАБРИКАМ. И порог, за которым нужно вместо конструктора выбирать небольшой МЕТОД-ФАБРИКУ, совсем не высок.

В библиотеке классов Java имеются интересные примеры. Все коллекции реализуют интерфейсы, которые отделяют клиента от деталей реализации. И все же они создаются прямыми вызовами конструкторов. А ФАБРИКА могла бы инкапсулировать иерархию коллекций. Методы ФАБРИКИ могли бы позволить клиенту запрашивать нужные ему свойства и возможности, а ФАБРИКА выбирала бы, экземпляр какого класса создать. Тогда код для создания коллекций стал бы более выразительным, а новые классы-коллекции можно было бы устанавливать, не нанося удар по всем программам на Java.

Но есть и случай, говорящий в пользу конкретных конструкторов. Во-первых, выбор реализации может оказаться слишком затратным для многих программ, так что эти программы могут предпочесть делать его самостоятельно. (Хорошо построенная ФАБРИКА, тем не менее, могла бы и учесть такие факторы.) В конце концов, классов-коллекций существует не очень много, и выбор сделать не так уж трудно.

Абстрактные типы коллекций сохраняют некоторую ценность, несмотря на отсутствие ФАБРИК, по причине особенностей их использования. Очень часто коллекции создаются в одном месте, а используются в другом. Это означает, что клиент, который в конце концов пользуется коллекцией, — добавляет, удаляет и извлекает содержимое — имеет право общаться только с интерфейсом и не зависеть от реализации. Обязанность выбора класса коллекции обычно падает на объект, владеющий коллекцией, или на ФАБРИКУ владеющего объекта.

Проектирование интерфейса

Разрабатывая декларативный заголовок ФАБРИКИ (неважно, автономной или ФАБРИКИ-МЕТОДА), не упускайте из виду два принципа.

- *Каждая операция должна быть единой и неделимой.* Все данные, которые необходимы для создания готового продукта-объекта, нужно передать за одну коммуникационную операцию с ФАБРИКОЙ. Придется также решить, что делать, если создание объекта сорвется, — например, в случае несоблюдения каких-то его инвариантов. Можно инициировать исключительную ситуацию или возвратить нулевое значение. Будьте последовательны и примите единый стандарт программирования для работы с ошибками создания объектов в ФАБРИКАХ.
- *Фабрика должна быть связана со своими аргументами.* Если неосторожно выбрать набор входных параметров, можно создать целую паутину взаимосвязей. На степень зависимости влияют операции, которые выполняются над аргументом. Если он просто вставляется в создаваемый объект, эта зависимость невелика. Но если при конструировании объекта из аргумента берутся фрагменты, зависимость становится сильнее.

Самые безопасные параметры — это те, которые поступают с нижних уровней архитектуры. Даже в пределах одного уровня существует тенденция разделения на подуровни: более элементарные объекты используются более сложными. (Это деление на уровни будет рассматриваться с разных сторон в главе 10, а затем снова в главе 16.)

Еще один хороший вариант параметра — это объект, который в модели является близкородственным генерируемому, так что между ними не создается новая взаимосвязь-зависимость. В предыдущем примере с объектом **Позиция товарного заказа (Purchase Order Item)** метод-фабрика принимает в качестве аргумента объект **Товар из каталога (Catalog Part)**, который имеет естественную ассоциацию с **Позицией (Item)**. Возникает прямая взаимосвязь между классом **Товарный заказ (Purchase Order)** и **Товаром (Part)**. Но эти три объекта и так образуют замкнутую концептуальную группу. АГРЕГАТ **Товарного заказа** уже и так ссылался на **Товар**. Поэтому передача управления корневому объекту АГРЕГАТА и инкапсуляция внутренней структуры АГРЕГАТА — это хороший выбор в данном случае.

Используйте абстрактный тип аргументов, а не конкретные их классы. ФАБРИКА привязана к конкретному классу продуцируемых объектов; нет нужды привязывать ее еще и к конкретным параметрам.

Где реализовать логику инвариантов?

Проследить, чтобы в создаваемом объекте или АГРЕГАТЕ выполнялись все инварианты — эта задача возложена на ФАБРИКУ. И все же нужно хорошенько подумать, прежде чем выносить свойственные объекту регламентные правила за его пределы. ФАБРИКА может делегировать проверку инвариантов самому своему продукту, и чаще всего лучше так и делать.

Но у фабрик устанавливаются особые отношения с создаваемыми ими объектами. Они уже знают внутреннюю структуру объектов, и сам смысл их существования в том и состоит, чтобы реализовать свой продукт. В некоторых обстоятельствах лучше как раз вынести логику инвариантов в ФАБРИКУ, не загромождая создаваемые объекты. Особенно удобно это делать с регламентными правилами АГРЕГАТОВ (которые распространяются на много объектов). А *неудобно* это делать с МЕТОДАМИ-ФАБРИКАМИ, которые включены в другие объекты модели.

Хотя в принципе инварианты проверяются в конце любой операции, иногда разрешенные над объектом преобразования просто не позволяют сделать такую проверку. Например, может существовать правило, регулирующее присвоение идентификационных данных объекту-СУЩНОСТИ. Но после создания объекта его данные могут оказаться неизменяемыми. Так, ОБЪЕКТЫ-ЗНАЧЕНИЯ целиком и полностью неизменяемы. Незачем объекту тащить на себе реализацию логики, которая никогда не будет применяться в течение его активного существования. В таких случаях инварианты лучше реализовать в ФАБРИКЕ, не усложняя ее продукт.

Отличия между фабриками сущностей и фабриками объектов-значений

ФАБРИКИ СУЩНОСТЕЙ и ФАБРИКИ ОБЪЕКТОВ-ЗНАЧЕНИЙ отличаются друг от друга двумя особенностями. ОБЪЕКТЫ-ЗНАЧЕНИЯ неизменяемы; продукт создается в окончательном виде. Поэтому операции ФАБРИКИ должны давать полное описание продукта. ФАБРИКИ СУЩНОСТЕЙ же склонны работать только с самыми существенными атрибутами, необходимыми для создания корректного АГРЕГАТА. Детали можно добавить и позже, если они не требуются немедленно для соблюдения инварианта.

Есть еще проблемы, связанные с присвоением идентификационных данных СУЩНОСТИ — помимо ОБЪЕКТОВ-ЗНАЧЕНИЙ. Как говорилось в главе 5, идентификатор может назначаться программой автоматически или предоставляться снаружи обычно пользователем. Если индивидуальность покупателя контролируется по номеру телефона, этот номер, очевидно, должен передаваться в виде аргумента в ФАБРИКУ. Соответственно, контроль над процессом присвоения идентификатора программой удобно возложить на ФАБРИКУ. Хотя фактическое генерирование уникального идентификационного номера обычно выполняется процедурой базы данных или другим инфраструктурным механизмом, ФАБРИКА знает, что именно запрашивать и куда это поместить.

Восстановление хранимых объектов

В предыдущих разделах ФАБРИКА играла свою роль только в самом начале цикла существования объекта. В какой-то момент большинство объектов либо попадают в базу данных, либо передаются по сети. Притом очень немногие технологии баз данных сохраняют объектный характер своего содержимого — в большинстве способов передачи и

хранения данных объект приводится к более ограниченному представлению. Поэтому восстановление объекта в памяти — это сложный процесс сборки отдельных частей заново в единое целое.

ФАБРИКА, используемая для восстановления объекта, очень похожа на ФАБРИКУ для его создания, если не считать двух основных отличий.

1. *Фабрика, восстанавливающая объект-сущность, не присваивает ему новый идентификационный номер.* Если бы она это делала, то предыдущее воплощение объекта потерялось бы. Поэтому идентификационные атрибуты должны содержаться во входных параметрах ФАБРИКИ, занимающейся восстановлением объекта.
2. *Фабрика, восстанавливающая объект, по-другому обрабатывает нарушение инварианта.* В ходе создания нового объекта фабрика просто сбрасывает его, если не удовлетворяется инвариант, но при восстановлении требуется более гибкий подход. Если объект уже существует где-то в системе (например, в базе данных), этот факт нельзя просто проигнорировать. Но нельзя игнорировать и нарушение регламентов. Должна существовать какая-то схема для разрешения таких противоречий, отчего восстановление становится более сложной задачей, чем создание новых объектов.

На рис. 6.16 и 6.17 показаны два способа восстановления. Некоторые из требующихся для этого операций удобно реализованы в технологиях отображения объектов (*object mapping*), если речь идет о восстановлении из базы данных. Если же требуется более сложное восстановление из другого источника, то лучше воспользоваться специально предназначенной для этого ФАБРИКОЙ.

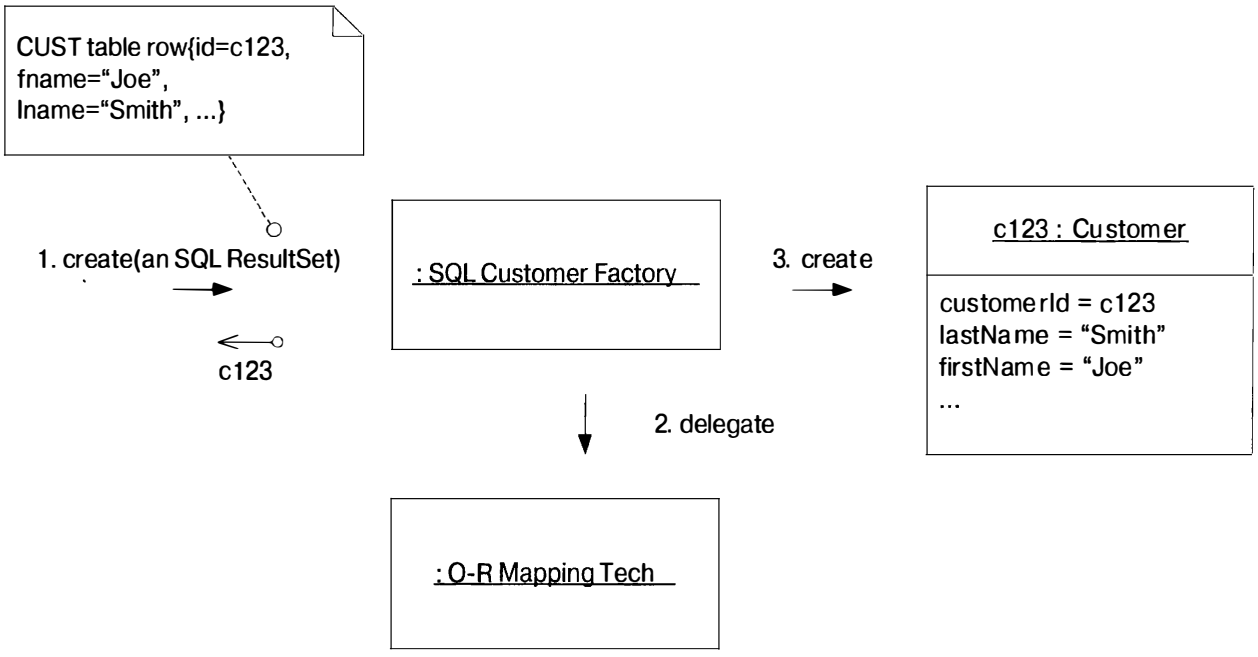


Рис. 6.16. Восстановление объекта-СУЩНОСТИ, извлеченного из реляционной базы данных

Подведем итоги. Для создания экземпляров объектов следует тщательно подобрать функциональные модули программы и явно определить их область действия. Это могут быть просто конструкторы, но часто возникает и потребность в более абстрактных или сложных механизмах создания экземпляров. Так в архитектуре программы появляется новый конструктивный элемент под названием ФАБРИКА (FACTORY). Обычно ФАБРИКИ явно не выражают никакую функциональную часть модели, но, тем не менее, входят в ее архитектуру как элементы, обеспечивающие четкую работу непосредственных смысловых объектов.

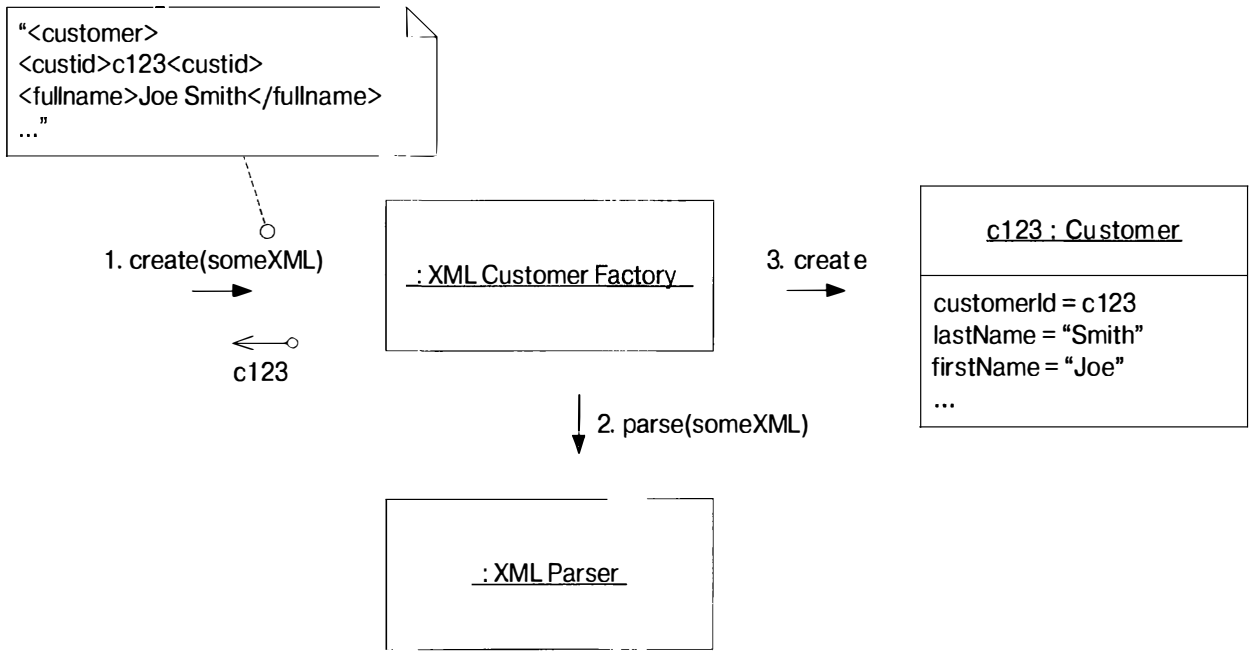
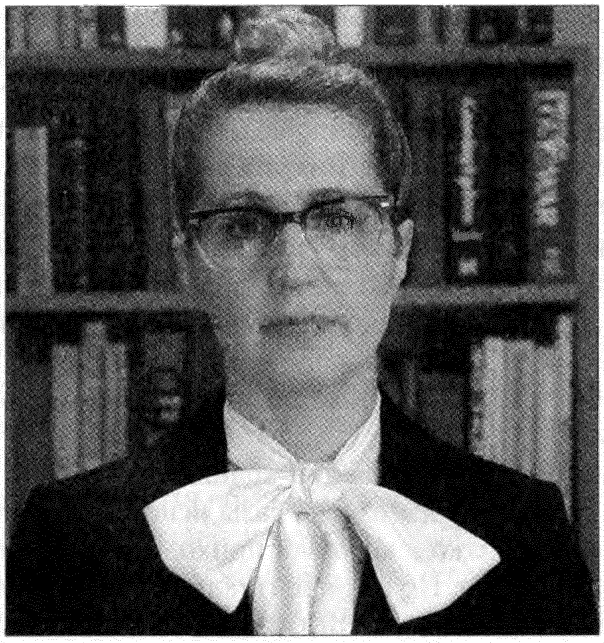


Рис. 6.17. Восстановление объекта-СУЩНОСТИ, переданного в формате XML

Фабрика инкапсулирует те преобразования в цикле существования объектов, которые связаны с их созданием и восстановлением. Но есть еще и преобразование, представляющее технические трудности и нетривиальное в архитектурной реализации — это сдача объектов на хранение и получение их оттуда. Это преобразование входит в обязанности еще одного конструктивного элемента модели — ХРАНИЛИЩА (REPOSITORY).

Хранилища



Благодаря наличию ассоциаций можно найти один объект по его взаимосвязям с другими объектами. Но для этого нужно иметь отправную точку, отталкиваясь от которой можно проследить СУЩНОСТЬ или ОБЪЕКТ-ЗНАЧЕНИЕ в середине их цикла существования.

* * *

Чтобы делать с объектом что бы то ни было, нужно иметь ссылку на него. Как получить эту ссылку? Один из способов — создать объект, поскольку при создании объекта возвращается ссылка на него. Второй способ — проследить ассоциацию. Начинаем с объекта, который нам уже известен, и запрашиваем у него информацию о связанном с ним объекте. Во всякой объектно-ориентированной программе это происходит постоянно. Именно в таких взаимосвязях заключается основная выразительность объектных моделей. Но нужно где-то взять тот самый отправной объект, с которого все начинается.

Мне однажды попался проект, в котором разработчики, будучи энтузиастами ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ, пытались реализовать *любые* обращения к объектам через создание или прослеживание ассоциаций! Их объекты находились в объектной базе данных, и они рассудили, что существующие концептуальные взаимоотношения между ними автоматически обеспечат нужные ассоциации. Достаточно, дескать, только провести подробный анализ и сделать всю модель предметной области связной. Это добровольно наложенное на себя условие привело разработчиков к созданию как раз такого бесконечного переплетения связей, которого мы пытаемся избежать на протяжении вот уже нескольких глав, тщательно реализуя СУЩНОСТИ и подбирая АГРЕГАТЫ. Эта стратегия долго не продержалась, но разработчикам не удалось заменить ее каким-нибудь другим внятным и строгим подходом. В результате они нагромоздили сиюминутных решений и понизили планку своих амбиций.

Подобный подход мало кому пришел бы в голову, не говоря уже о том, чтобы взяться за его реализацию, поскольку в большинстве проектов информация хранится в реляционных базах данных. Такая технология хранения делает естественным третий способ получения ссылки на объект: для поиска объекта в базе выполняется запрос к ней по его атрибутам или же отыскиваются отдельные составляющие объекта, после чего он восстанавливается.

Поиск по базе данных глобально доступен и позволяет непосредственно перейти к любому объекту. Нет никакой нужды в том, чтобы все объекты были взаимосвязаны — достаточно урезать сеть взаимосвязей до приемлемого, управляемого состояния. Выбор между отслеживанием ассоциаций и поиском по базе становится рядовым проектным решением, в котором приходится балансировать между независимостью поиска и связностью ассоциаций. Должен ли объект **Покупатель** (**Customer**) хранить коллекцию всех своих **Заказов** (**Orders**)? Или же **Заказы** следует разыскивать в базе данных, используя поиск по полю **Идентификатор покупателя** (**Customer ID**)? Проектируя объекты, нужно выбирать разумное сочетание поиска и отслеживания ассоциаций.

К сожалению, обычно у разработчиков не доходят руки до таких тонкостей проектирования — им хотя бы разобраться в том море механизмов, которое необходимо для того, чтобы провернуть трюк с сохранением объекта и извлечением его обратно, а затем и удалением из места хранения.

С технической точки зрения извлечение хранимого объекта это частный случай операции его создания, поскольку данные из базы используются фактически для сборки нового объекта. Действительно, код, который приходится для этого писать, не дает забыть об этой суровой реальности. Но с концептуальной точки зрения это всего лишь *середина* цикла существования объекта-СУЩНОСТИ. Ведь объект **Покупатель** (**Customer**) не стал представлять нового покупателя только потому, что объект положили в базу данных, а затем достали из нее. Чтобы не забывать об этом различии, мы и говорим о создании нового экземпляра на основе сохраненных данных как о *восстановлении* объекта.

Цель DDD состоит в том, чтобы научиться писать более качественные программы, сосредоточившись на модели предметной области, а не на программных технологиях. Пока

разработчик построит запрос SQL, передаст его в службу обработки запросов на инфраструктурном уровне, получит набор строк из таблицы базы данных, извлечет из них нужную информацию и передаст ее в конструктор или ФАБРИКУ, от его концентрации на модели ничего не останется. Так вырабатывается способ мышления об объектах всего лишь как о контейнерах данных, извлекаемых по запросам, и вся архитектура программы начинает ориентироваться на задачи обработки данных. Технические детали могут различаться, но главная проблема остается: клиентская часть взаимодействует с технологиями, а не с понятиями модели. Здесь могут оказаться полезными такие инфраструктуры, как УРОВНИ ОТОБРАЖЕНИЯ МЕТАДАННЫХ (METADATA MAPPING LAYERS) [13]. С их помощью облегчается преобразование результатов запросов в объекты, но и в этом случае программист по-прежнему думает о технических механизмах, а не о предметной области. Что еще хуже, если клиентский код напрямую обращается к базе данных, у программистов возникает искушение вообще выбросить такие конструкции модели, как АГРЕГАТЫ, или даже инкапсуляцию объектов, и заняться прямой обработкой извлекаемых данных. Регламентные правила предметной области все больше перетекают в код запросов к базе данных, а то и попросту отбрасываются. Объектные базы данных, конечно, снимают проблему преобразования, но механизмы поиска-то все равно носят механистический характер, и разработчиков продолжают соблазнять перспектива "вытаскивать" из базы любые объекты по своему усмотрению.

Клиентское приложение нуждается в удобных средствах получения ссылок на существующие объекты предметной области. Если инфраструктура позволяет это делать относительно легко, разработчики клиента добавляют в модель больше прослеживаемых ассоциаций, загромождая ее. С другой стороны, они могут с помощью запросов извлекать из базы данных в точности ту информацию, которая им нужна — например, достать несколько конкретных подобъектов, не обращаясь к корневому объекту АГРЕГАТА**. Таким образом операционная логика предметной области переносится в запросы и клиентский код, а роль** СУЩНОСТЕЙ **и** ОБЪЕКТОВ-ЗНАЧЕНИЙ **сводится к простым контейнерам данных. Техническая сложность реализации всей инфраструктуры доступа к базе данных быстро загромождает код клиента, вынуждая разработчиков урезать и упрощать уровень предметной области. В итоге модель становится бесполезной.**

Если придерживаться изученных нами до сих пор принципов проектирования, можно попробовать несколько сузить рамки проблемы доступа к объектам. Вот бы иметь такой метод доступа, который позволил бы не отступить от этих принципов и сохранить именно модель в центре внимания при разработке программы. Для начала не следует беспокоиться о временных объектах. Такие объекты (обычно ОБЪЕКТЫ-ЗНАЧЕНИЯ) проживают короткую жизнь — они используются в операциях создавших их клиентов, а затем сбрасываются. Что касается постоянных объектов, то для работы с ними не надо использовать запросы к базе данных, если их удобнее находить по цепочке ассоциаций. Например, адрес человека можно узнать из объекта **`Человек (Person)`**. И что самое важное, *обращение к любому объекту, внутреннему по отношению к* АГРЕГАТУ*, может выполняться только через корневой объект агрегата.*

Постоянные ОБЪЕКТЫ-ЗНАЧЕНИЯ обычно находятся путем отслеживания ассоциаций от какой-нибудь сущности, служащей корневым объектом для инкапсулирующего их АГРЕГАТА. Фактически доступ к ЗНАЧЕНИЮ через глобальный поиск часто не имеет смысла, поскольку нахождение ОБЪЕКТА-ЗНАЧЕНИЯ по его свойствам эквивалентно созданию нового экземпляра с теми же свойствами.

Впрочем, бывают и исключения. Например, если я планирую поездку с помощью транспортной онлайн-системы, то иногда сохраняю несколько потенциальных маршрутов, а позже возвращаюсь и выбираю один из них для резервирования билетов. Эти маршруты представляют собой ЗНАЧЕНИЯ (если бы существовало два маршрута, состоящих

из одних и тех же рейсов, то разницы между ними не было бы), но они ассоциированы с моим именем, и поэтому извлекаются в том же виде, в котором сформированы. Еще один случай — это перечислимые типы, область значений которых состоит из ограниченного количества заранее определенных величин или символов.

Надо сказать, что глобальный доступ к ОБЪЕКТАМ-ЗНАЧЕНИЯМ распространен значительно меньше, чем к сущностям. Так что если у вас возникает необходимость искать в базе данных существующий ОБЪЕКТ-ЗНАЧЕНИЕ, стоит подумать над тем, не представляет ли он на самом деле СУЩНОСТЬ, индивидуальность которой вы еще просто не осознали.

Из вышеизложенного должно быть ясно, что большинство объектов не требует глобального поиска для обращения к себе. А те, которые требуют, нужно соответствующим образом представить в архитектуре программы.

Теперь проблему можно сформулировать более точно.

Необходимо, чтобы какая-то часть постоянно существующих объектов была доступна через глобальный поиск по атрибутам. Доступ таким методом осуществляется к корневым объектам агрегатов, которые неудобно отслеживать по ассоциациям. Обычно это СУЩНОСТИ**, иногда** — ОБЪЕКТЫ-ЗНАЧЕНИЯ **со сложной внутренней структурой, а иногда значения из перечислимых типов. Предоставление такого доступа к другим объектам стирает основные различия между разновидностями объектов. Отсутствие ограничений на запросы к базе данных может нарушить инкапсуляцию объектов и** АГРЕГАТОВ **предметной области. Открытое использование технической инфраструктуры и механизмов доступа к базам данных усложняет работу клиентского приложения и знаменует отход от принципов** ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ**.**

Имеется множество приемов для решения технических проблем доступа к базам данных. Можно, например, инкапсулировать код SQL в ОБЪЕКТЫ-ЗАПРОСЫ (QUERY OBJECTS) или выполнять перевод из объектов в таблицы и назад через УРОВНИ ОТОБРАЖЕНИЯ МЕТАДАННЫХ [13]. Восстановление хранящихся объектов можно выполнять через ФАБРИКИ (об этом еще будет говориться). Эти и многие другие приемы дают возможность абстрагироваться от технических сложностей и тонкостей.

А теперь следует еще раз напомнить, что мы потеряли. Мы уже не думаем о понятиях нашей модели предметной области! Код уже не передает предметную суть выполняемых операций, а занимается пересылкой и обработкой данных. Шаблон ХРАНИЛИЩА (REPOSITORY) — это концептуально несложная архитектура, позволяющая инкапсулировать описанные выше технические решения и сконцентрировать внимание на прикладной модели.

ХРАНИЛИЩЕ представляет все объекты определенного типа в виде концептуального множества (обычно виртуального, эмулируемого). Оно работает аналогично коллекции, только с более развитым механизмом запросов. Можно добавлять и удалять объекты соответствующего типа, и скрытые механизмы ХРАНИЛИЩА будут автоматически помещать их в базу данных или удалять из нее. Введя это определение, получаем полный набор средств для доступа к корневым объектам АГРЕГАТОВ с самого начала и до конца их цикла существования.

Клиенты запрашивают объекты из ХРАНИЛИЩА, используя *запросные методы* (*query methods*), которые отбирают объекты по критериям, задаваемым клиентами, — обычно по значениям определенных атрибутов. ХРАНИЛИЩЕ выдает запрашиваемый объект, инкапсулируя механизмы запросов к базе данных и отображения метаданных. ХРАНИЛИЩА могут реализовать множество самых разных запросов, отбирая объекты в зависимости от установленных клиентами критериев. Они также могут возвращать информацию сводного характера, — например, сколько экземпляров объектов удовлетворяют тому или иному критерию. ХРАНИЛИЩЕ может даже выполнять вычисления по итогам запроса,

например, общую сумму или среднее каких-нибудь числовых атрибутов для объектов, полученных по этому запросу.

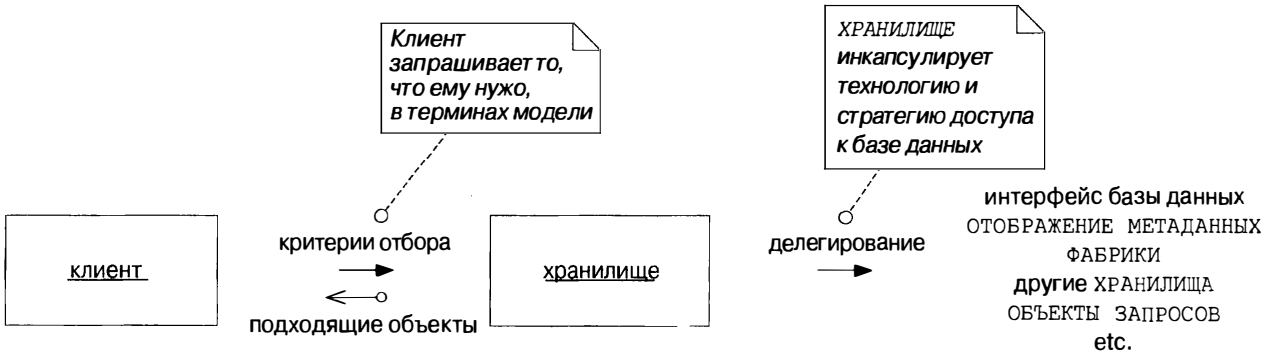


Рис. 6.18. ХРАНИЛИЩЕ, выполняющее поиск по запросу клиента

Наличие ХРАНИЛИЩА снимает с клиента огромную тяжесть — теперь он может общаться с простым, скрывающим технические подробности интерфейсом, и запрашивать нужную ему информацию в терминах модели. Для поддержки всего этого нужна развитая и сложная техническая инфраструктура, но сам интерфейс остается простым и концептуально привязанным к модели предметной области.

Для каждого типа объектов, к которым требуется глобальный доступ, введите объект-посредник, который может создать иллюзию, что все объекты такого типа объединены в коллекцию и находятся в оперативной памяти. Наладьте доступ через хорошо известный глобальный интерфейс. Реализуйте методы для добавления и удаления объектов, инкапсулирующие реальное помещение информации в базу данных и удаление ее оттуда. Реализуйте методы, которые будут выбирать объекты по заданным критериям и возвращать полностью сгенерированные и инициализированные объекты или коллекции объектов с атрибутами, подходящими под критерии, таким образом инкапсулируя реальные технологии хранения данных и выполнения запросов. Реализуйте ХРАНИЛИЩА **только для тех** АГРЕГАТОВ**, к корневым объектам которых требуется прямой доступ. Программа-клиент должна опираться на модель, а все операции хранения и обработки данных объектов должны быть переданы** ХРАНИЛИЩАМ**.**

* * *

У ХРАНИЛИЩ есть ряд важных преимуществ, ведь они:

- предоставляют клиентам простую модель для получения устойчиво существующих объектов и управления их жизненным циклом;
- убирают из операционной части приложения и модели предметной области необходимость в технической поддержке целостности объектов, разных вариантов технологий СУБД, и даже разных источников данных;
- выражают в себе проектные решения по способам доступа к объектам;
- позволяют легко заменить себя "заглушками" для целей тестирования (обычно заглушкой служит находящаяся в оперативной памяти коллекция).

Запросы к хранилищам

Все ХРАНИЛИЩА должны содержать методы, с помощью которых клиенты могут запрашивать объекты, соответствующие некоторым критериям. Но вот в организации такого интерфейса возможны варианты.

Самое простое в разработке ХРАНИЛИЩЕ содержит явно прописанные в коде запросы с конкретными параметрами. Предназначение запросов может быть самым разнообразным: извлечение СУЩНОСТИ по ее идентификационным данным (это реализовано практически в любом ХРАНИЛИЩЕ); получение коллекции объектов по значению какого-либо атрибута или сложной комбинации параметров; отбор объектов по диапазону значений атрибутов (например, по датам); и даже различные расчеты, не выходящие за пределы общей компетенции ХРАНИЛИЩ (в виде интерфейса к операциям используемой СУБД).

Большинство таких запросов возвращает объект или коллекцию объектов, но в рамки концепции вполне вписывается и возврат результатов различных сводно-статистических вычислений, — например, количества объектов или суммы значений тех числовых атрибутов, которые в модели задействованы именно в такой операции.

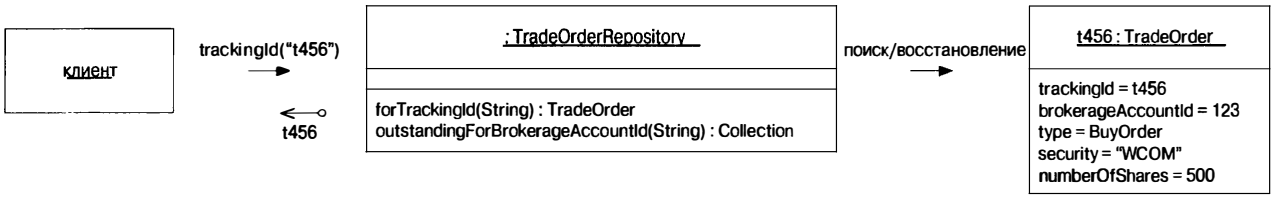


Рис. 6.19. Явно прописанные запросы в простейшем ХРАНИЛИЩЕ

Запросы, прописываемые явно, можно строить для любой инфраструктуры без особых затрат, потому что они делают всего-навсего то, что клиент все равно хочет от системы.

А вот в проектах с большим количеством запросов можно попытаться построить не просто ХРАНИЛИЩЕ, а целую архитектурную среду, ассоциированную с ним, в которой можно было бы составлять более гибкие запросы. Для этого потребуются кадры, знакомые с нужными технологиями, а также соответствующая техническая инфраструктура.

Одно из особенно удачных решений для обобщения ХРАНИЛИЩ путем создания архитектурной среды состоит в том, чтобы строить запросы на основе СПЕЦИФИКАЦИЙ (SPECIFICATION). СПЕЦИФИКАЦИЯ позволяет клиенту описывать (т.е. задавать, *специфицировать*), что именно ему нужно, и при этом не беспокоиться, как именно это делается. В процессе этого создается объект, который фактически и осуществляет нужный выбор. Этот архитектурный шаблон будет рассматриваться подробно в главе 9.

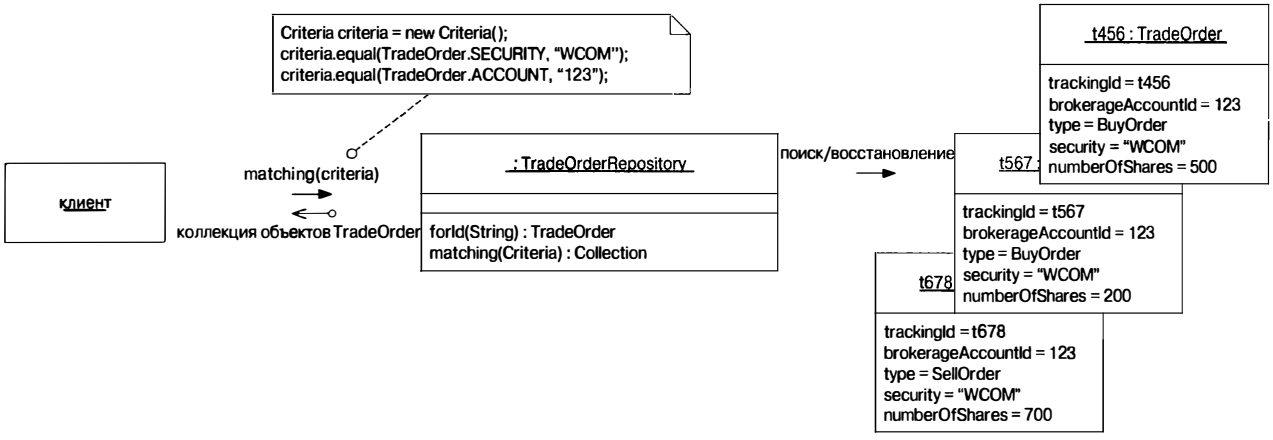


Рис. 6.20. Гибкая декларативная СПЕЦИФИКАЦИЯ критериев поиска в сложном ХРАНИЛИЩЕ с расширенными возможностями

Запросы на основе СПЕЦИФИКАЦИЙ составляются гибко и удобно. В зависимости от имеющейся инфраструктуры, архитектурная среда для хранилищ может быть или совсем простой, или непомерно усложненной. Проектирование таких хранилищ и связан-

ную с ним техническую проблематику подробнее рассматривают Роб Ми (Rob Mee) и Эдвард Хайет (Edward Hieatt) в [13].

Даже если хранилище спроектировано на выполнение гибких запросов, оно должно позволять также и добавлять специализированные, явно прописанные запросы. Это могут быть вспомогательные методы, инкапсулирующие часто используемые запросы, или такие, которые возвращают не сами объекты, а, например, результаты определенных математических операций с ними. Среды, которые не предусматривают такую возможность, в конце концов, либо "замутняют" чистоту архитектуры предметной области, либо просто игнорируются разработчиками.

Клиентам безразлична реализация хранилищ, а разработчикам — нет

Реализация технологии длительного хранения и поддержания целостности объектов дает возможность приложению-клиенту быть очень простым и совершенно независимым от конкретной реализации ХРАНИЛИЩА. Но инкапсуляция инкапсуляцией, а часто бывает, что разработчикам необходимо знать, что происходит там внутри, "в глубине". ХРАНИЛИЩА могут работать и использоваться очень по-разному, и вопросы быстродействия часто играют ключевую роль.

Кайл Браун (Kyle Brown) рассказал мне историю о том, как к нему обратились для решения проблем с системой управления производством на основе WebSphere, которую он как раз устанавливали в ее рабочей среде. Система загадочно съедала всю память после нескольких часов работы. Кайл просмотрел код и нашел причину. В какой-то момент в системе собиралась сводная информация по всем учитываемым объектам на предприятии. Разработчики реализовали это в виде запроса под названием "все объекты", который создавал и инициализировал экземпляры всех объектов, а потом отбирал то, что было нужно. Получалось так, как будто вся база данных одним махом оказывалась в памяти! При тестировании этой проблемы не было, потому что объем тестовых данных был невелик.

Здесь недосмотр очевиден, но серьезные проблемы могут возникнуть и из-за гораздо более мелких недочетов. Разработчикам необходимо знать технические характеристики операций, инкапсулированных в объектах, — пусть и не обязательно мельчайшие подробности их реализации. Если компонент хорошо спроектирован, ему можно дать такую характеристику. (Это одна из главных тем в главе 10.)

Как говорилось в главе 5, на выбор тех или иных решений в моделировании могут повлиять особенности и ограничения инфраструктурной технологии. Например, наличие реляционной базы данных накладывает практические ограничения на глубокие сложносоставные объектные структуры. В общем, разработчики должны иметь достаточный доступ к потоку информации с обоих уровней: как по вопросам использования ХРАНИЛИЩА, так и по технической реализации его запросов.

Реализация хранилища

Конкретные реализации ХРАНИЛИЩ могут сильно отличаться друг от друга в зависимости от имеющейся инфраструктуры и технологии контроля существования объектов. В идеале было бы хорошо спрятать от приложения-клиента (но не от разработчика этого клиента) все детали операций, чтобы код клиента оставался одним и тем же независимо от того, хранятся ли данные в объектной базе, реляционной базе или просто в оперативной памяти. ХРАНИЛИЩЕ же будет делегировать задания соответствующим службам инфраструктуры для выполнения порученной ему работы. Инкапсуляция механизмов

хранения данных, их извлечения и выполнения запросов — это самое основное в реализации ХРАНИЛИЩА.

Концепцию ХРАНИЛИЩА можно адаптировать ко многим ситуациям. Возможности ее реализации настолько разнообразны, что здесь будет приведено только несколько общих принципов, которые полезно помнить.

- *Абстрагируйте тип.* ХРАНИЛИЩЕ как бы "содержит в себе" все экземпляры определенного типа, но это не значит, что для каждого класса надо иметь свое хранилище. В качестве типа можно использовать абстрактный надкласс из иерархии. Например, **Товарная заявка (TradeOrder)** может быть как **Заявкой на покупку (BuyOrder)**, так и **Заявкой на продажу (SellOrder)**. Тип также может представлять собой интерфейс, реализаторы которого даже не связаны иерархически. Но это может быть и один конкретный класс. Не забывайте, что всегда можно встретить препятствия на пути реализации всего этого из-за отсутствия подобного полиморфизма в технологии СУБД.
- *Извлекайте преимущества из независимости от клиента.* Реализация ХРАНИЛИЩА предоставляет значительно больше свободы для внесения изменений, чем тот случай, когда клиент вызывает механизмы напрямую. Этим можно воспользоваться для оптимизации быстродействия, варьируя запросы или кэшируя объекты в памяти, по желанию меняя общую методику хранения и поддержания целостности объектов. Можно также облегчить тестирование клиентского кода и объектов модели предметной области, построив легко управляемую симуляцию ХРАНИЛИЩА с хранением объектов в оперативной памяти.

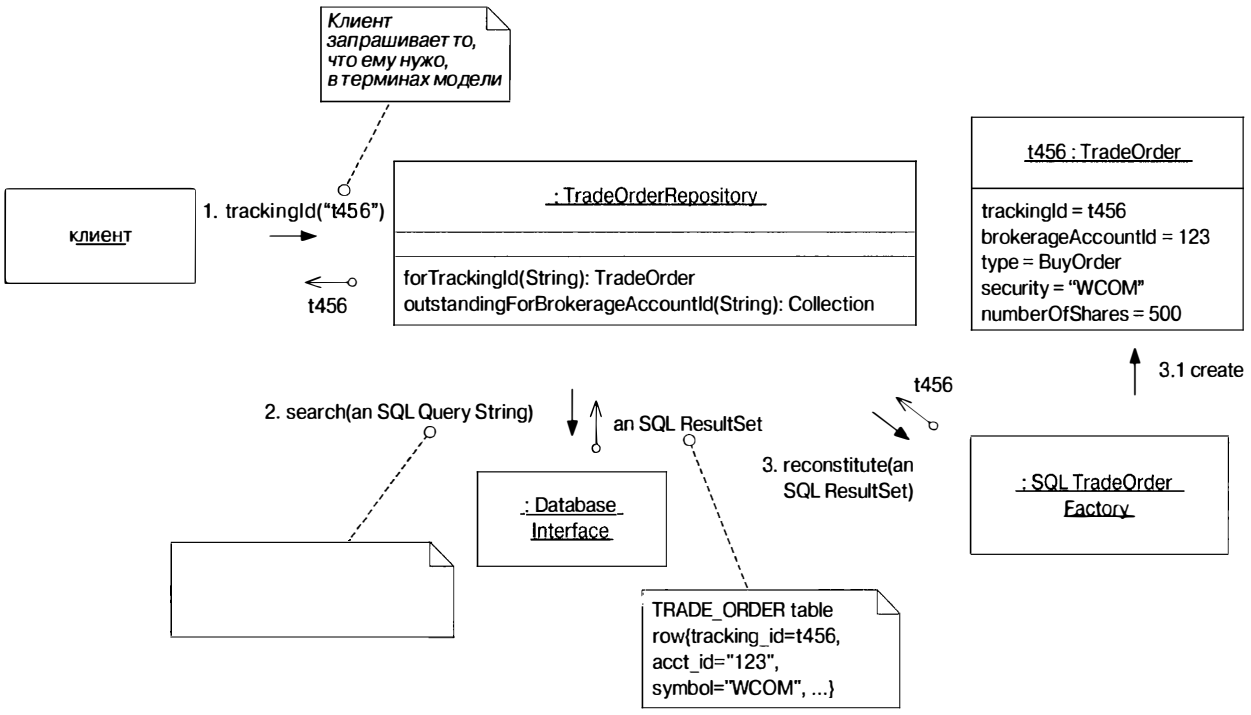


Рис. 6.21. Инкапсуляция реальной системы хранения данных в объекте-ХРАНИЛИЩЕ

- *Оставьте контроль транзакций клиенту.* Хотя именно ХРАНИЛИЩЕ помещает данные в базу и извлекает их оттуда, оно, как правило, не должно контролировать их завершение (т.е. выполнять фиксацию транзакции). Конечно, есть искушение, например, зафиксировать транзакцию после сохранения данных, но у клиента на-

верняка есть собственный контекст для корректной инициализации и завершения отдельных рабочих операций. Контроль транзакций со стороны клиента значительно облегчается, если ХРАНИЛИЩЕ в это дело не вмешивается.

Обычно разработчики размещают архитектурную среду для поддержки ХРАНИЛИЩ на инфраструктурном уровне. Кроме обеспечения связи с компонентами инфраструктуры, которые расположены ниже, надкласс ХРАНИЛИЩА может также реализовать некоторые важнейшие запросы, особенно когда имеется механизм гибкого их построения. К сожалению, в такой системе типов, как у Java, приходится типизировать возвращаемые объекты, как "Объекты", оставляя клиенту работу по приведению их к объявленному типу ХРАНИЛИЩА. Но это, конечно, приходится делать с запросами, которые в Java и так возвращают коллекции.

Некоторые дополнительные рекомендации по программированию хранилищ и технические шаблоны для их поддержки (такие, как "объект-запрос", *query object*) можно найти в [13].

Работа в рамках архитектурной среды

Прежде чем программировать что-нибудь вроде ХРАНИЛИЩА, необходимо хорошенько обдумать инфраструктуру, с которой приходится работать, — особенно архитектурную среду, если она есть. Можно обнаружить, что среда предоставляет такие возможности, с помощью которых легко построить ХРАНИЛИЩЕ, а бывает и так, что она только мешает, и чем дальше, тем больше. Может оказаться, что в архитектурной среде уже определены адекватные шаблоны для поддержания существования объектов, но иногда получается так, что готовые шаблоны вообще ничем не похожи на ХРАНИЛИЩА.

Пусть, например, проект строится на основе J2EE. Если поискать концептуальное сходство между этой средой и ПРОЕКТИРОВАНИЕМ ПО МОДЕЛИ (и при этом помнить, что объект Java Bean и объект-СУЩНОСТЬ — не одно и то же[2]), то можно, например, волевым решением назначить объекты Java Bean корневыми объектами АГРЕГАТОВ. Конструкция в архитектурной среде J2EE, обеспечивающая доступ к таким объектам, называется *EJB Home*. Попытка выдать ее за ХРАНИЛИЩЕ может вызвать и другие проблемы.

В целом, можно посоветовать "не плыть против течения" архитектурной среды. Пытайтесь придерживаться принципов DDD и при этом не отвлекаться на частности, когда среда работает против вас. Ищите сходство между концепциями предметно-ориентированного проектирования и принципами устройства среды, в которой работаете. Все это, конечно, справедливо в предположении, что вы не имеете права уклониться от работы со средой. Во многих проектах на основе J2EE объекты Java Bean вообще не используются. Если у вас есть свобода выбора, работайте с теми средами или их фрагментами, которые согласуются с принятым вами архитектурным стилем.

Связь с фабриками

ФАБРИКА ведает началом существования объекта, а ХРАНИЛИЩЕ помогает работать с ним в середине и конце его жизни. Если объекты находятся в оперативной памяти или хранятся в объектной базе данных, то тут все просто. Но обычно хотя бы часть данных программы сохраняется в реляционной базе, файле и других необъектных системах. В таких случаях извлеченные из этих мест хранения данные приходится восстанавливать в объектную форму.

---

[2] Т.е. соответственно, *entity bean* и просто *entity*. Против этой путаницы и предостерегает автор. — *Примеч. перев.*

Поскольку в этом случае ХРАНИЛИЩЕ фактически создает объекты по имеющимся данным, многие считают, что ХРАНИЛИЩА — и есть ФАБРИКИ. С технической точки зрения так оно и есть. Но все-таки полезно на первом плане держать концептуальную модель, а с ее позиций, как уже говорилось, восстановление хранимого объекта не есть создание нового. В методике проектирования, основанной на предметной области, ФАБРИКИ и ХРАНИЛИЩА выполняют разные функции. ФАБРИКА создает новые объекты; ХРАНИЛИЩЕ находит и извлекает старые. ХРАНИЛИЩЕ должно давать клиентам иллюзию, что объекты хранятся прямо в памяти. Бывает, что объекты приходится восстанавливать (да, и при этом создавать новые экземпляры), но концептуально это те же самые объекты, которые уже существовали — это просто середина их жизненного цикла.

Чтобы примирить разные точки зрения, достаточно сделать так, чтобы ХРАНИЛИЩЕ *делегировало* создание объектов ФАБРИКЕ, которая также (теоретически, хотя на практике и редко) могла бы создавать и совсем новые объекты "с нуля".

Четкое разделение этих обязанностей помогает также снять с ФАБРИКИ всякую ответственность за поддержание целостности (непрерывности существования) объекта. Работа ФАБРИКИ — создать объект любой требуемой сложности на основе данных. Если в результате получается новый объект, об этом должен знать клиент, который при желании может добавить его в ХРАНИЛИЩЕ, а оно уже инкапсулирует операции по сохранению объекта в базе данных.

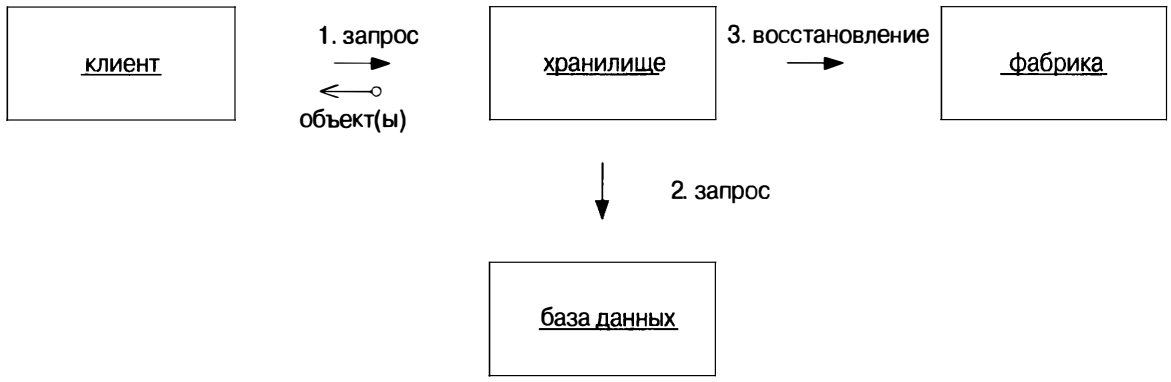


Рис. 6.22. ХРАНИЛИЩЕ восстанавливает существующий объект с помощью ФАБРИКИ

Рис. 6.23. Помещение нового объекта в ХРАНИЛИЩЕ

Искушение объединить ФАБРИКУ и ХРАНИЛИЩЕ появляется еще в одном случае — при желании реализовать функцию "поиска или создания". При этом клиент описывает, какой объект ему нужен, и если поиск показывает, что такого объекта еще не существует, то он создается и предоставляется клиенту. Подобных функций следует избегать. В лучшем случае ее наличие создает совсем небольшие удобства, но даже кажущаяся ее полезность исчезает вовсе, если в программе делается различие между СУЩНОСТЯМИ и ОБЪЕКТАМИ-ЗНАЧЕНИЯМИ. Если клиенту нужен ОБЪЕКТ-ЗНАЧЕНИЕ, он обращается к фабрике и получает новый. Как правило, различие между новым и уже существующим объектами играет важную роль в предметной области, и если средства архитектурной среды позволяют сымитировать отсутствие такого различия, то на самом деле они только запутывают дело.

Проектирование объектов для реляционной базы данных

Самой распространенной необъектной компонентой программных систем, которые в основном следуют объектно-ориентированному подходу, является реляционная база данных. Ее наличие порождает проблемы смешения парадигм (см. главу 5). Но база данных более тесно связана с объектной моделью, чем большинство прочих компонентов. База данных не просто имеет дело с объектами — она хранит в себе постоянную форму тех данных, которые образуют эти самые объекты. Уже немало написано о технических трудностях и особенностях проекции (отображения) объектов на реляционные базы данных, эффективного их хранения и извлечения. В частности, этот вопрос рассматривается в книге [13]. Существуют достаточно отлаженные средства для построения соответствий между этими двумя формами данных и управления ими. Не считая технических трудностей, некорректное построение такого соответствия может иметь существенное влияние и на саму объектную модель.

Наиболее распространены три случая.

1. База данных является в основном хранилищем объектов.
2. База данных была разработана для другой системы.
3. База данных разработана для этой системы, но выступает в роли, отличной от хранилища объектов.

Если структура базы данных специально проектируется для хранения объектов, то стоит и потерпеть некоторые ограничения в модели ради простоты соответствия объектов. Если нет других требований к структуре базы, ее можно спроектировать так, чтобы легче и эффективнее было поддерживать ее агрегатную целостность в процессе обновления. Вообще-то, структура таблиц реляционной базы данных не обязана отражать модель предметной области. Средства отображения данных сами по себе достаточно богаты возможностями, чтобы сгладить любые существенные отличия. Проблема в том, что иметь много накладывающихся друг на друга моделей не очень-то удобно и слишком сложно. К этому случаю применима та рекомендация, которую мы упоминали в разговоре о преимуществах ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ — избегать разделения двух процессов, анализа проблемы и проектирования модели. Да, для этого требуется частично пожертвовать сложностью объектной модели, а иногда пойти на компромисс и в структуре базы данных (например, избирательно применить денормализацию), но без этого есть риск потерять тесную связь между моделью и программной реализацией. Этот подход не требует упрощенного отображения "один объект-одна таблица". В зависимости от возможностей имеющихся средств отображения данных, возможно агрегирование и комбинирование объектов. Но очень важно, чтобы отображение сохраняло про-

зрачный характер — чтобы его можно было легко понять из чтения кода или названий отображаемых ячеек данных.

- Если база данных выступает в основном хранилищем объектов, не позволяйте модели данных и объектной модели "расходиться слишком далеко", вне зависимости от богатства средств отображения. Пожертвуйте частью отношений между объектами ради того, чтобы сделать модель ближе к реляционной. Не бойтесь применять такие формально реляционные стандарты, как нормализация, если это помогает в отображении данных.
- Процессы, протекающие вне объектной системы, вообще не должны иметь доступа к хранилищу объектов, потому что они могут нарушить накладываемые объектами инвариантные ограничения. Кроме того, предоставление им права доступа заблокирует модель данных от изменений, и это еще скажется, когда придет время рефакторинга.

С другой стороны, во многих случаях данные поступают из устаревшей или внешней системы, которая никогда и не задумывалась как хранилище объектов. В такой ситуации в одной системе фактически соседствуют две модели предметной области. Этот вопрос подробно разбирается в главе 14. Иногда имеет смысл приспособиться к модели, принятой в другой системе, а иногда, наоборот, — сделать свою модель совершенно другой.

Еще одна причина, по которой приходится делать исключения — это быстродействие. Для решения проблем в этой области программисту приходится идти на многие ухищрения.

Но для важного и распространенного случая, когда реляционная база данных служит постоянным хранилищем объектов из объектно-ориентированной предметной области, лучше всего применять самый прямой подход. Строка таблицы должна содержать объект — возможно, вместе с его подобъектами в виде АГРЕГАТА. Внешний ключ в таблице должен транслироваться в ссылку на другой объект-СУЩНОСТЬ. Если и встречается необходимость иногда отойти от этого прямого подхода, это не должно приводить к полному забвению принципа прямого соответствия.

Привязать объектную и реляционную составляющую к одной и той же модели помогает ЕДИНЫЙ ЯЗЫК. Имена и ассоциации элементов в объектах должны до мелочей соответствовать именам и ассоциациям в реляционных таблицах. Хотя в присутствии мощных средств отображения данных это может показаться несущественным, даже небольшие различия в отношениях между данными могут вызвать большую путаницу.

Традиция рефакторинга, которая все больше овладевает объектно-ориентированным миром, пока не слишком сильно повлияла на проектирование реляционных баз данных. Более того, серьезные проблемы переноса данных делают частые изменения нежелательными. Это может затормозить рефакторинг объектной модели, но если модель базы данных и объектная модель начинают расходиться, то может быстро потеряться прозрачность, наглядность преобразования данных.

Наконец, могут быть и причины для введения такой структуры базы данных, которая решительно отличается от объектной модели, пусть даже база специально создавалась именно для данной программной системы. База данных может также использоваться другой программой, в которой вообще не инициализируются экземпляры объектов. Такая база может практически не требовать изменений даже тогда, когда поведение объектов быстро меняется. Тогда возникает искушение углубить разрыв между системой и базой данных. Часто это делается непреднамеренно — разработчикам просто не удается вести базу данных "в ногу" с моделью. Если же такой разрыв выбирается сознательно, в результате вполне может получиться аккуратная и экономная структура таблиц базы, а не корявое нечто, порожденное многочисленными попытками привязать базу данных к самой последней версии объектной модели.

ГЛАВА 7

Работа с языком: расширенный пример

Предыдущие три главы были посвящены введению в язык шаблонов, пригодный для отшлифовки мельчайших деталей моделей и поддержки строгой методики ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN). В приведенных ранее примерах архитектурные шаблоны большей частью применялись по одному за раз, но в реальном проекте их придется использовать одновременно. В этой главе представлен один большой и подробный пример (хотя, конечно, намного более простой, чем любой реальный проект). В этом примере мы пройдем по цепочке последовательных усовершенствований модели и архитектуры программы точно так же, как гипотетическая группа разработчиков могла бы решать проблемы программной реализации и удовлетворять требования заказчика, попутно выстраивая архитектурный проект на основе модели предметной области. При этом будет показано, какие факторы действуют в том или ином случае и как шаблоны из части II помогают с ними справляться.

Введение в систему управления доставкой

Мы разрабатываем новую программу для компании, занимающейся доставкой товарных грузов своим клиентам. Первоначальный список требований состоит из трех функций.

1. Отслеживать ключевые манипуляции с грузом клиента.
2. Оформлять заказ заранее.
3. Автоматически высылать клиенту счет-фактуру по достижении грузом некоторого операционного пункта маршрута.

В реальном проекте пришлось бы потратить немало времени и итераций на то, чтобы прийти к простой и ясной модели. Подробно этот исследовательский процесс будет рассмотрен в части III. А здесь мы начнем с модели, которая выражает нужные понятия в разумной и естественной форме, а потом сосредоточимся на проработке деталей архитектуры.

Эта модель служит для организации знания из предметной области и предоставляет разработчикам рабочий язык. Теперь можно сформулировать такие утверждения.

> *В работе с Грузом (Cargo) участвует несколько Клиентов (Customers), каждый из которых играет свою роль (role).*
>
> *Должна задаваться (be specified) цель (goal) доставки груза.*
>
> *Цель (goal) доставки груза достигается в результате последовательности Переездов (Carrier Movement), которые удовлетворяют Заданию (Specification)*[1].

[1] "Спецификация" тут подходит хуже, чем "задание" по той причине, что автор естественным образом ставит рядом однокоренные слова: глагол *specify* (*задавать*) и существительное *specification* (задание, спецификация). Поэтому в данном случае лучше и по-русски использовать однокоренные слова. — *Примеч. перев.*

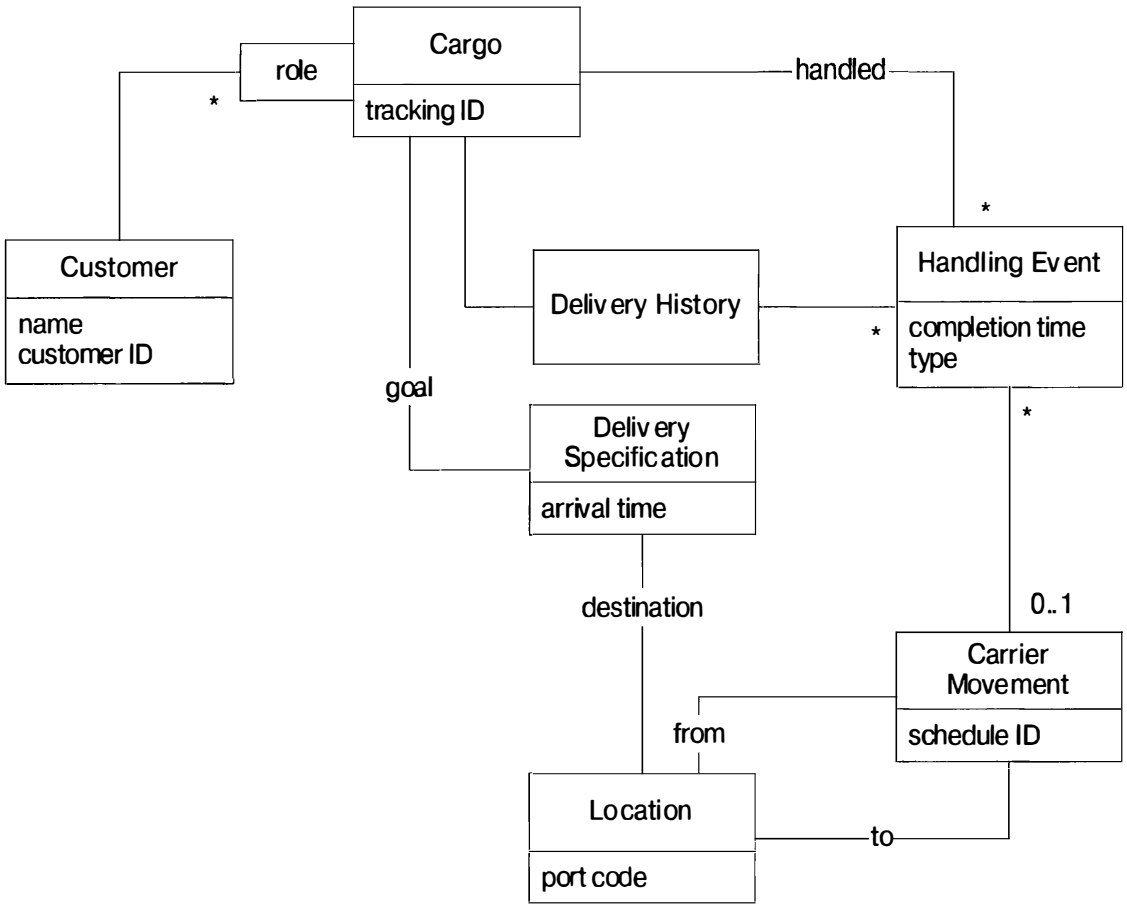


Рис. 7.1. Диаграмма классов, представляющая модель обслуживания доставки грузов

Каждый объект в этой модели имеет четкий смысл.

Манипуляция (Handling Event) — это дискретное действие, выполняемое над **Грузом (Cargo)**, например, погрузка его на судно или "проведение" через таможню. Этот класс, возможно, придется развить в целую иерархию различных событий, таких как погрузка, разгрузка, востребование получателем.

Задание на доставку (Delivery Specification) определяет цель доставки груза — это как минимум пункт следования и дата прибытия, но могут быть и дополнительные данные. Этот класс следует шаблону SPECIFICATION (СПЕЦИФИКАЦИЯ) (см. главу 9).

Эти обязанности можно было бы возложить на объект **Груз**, но абстракция в виде **Задания на доставку** дает как минимум три преимущества.

1. Если бы не было **Задания**, объект **Груз** отвечал бы за множество детальных атрибутов и ассоциаций, задающих цели доставки груза. Это загромоздило бы **Груз** и сделало бы его трудным для понимания и изменения.
2. При объяснении модели в целом такая абстракция позволяет легко и безопасно скрывать подробности. Например, в **Задании** могут быть инкапсулированы и другие критерии, но на общих схемах модели их рисовать не обязательно. Схема сообщает читателю, что на доставку имеется **Задание**, а подробности его в данный момент неважны (и могут быть легко изменены позже).
3. Такая модель более выразительна. Добавлением в нее **Задания на доставку** мы как бы говорим, что определяются не точные средства доставки **Груза**, а конечная цель, которая преследуется этой доставкой.

Роль предназначена для того, чтобы разделять обязанности, исполняемые тем или иным **Клиентом** в процессе доставки груза. Один из них будет "отправителем", другой — "получателем", третий — "плательщиком" и т.п. Для конкретного **Груза** только один **Клиент** может выступать в той или иной роли, поэтому ассоциация меняет свой тип с "многие ко многим" на "многие к одному". *Роль* можно реализовать просто как текстовую строку, хотя если от нее понадобится более сложное поведение, ее можно сделать и классом.

Переезд представляет одно перемещение конкретного **Транспортного средства** (**Carrier**), такого как судно или грузовик, из одного **Местоположения** (**Location**) в другое. Грузы переезжают из одного места в другое благодаря тому, что их грузят на **Транспортные средства** на время одного или нескольких **Переездов**.

История доставки (**Delivery History**) описывает, что в действительности происходило с **Грузом** — в отличие от **Задания на доставку**, которое описывает желаемое. Объект **История доставки** может вычислить текущее **Местоположение** нужного Груза, проанализировав последнюю погрузку или разгрузку, а также пункт назначения соответствующего **Переезда**. Успешная доставка заканчивается тем, что объект **История доставки** удовлетворяет цели, поставленные в **Задании на доставку**.

Итак, все понятия, необходимые для удовлетворения вышеописанных требований, в этой модели имеются. Конечно, нужны еще подходящие механизмы для поддержания непрерывности существования объектов, поиска нужных объектов и т.п. Эти вопросы программной реализации в модели не рассматриваются, а вот в архитектуре программы они должны быть решены.

И все же, чтобы стать основой для качественной программной реализации, эту модель предстоит сделать еще яснее и строже.

Не забывайте — как правило, усовершенствование модели, проектирование и реализация идут параллельно в итерационном процессе разработки программы. Изменения в модели должны мотивироваться не только удобством проектирования программы, но и необходимостью отразить в ней новые знания о предметной области. Однако в этой главе для упрощения изложения мы начнем уже со сравнительно зрелой модели, а изменения в ней будут вызваны только привязкой ее к практической реализации на основе рассмотренных ранее структурных шаблонов.

Изоляция предметной области: добавление прикладных операций

Чтобы не смешивать обязанности объектов предметной области с функциями других частей системы, применим МНОГОУРОВНЕВУЮ АРХИТЕКТУРУ и выделим *уровень предметной области*.

Не вдаваясь в подробности анализа, назовем три прикладные функции, которые можно передать трем классам операционного уровня.

1. **Маршрутный запрос** (**Tracking Query**), имеющий доступ к прошлым и нынешним манипуляциям с конкретным **Грузом**.
2. **Служба резервирования** (**Booking Application**)[2], позволяющая заказать новый **Груз** и приготовить систему к его обработке.

---

[2] Не следует путать употребляемое здесь слово *служба* с термином *service* — в данном контексте оно используется в общем, а не программном смысле, хотя исполняемые функции аналогичны. В оригинале слово *application* тоже создает путаницу с термином "приложение, прикладная программа". — *Примеч. перев.*

3. **Служба регистрации событий** (**Incident Logging Application**), фиксирующая любые манипуляции с **Грузом** (т.е. выводящая информацию, найденную **Маршрутным запросом**).

Эти классы операционного уровня являются всего лишь координаторами. Они не должны сами генерировать ответы на задаваемые вопросы — это работа уровня предметной области.

Отделение сущностей от значений

Рассмотрим по очереди каждый объект и выясним, кто из них представляет собой индивидуальные единицы, требующие отслеживания идентичности, а кто — просто набор значений. Вначале пройдем по четко определенным классам, а потом обратимся к менее очевидным случаям.

Клиент (Customer)

Начнем с самого простого. Объект **Клиент** представляет лицо физическое (человека) или юридическое (фирму), т.е. самую что ни на есть индивидуальную сущность. У такого объекта, очевидно, есть идентичность, важная для пользователя, поэтому в модели он представляет собой СУЩНОСТЬ. Как отслеживать эту СУЩНОСТЬ? В некоторых случаях подойдет идентификационный налоговый номер, но для иностранной компании этот способ не годится. Этот вопрос требует консультации со специалистом по предметной области. Мы обращаемся к сотруднику фирмы по доставке и узнаем, что у фирмы уже есть база данных клиентов, в которой каждому **Клиенту** при первом же деловом контакте присваивается индивидуальный кодовый номер. Этот код уже используется для работы в фирме, поэтому принятие его на вооружение в нашей программе позволит сохранить совместимость систем. Вначале код будет вводиться вручную.

Груз (Cargo)

Два одинаковых контейнера должны различаться, так что объекты **Груз** являются СУЩНОСТЯМИ. На практике все компании, занимающиеся доставкой грузов, присваивают контрольные идентификационные номера каждому месту груза. Такие номера генерируются автоматически, видны пользователю и в данном случае, скорее всего, сообщаются клиенту в момент резервирования заказа.

Манипуляция (Handling Event) и Переезд (Carrier Movement)

Нас интересует каждое такое событие, поскольку нам необходимо знать, что происходит с грузами. Это отражение событий реального мира, которые обычно не взаимозаменяемы, так что объекты будут СУЩНОСТЯМИ. Каждый **Переезд** будет идентифицироваться по коду, который берется из расписания перевозок.

В дальнейших консультациях со специалистом выясняется, что **Манипуляции** можно однозначно идентифицировать по совокупности номера **Груза**, времени выполнения и типа операции. Например, один и тот же **Груз** нельзя одновременно погружать и разгружать.

Местоположение (Location)

Два географических пункта с одинаковыми названиями не обязаны быть одним и тем же. Уникальный ключ можно было бы построить из широты и долготы. Но это было бы не очень удобно, поскольку для большинства операций в этой предметной области географические измерения не представляют никакого интереса, да и сложноваты техниче-

ски. Более вероятно, что объект **Местоположение** будет частью некоей специальной географической модели, в которой те или иные географические пункты будут различаться по маршрутам перевозок или еще каким-нибудь специфическим для предметной области способом. Поэтому достаточно будет произвольного, внутреннего, автоматически генерируемого идентификатора.

История доставки (Delivery History)

Тут дело обстоит сложнее. **Истории доставки** не взаимозаменяемы, т.е. являются СУЩНОСТЯМИ. Но любая **История доставки** однозначно ассоциирована с конкретным **Грузом**, и поэтому фактически не имеет собственной значимой идентичности. Ее индивидуальность заимствована у **Груза**, который является ее владельцем. Все это станет яснее, когда мы приступим к моделированию агрегатов.

Задание на доставку (Delivery Specification)

Хотя **Задание на доставку** представляет цель перевозки **Груза**, эта абстракция на самом деле не зависит от **Груза**, а выражает некоторое гипотетическое состояние одной из **Историй доставки**. Мы надеемся, что **История доставки**, ассоциированная с **Грузом**, со временем станет соответствовать **Заданию на доставку**, ассоциированному с тем же **Грузом**. Если два **Груза** направляются в одно и то же место, они могут совместно использовать одно и то же **Задание на доставку**, но не могут этого делать с **Историями доставки**, пусть даже вначале они выглядят одинаково (обе пустые). Итак, **Задания на доставку** представляют собой ОБЪЕКТЫ-ЗНАЧЕНИЯ.

Роль и другие атрибуты

Роль несет определенную информацию о той ассоциации, которой она соответствует, но у нее нет ни истории, ни непрерывности существования. Это ОБЪЕКТ-ЗНАЧЕНИЕ, и его могут совместно использовать разные пары **Груз–Клиент**.

Другие атрибуты, такие как метки времени или имена, тоже представляют собой ОБЪЕКТЫ-ЗНАЧЕНИЯ.

Проектирование ассоциаций в модели

Ни одна из ассоциаций на исходной схеме не имеет приоритетного направления, а ведь двунаправленные ассоциации трудно переводить в программные конструкции. Кроме того, выделение такого направления обычно выражает новое знание о предмете, что углубляет и саму модель.

Можно дать **Клиенту** прямую ссылку на каждый **Груз**, который он отправляет. Но для постоянного клиента, регулярно делающего новые заказы, это будет обременительно. Кроме того, понятие **Клиента** не привязано к конкретному **Грузу**. В большой системе **Клиент** может играть сразу много ролей по отношению ко многим объектам. Поэтому лучше не обременять его такими специфическими обязанностями. Если нам нужна функция поиска **Грузов** по **Клиентам**, это можно сделать по запросу к базе данных. К этому вопросу мы еще вернемся позже в этой главе — в разделе, посвященном ХРАНИЛИЩАМ.

Если бы наша программа вела учет грузовых судов, направление ассоциации от **Переезда** к **Манипуляции** имело бы важное значение. Но в нашем деле имеет значение только учет **Грузов**. И направление ассоциации только от **Манипуляции** к **Переезду** отражает правильное понимание предметной области. Также при этом упрощается про-

граммная реализация, сводясь к простой ссылке на объект, поскольку направление к множеству объектов отключено.

Обоснование остальных решений приведено на рис. 7.2.

В нашей модели имеется одна циклическая ссылка: **Груз** знает свою **Историю доставки**, которая содержит последовательность **Манипуляций**, которые в свою очередь указывают на **Груз**. Циклические ссылки возникают естественным образом во многих предметных областях, и в архитектуре программ они тоже бывают необходимы, но работать с ними не так-то просто. Можно попробовать, используя чисто технические средства, избежать необходимости хранить одну и ту же информацию в двух местах, между которыми требуется синхронизация. В нашем случае можно реализовать простой, хотя и ненадежный вариант (на Java) в исходном прототипе: включить в **Историю доставки** объект **Список** (**List**), содержащий **Манипуляции**. Но в какой-то момент нам, вероятно, придется отказаться от этой коллекции в пользу поиска по базе данных с **Грузом** в качестве ключа. Этот вопрос еще всплывет и при выборе ХРАНИЛИЩ. Если запрос на просмотр истории делается сравнительно нечасто, такой подход должен дать хорошее быстродействие, упростить доработку и снизить затратность добавления новых **Манипуляций**. Если же такй запрос встречается очень часто, то лучше иметь прямой указатель. Здесь идет речь о балансе между простотой реализации и быстродействием. Модель же остается одной и той же; в ней есть циклическая ссылка и двунаправленность ассоциации.

Границы агрегатов

Объекты **Клиент**, **Местоположение** и **Переезд** имеют собственную идентичность и совместно используются многими **Грузами**, поэтому они должны стать корневыми в своих собственных АГРЕГАТАХ, содержащих как их атрибуты, так и другие объекты, которые мы пока не будем рассматривать в подробностях. **Груз** также представляет собой очевидный корневой объект АГРЕГАТА, хотя не сразу понятно, где же провести четкую границу для него.

АГРЕГАТ **Груз** имеет право объединять в себе все, что не существовало бы, не будь в природе этого **Груза** — т.е. **Историю доставки**, **Задание на доставку**, а также **Манипуляции**. С **Историей доставки** все ясно. Никто не будет разыскивать ее в базе, не интересуясь при этом самим **Грузом**. Итак, прямой глобальный доступ к ней не нужен, а ее индивидуальность непосредственно происходит от **Груза**, поэтому **История доставки** идеально укладывается в границы **Груза** как АГРЕГАТА, и при этом не обязана быть корневым объектом. **Задание на доставку** является ОБЪЕКТОМ-ЗНАЧЕНИЕМ, и от включения его в тот же АГРЕГАТ **Груз** никаких осложнений возникнуть не должно.

А вот **Манипуляция** — дело другое. Раньше мы уже рассмотрели два возможных запроса к базе данных, которые могли бы разыскивать такие данные. Один, в котором **Манипуляции** разыскиваются по базе для определенной **Истории доставки** вместо того, чтобы извлекаться из коллекции, имел бы локальный характер в АГРЕГАТЕ **Груз**. Другой использовался бы для нахождения всех погрузочных и подготовительных операций с грузом, соответствующих конкретному **Переезду**. Во втором случае возникает такое ощущение, что действия по манипулированию **Грузом** имеют самостоятельную объектную ценность даже отдельно от самого **Груза**. Поэтому **Манипуляция** должна быть корневым объектом своего собственного АГРЕГАТА.

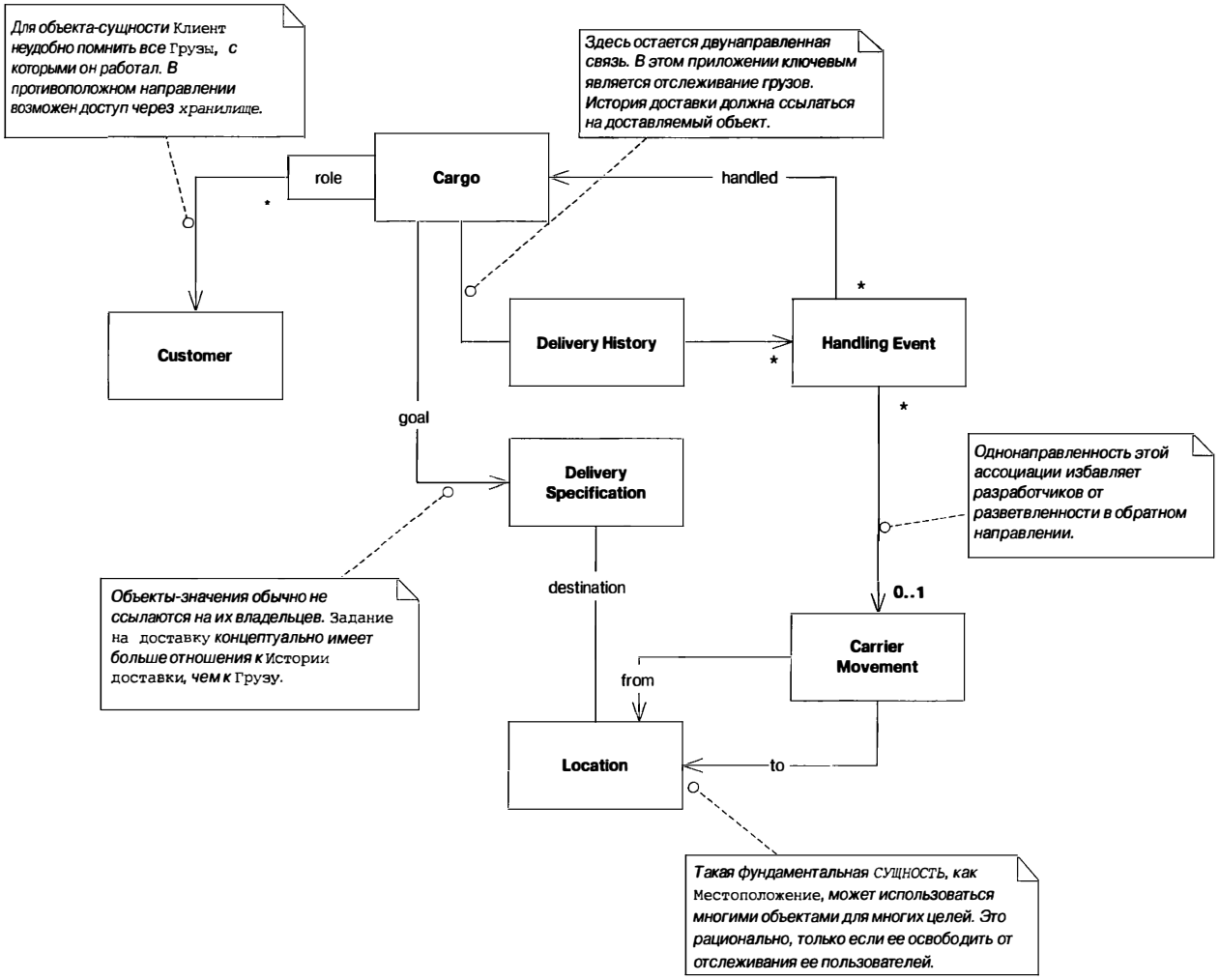


Рис. 7.2. Ограничение допустимых направлений в некоторых ассоциациях

Выбор хранилищ

В архитектурном проекте нашей программы имеется пять СУЩНОСТЕЙ, являющихся корневыми объектами АГРЕГАТОВ. Поэтому достаточно ограничиться только этими объектами — остальным все равно нельзя иметь свои ХРАНИЛИЩА.

Чтобы решить, кто из этих кандидатов имеет право на хранилище, необходимо вернуться к списку функциональных требований к программе. Чтобы сделать заказ через **Службу резервирования** (**Booking Application**), пользователь программы должен выбрать одного или нескольких **Клиентов**, которые будут играть различные роли (отправитель, получатель и т.д.). Итак, нам нужно **Хранилище клиентов** (**Customer Repository**). Необходимо также будет задать **Местоположение** пункта, куда нужно отправить **Груз**, для чего следует создать **Хранилище местоположений** (**Location Repository**).

Служба регистрации событий (**Incident Logging Application**) должна дать пользователю возможность найти **Переезд**, на котором сейчас находится **Груз**, поэтому необходимо и **Хранилище переездов** (**Carrier Movement Repository**). Пользователь должен также иметь возможность сообщить системе, какой **Груз** перевозится, так что потребуется и **Хранилище грузов** (**Cargo Repository**).

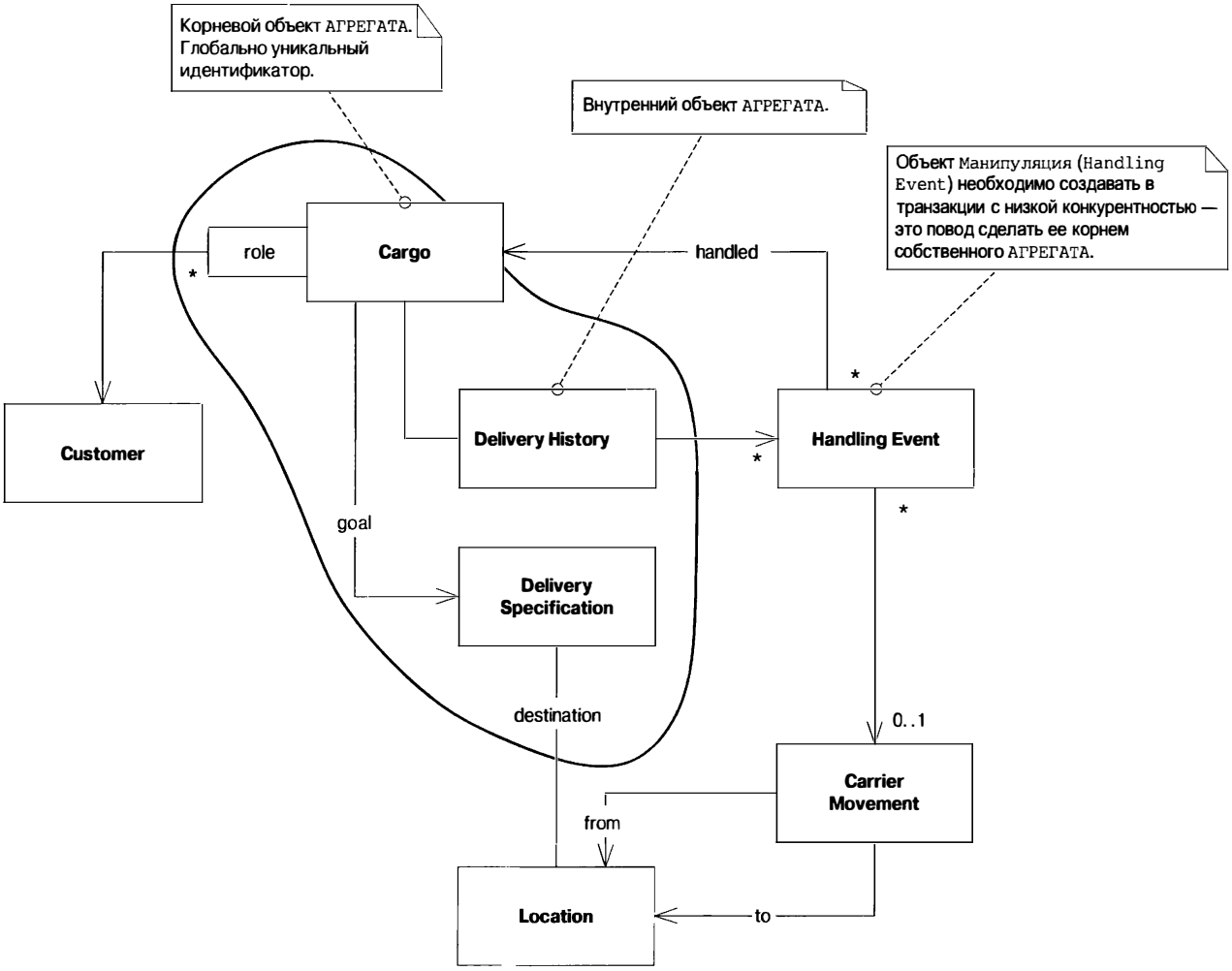


*Рис. 7.3. Границы АГРЕГАТОВ в модели. **(Примечание**: подразумевается, что любая СУЩНОСТЬ за пределами очерченной границы является корнем своего собственного АГРЕГАТА.)*

Хранилища манипуляций (Handling Event Repository) пока не будет, потому что в первом приближении мы решили реализовать ассоциацию с **Историей доставки** в виде коллекции, а в требованиях к программе ничего не сказано о том, должна ли она уметь выяснять, что перевозится в том или ином **Переезде**. Но оба обстоятельства могут измениться, и тогда может понадобиться ХРАНИЛИЩЕ.

Проход по сценариям

Для проверки правильности всех принятых проектных решений нам придется регулярно шаг за шагом проходить по рабочим сценариям программы, чтобы убедиться, что ее задачи решаются эффективно.

Пример рабочей функции: изменение места назначения груза

Нам звонит **Клиент** и говорит: “Ой, мы попросили доставить наш груз в Хакенсак, но на самом деле он нам нужен в Хобокене”. Мы выполняем все пожелания наших клиентов, так что система должна обеспечить подобное изменение маршрута.

Задание на доставку — это ОБЪЕКТ-ЗНАЧЕНИЕ, поэтому проще будет его сбросить и завести новый, а потом сделать нужную замену в объекте **Груз (Cargo)** с помощью *set*-метода.

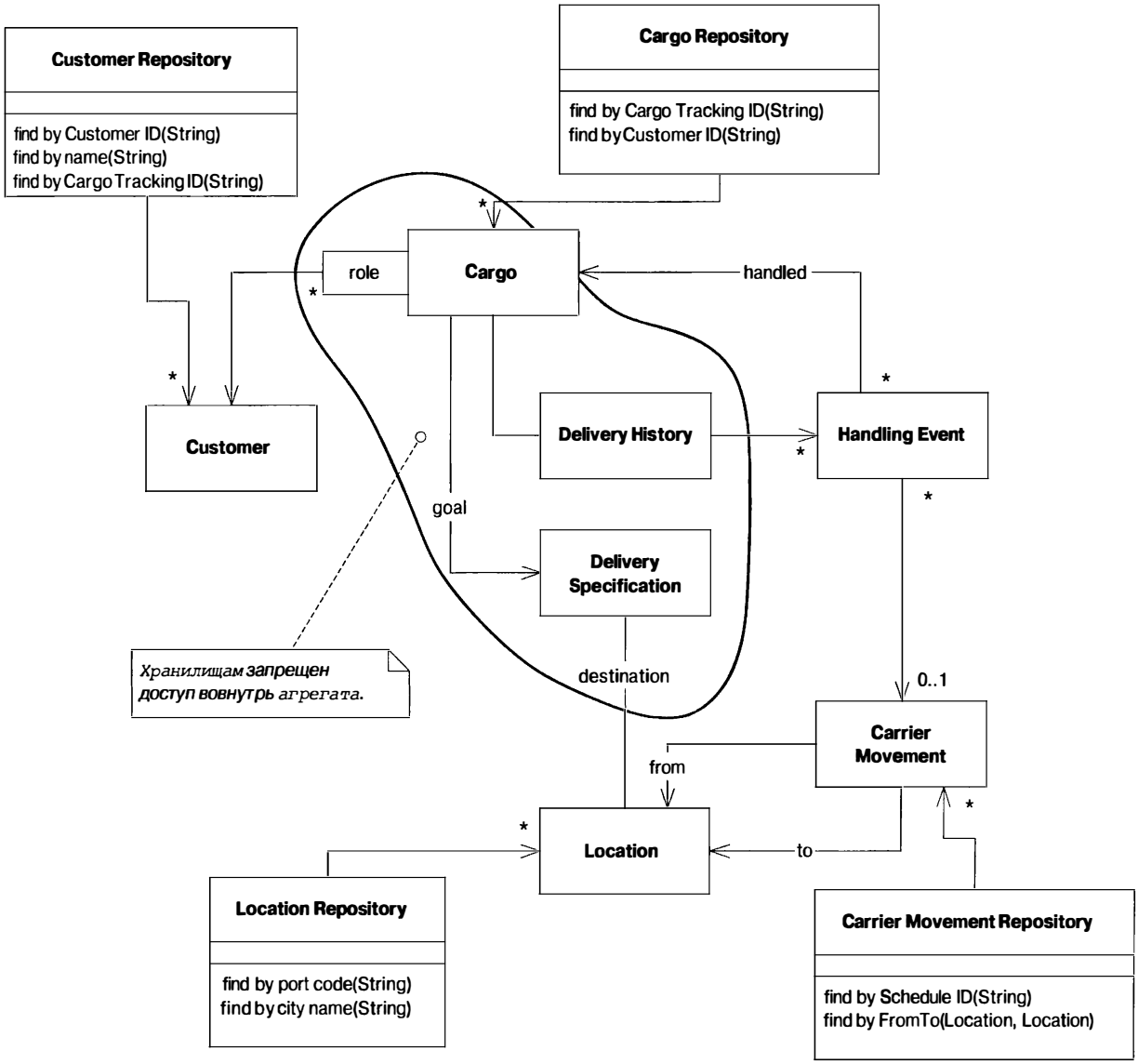


Рис. 7.4. ХРАНИЛИЩА предоставляют доступ к корневым объектам выбранных АГРЕГАТОВ

Пример рабочей функции: повторение заказов

Пользователи говорят, что заказы, поступающие от одних и тех же клиентов, имеют тенденцию практически повторяться, поэтому имеет смысл использовать данные старых **Грузов** как прототипы для новых. В программе должна быть возможность найти **Груз** в ХРАНИЛИЩЕ и выбрать команду создания нового **Груза** на основе выбранного. Применим для этого архитектурный шаблон PROTOTYPE (ПРОТОТИП) [14].

Груз это СУЩНОСТЬ и корневой объект АГРЕГАТА. Поэтому копировать его надо тщательно; необходимо проанализировать, что должно случиться с каждым объектом или атрибутом, заключенным в границы этого АГРЕГАТА. Итак, по порядку.

- **История доставки**. Нужно создать новый пустой объект, поскольку история старого объекта здесь неприменима. Обычно так и бывает с СУЩНОСТЯМИ внутри границ АГРЕГАТОВ.
- **Роли клиентов**. Необходимо скопировать **Карту (Map)** или другую используемую коллекцию, которая содержит ссылки на **Клиентов** по ключам — причем вместе с самими ключами, поскольку **Клиенты**, весьма вероятно, будут играть те же роли

в новом заказе. Однако нужно проявить осторожность и не копировать *сами объекты*. В конце концов у нас должны остаться ссылки на те же объекты-**Клиенты**, на которые ссылался старый объект **Груз**, потому что это СУЩНОСТИ за пределами АГРЕГАТА.

- **Контрольный номер**. Новому заказу нужно присвоить новый контрольный идентификационный номер из того же источника, из которого мы бы получали его при создании целиком нового **Груза**.

Следует обратить внимание, что мы скопировали все внутри границ АГРЕГАТА **Груз** и внесли кое-какие изменения в копию, но *не затронули никаких данных за пределами этого* АГРЕГАТА.

Создание объектов

Фабрики и конструкторы для объекта Груз

Даже если у нас есть сложно устроенная ФАБРИКА для объекта **Груз (Cargo)** или если в качестве ФАБРИКИ используется другой **Груз**, как это было в сценарии повторения заказов, нам все равно нужен хотя бы простейший конструктор. От конструктора требуется производить на свет объект, соблюдающий инварианты, или хотя бы, в случае СУЩНОСТИ, поддерживающий собственную уникальную идентичность.

На основании этих проектных решений можно создать следующий метод-ФАБРИКУ в объекте **Груз**.

```
public Cargo copyPrototype(String newTrackingID)
```

Или же можно включить этот метод в автономную ФАБРИКУ.

```
public Cargo newCargo(Cargo prototype, String newTrackingID)
```

Автономная ФАБРИКА могла бы также инкапсулировать процесс получения нового, автоматически сгенерированного идентификатора для нового **Груза**; в этом случае ей понадобился бы всего один аргумент.

```
public Cargo newCargo(Cargo prototype)
```

Из этих ФАБРИК будет возвращаться один и тот же результат: объект **Груз** с пустой **Историей доставки (Delivery History)** и нулевым **Заданием на доставку (Delivery Specification)**.

Двунаправленная ассоциация между **Грузом** и его **Историей доставки** означает, что ни **Груз**, ни **История доставки** не является завершенным объектом, если не указывает на другой, а поэтому создавать их нужно вместе. Напомним, что **Груз** — корневой объект АГРЕГАТА, включающего **Историю доставки**. Поэтому можно разрешить конструктору или ФАБРИКЕ **Груза** создавать **Историю доставки**. Конструктор **Истории доставки** будет принимать **Груз** в качестве аргумента, и результат будет выглядеть примерно так.

```
public Cargo(String id) {
    trackingID = id;
    deliveryHistory = new DeliveryHistory(this);
    customerRoles = new HashMap();
}
```

В результате получаем новый **Груз** с новой **Историей доставки**, указывающей на все тот же **Груз**. Конструктор **Истории доставки** используется исключительно корневым объектом ее АГРЕГАТА, т.е. **Груза**, чтобы создание **Груза** было полностью инкапсулировано.

Добавление объекта Манипуляция

Всякий раз, когда груз подвергается манипуляциям в реальном мире, кто-то из пользователей вводит новый объект **Манипуляция (Handling Event)**, используя **Службу регистрации событий (Incident Logging Application)**.

В каждом классе должны быть свои простейшие конструкторы. Поскольку **Манипуляция** представляет собой СУЩНОСТЬ, в конструктор должны передаваться все атрибуты, определяющие ее индивидуальность. Как уже говорилось, **Манипуляция** однозначно определяется по сочетанию идентификатора **Груза**, времени выполнения и типа события. Кроме этого, в **Манипуляции** имеется еще только один атрибут — ассоциация с **Переездом** (причем даже не у всех типов **Манипуляций**). Вот простейший конструктор, создающий корректный объект **Манипуляция (Handling Event)**:

```
public HandlingEvent(Cargo c, String eventType, Date timeStamp) {
    handled = c;
    type = eventType;
    completionTime = timeStamp;
}
```

Как правило, атрибуты СУЩНОСТИ, не обязательные для ее идентификации, можно добавить позже, а не при создании. В нашем случае все атрибуты **Манипуляции** будут определены в первой транзакции, после чего больше изменяться не будут (кроме разве что исправления ошибок ввода даты). Поэтому будет удобно и более выразительно с точки зрения клиентского кода добавить в объект **Манипуляция** простой МЕТОД-ФАБРИКУ для каждого типа события, который бы принимал все нужные аргументы. Например, "событие погрузки" не может обойтись без объекта **Переезд (Carrier Movement)**.

```
public static HandlingEvent newLoading(
   Cargo c, CarrierMovement loadedOnto, Date timeStamp) {
      HandlingEvent result =
         new HandlingEvent(c, LOADING_EVENT, timeStamp);
      result.setCarrierMovement(loadedOnto);
      return result;
}
```

В нашей модели **Манипуляция** — это абстракция, которая может инкапсулировать много различных специализированных классов **Манипуляций**, от погрузки и разгрузки до опечатывания, помещения на хранение и других операций, не связанных с **Переездами**. Можно создать для этой цели много подклассов, можно применить сложную инициализацию, а можно сочетать то и другое. Добавляя МЕТОДЫ-ФАБРИКИ в базовый класс **Манипуляция** для каждого типа манипуляций с грузами, мы абстрагируем создание экземпляров и освобождаем клиентский код от необходимости знать внутреннее устройство класса. Пусть ФАБРИКА отвечает за то, чтобы правильно распознать, экземпляр какого класса нужно создать и как именно его инициализировать.

Но, к сожалению, не все так просто. Создание экземпляров осложняется циклическими ссылками: из **Груза** на **Историю доставки**, затем на **Событие истории**

(**History Event**) и в конце концов снова на **Груз**. **История доставки** содержит коллекцию **Манипуляций**, относящихся к соответствующему **Грузу**, и новый объект можно добавлять в эту коллекцию только в рамках транзакции. Если этот обратный указатель не создать, связи между объектами будут разорваны.

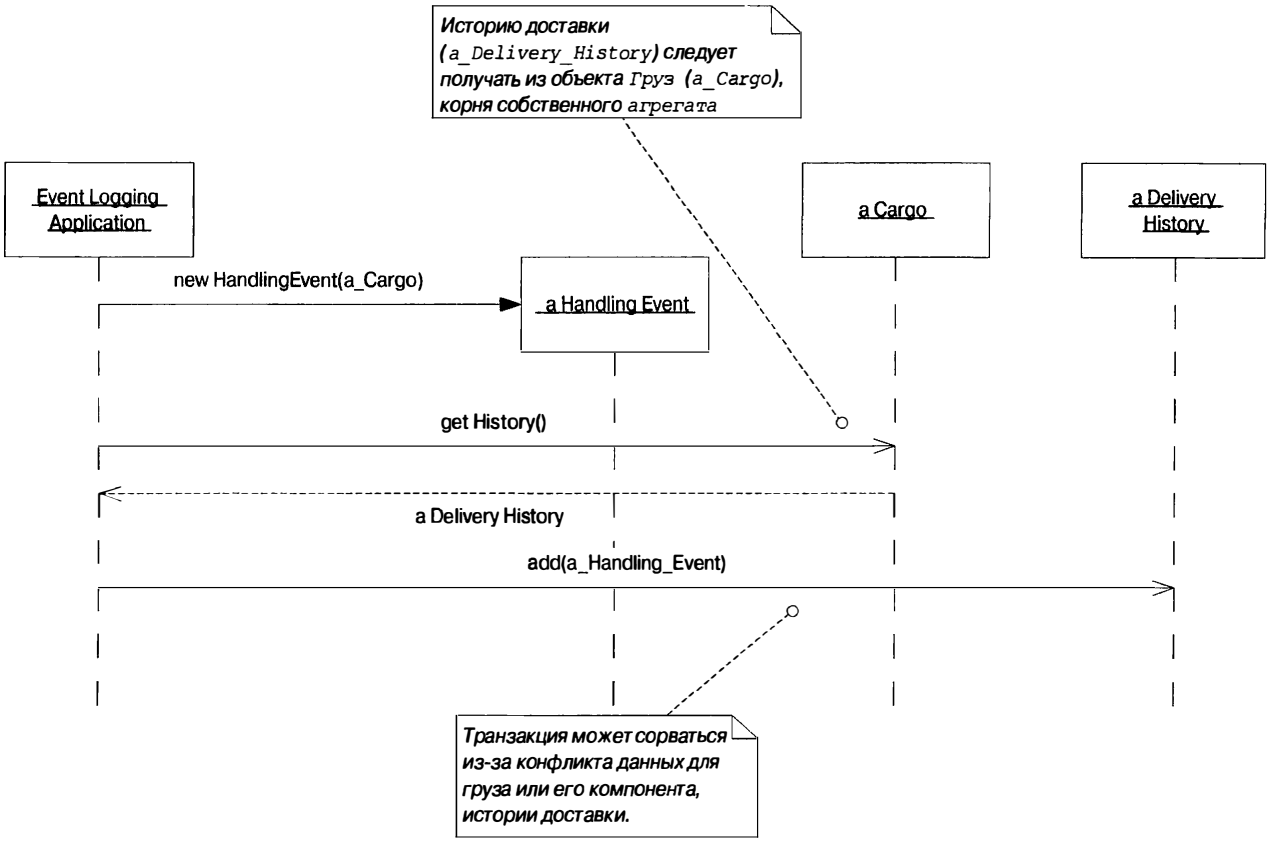


*Рис. 7.5. При добавлении **Манипуляции** ее необходимо внести в **Историю доставки***

Создание обратного указателя можно инкапсулировать в ФАБРИКЕ (и сохранить на уровне предметной области, где ему и место). Но сейчас мы рассмотрим альтернативный подход, который позволит совсем убрать эту "неуклюжую" взаимосвязь.

Перерыв на рефакторинг: альтернативный агрегат Груз

Моделирование предметной деятельности и проектирование программной архитектуры — это движение не только вперед. Без регулярного рефакторинга, отражающего новое понимание модели и улучшающего качество ее реализации, этот процесс очень скоро будет не нужен.

В настоящий момент спроектированная нами архитектура имеет несколько недостатков, хотя она работает и действительно отражает модель. Проблемы, которые в начале проектирования казались маловажными, теперь начинают раздражать. Вернемся к одной из таких проблем и попробуем кое-что изменить.

Необходимость обновлять **Историю доставки** при добавлении **Манипуляции** вовлекает в транзакцию АГРЕГАТ **Груз**. Если другой пользователь в это же самое время вносит в **Груз** изменения, то транзакция **Манипуляции** срывается или задерживается. Ввод **Манипуляции** это операционная деятельность, которая должна выполняться бы-

стро и просто, поэтому следует выдвинуть важное требование — вводить **Манипуляции** *без конкурентности, конфликтов данных*. Это требует от нас пересмотреть архитектуру программы.

Если заменить коллекцию **Манипуляций** в **Истории доставки** на запрос к базе данных, это позволит нам добавлять новые **Манипуляции**, не создавая проблем с транзакционной целостностью за пределами АГРЕГАТА. Такое изменение в программе позволит завершать подобные транзакции без вмешательства. Если вводится много **Манипуляций**, а запросов выполняется сравнительно немного, этот подход становится более эффективным. На самом деле же, если на техническом уровне для хранения данных используется реляционная база, то, скорее всего, для эмуляции коллекции все равно используется запрос. Замена коллекции запросом позволяет также облегчить согласование циклических ссылок между **Грузом** и **Манипуляцией**.

Чтобы реализовать механизм запросов, добавим в программу ХРАНИЛИЩЕ для МАНИПУЛЯЦИЙ. **Хранилище манипуляций (Handling Event Repository)** обеспечит выполнение запросов по **Манипуляциям**, относящимся к определенному **Грузу**. Кроме того, оно может обслуживать запросы, оптимизированные на поиск определенной специализированной информации. Например, если часто выполняются обращения к **Истории доставки** для просмотра последней погрузки или выгрузки, чтобы по ним определить текущее состояние **Груза**, то можно скомпоновать такой запрос, который будет возвращать только нужную **Манипуляцию**. А если понадобится, например, запросить все **Грузы**, погруженные на транспорт для конкретного **Переезда**, такой запрос тоже можно будет легко добавить.

Из-за всего этого у **Истории доставки** теперь нет постоянно хранимого состояния — на этом этапе нет необходимости держать его под рукой. **Историю доставки** легко воспроизвести всякий раз, когда она нужна для ответа на определенный вопрос. И хотя СУЩНОСТЬ в этом случае переживает регулярное восстановление, непрерывность ее существования между различными воплощениями обеспечивается ассоциацией с одним и тем же **Грузом**.

Теперь организовать и поддерживать циклическую ссылку ничего не стоит. **Фабрика грузов (Cargo Factory)** упрощается, поскольку больше не обязана добавлять пустую **Историю доставки** к новым экземплярам объектов. Можно немного сэкономить на объеме базы данных, причем реальное количество постоянно хранимых объектов может уменьшиться значительно, а это ограниченный ресурс в некоторых объектных базах данных. Если обычная практика такова, что пользователь редко запрашивает статус **Груза** до его прибытия к месту назначения, то при этом подходе удается избежать большого объема ненужной работы.

С другой стороны, если используется объектная база данных, то прослеживание ассоциации или коллекции выполняется, скорее всего, намного быстрее, чем запрос к ХРАНИЛИЩУ. Если в ходе работы часто запрашивается полная история манипуляций с грузом, а не время от времени — его последнее местоположение, то соображения быстродействия могут оказаться в пользу коллекции. Кроме того, не забудем, что добавленную в программу гипотетическую возможность ("Что перевозится в этом **Переезде**?") пока никто не просил, и может вовсе не попросить, так что не нужно слишком тратиться на ее реализацию.

Подобные альтернативные варианты и компромиссы встречаются повсеместно, и даже в этой упрощенной системе можно найти множество примеров. Но важно понимать, что все это степени свободы маневра внутри одной и той же модели. Правильно выполнив моделирование ОБЪЕКТОВ-ЗНАЧЕНИЙ, СУЩНОСТЕЙ и их АГРЕГАТОВ,

мы снизили побочные эффекты от изменений архитектуры. Например, в нашем случае все изменения инкапсулированы в границах АГРЕГАТА **Груз**. Правда, потребовалось еще введение **Хранилища манипуляций**, но при этом не пришлось ничего переделывать в самих **Манипуляциях** (правда, в зависимости от особенностей технической реализации ХРАНИЛИЩ кое-какая переделка все-таки может понадобиться).

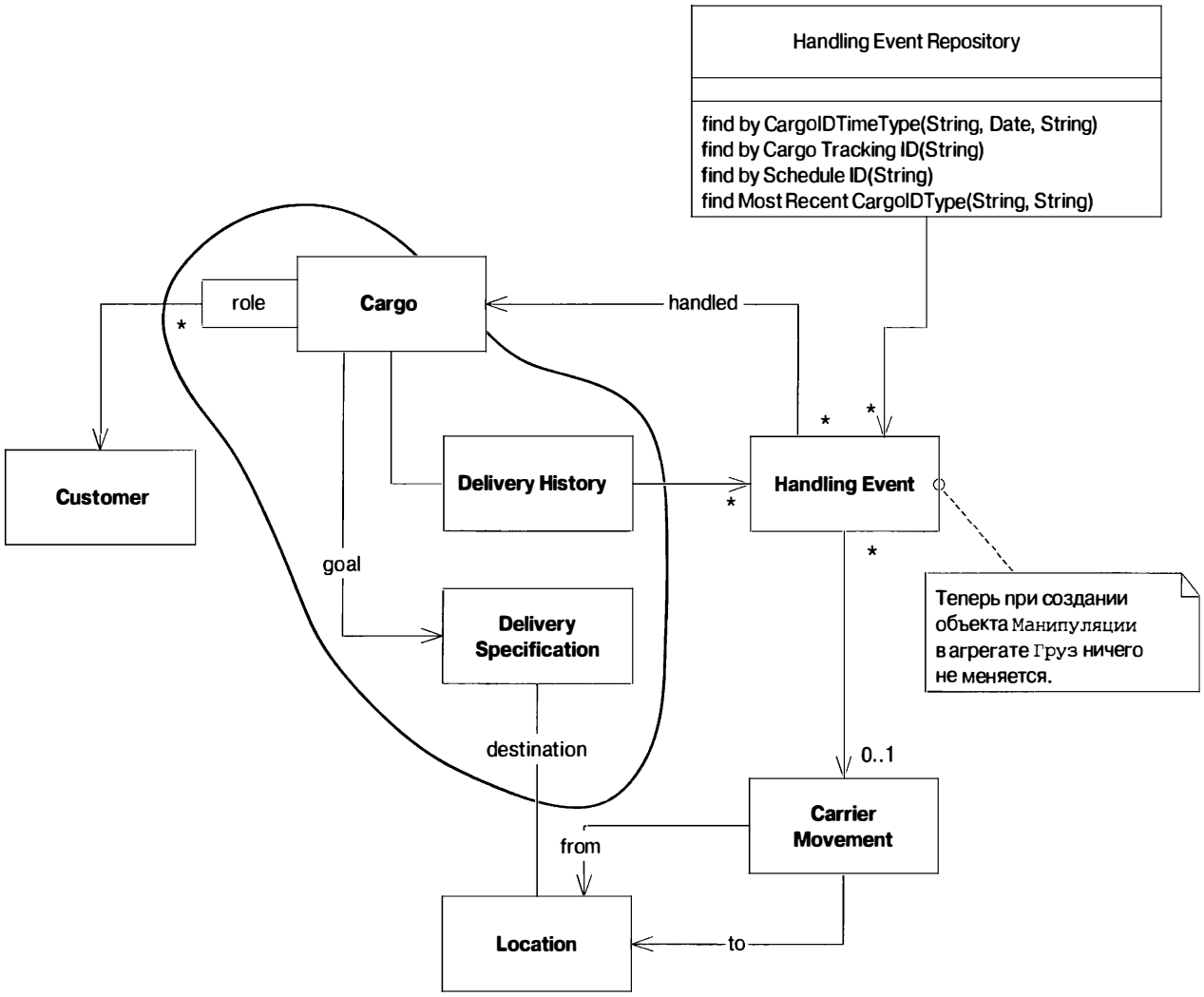


*Рис. 7.6. Реализация коллекции **Манипуляций** (**Handling Events**) в **Истории доставки** (**Delivery History**) в виде запроса к базе данных облегчает добавление новых Манипуляций и ликвидирует конфликты параллельно вводимых данных в АГРЕГАТЕ **Груз** (**Cargo**)*

Модули в модели грузопоставок

Пока что в нашей работе участвовало так мало объектов, что вопрос о разбиении на модули даже не поднимался. Теперь давайте рассмотрим несколько более "громоздкую" часть модели поставок (хотя, конечно, по-прежнему упрощенную) и оценим, как ее организация в виде МОДУЛЕЙ повлияет на модель.

На рис. 7.7 показано аккуратное разбиение модели, выполненное гипотетическим энтузиастом идей нашей книги. Эта схема напоминает нам о проблеме модульного членения под влиянием технической инфраструктуры, поставленной в главе 5. В данном случае объекты группируются согласно архитектурным шаблонам, по которым они построены. В результате сваливаются в одну кучу объекты, концептуально имеющие между

собой мало общего (обладающие *низкой связностью*), тогда как между модулями "протягивается" множество ассоциаций (имеет место *высокая зависимость*). Деление на модули несет какую-то информацию, но это не информация о процессе поставок — это информация о том, какую книжку читал программист, когда писал эту программу.

Группировка по структурному признаку (шаблонам) может показаться очевидной и грубой ошибкой, но на самом деле этот критерий ничуть не хуже, чем отделение постоянно существующих объектов от временных или любая другая методика, никак не связанная с предметным смыслом самих объектов.

Вместо того чтобы заниматься подобным, нам следует искать связанные понятия и сосредоточиться на цели проекта. Как и в случае менее масштабных проектных решений, существует много способов сделать это. На рис. 7.8 показан самый "прямолинейный".

Имена модулей на рис. 7.8 пополняют язык разработчиков. Наша компания выполняет *доставку* по адресам *клиентов*, поэтому мы имеем право взимать с них *оплату*. Торгово-обслуживающий персонал напрямую имеет дело с *клиентами*, заключая с ними *договоры*. Технический персонал непосредственно выполняет операции *доставки*, привозя груз к месту назначения. Бухгалтерия ведает вопросами *оплаты*, выставляя счета-фактуры согласно оговоренным в договоре *клиента* ценам. Такова информация, которую несет данный набор модулей.

Конечно, это интуитивное разбиение можно дальше улучшать методом последовательных приближений, а то и вовсе заменить на другое, но в нынешнем виде оно уже помогает в ПРОЕКТИРОВАНИИ ПО МОДЕЛИ и вносит вклад в развитие ЕДИНОГО ЯЗЫКА.

Новая функция: распределение заказов

До этого момента мы занимались "подгонкой" модели под первоначальные требования. Теперь нам предстоит добавить некоторые новые и важные функции.

Отдел сбыта воображаемой компании-поставщика товаров пользуется другими программами для управления связями с клиентами, прогнозирования сбыта и т.п. Одна из функций состоит в управлении производительностью труда: фирма имеет возможность ввести допустимое для нее распределение заказов по видам товаров, их происхождению и месту назначения, а также по другим факторам, которые можно ввести как дополнительные категории. Фактически это управление сбытом товаров и услуг с целью не допустить торможения более выгодных деловых операций менее выгодными грузами и в то же время избежать как недозаказа (неполного использования транспортных мощностей), так и перезаказа (из-за чего возникают заторы и конфликты при перевозке, и это портит отношения с клиентами).

Теперь фирма хочет, чтобы эту функцию интегрировали с системой оформления заказов. Когда поступает заказ, он должен сравниваться с имеющимся распределением, чтобы выяснить, принимать его или нет.

Нужная для этого информация располагается в двух местах, которые необходимо опрашивать, используя **`Службу резервирования (Booking Application)`**, которая и будет решать, принять заказ или нет. Примерная схема движения информации будет выглядеть так.

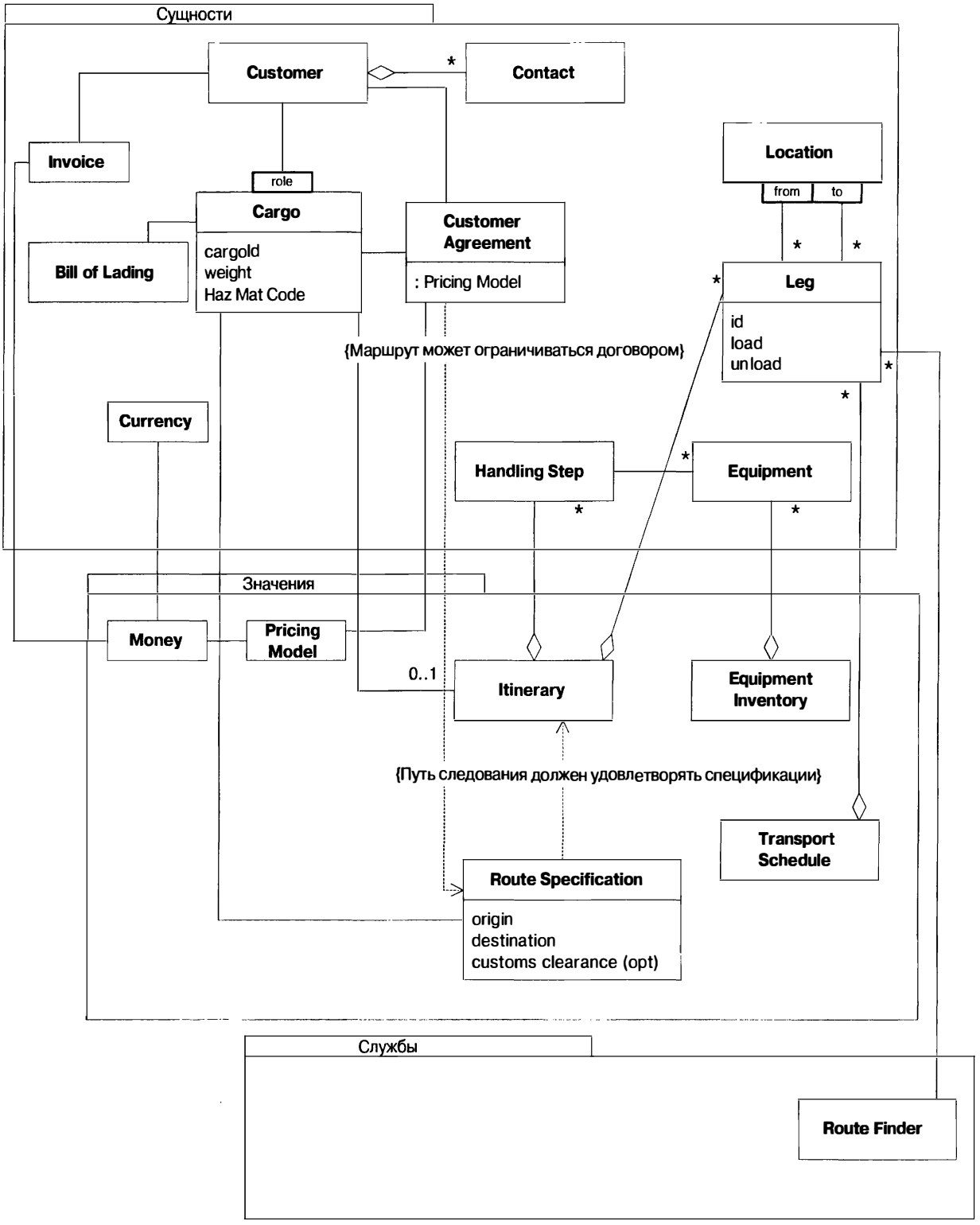


Рис. 7.7. Деление на МОДУЛИ, не несущее знаний о предметной области

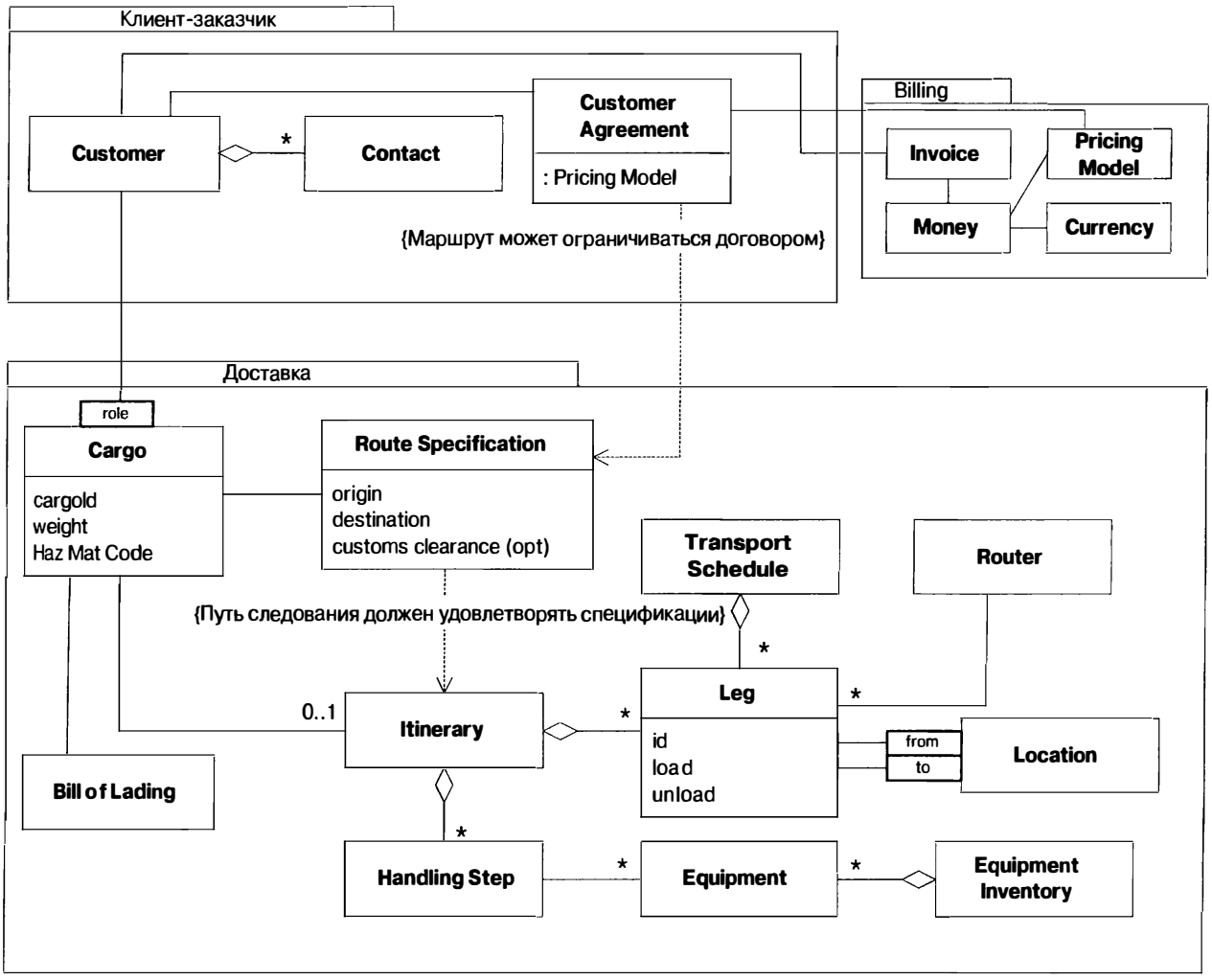


Рис. 7.8. Деление на МОДУЛИ на основе самых общих понятий модели

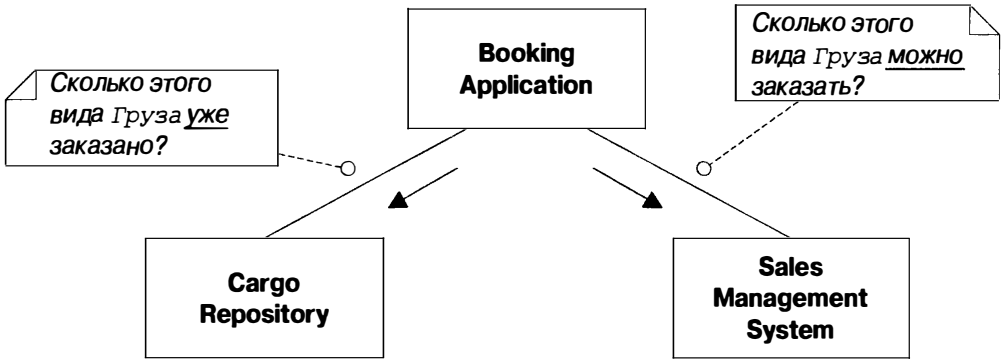


Рис. 7.9. ***Службе резервирования (Booking Application)*** *необходима информация из* ***Системы управления поставками (Sales Management System)*** *и из ХРАНИЛИЩ уровня модели предметной области*

Связь между двумя системами

Система управления поставками не создана на основе этой модели, с которой мы сейчас работаем. Если **Служба резервирования** будет взаимодействовать с ней напрямую, то нам придется приспосабливаться к архитектуре и технологиям той другой системы. Из-за этого станет труднее ПРОЕКТИРОВАТЬ ПО МОДЕЛИ и придерживаться

ЕДИНОГО ЯЗЫКА. Чтобы этого не делать, давайте разработаем новый класс, в обязанности которого будет входить трансляция между нашей моделью и языком **Системы управления поставками**. Мы не будем создавать трансляторный механизм самого общего назначения. Новый класс будет открывать для обозрения только те возможности, которые нам нужны, и абстрагировать их заново в терминах нашей модели. Таким образом, этот класс будет выступать в качестве ПРЕДОХРАНИТЕЛЬНОГО УРОВНЯ (ANTICORRUPTION LAYER), который рассматривается позже в главе 14.

Поскольку этот класс — интерфейс к **Системе управления поставками**, на первый взгляд может показаться, что ему нужно дать имя **Интерфейс управления поставками (Sales Management Interface)** или что-то в этом роде. Но тогда мы упустим возможность переформулировать проблему на нашем языке в виде, более удобном для нас. Давайте определим СЛУЖБУ для каждой из функций распределения заказов, которые нам потребуются от другой системы. А реализуем мы эти службы с помощью класса, имя которого будет отражать его обязанности в нашей системе: **Контролер распределения (Allocation Checker)**.

Если понадобится еще какая-нибудь интеграция помимо этой (например, использование базы данных клиентов из **Системы управления поставками** вместо нашего собственного ХРАНИЛИЩА **Клиентов**), то можно создать еще один транслятор, СЛУЖБЫ которого будут выполнять соответствующие обязанности. Может оказаться полезным и наличие класса более низкого уровня наподобие **Интерфейса системы управления поставками (Sales Management System Interface)**, который бы управлял механизмами коммуникации с другой системой, но не отвечал бы за трансляцию смысловых понятий. К тому же он был бы спрятан за **Контролером распределения**, следовательно, не виден в модели предметной области.

Усовершенствование модели: введение подразделений

Итак, мы наметили контуры взаимодействия двух систем. Теперь поставим задачу: какой интерфейс нужно иметь, чтобы ответить на вопрос “Сколько можно заказать **Груза** данной разновидности?”. Проблема состоит в том, чтобы определить, что такое “разновидность” **Груза**, поскольку в модели предметной области никакой классификации **Грузов** еще нет. В **Системе управления поставками** классификация **Грузов** на категории выполняется по списку ключевых слов, и наши типы должны соответствовать этому списку. В качестве аргумента можно было бы передать коллекцию строк, но при этом упускается другая возможность: заново абстрагировать предметную область другой системы. Необходимо обогатить нашу модель знанием о том, что существуют разные категории грузов. И проработать новое понятие модели нужно совместно со специалистом в предметной области.

Иногда (как будет говориться в главе 11) идею проектного решения в модели может подсказать *аналитический шаблон* (*analysis pattern*). В [11] описывается шаблон, помогающий решить эту проблему: УЧАСТОК РАБОТ (ENTERPRISE SEGMENT). УЧАСТОК РАБОТ представляет собой набор мер и критериев для логического выделения той или иной части из прикладной деятельности, моделируемой в программе. Среди этих мер-критериев могут быть такие, которые мы уже упоминали в связи с деятельностью по доставке грузов, а также меры времени, — например, месяцы или дни. Использование этого понятия в нашей модели распределения заказов придает модели больше выразительности и упрощает интерфейс. Класс с именем **Участок работ (Enterprise Segment)** появится в нашей модели и программной архитектуре как дополнительный ОБЪЕКТ-ЗНАЧЕНИЕ, который нужно будет создать для каждого **Груза**.

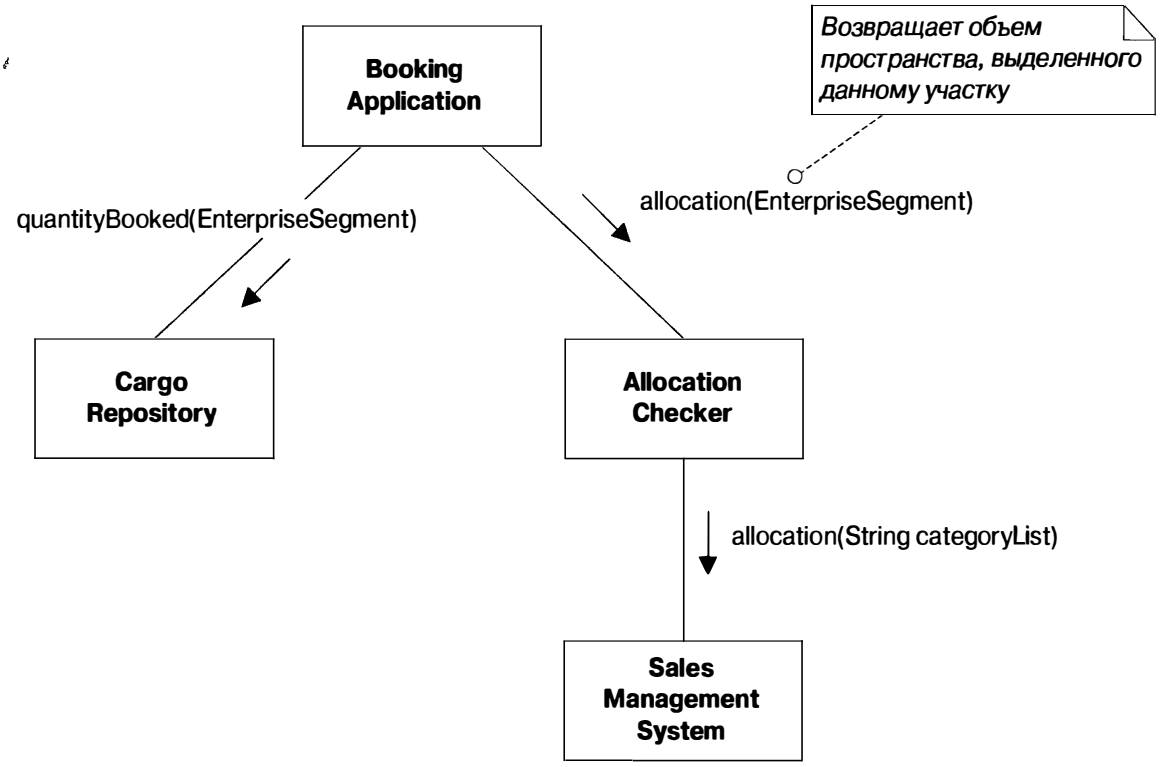


Рис. 7.10. ***Контролер распределения (Allocation Checker)*** *выступает в роли* ПРЕДОХРАНИТЕЛЬНОГО УРОВНЯ *(ANTICORRUPTION LAYER), обеспечивая избирательный интерфейс к* ***Системе управления поставками (Sales Management System)*** *в терминах нашей модели*

Контролер распределения обеспечивает трансляцию между **Участками работ** и именами категорий во внешней системе. Соответственно, **Хранилище грузов** должно также обслуживать запросы по **Участкам работ**. В обоих случаях объект **Участок работ** помогает выполнять эти операции, не нарушая его инкапсуляции и не усложняя их собственной реализации. (Следует помнить, что **Хранилище грузов** выдает в ответ на запрос результат подсчета, а не коллекцию экземпляров.)

В этой архитектуре все еще есть несколько недостатков.

1. Мы передали **Службе резервирования** работу по применению следующего правила (делового регламента): "Заказ на доставку **Груза** принимается, если выделенное для его **Участка работ** пространство больше, чем уже заказанный объем плюс объем нового **Груза**". Но следить за исполнением делового регламента — это обязанность уровня предметной области, которую не следует выносить на операционный уровень.
2. Неясно, как именно **Служба резервирования** порождает **Участок работ**.

Обе эти обязанности кажутся естественными для **Контролера распределения**. Если изменить его интерфейс, эти две СЛУЖБЫ можно разделить и тем самым добиться более четкого взаимодействия.

Единственное серьезное ограничение, накладываемое такой интеграцией, состоит в том, что **Система управления поставками (Sales Management System)** не должна использовать такие критерии-меры, которые Контролер распределения не может преобразовать в **Участки работ**. (Без применения шаблона УЧАСТОК РАБОТ аналогичное ограничение вынуждало бы систему управления поставками пользоваться только

теми критериями, которые допустимы в запросах к **Хранилищу грузов**. Этот подход имеет право на существование, но тогда функции системы управления поставками "просачиваются" в другие части предметной области. В нашей же архитектуре Хранилище грузов должно уметь работать только с **Участками работ**, а эффект изменений в системе управления поставками доходит только до **Контролера распределения**, к торый с самого начала и задумывался как ФАСАДНЫЙ ОБЪЕКТ.)

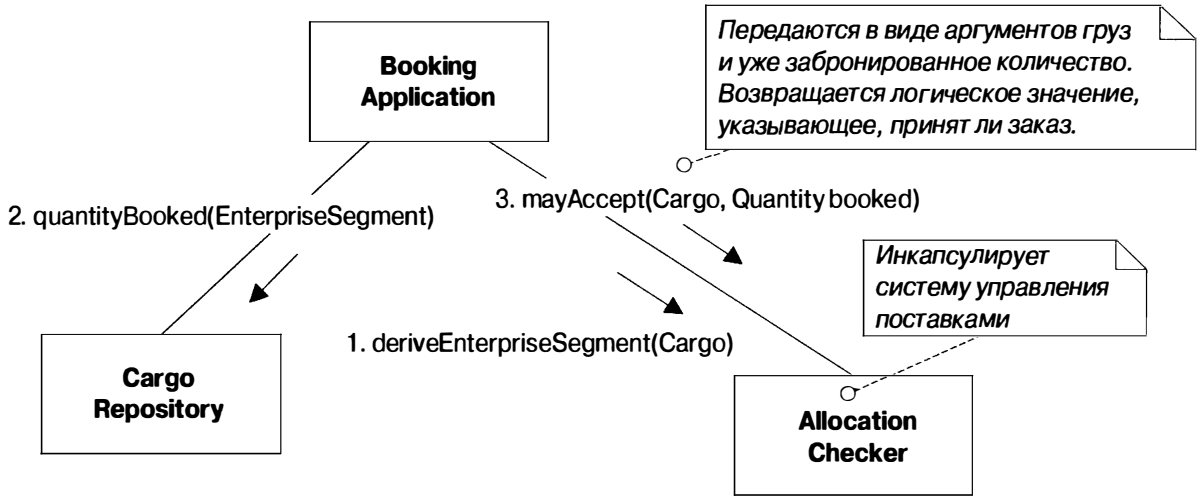


Рис. 7.11. Передача обязанностей уровня предметной области от ***Службы резервирования*** *(****Booking Application****)* ***Контролеру распределения*** *(****Allocation Checker****)*

Оптимизация быстродействия

Интерфейс **Контролера распределения** — это единственная его часть, непосредственно связанная с остальными объектами архитектуры предметной области. Но вот его внутренняя реализация предоставляет определенные возможности для решения проблем быстродействия, если таковые возникнут. Например, если **Система управления поставками** работает на другом сервере (физически находящемся в другом месте), то затраты на пересылку данных могут оказаться существенными, ведь на каждую проверку **Контролера** выполняются два обмена сообщениями. Альтернативы второму сообщению, которое требует от **Системы управления поставками** ответить на вопрос, можно или нет принять заказ на определенный груз, нет. Но первое сообщение, которое создает **Участок работ** для данного груза, основано на относительно статических данных и операциях, сравнимых с собственно операциями распределения. Одно из возможных проектных решений состоит в том, чтобы кэшировать эти данные на сервере, где расположен **Контролер распределения**, и тем самым уменьшить объем пересылки сообщений вдвое. Но эта оптимизация не проходит даром: структура программы усложняется, и теперь нужно каким-то образом синхронизировать дублирующиеся данные. Но если для распределенной системы именно быстродействие является критическим параметром, гибкость размещения данных может оказаться удачным проектным решением.

Итоги

Вот и все. Интеграция приложений могла бы превратить нашу простую, концептуально связную архитектуру в хаотическую кашу. Но с помощью предохранительного уровня, служб и разделения на участки работ нам удалось интегрировать функции **Сис-**

темы управления поставками в нашу систему резервирования заказов без нарушения стиля и структуры, а также с развитием предметной области.

И последний вопрос по проектированию архитектуры: почему бы не передать **Грузу** обязанности по созданию **Участка работ**? На первый взгляд, это решение кажется правильным: если все нужные для создания данные находятся в объекте **Груз**, то можно было бы сделать и сам объект **Участок работ** производным атрибутом **Груза**. К сожалению, не все так просто. **Участки работ** определяются так, чтобы их границы проходили удобно с точки зрения деловой стратегии предприятия. Одни и те же СУЩНОСТИ могут подразделяться на логические фрагменты по-разному в зависимости от целей деятельности. Мы выделили свой участок работ для конкретного **Груза** с целью *резервирования заказа*, но, например, с точки зрения налогового учета он мог бы занимать совершенно другой участок. Даже и те **Участки работ**, которые связаны с распределением заказов, могут со временем меняться, если **Система управления поставками** подвергнется реструктуризации, например, в связи с новой деловой стратегией. Тогда **Грузу** пришлось бы иметь определенные знания о **Контролере распределения**, что далеко выходило бы за пределы его концептуальных обязанностей и перегружало бы его методами для создания конкретных типов **Участков работ**. Поэтому обязанности по созданию таких значений естественным образом должны ложиться на тот объект, который знает правила сегментирования деятельности, а не на тот, который содержит подчиняющиеся этим правилам данные. Правила можно было бы вынести в отдельный объект "**Стратегия**" ("**Strategy**"), который можно передать в **Груз** и тем самым позволить ему создать **Участки работ**. Это решение, похоже, выходит за пределы поставленных перед нами задач, но оно могло бы стать одним из вариантов будущих проектных решений, так как не должно иметь никаких пагубных последствий.

III

Углубляющий рефакторинг

В части II этой книги была заложена основа для поддержания точного соответствия между моделью и ее программной реализацией. Использование проверенного набора базовых структурных элементов наряду с единым и последовательным языком привносит в работу программиста систематичность.

Конечно, основная трудность состоит в том, чтобы на практике построить четкую и точную модель, заключающую в себе тонкие соображения специалистов и реально помогающую в проектировании программного обеспечения. В идеале мы надеемся разработать такую модель, чтобы она способствовала глубокому пониманию предметной области. Это делается для того, чтобы программное обеспечение лучше соответствовало образу мышления специалистов и более адекватно отвечало потребностям пользователей. В этой части книги мы попытаемся четче поставить эту цель, описать процесс ее достижения и разъяснить некоторые структурные принципы и шаблоны, позволяющие адаптировать архитектуру программы как к выполнению ее задач, так и к потребностям разработчиков.

В процессе разработки полезных моделей необходимо понять три истины.

1. Создание сложных, хорошо проработанных моделей предметных областей возможно, и они стоят затраченного на них труда.
2. Практически не существует другого способа построить такую модель, кроме итерационного процесса рефакторинга с тесным взаимодействием между специалистами предметной области и программистами, желающими узнать о ней больше.
3. Реализация и успешное использование таких моделей могут потребовать высокой квалификации в области проектирования и моделирования.

Уровни рефакторинга

Рефакторинг — это такая реструктуризация программы, в результате которой не изменяются ее функциональные возможности. Вместо того чтобы планировать масштабные проектные решения, разработчики последовательно вносят в код небольшие, дискретные структурные изменения, после которых выполняемые программой функции не изменяются, а структура программы становится более понятной и гибкой. Можно сравнительно безопасно экспериментировать с кодом, если в наличии есть набор автоматизированных модульных тестов. Такая процедура избавляет разработчиков от необходимости планировать далеко наперед.

Но практически вся литература по рефакторингу посвящена механическим изменениям в коде, улучшающим удобство чтения или какие-нибудь мелкие детали. А вот методика "рефакторинга по шаблонам"[1] позволила бы переориентировать этот процесс на задачи более высокого уровня, в которых разработчик ищет возможность применить к программе общепринятые архитектурные шаблоны. И все-таки даже это — в основном технический взгляд на качество программной архитектуры.

Чтобы рефакторинг радикально повлиял на надежность и жизнеспособность системы, он должен либо мотивироваться углубленным пониманием предметной области, либо улучшать качество представления модели в результате использования программного кода. Эта разновидность рефакторинга, впрочем, не отменяет рефакторинг по архитектурным шаблонам или микрорефакторинг (и тот, и другой должны выполняться непрерывно). Наоборот, она накладывается на них как более высокий уровень рефакторинг

[1] Архитектурные шаблоны как объект рефакторинга вкратце упоминались в [Gamma et al., 1995]. Затем уже Джошуа Кериевский (Joshua Kerievsky) придал методике рефакторинга по шаблонам более зрелую и полезную форму [Kerievsky, 2003].

по углубленной модели, или углубляющий рефакторинг. Рефакторинг на основе новых знаний о предметной области часто состоит из последовательности микрорефакторингов, но побуждением к нему служит не просто желание улучшить код. Микрорефакторинг в данном случае — способ разбить процесс движения к углубленной и осмысленной модели на технически удобные стадии. Цель состоит в том, чтобы разработчик понимал не только, *что* делает код, но и *почему* код делает то, что он делает, и чтобы это понимание было вызвано постоянным общением со специалистами в предметной области.

В каталоге из книги по рефакторингу [12] перечислено большинство обычных приемов микрорефакторинга. Поводом для применения каждого из них обычно становится какая-то проблема в самом коде. В противоположность им модели предметной области подвергаются настолько разнообразным трансформациям по мере приобретения новых знаний и более глубокого понимания предмета, что полный их каталог составить попросту невозможно.

Моделирование — деятельность неструктурированная и неформальная по своей сути, как всякое исследование. Углубляющий рефакторинг[2] выполняется так, как это диктуют разработчику новые знания предметной области и новые, более глубокие выводы из этих знаний. Конечно, неплохо иметь под рукой опубликованное собрание удачных моделей (об этом говорится в главе 11), но оно может и увести в сторону, поскольку моделирование предметной области нельзя сводить к набору готовых рецептов или применению инструментов из стандартного набора. Моделирование и проектирование архитектуры программ[3] требуют творческого подхода. В следующих шести главах будут прорабатываться некоторые специфические методики для усовершенствования моделей предметных областей, а также рассматриваться программные архитектуры, на основе которых они созданы.

Углубленные модели

Традиционно методику объектного анализа объясняют так: надо найти в техническом задании на программу (т.е. документации, которая дает список требований к ней) ключевые существительные и глаголы, после чего выбрать их в качестве объектов и методов первого приближения. Это объяснение объектного моделирования будет слишком упрощенным, но для преподавания начинающим вполне сойдет. Однако, по правде говоря, модели первого приближения обычно страдают наивностью и построены на основе поверхностного знания о предмете.

Например, как-то раз я работал над программой для обслуживания грузоперевозок и предложил в качестве объектов первоначальной модели корабли и контейнеры. В самом деле, ведь корабли перемещаются с места на место, а контейнеры то ассоциируются с ними, то теряют эту ассоциацию посредством операций погрузки и разгрузки. Да, физическая работа по перемещению грузов описана довольно точно, но вот с точки зрения программного обеспечения, нужного для ведения дела, такая модель бесполезна.

---

[2] Автор употребляет выражение *refactoring toward deeper insight*, т.е. "рефакторинг в направлении, которое диктует углубляющееся понимание модели". — *Примеч. перев.*

[3] В сообществе программистов, как и в других, часто употребляется слово "дизайн" как калька с англ. *design*. Но в русском языке это слово обозначает создание красивого визуального оформления, тогда как в книге речь идет о разработке надежной, работоспособной структуры. Поэтому (возможно, к неудовольствию многих программистов) я избегаю слова "дизайн". Более точно говорить "проектирование" или даже "конструирование", но только в отношении процесса, тогда как "архитектура" обозначает его результат. — *Примеч. перев.*

Постепенно, за много итераций, через несколько месяцев совместной работы со специалистами по перевозкам, мы создали совершенно другую модель. Неспециалисту она казалась гораздо менее очевидной, но зато специалисты были довольны. Акценты сместились на деловые операции по доставке груза.

Корабли никуда не делись из модели, но теперь это были абстракции в виде "рейсов" — отдельных перевозок на корабле, поезде или другом транспортном средстве. Сам по себе корабль как объект стал второстепенным; его могли заменить в последний момент из-за необходимости ремонта или по скользящему графику, тогда как рейс проходил строго по плану. Контейнер же исчез из модели вообще. Точнее, он все-таки фигурировал в программе управления грузами в другой, более сложной форме, но в контексте исходной программы он превратился в мелкую подробность выполнения деловой операции. Физическое перемещение груза отступило на второй план по сравнению с передачей юридической ответственности за груз. А на первый план вышли менее очевидные объекты, такие как "транспортные накладные".

Когда новые моделировщики появлялись в проекте, каковы были их первые предложения? Да все те же: надо добавить недостающие классы "корабль" и "контейнер". Но эти люди не были глупы — они просто еще не приобрели знакомство с предметом.

Углубленная модель дает яркое представление о наиболее важных идеях, знаниях, характерных для специалистов, в то же время отбрасывая второстепенные, поверхностные аспекты предметной области. В этом определении не упоминается абстрагирование. Обычно в углубленной модели есть абстрактные элементы, но могут присутствовать и очень даже конкретные, если они бьют в самую суть проблемы.

Модель, действительно соответствующая духу предметной области, отличается всесторонностью, простотой и наглядностью. Одной из особенностей такой модели почти всегда является простой, хотя и достаточно абстрактный, общий язык, употребление которого нравится специалистам.

Углубленная модель и гибкая архитектура

В процессе постоянного рефакторинга от архитектуры программы требуется поддержка возможности изменений. В главе 10 изучается вопрос, как максимально приспособить программную архитектуру для внесения изменений, а также интегрирования с другими частями системы.

Легкость в использовании и удобство внесения изменений связаны с определенными характерными особенностями программной архитектуры. Они не особенно сложны, но требуют тщательной проработки. "Гибкая архитектура" и способы ее построения как раз и составляют предмет главы 10.

Нельзя не отметить большую удачу: само пошаговое преобразование модели и кода (при условии, что каждый шаг отражает новое знание и новые выводы) дает нам нужную свободу маневра и простые способы выполнять нужные операции именно в те моменты, когда изменения особенно необходимы. Хорошая перчатка проявляет гибкость там, где должны гнуться пальцы, а в остальных местах она жесткая и защищает руку. Поэтому хотя в такой методике моделирования и проектирования многое делается методом проб и ошибок, вносить изменения со временем становится удобнее, и повторяющиеся изменения "двигают" нас к гибкой архитектуре.

Кроме удобства внесения изменений в код, гибкая архитектура способствует еще и усовершенствованию самой модели. ПРОЕКТИРОВАНИЕ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN) стоит на двух "столпах": пусть углубленная модель делает возможной выразительную архитектуру, но ведь и сама архитектура помогает в развитии предметного зна-

ния, заключенного в модели, если достаточная ее гибкость позволяет разработчику экспериментировать, а достаточная наглядность — ясно видеть, что происходит. Эта вторая половина цикла обратной связи весьма важна, поскольку нужная нам модель — это не просто красивый набор идей, а основа работы программной системы.

Процесс познания

Чтобы построить программную архитектуру, хорошо приспособленную к решению прикладной задачи, вначале следует создать модель, которая бы выражала основные понятия (концепции) предметной области. Активный поиск этих концепций и внедрение их в архитектуру составляют тему главы 9.

Из-за тесной связи между моделью и архитектурой процесс моделирования затягивается, когда становится трудно выполнять рефакторинг над кодом. В главе 10 обсуждается вопрос, как правильно писать программы, которые другим разработчикам (да и себе самому тоже) было бы легко дорабатывать и расширять их функциональные возможности. Такая работа всегда сопровождается усовершенствованием модели, поэтому часто требует сложных методов проектирования архитектуры и более строгого определения основных понятий.

Для выстраивания нового знания и новых понятий в виде модели обычно приходится полагаться на творческие способности, а также метод проб и ошибок. Но иногда оказывается, что можно следовать шаблонному образцу, который кто-то до вас уже открыл и описал. В главах 11 и 12 рассматривается применение "аналитических шаблонов" (*analysis patterns*) и "архитектурных [проектных] шаблонов" (*design patterns*)[4]. Такие шаблоны — это не готовые решения, но с их помощью можно сузить область поиска и более продуктивно осуществить переработку знания.

Но все же я начну часть III с самого волнующего события во всем предметно-ориентированном проектировании. Бывает так, что как только выстроена хорошо спроектированная по модели архитектура и подобран хороший набор четких понятий, происходит качественный скачок — открывается возможность превратить программу в нечто более выразительное и разностороннее, чем могло ожидаться ранее. Это может означать как добавление новых функциональных возможностей, так и замену большой порции негибкого, маловыразительного кода на простое и гибкое выражение углубленной модели предметной области. Такие скачки, конечно, случаются не каждый день. Но они настолько ценны сами по себе, что упускать их нельзя — нужно суметь распознать потенциальную возможность такого скачка и в полной мере ее реализовать.

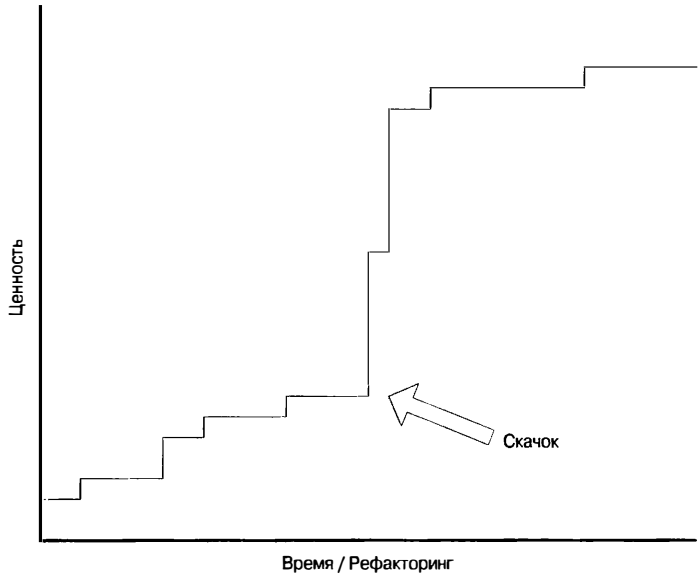
В главе 8 рассказывается подлинная история проекта, в котором процесс рефакторинга по углубленной модели привел к подобному скачку. Такое событие заранее спланировать нельзя, но, тем не менее, это хороший повод задуматься о вопросах рефакторинга предметной области.

---

[4] К сожалению, это выражение уже получило в русском языке неточный перевод-кальку "шаблоны проектирования". Между тем неплохо передает суть явления выражение "архитектурный образец". Здесь же используется противоположный смысл слов *analysis* (анализ) и *design* (проектирование, разработка) как двух взаимодополняющих видов деятельности: аналитической и синтетической. *Примеч. перев.*

Глава 8

Качественный скачок



Распределение выгоды от рефакторинга имеет нелинейный характер. Как правило, если приложенные усилия невелики, то и результат будет скромным. Небольшие усовершенствования имеют тенденцию накапливаться, бороться с энтропией и не давать программе скатиться в состояние окаменелой древности. Но все же наиболее важные качественные улучшения происходят скачкообразно, и ударная волна от них распространяется по всему проекту.

Медленно, но уверенно разработчики ассимилируют знание и перерабатывают его в компактную форму модели. Углубленные модели могут возникать постепенно, после серии небольших операций рефакторинга, каждый раз выполняемых над одним объектом: здесь откорректировали ассоциацию между тем и этим, там передали обязанности оттуда туда.

Но бывает, что постепенный рефакторинг открывает дорогу чему-то новому совсем не так систематично. Каждое усовершенствование кода и модели открывает разработчику глаза на что-то новое. А ясность восприятия создает потенциальную возможность для качественно нового вывода или идеи. И поток модификаций вдруг приводит к модели, которая соответствует реальности и приоритетам пользователя на более глубоком уровне, чем ранее. Универсальность и наглядность внезапно возрастают, а сложность при этом куда-то испаряется.

Такого рода скачок — это не стандартный прием. Это событие. Трудность заключается в том, чтобы понять, что же происходит и что в связи с этим делать. Чтобы донести до читателя, на что похоже это ощущение, я расскажу реальную историю о проекте, над которым я работал несколько лет назад, — историю о том, как нам удалось построить очень ценную и глубокую модель.

История успеха

Просидев над рефакторингом долгую нью-йоркскую зиму, мы пришли к модели, в которой заключались некоторые основные знания о предметной области, а также к архитектуре, которая более-менее давала возможность работать. А работали мы над центральной частью большой программы по управлению синдицированными кредитами в инвестиционном банке.

Если компания Intel хочет построить завод стоимостью миллиард долларов, ей для этого потребуется слишком большой кредит, который не сможет выдать в одиночку ни одна кредитная организация. Поэтому кредиторы формируют *синдикат*, объединяющий их ресурсы для совместной поддержки *предприятия* (см. примечание ниже). Инвестиционный банк обычно главенствует в таком синдикате, координируя финансовые транзакции и оказывая другие услуги. Наш проект заключался в разработке программного обеспечения для обслуживания этого процесса.

Что такое предприятие

"Предприятие" в данном контексте — это не заводские помещения с оборудованием. В этом проекте, как и во многих других, мы обогатили словарь нашего ЕДИНОГО ЯЗЫКА специализированной терминологией из языка специалистов по предметной области. В мире коммерческих банков *предприятием* (*facility*) именуется *обязательство фирмы предоставлять заемные средства*. Даже ваша кредитная карточка является предприятием, дающим вам право по первому требованию брать в долг средства (не более предельно допустимой суммы) под заранее оговоренный процент. Пользуясь карточкой, вы берете кредит и создаете задолженность; каждый новый случай оплаты карточкой — это *выборка средств* (*drawdown*) вашим предприятием, увеличивающая задолженность. Затем вы выплачиваете основную сумму кредита; возможна также выплата ежегодной платы за пользование карточкой. Эта плата взимается с вас за предоставление привилегии, т.е. самой карточки (предприятия), и поэтому не зависит от размера задолженности.

Модель неплоха, но...

Поначалу мы были настроены оптимистично. За четыре месяца до того на нас свалилась беда в виде унаследованной нами совершенно нерабочей базы кода, но мы реконструировали ее в архитектуру, основанную на модели.

Модель, показанная на рис. 8.1, представляет основной случай в очень простом виде. Здесь **`Вклад в кредит (Loan Investment)`** — производный объект, который представляет долевой вклад конкретного инвестора в общий **`Кредит (Loan)`**, пропорциональный его доле в **`Предприятии (Facility)`**.

Но были и тревожные признаки. Мы постоянно "спотыкались" о неожиданные требования, которые усложняли структуру программы. Вот один из главных примеров: постепенно до нас дошло, что доли в **`Предприятии`** имеют только *рекомендательный характер* по участию в любых выборках средств по кредиту.

Когда заемщик запрашивает свои деньги, глава синдиката может востребовать соответствующие доли у любых его членов. После такого требования член синдиката обычно выкладывает нужную сумму, но он может и вступить в переговоры с другим членом синдиката о том, чтобы вложить меньшую (или большую) сумму. Мы приспособились к этому обстоятельству, введя в модель **`Кредитные поправки (Loan Adjustments)`**.

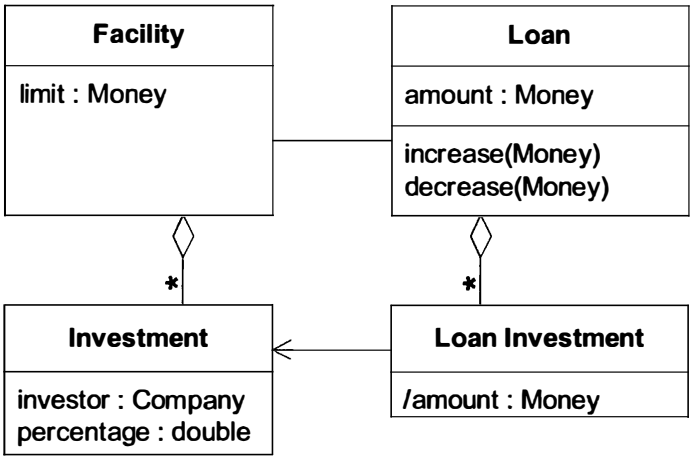


Рис. 8.1. Модель с фиксированными долями кредиторов

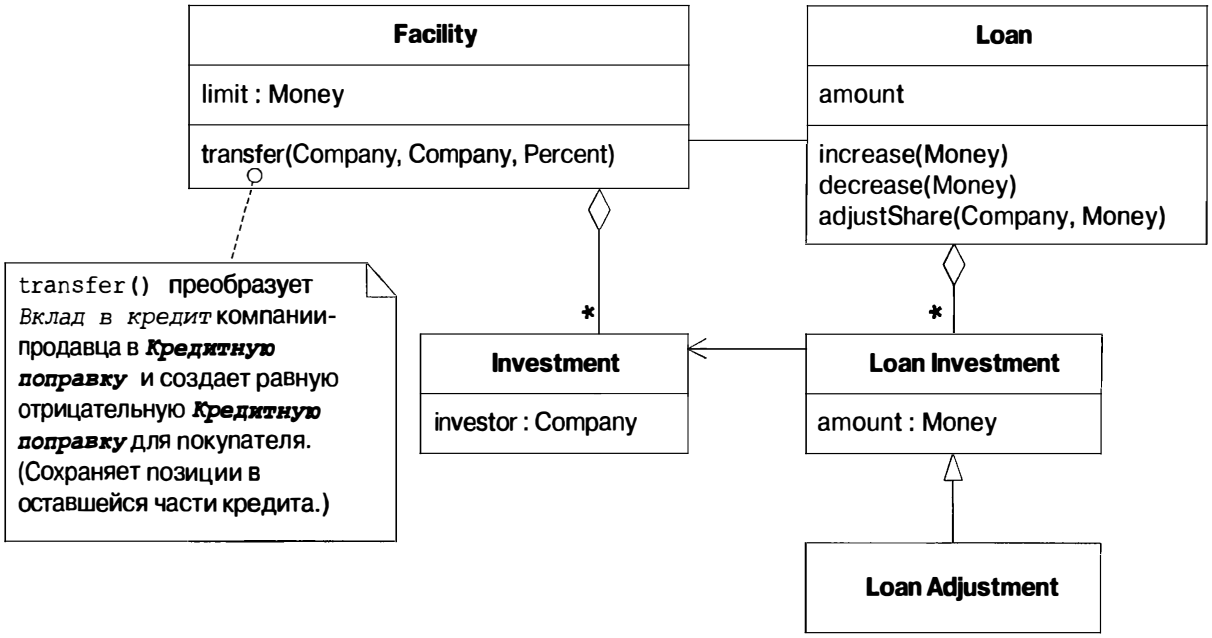


Рис. 8.2. Последовательные поправки к модели для решения проблем. ***Кредитные поправки (Loan Adjustments)*** *описывают вычеты из доли, которую кредитор изначально соглашался предоставить* ***Предприятию (Facility)***

Усовершенствования такого рода позволяли нам решать проблемы в текущем режиме, по мере прояснения правил, по которым выполнялись различные транзакции. Но сложность архитектуры возрастала, а быстро выйти на надежную, устойчивую функциональность не получалось.

Еще больше нас беспокоили небольшие погрешности округления, которые никак не удавалось "подавить" все более сложными алгоритмами. Конечно, в стомиллионной сделке никого не волнует, куда девается несколько лишних центов, но банкиры склонны не доверять программам, которые не умеют скрупулезно учитывать такие центы. Мы начали подозревать, что наши трудности — это только симптомы фундаментальных проблем с архитектурой.

Скачок

Наконец на нас снизошло озарение и мы поняли, что же было не так. В нашей модели **Предприятие (Facility)** и доли **Кредита (Loan)** были связаны таким образом, который *не соответствовал реальному положению дел.* Это открытие имело широкий резонанс. С помощью специалистов по банковскому делу, под их одобрительные кивки и — чего греха таить — вопросы, почему мы так долго возились, мы набросали новую модель на доске. Хотя ее детали еще не выкристаллизировались, мы уже знали, какая новая особенность является критически важной для модели: доли в **Кредите** и доли в **Предприятии** могли изменяться независимо одна от другой. Сделав этот фундаментальный вывод, мы приступили к прохождению многочисленных сценариев, используя наглядное представление новой модели, которое выглядело примерно так.

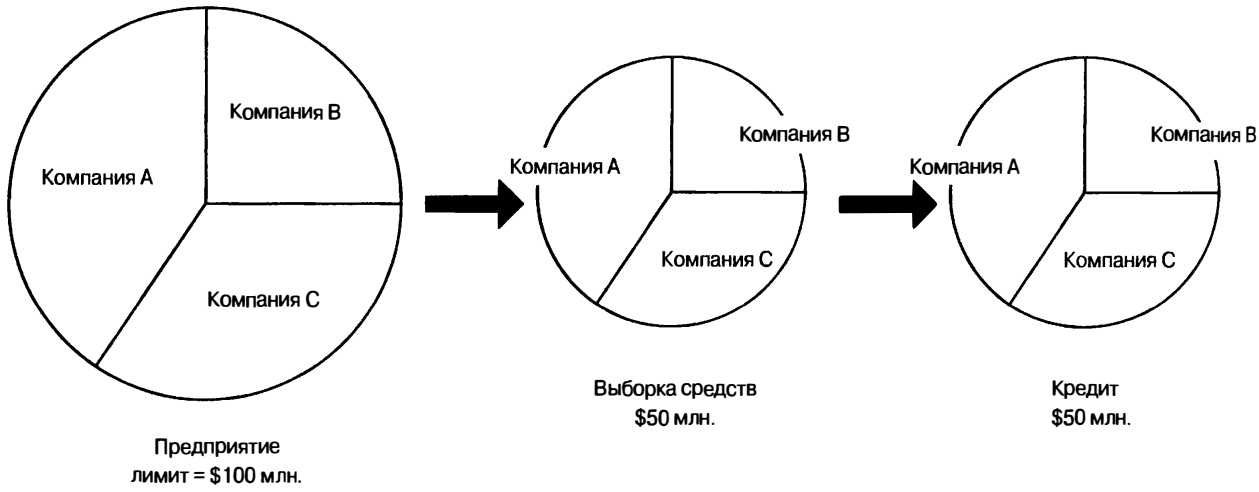


*Рис. 8.3. Распределение выборки средств на основе долей в **Предприятии (Facility)***

На рис. 8.3 показано, что заемщик решил получить начальную сумму в 50 млн. долларов из тех 100 млн., которые ему обещаны в рамках данного **Предприятия**. Трое кредиторов делят этот взнос в долях, пропорциональных их заявленным долям в **Предприятии**. В итоге пятидесятимиллионный **Кредит** выдается кредиторами в показанной пропорции.

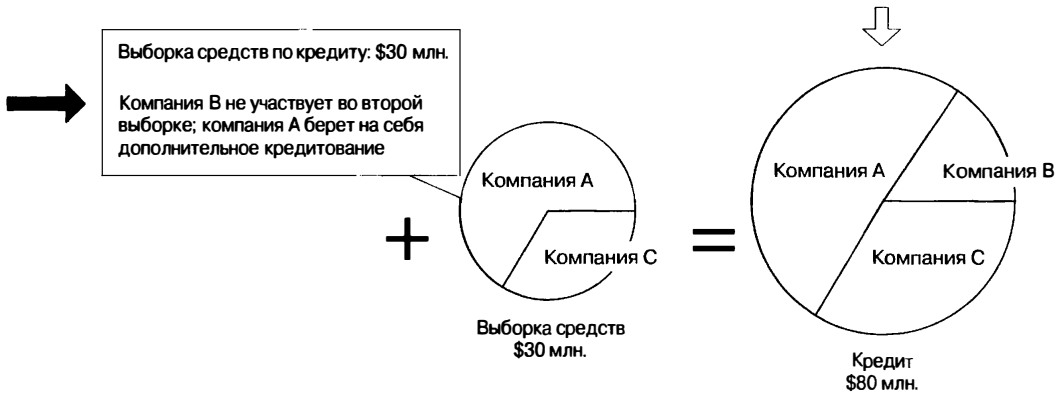


Рис. 8.4. Кредитор B не участвует во второй выборке средств

Далее, на рис. 8.4 заемщик получает еще 30 млн. долларов, доводя освоенный им **Кредит** до 80 млн.; это все еще меньше, чем кредитный лимит **Предприятия** в 100 млн. На этот раз компания B решает не участвовать в выборке средств по кредиту и предоставляет компании A покрыть дополнительную долю. Эти инвестиционные решения от-

ражены в долевом соотношении выборки средств. После того как вторая выборка добавляется к **Кредиту**, его доли перестают быть пропорциональными долями в **Предприятии**. Это обычная ситуация.

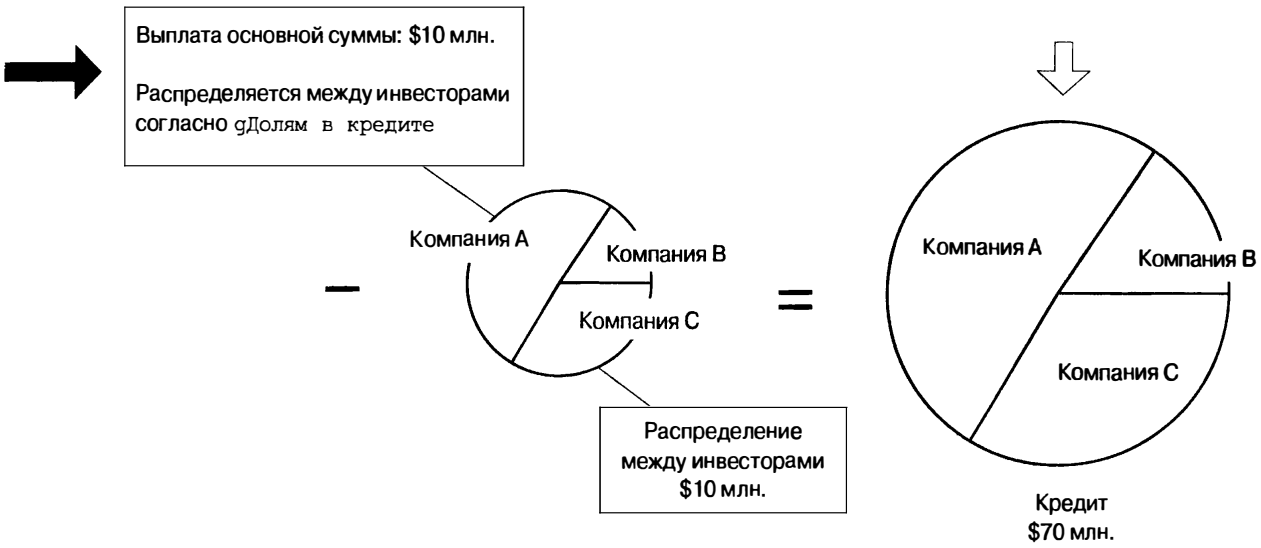


Рис. 8.5. Выплаты основной суммы всегда распределяются пропорционально долям в фактическом ***Кредите***

Когда заемщик выплачивает **Кредит**, возвращенные средства распределяются между кредиторами в соответствии с их долями в фактическом **Кредите**, а не в **Предприятии**. Аналогично, и процентные выплаты будут делиться между кредиторами пропорционально долям в **Кредите**.

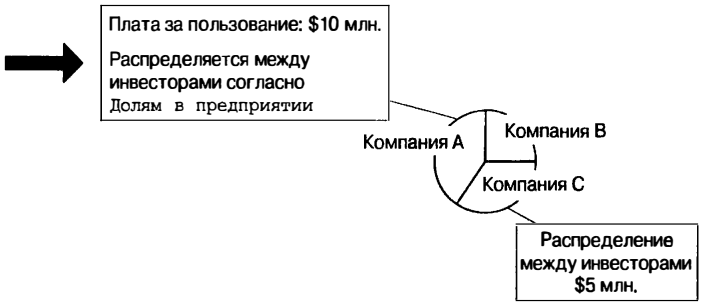


Рис. 8.6. Плата за пользование всегда распределяется пропорционально долям в ***Предприятии (Facility)***

С другой стороны, когда заемщик вносит плату за привилегию иметь доступ к **Предприятию**, эти средства делятся пропорционально долям кредиторов в **Предприятии** независимо от того, кто фактически дал деньги в кредит. Сам **Кредит** этими выплатами не изменяется. Существуют даже сценарии, в которых кредиторы перераспределяют плату за пользование независимо от процентных выплат.

Углубленная модель

Мы сделали два глубоких вывода относительно предметной области. Первый состоял в том, что наши "Вклады" и "Вклады в кредит" — это просто частные случаи более общего и фундаментального понятия *доли* (*share*). Доли в предприятии, доли в кредите, доли в *распределении выплат* везде доли. На доли делится все, что может быть поделено.

Спустя несколько хлопотных дней я набросал модель долевого участия, пользуясь для нее тем языком, на котором мы разговаривали со специалистами, и основываясь на проработанных совместно с ними сценариях.

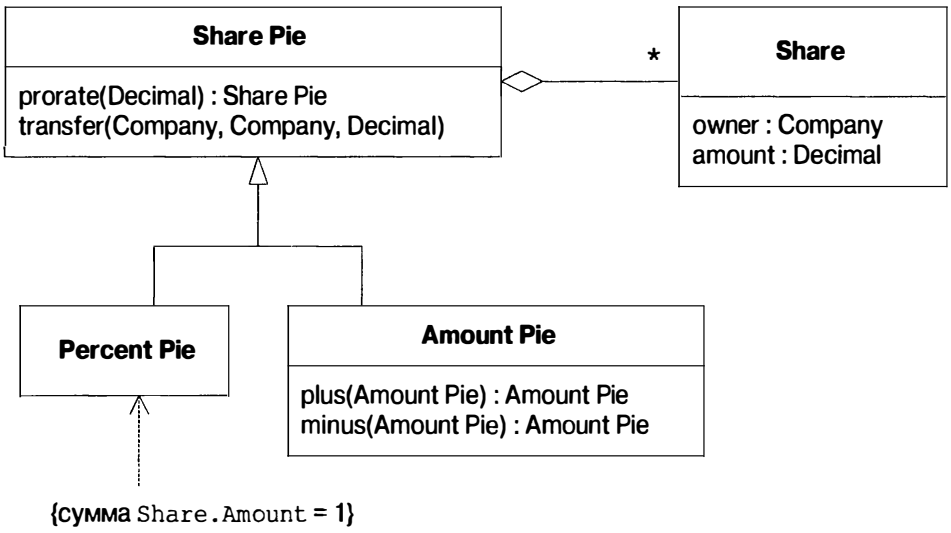


Рис. 8.7. Абстрактная модель долевого участия

В дополнение к ней я построил и примерную модель кредита.

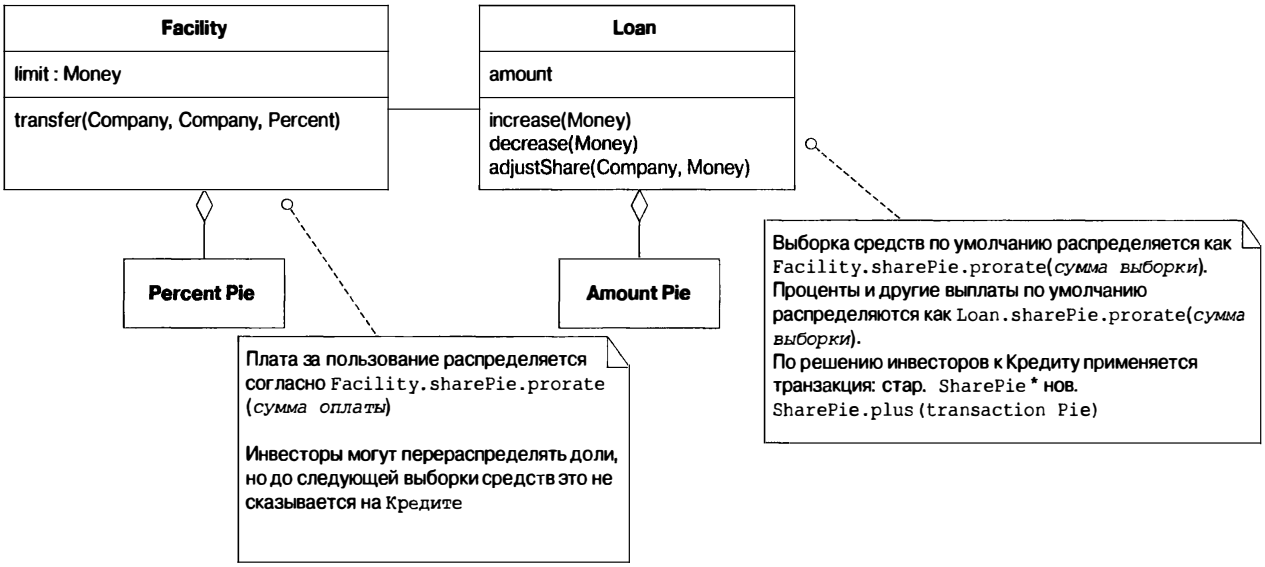


Рис. 8.8. Модель кредита с использованием ***Распределения долей (Share Pie)***

В этих моделях больше не было специализированных объектов для долей **Предприятия** или **Кредита**. И те, и другие сведены в более наглядное **Распределение долей (Share Pie)**[1]. Такое обобщение позволяет ввести общую "арифметику долей", что очень сильно упрощает вычисление долей в любой транзакции и делает подобные расчеты более выразительными, лаконичными и легко комбинируемыми.

Но главное, из-за чего исчезли все предыдущие проблемы, — это устранение неуместного ограничения в модели. **Доли** в **Кредите** больше не были привязаны к **Долям**

[1] *Pie* (дословно "пирог") означает круговую диаграмму, на которой показано соотношение долей в каком-либо общем количестве. — *Примеч. перев.*

Предприятии, но при этом все уместные ограничения (общая сумма, распределение платы за пользование кредитом) никуда не делись. **Распределение долей** в **Кредите** можно было регулировать напрямую, так что **Кредитные поправки** (**Loan Adjustments**) больше не требовались, и это устранило немалую часть специализированных проверок.

Надобность в понятии **Вклада в кредит** (**Loan Investment**) тоже отпала, и в этот момент до нас, наконец, дошло, что "вклад в кредит" — это вообще не банковский термин. Кстати, специалисты-банкиры уже говорили нам несколько раз, что они не понимают этого термина. Но они верили нам как программистам и предполагали, что он полезен для технических целей при написании программы. А мы-то как раз выдумали его из-за недостаточного понимания предметной области.

И вдруг оказалось, что на основе нашего нового видения предмета мы можем выполнять любой рабочий сценарий, который нам встречался, практически без усилий и гораздо проще, чем раньше. Что касается схем наших моделей, то *они стали совершенно понятны специалистам, тогда как раньше те жаловались на "излишне технические" схемы.* Даже из набросков на доске было видно, что самые сложные проблемы округления нам теперь нипочем. Это позволило мгновенно написать кое-какой сложный код для реализации округления.

Наша новая модель работала хорошо. Даже отлично.

А мы чувствовали себя отвратительно!

Трезвое решение

Разумно было бы полагать, что в этот момент нам стало легче. Но это было не так. Для проекта были установлены жесткие временные рамки, и мы уже находились в цейтноте. Главным чувством, которое мы испытывали, был страх.

Догматы рефакторинга говорят нам, что двигаться нужно мелкими шажками, непрерывно сохраняя работоспособность кода. Но для рефакторинга нашего кода по новой модели потребовалось бы изменить очень многое во вспомогательных частях кода, а работоспособных промежуточных состояний программы между началом и концом этого процесса практически не было. Мы видели кое-какие небольшие усовершенствования, которые можно было бы внести, но ни одно из них не приближало нас к новой концептуальной модели. Мы видели последовательность шагов, которые привели бы нас к ней, но по ходу дела целые фрагменты приложения временно перестали бы работать. Это было время, когда автоматизированное тестирование еще не вошло в обиход в таких проектах. У нас такого тестирования не было, и непредвиденные сбои были неизбежны. Ко всему прочему, требовались новые усилия, предстояли новые объемы работ, а мы уже устали от месяцев мышиной возни.

Именно в это время у нас состоялась встреча с руководителем проекта, которую я никогда не забуду. Наш руководитель был человек умный и решительный. Он задал нам несколько вопросов.

В.: Сколько потребуется времени, чтобы вернуться к полному текущему набору рабочих функций, но в новой архитектуре?

О.: Примерно три недели.

В.: Можно ли решить наши проблемы без этого?

О.: Возможно, но никакой гарантии дать нельзя.

В.: Можно ли будет продвинуться дальше в следующей версии программы, если не сделать этого сейчас?

О.: Без этих изменений движение вперед сильно замедлится. И вносить изменения станет намного труднее, как только мы установим всю базу кода.

В.: Уверены ли *вы лично*, что все переделать будет правильно?

О.: Мы знаем, что все висит на волоске, и справимся по-старому, если придется. К тому же мы устали. Но ответ будет — *да, это более простое решение, которое гораздо лучше соответствует цели проекта.* Если брать дальнюю перспективу, то сделать все по-новому менее рискованно.

Тогда руководитель дал нам "добро" и сказал, что организационные вопросы он решит. Я неизменно восхищаюсь той смелостью и доверием к нам, с которыми он принял это решение. Мы закусили удила и сделали все за три недели. Работы было очень много, но прошло все на удивление гладко.

Воздаяние

В проекте прекратились загадочные и неожиданные изменения в требованиях к программе. Реализация операций округления, не очень-то простая сама по себе, стала стабильной и осмысленной. Мы выпустили первую версию и не видели препятствий на пути ко второй. А мой нервный срыв, хотя и маячил совсем рядом, но все-таки не состоялся.

По мере разработки второй версии **`Распределение долей (Share Pie)`** стало унифицирующим фактором всего приложения. Как программисты, так и специалисты по финансам пользовались этим понятием при обсуждении системы. *Маркетологи с его помощью объясняли потенциальным клиентам возможности программы.* А клиенты "подхватывали" понятие вслед за ними и включались в обсуждение программы. Так оно вошло в ЕДИНЫЙ ЯЗЫК проекта, поскольку выражало самую суть такого явления, как синдицированные банковские кредиты.

Потенциал

Когда появляется перспектива качественного скачка к углубленной модели, она часто кажется пугающей. Столь решительное изменение имеет больший потенциал, но оно и более рискованно, чем большинство методов рефакторинга. Да и время для него может быть неподходящим.

Как бы нам этого ни хотелось, прогресс не идет плавно и последовательно. Переход к действительно глубокой по смыслу модели — это огромный качественный сдвиг в мышлении, который требует радикального изменения архитектуры. Во многих программных проектах самые важные достижения в модели и архитектуре связаны именно с такими скачками.

Концентрация на основах

Не пытайтесь насильно вызвать качественный скачок, это только парализует работу. Потенциал для скачка обычно возникает после ряда небольших рефакторинговых модификаций. Большую часть работы занимают мелкие фрагментарные изменения, и понимание модели тоже растет постепенно в ходе каждого последовательного улучшения.

Чтобы подготовить почву для скачка, сосредоточьтесь на переработке знаний и культивировании надежного ЕДИНОГО ЯЗЫКА. Нащупывайте важнейшие понятия предметной области и выражайте их в модели в явном виде (см. главу 9). Улучшайте архитектуру программы, делая ее более гибкой (см. главу 10). Выполняйте дистилляцию модели (см. главу 10). Работайте с этими верными средствами, улучшайте понимание и наглядность — все это обычно предшествует качественному скачку.

Не чурайтесь скромных, небольших изменений, которые постепенно улучшают модель, даже если они ограничены одной и той же общей концептуальной средой. Не стопорите работу, заглядывая слишком далеко вперед — просто не теряйте бдительности, чтобы не упустить шанс.

Каскад озарений

Сделанный нами качественный скачок прекратил наши бесплодные мучения, но это был еще не конец. Углубленная модель открыла нам неожиданные перспективы по обогащению возможностей программы и улучшению ее архитектуры.

Буквально через несколько недель после выпуска версии программы с **Распределением долей (Share Pie)** мы заметили в модели еще один "неуклюжий" элемент, из-за которого усложнялась программная реализация. Не хватало важного объекта-СУЩНОСТИ, отсутствие которого вынуждало нас раздавать его обязанности другим объектам. В предметной области существовали важные правила, регулировавшие выборку кредита, оплату пользования им и т.п. Вся эта система правил (деловых регламентов) была втиснута в разные методы объектов **Предприятие (Facility)** и **Кредит (Loan)**. Пока в модели не было **Распределения долей**, эти проблемы с архитектурой практически не были заметны, однако наш новый взгляд на модель делал их очевидными. Мы начали замечать, что в обсуждениях проскакивают новые термины, которых не было в модели, — например, "транзакции" (финансовые). И вот наконец нам стало ясно, что эти термины появились из-за неявного присутствия понятия-объекта, обязанности которого были распределены между сложными методами других объектов.

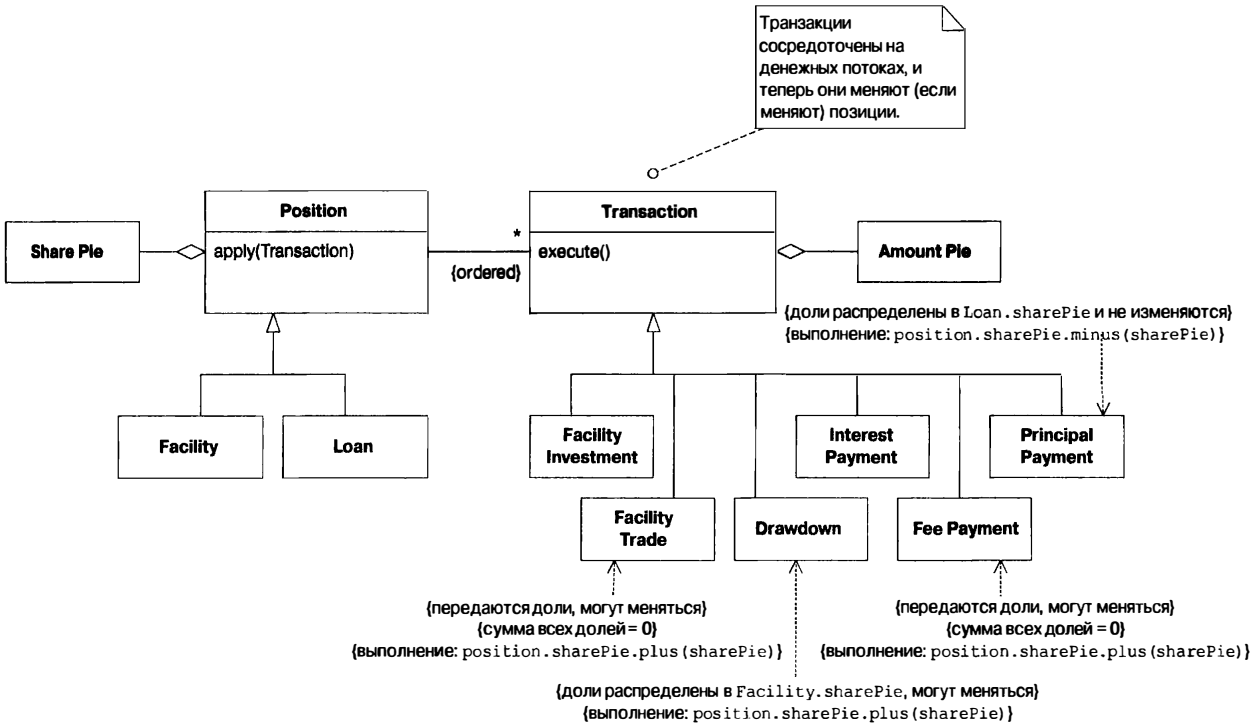


*Рис. 8.9. Еще один качественный скачок в модели, последовавший через несколько недель. Ограничения на **Транзакции (Transactions)** стали формулироваться легко и точно*

Проделав работу, аналогичную описанной выше (хотя, к счастью, и не в таком цейтноте), мы вышли на очередной уровень усовершенствования модели. В новой модели неяв-

ные понятия стали явными, приобретя различные формы **Транзакций** (**Transactions**), и в то же время упростились **Позиции** (**Positions**) — это абстракции, объединяющие в себе **Предприятия** (**Facilities**) и **Кредиты** (**Loans**). Стало легко определять различные необходимые транзакции, а также правила их выполнения, процедуры переговоров и утверждения; все это выражалось в сравнительно наглядном, доступном для понимания коде.

Как это часто бывает после настоящего качественного скачка к усовершенствованной модели, простота и наглядность новой архитектуры в сочетании с улучшенным качеством коммуникации, благодаря новому ЕДИНОМУ ЯЗЫКУ опять привели к еще одному качественному скачку в моделировании.

В результате мы ускорили темп разработки программы на том этапе, на котором большинство проектов вместо ускорения тонет, как в болоте, в сложности и объеме всего написанного ранее.

ГЛАВА 9

Перевод неявных понятий в явные

Углубленная модель — это звучит здорово, но как ее построить? Углубленная модель обладает потенциалом, потому что содержит основные понятия и абстракции, которые в сжатой и гибкой форме выражают важные знания о деятельности пользователя, его проблемах и методах решения этих проблем. Первым шагом на пути к такой модели будет представление ключевых понятий предметной области в виде, пригодном для моделирования. Усовершенствование путем последовательных приближений, переработки знаний и рефакторинга — все это будет позже. Но процесс только тогда получает настоящий толчок, когда удается выявить, определить и включить как в модель, так и в архитектуру важное понятие (концепцию) из предметной области.

Многие преобразования моделей предметных областей и соответствующего кода удается осуществить только после того, как разработчики выявляют понятие, косвенно фигурировавшее в обсуждениях или неявно присутствовавшее в архитектуре, а затем дают ему явное представление в модели через один или несколько объектов или взаимосвязей.

Бывает так, что перевод неявного понятия в явное представляет собой качественный скачок, приводящий к углубленной модели. Впрочем, чаще такой скачок происходит позже, после выявления и включения в модель определенного количества важных понятий, после нескольких процедур рефакторинга для корректировки обязанностей объектов, их взаимосвязей с другими объектами, и даже смены имен. И тут все, наконец, встает на свои места. Но начинается весь этот процесс именно с выявления неявно присутствующих в прикладной области важных понятий в какой-то форме, пусть самой грубой.

Извлечение понятий

Разработчикам поневоле приходится вырабатывать чувствительность к признакам, за которыми скрывается наличие неявных понятий, а иногда даже активно выискивать таковые. Большинство открытий этого рода происходит благодаря тому, что разработчики прислушиваются к употребляемому в группе языку, критически изучают “неуклюжие” места в коде и кажущиеся противоречия в утверждениях специалистов, перелопачивают литературу по предметной области, и к тому же очень много экспериментируют.

Внимание к языку

Возможно, вы вспомните подобное из своей практики: пользователи в своем отчете постоянно упоминают нечто, собранное из атрибутов разных объектов и даже из результатов прямого запроса к базе данных. Такой же набор данных собирает одна из частей вашего приложения для вывода отчета, представления каких-то данных, генерирования

какой-нибудь сводки. Но потребность в отдельном объекте для всего этого у вас никогда не возникала. Возможно, вы даже не понимали смысла, который пользователи вкладывают в какой-то термин, и не осознавали его значимости.

А потом на вас снисходит озарение. Название той совокупности данных, которая упоминалась в отчете, оказывается, обозначает важное понятие предметной области. В крайнем возбуждении вы делитесь со специалистами своими открытиями. Они, соответственно, испытывают облегчение, что вы, наконец, пришли к этому, или зевают от скуки, поскольку давно считали это само собой разумеющимся. В любом случае вы начинаете чертить на доске схемы моделей вместо того, чтобы показывать ваши идеи на пальцах, как делали раньше. Пользователи поправляют вас в деталях новой модели, но вы уже ясно видите, как изменилось качество обсуждения. Теперь вы с пользователями лучше понимаете друг друга, и демонстрация различных взаимодействий в модели для прохождения тех или иных рабочих сценариев стала более естественной. Язык модели предметной области стал более мощным. Выполняя над кодом рефакторинг для реализации новой модели, вы обнаруживаете, что архитектура программы стала более четкой и ясной.

Слушайте язык, на котором говорят специалисты предметной области. Есть ли у них термины, которые кратко выражают нечто сложное? Поправляют ли они вас (пусть даже тактично)? Исчезает ли у них с лиц выражение озадаченности, когда вы употребляете какую-то конкретную фразу? Все это — намеки на существование понятия, которое может оказаться полезным для модели.

Это отнюдь не старый подход "делаем все существительные объектами". Здесь вы слышите новое слово, и оно становится ориентиром; вы идете в новом направлении, общаясь и перерабатывая знание с целью выкристаллизировать четкое и полезное понятие. Когда пользователи программы или специалисты в предметной области пользуются лексиконом, которого нет в модели, — это тревожный знак. А еще более тревожный — когда термины, отсутствующие в модели, употребляют как разработчики, так и специалисты.

Впрочем, на такую ситуацию лучше смотреть как на возможность развития. Единый язык складывается из слов и выражений, которые употребляются в устной речи, в документации, на схемах модели, и даже в коде. Если какой-то термин не встречается в архитектуре программы, то появляется возможность улучшить и модель, и архитектуру, включив его в словарь.

Пример

Пропущенное понятие в модели грузопоставок

Итак, разработчики написали работающее приложение, которое могло заказать доставку груза. Дальше они приступили к разработке приложения "операционной поддержки", в задачу которого входила помощь при манипулировании нарядами-заказами на погрузку и разгрузку в начале и конце маршрута, а также при перемещениях груза с одного транспортного средства на другое.

Программа заказа доставки использовала специальную службу маршрутизации, чтобы спланировать перевозку груза. Каждый отрезок маршрута хранился в строке базы данных, где указывался идентификатор рейса (т.е. одного перемещения конкретного транспортного средства), к которому был приписан груз, место погрузки и место разгрузки.

Давайте подслушаем фрагмент разговора (в сильно сокращенном виде) между программистом (**П.**) и специалистом по грузоперевозкам (**С.**).

П. Хочу проверить, все ли нужные для операционного приложения данные находятся в таблице "заказов на доставку груза" (`cargo_bookings`).

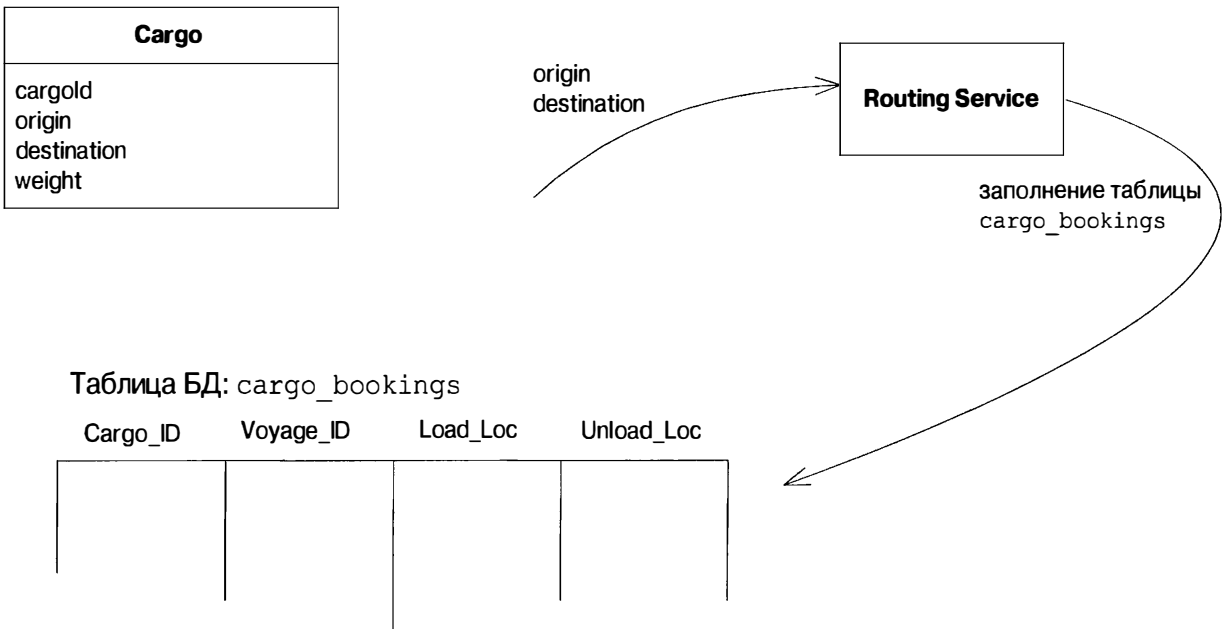


Рис. 9.1.

С. Потребуется полное описание пути следования **Груза (Cargo)**. Какая информация находится в таблице сейчас?

П. Идентификатор груза, рейс, место погрузки и место разгрузки для каждого участка пути.

С. А дата? Она понадобится операционному персоналу, чтобы договариваться об операциях перемещения груза.

П. Дату можно взять из расписания рейса. У нас нормализованная таблица.

С. Да, это нормальная практика — знать эту дату. Пути следования груза используются операционным персоналом для планирования предстоящих перемещений грузов.

П. Ну да... Ладно, доступ к датам у них точно будет. Программа управления операциями сможет выдавать полную последовательность погрузок и разгрузок с датой каждой манипуляции, т.е. то, что вы вроде бы называете "путем следования".

С. Вот и хорошо. Путь следования — это главное, что им понадобится. А, кстати, знаете, в программе заказа доставки грузов есть пункт меню, который выводит путь следования на печать или отправляет его клиенту по электронной почте. Можно этим как-то воспользоваться?

П. По-моему, это делается просто для отчета. Основой для операционного приложения это сделать нельзя.

[***Программист*** *задумывается, потом его осеняет.*]

П. Так ведь путь следования — это на самом деле связующее звено между заказом груза и операциями над ним!

С. Да, и еще это определенная связь с клиентом.

П. [*Набрасывает схему на доске.*] Это должно быть что-то в этом роде?

С. Да, в основном это правильно. Для каждого участка надо знать рейс, место погрузки и выгрузки, время.

П. Итак, как только мы создали объект **Участок пути (Leg)**, он может извлекать данные о времени из расписания рейса. Мы можем сделать объект **Путь следования (Itinerary)** нашей главной точкой контакта с операционным приложением. И еще мы можем переписать вывод отчета о пути следования так, чтобы в нем использовался этот объект. Этим мы вернем логику предметной области на место, на уровень предметной области.

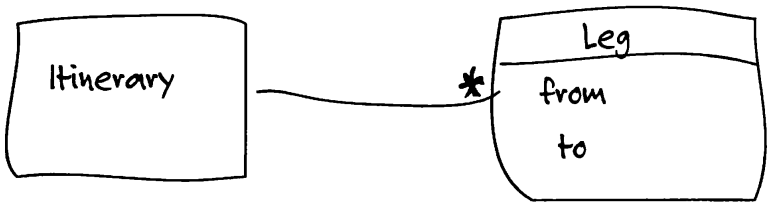


Рис. 9.2.

С. Я не до конца понял все, что вы сказали, но согласен, что два основных применения **Пути следования** — это вывод отчета при заказе и управление операциями с грузом.

П. О, кстати! Мы можем сделать так, чтобы интерфейс **Службы маршрутизации (Routing service)** возвращал объект пути следования. Тогда не надо будет помещать эти данные в таблицу базы, и системе маршрутизации вообще не нужно будет знать ничего о таблицах.

С. Как-как?

П. Я имею в виду, чтобы система маршрутизации просто возвращала **Путь следования**. Тогда его может сохранить в базе данных программа заказа доставки грузов в процессе сохранения остальных данных заказа.

С. А что, сейчас это делается не так?!

Дальше программист пошел разговаривать с другими программистами, занятыми маршрутизацией. Они оговорили изменения в модели и последствия их для архитектуры программы, по мере необходимости обращаясь к специалистам. В итоге получилась показанная на рис. 9.3 схема.

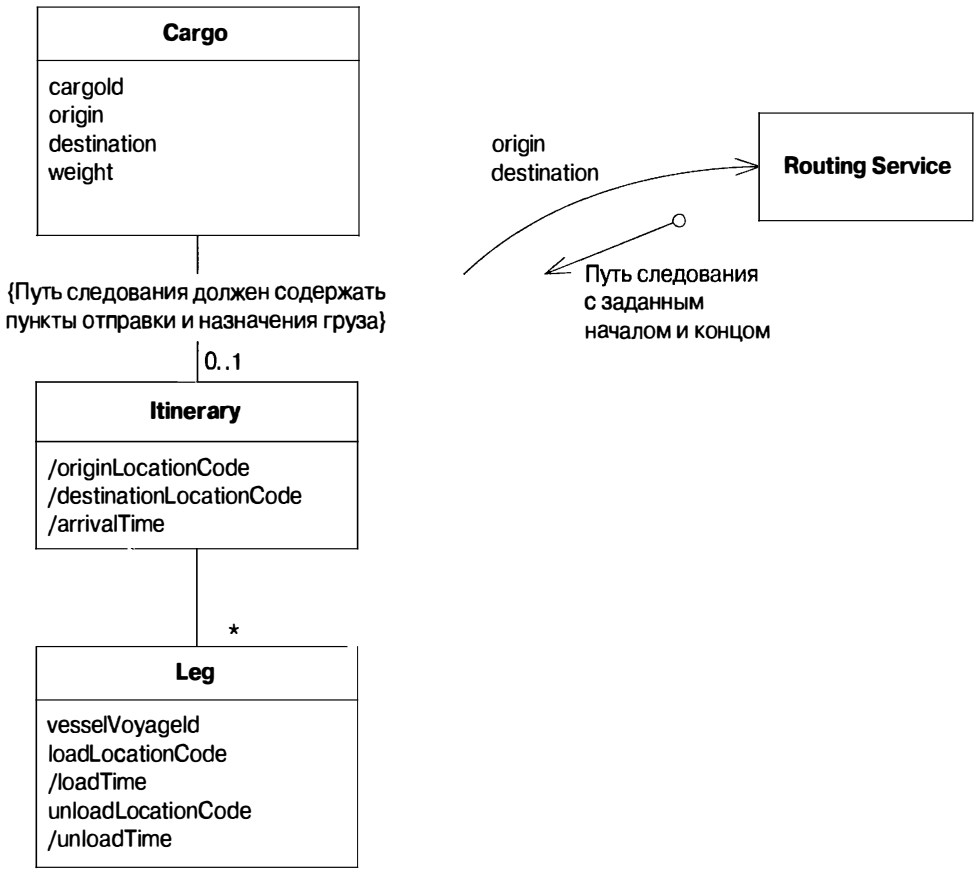


Рис. 9.3.

Потом программисты выполнили рефакторинг кода, чтобы отразить в нем новую модель. Это было проделано за два или три этапа, но быстро, в течение недели — если не считать упрощения отчета о пути следования в программе заказа на доставку, с которым справились чуть позже, на следующей неделе.

Итак, программист достаточно внимательно слушал специалиста по перевозкам, так что ему удалось заметить важность понятия "путь следования". Фактически все нужные для него данные уже собирались в программе в одно место, да и операционная часть объекта уже подразумевалась в отчете, но только введение в модель явного объекта **Путь следования** открыло новые возможности.

Преимущества рефакторинга для нового объекта **Путь следования** (**Itinerary**) таковы.

1. Более выразительное определение интерфейса **Службы маршрутизации** (**Routing Service**).
2. Независимость **Службы маршрутизации** от таблиц баз данных с заказами.
3. Более четкое оформление связи между программой заказа на доставку и программой операционной поддержки (совместное использование объекта **Путь следования**).
4. Снижение дублирования данных и операций, поскольку **Путь следования** определяет время разгрузки/погрузки как для отчета о заказе, так и для программы управления операциями с грузами.
5. Удаление предметной логики из отчета о заказах и передача ее на изолированный уровень предметной области.
6. Расширение ЕДИНОГО ЯЗЫКА, позволяющее более точное обсуждение модели и архитектуры между программистами и специалистами по перевозкам, а также между самими программистами.

Выявление узких мест

Нужные понятия не всегда сами всплывают на поверхность в дискуссиях или документации. Бывает, что приходится копать глубже и проявлять изобретательность. А копать обычно нужно в самых "неуклюжих", неудобных местах программной архитектуры — там, где процедуры делают сложные вещи, которые трудно объяснить; там, где при реализации каждого нового требования сложность постоянно возрастает.

Иногда бывает трудно даже понять, что в модели пропущено то или иное понятие. Вроде бы имеющиеся объекты справляются со всеми обязанностями, и только некоторые из них выполняются "неуклюже". Или же ощущается нехватка чего-то важного, но вот решение проблемы в модели никак не дается.

В таких случаях нужно активно привлекать к поискам специалистов по предмету. Если вам повезет, им может понравиться возня с идеями и экспериментирование с моделью. Если повезет не так сильно, идеи придется производить на свет вам, программистам, а специалисты в предметной области послужат вам для проверки: по выражениям их лиц вы поймете, испытывают ли они неудобство или проблески понимания.

Пример

Вычисление процентов: трудный способ

Следующая история происходит в гипотетической финансовой компании, которая вкладывает средства в коммерческие кредиты и другие активы, приносящие проценты.

Программа, следящая за такими вложениями и доходами от них, разрабатывалась постепенно, "прирастая" по одной рабочей функции. Каждую ночь один из компонентов запускается в пакетном режиме, вычисляет все проценты и сборы, полученные за день, и фиксирует их должным образом в программе бухгалтерского учета фирмы.

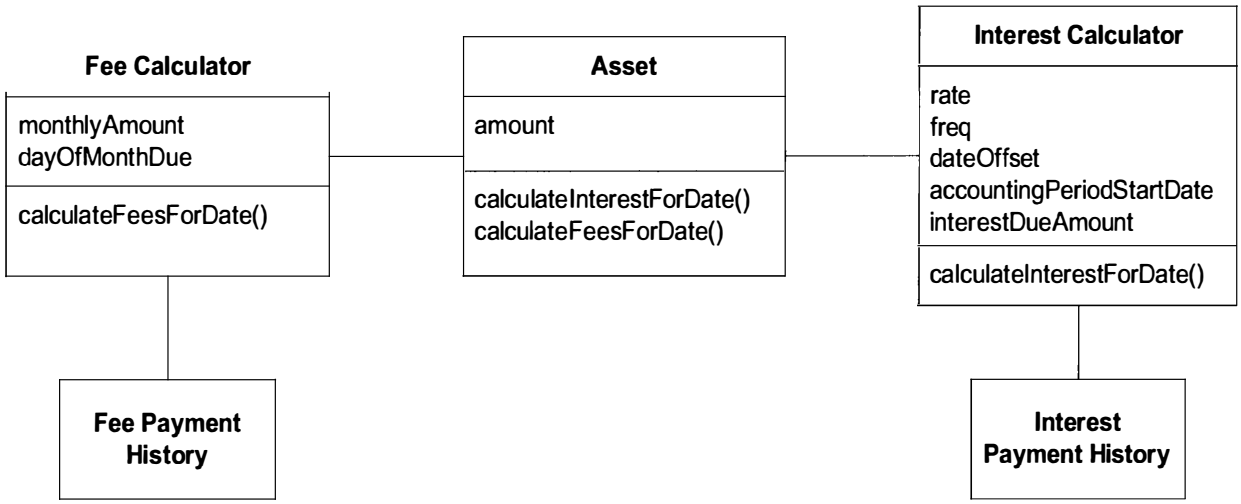


Рис. 9.4. Неудобная модель

Ночной пакетный сценарий перебирает все **Активы (Assets)** и каждому отдает распоряжение "рассчитать проценты от даты", `calculateInterestForDate()`, соответствующей этому дню. Сценарий получает вычисленное значение (доход) и передает его вместе с именем конкретной учетной книги в СЛУЖБУ, которая представляет *public*-интерфейс программы бухучета. Эта программа вносит заданную сумму в указанную книгу. Аналогичным образом сценарий перебирает все **Активы**, собирая с них полученные за день платежи, и помещает цифры в другую книгу.

Один из программистов все это время боролся[1] с возрастающей сложностью вычисления процентов. Он начал подозревать, что должна существовать модель, лучше приспособленная к этой задаче, и пригласил облюбованного им специалиста по предметной области, чтобы тот помог ему покопаться в проблемной части приложения.

Программист (П.). Наш **Калькулятор процентов (Interest Calculator)** ведет себя все хуже и хуже.

Специалист (С.). Это сложная часть работы. У нас есть много случаев, которые даже еще не реализованы.

П. Я это знаю. Мы можем добавить новые виды процентных доходов, заменив **Калькулятор процентов**. Но сейчас главная проблема состоит в другом — во всех этих особых случаях, когда проценты не платятся по графику.

С. Ничего такого "особого" в этих случаях нет. График платежей устроен очень гибко.

П. Когда в свое время мы выделили **Калькулятор процентов** из **Актива**, это очень помогло. Нам нужно и дальше разбивать его.

---

[1] У автора в примерах с программным обеспечением для учета дохода от финансовых активов фигурирует женщина-программист употребляется местоимение "она". Возможно, это следование принципу политкорректности. Но в русском переводе это создает определенные неудобства, в отличие от английского, где разница только в местоимении. А так как этот пример гипотетический, я счел возможным заменить гипотетическую женщину на гипотетического мужчину-программиста. — *Примеч. перев.*

С. Хорошо.

П. Я тут подумал: может быть, вы описываете вычисление процентов в каком-то своем стиле.

С. Что вы имеете в виду?

П. Ну, например, мы в программе отслеживаем подлежащий погашению процент, не уплаченный на протяжении расчетного периода. У вас есть для него какой-нибудь термин?

С. Мы, собственно, так не делаем. Начисленный к уплате процент и фактический платеж — это совершенно разные статьи.

П. Так вам эта цифра не нужна?

С. Ну, иногда мы можем на нее взглянуть, но вообще мы не так ведем дела.

П. Ладно, так если платеж и процент — это разные вещи, может, нам нужно и моделировать их отдельно? Что скажете, например, об этом? [*Рисует на доске.*]

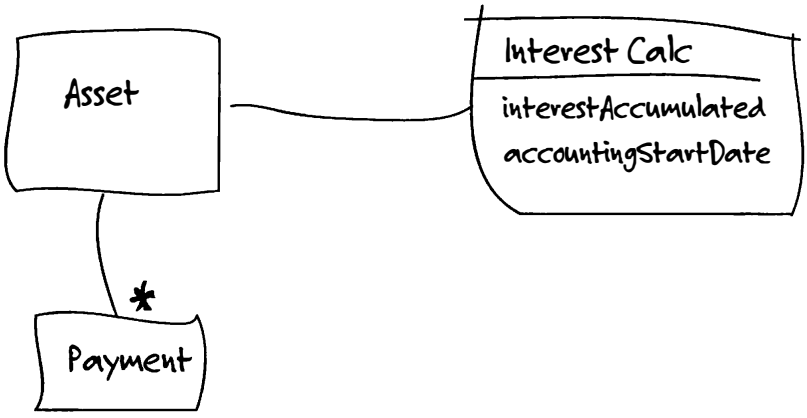


Рис. 9.5.

С. Вроде нормально. Но вы просто перенесли его из одного места в другое.

П. Да, но теперь **`Калькулятор процентов (Interest Calculator)`** отслеживает только начисленные проценты, а **`Платеж (Payment)`** содержит отдельное число. Особого упрощения это не дает, но как будто лучше отражает вашу деловую практику, разве нет?

С. А, понятно. Кстати, можно нам иметь и историю начисления процентов тоже? Аналогично **`Истории платежей (Payment History)`**.

П. Да, у нас уже запрашивали эту новую возможность. Но ее можно было бы добавить в исходную архитектуру.

С. Вот как. Что ж, когда я увидел такое разделение между **`Историей платежей`** и процентов, я подумал, что вы разделяете проценты, для того чтобы организовать их аналогично **`Истории платежей`**. Вы что-нибудь знаете об учете по методу начислений (*accrual basis accounting*)?

П. Объясните, пожалуйста.

С. Каждый день или в любой момент согласно графику у нас происходит начисление процентов, которое проводится через книгу. Платежи проводятся по-другому. То, что у вас тут нарисовано, выглядит каким-то "неуклюжим".

П. Так вы говорите, что если вести список "начислений" (*accruals*), то можно взять итог или... "провести" их по мере необходимости?

С. Провести, вероятно, по дате начисления, но взять итоговую сумму да, в любой момент. Комиссионные сборы (*fees*) устроены так же, но проводятся, конечно, через другую книгу.

П. Вообще-то, вычисление процентов упростилось бы, если выполнять его только за один день или за период. Взяли, да и посчитали все одним махом. Как насчет такой модели?

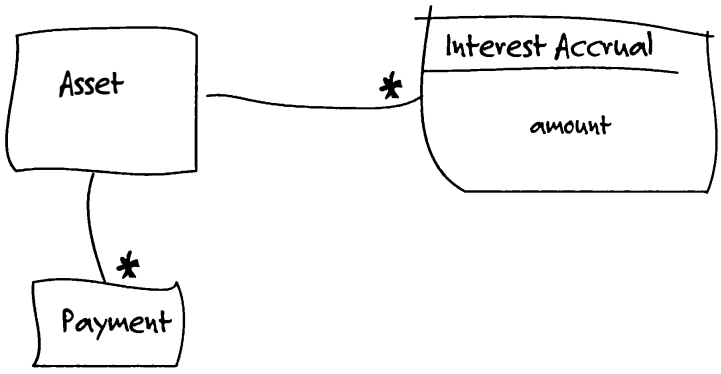


Рис. 9.6.

С. Да, конечно. Вот это похоже на правду. Не совсем понимаю, почему это будет легче для вас. Но в целом, ценность любого актива состоит в том, что по нему могут накапливаться проценты, сборы и т.д.

П. Вы сказали, комиссионные сборы устроены таким же образом? Но они... как вы там говорили... проводятся по другой книге?

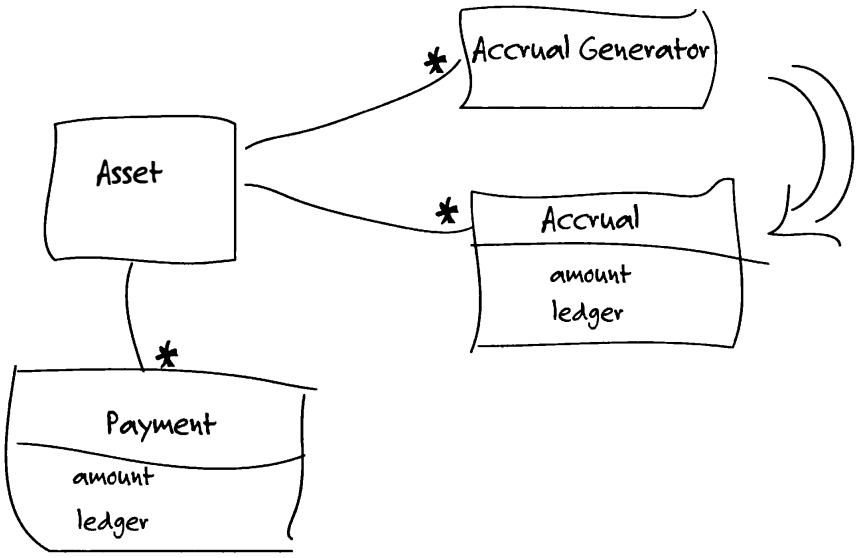


Рис. 9.7.

П. В этой модели у нас вычисление процентов или, скорее, логика расчета начислений из **Калькулятора процентов** отделена от их учета. И до сих пор я не замечал, как много дублирования в **Калькуляторе сборов (Fee Calculator)**. А теперь можно легко добавлять различные виды сборов.

С. Да, вычисления и раньше выполнялись правильно, но теперь я хорошо вижу, что происходит.

Поскольку классы **Калькуляторов** не были непосредственно связаны с другими частями архитектуры, рефакторинг оказался довольно простым в исполнении. Разработчик смог переписать модульные тесты с использованием нового языка за несколько часов, и новая архитектура заработала к концу следующего дня. В конце концов, у него получилось вот что.

В новой версии приложения (после рефакторинга) ночной пакетный сценарий приказывает каждому **Активу (Asset)** выполнить "расчет начислений по дату", `calculateAccrualsThroughDate()`. При этом возвращается коллекция **Начислений (Accruals)**, и каждая сумма заносится в указанную книгу учета.

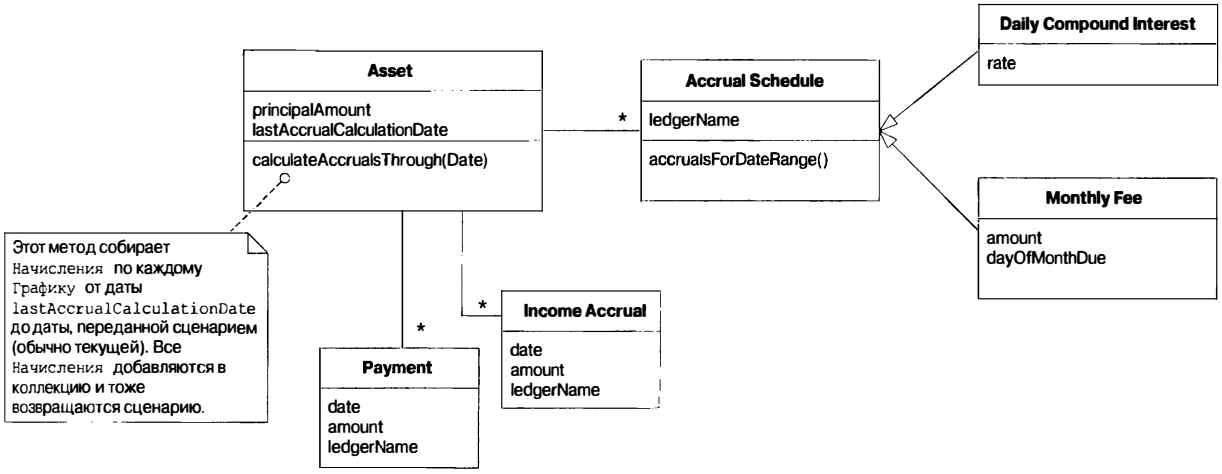


Рис. 9.8. Углубленная модель после рефакторинга

Новая модель имеет ряд преимуществ. Результаты внесенных изменений.

1. Единый язык обогатился термином "начисление" (*accrual*).
2. Начисления отделены от фактических платежей.
3. Знание предметной области (например, по какой книге выполнять бухгалтерскую проводку) перемещено из сценария в изолированный уровень прикладной модели.
4. Проценты и сборы сведены вместе таким образом, который соответствует фактическому ведению дел и устраняет дублирование в коде.
5. Создана возможность легко добавлять новые варианты процентов и сборов в виде **Графиков начислений (Accrual Schedules)**.

На этот раз программисту пришлось потрудиться, чтобы найти нужные понятия. Он видел неудобство вычисления процентов и приложил сознательные усилия, чтобы найти более глубокий по смыслу ответ.

К счастью, у разработчика был толковый и заинтересованный партнер-специалист по банковскому делу. Если бы знания приходили из более пассивного источника, ему пришлось бы отсечь больше тупиковых направлений и привлечь других разработчиков для совместного "мозгового штурма". Это замедлило бы прогресс в работе, хотя и не остановило бы его.

Размышление над противоречиями

Различные специалисты имеют свои взгляды на предмет, основанные на их личном опыте и потребностях. Даже один и тот же человек может предоставить такую информацию, которая после логического анализа окажется противоречивой. Эти докучливые противоречия, которые постоянно попадаются на глаза при анализе требований к программе, могут оказаться прекрасными ключами к углубленным моделям. Некоторые из них возникают из-за разночтений в терминологии или просто по недоразумению. Но в любом противоречии между двумя фактическими утверждениями специалистов есть и сухой ценный остаток.

Как-то раз астроном Галилей сформулировал парадоксальное утверждение. Наши чувства ясно свидетельствуют, что Земля стоит на месте: людей не сдувает с нее, они не уносятся прочь. Тем не менее Коперник убедительно показал, что Земля движется вокруг Солнца с большой скоростью. Разрешение этого противоречия порождает важнейшее знание о законах природы.

Галилей провел мысленный эксперимент. Если всадник уронит шар со скачущей лошади, куда тот упадет? Конечно, шар будет двигаться вместе с лошадью, пока не упадет

на землю ей под ноги, как если бы лошадь стояла на месте. Из этого Галилей вывел первоначальное понятие об инерционных системах отсчета, разрешив парадокс и выработав гораздо более полезную модель физики движения.

Что ж, наши противоречия обычно вовсе не так интересны, а выводы из них — не столь масштабны. И все же именно такой стиль мышления часто помогает пробиться через поверхностный уровень проблемы к более глубокому ее видению.

Все противоречия, как правило, на практике устранить не получается, да это и не всегда бывает желательно. (В главе 14 подробно рассматривается, как принимать решения по этому поводу и что делать с результатами.) Но даже если противоречие остается неустраненным, все равно бывает полезно обдумать, как два противоположных утверждения могут быть справедливыми в отношении некоей внешней реальности.

Чтение книг

Разыскивая подходящие понятия для моделей, не пропустите очевидного. Во многих областях знания существует литература, рассказывающая об основных понятиях и необходимых общих знаниях по предмету. Конечно, придется поработать и с собственными специалистами, чтобы выделить ту часть знания, которая непосредственно относится к проблеме, и переработать ее в нечто пригодное для реализации в объектно-ориентированной программе. Но тем не менее будет нелишним начать со связного и глубоко продуманного взгляда на предмет, изложенного в литературе.

Пример

Вычисление процентов: изучаем литературу

Представим себе другой сценарий для программы отслеживания кредитных капиталовложений, которая рассматривалась в прошлом примере. Как и раньше, история начинается с того, что программист понимает: архитектура программы становится все более неудобной, особенно **`Калькулятор процентов (Interest Calculator)`**. Но в этом случае обязанности специалиста по предметной области не имеют никакого отношения к проблеме, и ему нет интереса помогать в разработке программного обеспечения как таковой. Соответственно, программист не может обратиться к специалисту и вместе с ним "нащупать" недостающие понятия, которые, по-видимому, прячутся где-то под поверхностью.

Вместо этого наш разработчик идет в книжный магазин. Поискав немного, он находит и уносит с собой приглянувшееся ему введение в бухгалтерский учет. В книге он находит целую систему четко определенных понятий. Следующий отрывок дает ему особенно много пищи для размышлений.

> ***Метод начислений при учете доходов и издержек.*** *В этом методе доход учитывают в момент его начисления, даже если он еще не выплачен. Все издержки также учитываются в момент начисления, будь то фактическая их выплата или выставление счета к оплате. Любые долговые обязательства, включая налоги, проходят по статье расходов.*[2]

Теперь наш разработчик больше не должен заново изобретать бухгалтерский учет. Еще немного обсудив вопрос с другим программистом, он приходит к такой модели.

---

[2] Каплан С. "Финансы и бухгалтерский учет: как вести учет своих финансов, не имея специального образования и ученой степени". Изд-во Adams Media, 2000 г. (*Finance and Accounting: How to Keep Your Books and Manage Your Finances Without an MBA, a CPA or a Ph.D.*, by Suzanne Caplan, Adams Media, 2000).

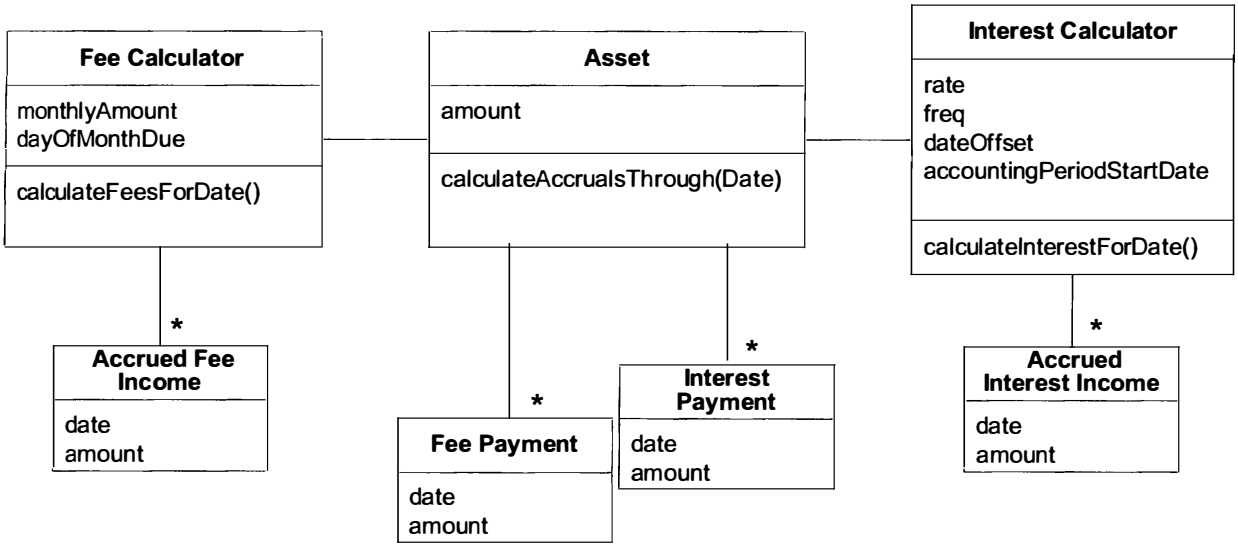


Рис. 9.9. Слегка углубленная модель, построенная на основе информации из книги

Он еще не понял, что генераторами дохода являются **Активы** (**Assets**), поэтому в модели все еще присутствуют **Калькуляторы**. Знание о книгах учета находится на операционном уровне вместо уровня предметной области, где оно более уместно. Но он уже отделил фактический платеж от начисления дохода, что было самым проблематичным местом в модели. В модель и в ЕДИНЫЙ ЯЗЫК введено слово "начисление" (*accrual*). Позднее можно будет в итерационном режиме внести дальнейшие усовершенствования.

Когда разработчик наконец смог поговорить со специалистом по финансам, тот был приятно удивлен — впервые программист проявил интерес к его профессии. Ввиду особенностей распределения обязанностей им никогда раньше не приходилось вместе сидеть над моделью, как это было в предыдущем примере. Но поскольку приобретенные этим программистом познания позволили ему задавать более грамотные вопросы, специалист стал более внимательно прислушиваться к нему и постарался отвечать на вопросы быстрее и точнее.

Конечно, не следует считать, будто чтение и консультации со специалистами — это варианты "или-или". Даже при самой разносторонней поддержке со стороны специалистов полезно почитать литературу, чтобы составить себе представление о теории предметной области. В большинстве видов деятельности нет моделей настолько четких, как в бухгалтерском учете или финансах, но во многих встречаются специалисты-мыслители, организовавшие и представившие в абстрактной форме общепринятые процедуры их профессии.

Еще один вариант, который есть у программиста — это почитать публикации других профессиональных программистов, имеющих опыт разработки программного обеспечения для нужной предметной области. Например, глава 6 из [11] могла бы направить нашего программиста в совершенно другом направлении, не обязательно лучшем или худшем. Из такой литературы вряд ли можно почерпнуть готовое решение. Скорее, она дала бы программисту несколько отправных точек для его собственных экспериментов, а также сводный опыт предшественников, уже побывавших на этой территории. И тогда не пришлось бы заново изобретать велосипед. Этот вариант рассматривается более подробно в главе 11.

Метод проб и ошибок

Приведенные здесь примеры не передают в полной мере того, сколько проб и ошибок нужно сделать на пути к успеху. Я в дискуссии вылавливал, наверное, с полдюжины разных подсказок, прежде чем попадалась одна достаточно ясная и полезная, с которой можно было поработать на модели. Но потом даже этот ключ к решению минимум один раз приходилось заменять другим, по мере того как накопление опыта и переработка знаний подсказывали новые идеи. Тому, кто занимается моделированием и проектированием программных систем, нельзя слишком привязываться к своим собственным идеям.

Все эти изменения направления работ — не пустые слова. Каждое из них что-то добавляет к пониманию модели. После каждого рефакторинга архитектура становится более гибкой, податливой к следующим изменениям, готовой к компромиссам там, где это оказывается необходимым.

В принципе другого выбора-то и нет. Выяснить, что работает, а что нет, можно только одним способом — экспериментировать. Попытки избежать лишних ложных шагов только ухудшают результат, поскольку ограничивают опыт. И времени на это может уйти еще больше, чем на серию быстрых экспериментов.

Моделирование неочевидных понятий

Объектно-ориентированная парадигма требует от нас искать и изобретать понятия определенного вида. Плотью большинства объектных моделей являются реально существующие вещи, пусть даже такие абстрактные, как бухгалтерские “начисления”, а также действия, предпринимаемые этими вещами. Это и есть “существительные и глаголы”, о которых написано в любой вводной книге по объектно-ориентированному проектированию программ. Но в модель можно также включить явно определенные понятия, относящиеся и к другим категориям.

Здесь мы рассмотрим три такие категории, которые для меня не были очевидными, когда я начинал работать с объектами. По мере того как я знакомился с ними, мои проекты становились все качественнее.

Явные условия-ограничения

Условия-ограничения образуют особо важную категорию понятий модели. Они часто существуют в неявном виде, и перевод их в явную форму может сильно улучшить концептуальную архитектуру.

Иногда ограничения естественным образом вписываются в какой-нибудь объект или метод. Так, объект “Ведро” (*Bucket*) должен гарантировать соблюдение следующего инварианта: он не должен содержать больше вещества, чем позволяет его емкость.

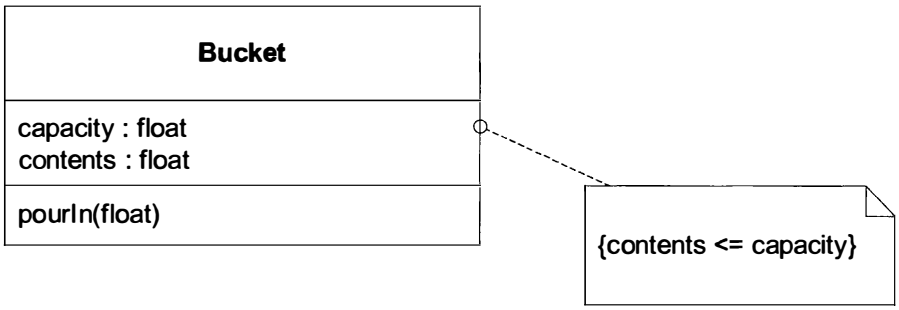


Рис. 9.10.

Такой простой инвариант можно соблюсти, применяя проверку условия в каждой операции, от которой может измениться объем содержимого ведра.

```
class Bucket {
   private float capacity;
   private float contents;

   public void pourIn(float addedVolume) {
      if (contents + addedVolume > capacity) {
         contents = capacity;
      } else {
         contents = contents + addedVolume;
      }
   }
}
```

Логика этого ограничения настолько проста, что вводимое правило тривиально. Однако его так же легко и потерять в более сложном классе. Давайте выделим его в отдельный метод под именем, которое четко и в явном виде выражало бы смысл накладываемого ограничения.

```
class Bucket {
   private float capacity;
   private float contents;

   public void pourIn(float addedVolume) {
      float volumePresent = contents + addedVolume;
      contents = constrainedToCapacity(volumePresent);
   }

   private float constrainedToCapacity(float volumePlacedIn) {
      if (volumePlacedIn > capacity) return capacity;
         return volumePlacedIn;
   }
}
```

В обеих версиях кода проверяется и форсируется соблюдение ограничения, но во втором присутствует более явная связь с моделью (что есть основное требование ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ). Это простое правило было вполне понятно и в исходной форме, но при проверке соблюдения более сложных правил они начинают загромождать объект или операцию, к которым применяются, как это происходит с любым неявно вводимым понятием. Выделение условия-ограничения в собственный метод позволяет дать ему информативное имя, благодаря которому ограничение становится явным и заметным в архитектуре программы. Теперь это уже предмет с именем, который можно обсуждать. Такой подход также предоставляет ограничению достаточное смысловое пространство. Более сложное правило запросто может разрастись в метод более "длинный", чем тот, который его вызывает (в нашем случае метод `pourIn()`). Тогда вызывающий метод остается простым и фокусируется на своих прямых обязанностях, а сложность ограничения может возрастать как угодно по мере надобности.

Отдельный метод дает условию-ограничению некоторое пространство для роста, но бывает много случаев, когда связь просто нельзя "скомпоновать" в один метод. Или же, даже если метод остается простым, он может требовать для себя информацию, которая объекту не нужна для выполнения его основных обязанностей. То есть, правило может элементарно не вписываться ни в один существующий объект.

Вот несколько тревожных свидетельств о том, что условие-ограничение искажает архитектуру объекта, в котором она инкапсулирована.

1. Для проверки ограничения требуются такие данные, которые без него излишни для определения объекта.
2. В нескольких объектах фигурируют связанные между собой правила, из-за чего приходится вводить дублирование или наследование объектов, тогда как при отсутствии этих правил они не являются родственными.
3. Вокруг ограничений идет множество дискуссий об архитектуре и функциональных требованиях, а в программной реализации они все прячутся в процедурном коде.

Когда ограничения "затемняют" основные обязанности объекта или когда ограничение играет важную роль в предметной области, но не в модели, его можно выделить в отдельный объект или даже промоделировать в виде набора объектов и отношений. (Подробно и не слишком формализовано об этом говорится в [23]).

Пример

Краткий обзор: избыточное резервирование

В главе 1 мы встречались с обычной деловой практикой при грузоперевозках — резервированием на 10% больше груза, чем его может перевезти транспортное средство. (Опыт научил фирмы, занимающиеся поставками и перевозками, что этот избыток компенсируется отменами заказов в последний момент, так что их транспорт уходит практически полным.)

Это условие-ограничение на ассоциацию между **Рейсом** (`Voyage`) и **Грузом** (`Cargo`) было сформулировано в явном виде как на схемах, так и в коде, путем добавления нового класса, представляющего само ограничение.

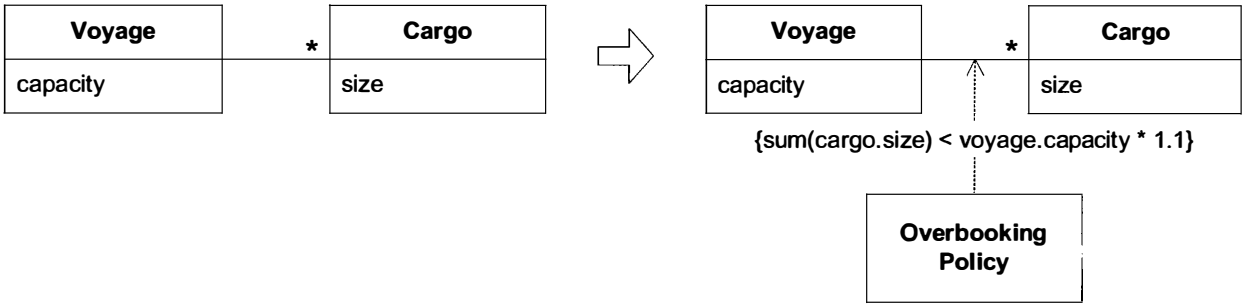


Рис. 9.11. Модель с выделением правила в явном виде

Процессы как объекты предметной области

Заранее договоримся, что мы *не хотим* делать процедуры главным аспектом нашей модели. Процедуры должны инкапсулироваться объектами, поэтому давайте выражаться в терминах целей, обязанностей, предназначения.

Я говорю здесь о процессах, которые существуют в предметной области и которые мы обязаны представить в модели. Когда таковые возникают на практике, они имеют тенденцию приводить к "неуклюжим" объектным архитектурам.

Первый пример в этой главе посвящен системе управления грузопоставками, которая маршрутизировала грузы. Процесс маршрутизации это нечто, имеющее смысл в пред-

метной области. Одним из способов выразить это нечто является СЛУЖБА (SERVICE), которая может инкапсулировать очень сложные алгоритмы.

Если существует несколько способов выполнения того или иного рабочего процесса, возникает идея сделать сам алгоритм или некую его ключевую часть отдельным объектом. Тогда выбор между процессами становится выбором между объектами, каждый из которых представляет отдельную СТРАТЕГИЮ (STRATEGY). (Использование СТРАТЕГИЙ в предметной области более подробно рассматривается в главе 12.)

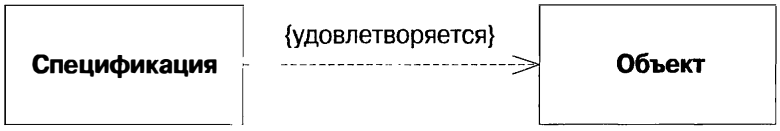
Основное отличие между процессом, который следует сделать отдельным объектом, и процессом, который должен быть скрыт "в недрах" кода, состоит вот в чем: говорят ли об этом процессе специалисты по предметной области, или он составляет не более чем часть внутреннего механизма программы?

Условия-ограничения и процессы — это две широкие категории понятий моделей, которые далеко не всегда приходят на ум при программировании на объектно-ориентированных языках. А между тем они могут сделать архитектуру более четкой благодаря тому, что мы начнем думать о них как об элементах модели.

Некоторые полезные категории понятий имеют более узкий характер. В завершение этой главы я расскажу об одной намного более специфической, хотя и вполне обычной, категории. Для выражения некоторых разновидностей правил или регламентов служат СПЕЦИФИКАЦИИ (SPECIFICATIONS), которые высвобождают эти правила в явном виде из системы условных проверок и помещают в модель на видное место.

Я разработал шаблон СПЕЦИФИКАЦИЯ в сотрудничестве с Мартином Фаулером [9]. Обманчивая простота самого понятия сопровождается многими тонкостями на операционном уровне и в реализации, так что в этой главе будет много подробностей. Еще больше на эту тему будет говориться в главе 10, где этот шаблон будет дополнен. Ознакомившись с введением в шаблон, идущий далее раздел "Применение и реализация спецификаций" можно пока что прочитать бегло и отложить до тех пор, пока у вас не возникнет потребность в практическом применении этих знаний.

Спецификация



Во всех программах встречаются логические (булевы) проверки, которые фактически являются элементами несложных регламентных правил. Пока такие правила достаточно просты, их можно реализовать с помощью проверочных методов, например, `anIterator.hasNext()` (*счетчикЦикла.имеетСледующееЗначение*) или `anInvoice.IsOverdue()` (*счетФактура.просрочен*). В классе **Счет-фактура (Invoice)** код, помещенный в метод `isOverdue()` (*просрочен*), реализует алгоритм проверки некоторого правила и выдачи ответа в виде логического значения.

```
public boolean isOverdue() {
   Date currentDate = new Date();
   return currentDate.after(dueDate);
}
```

Но не все правила так просты. В том же классе **Счет-фактура (Invoice)** может присутствовать еще один метод-правило, `anInvoice.isDelinquent()` (*счетФактура.неОплачен*). Этот метод, вполне возможно, начинается с проверки просроченности счета, но

не заканчивается на ней. Длительность предоставляемого срока оплаты, например, может зависеть от состояния банковского счета клиента. По некоторым неоплаченным счетам-фактурам должно высылаться второе извещение, а другие счета должны передаваться в коллекторское агентство. А еще есть кредитно-платежная история клиента, политика компании по разным видам услуг и товаров... и вот уже очевидный смысл объекта **Счет-фактура** как запроса на оплату теряется в массе кода, занимающегося проверкой правил. Кроме того, класс **Счет-фактура** быстро обрастает всевозможными связями с классами и подсистемами прикладной модели, которые также не подчеркивают его основной смысл.

В попытках спасти класс **Счет-фактура** (**Invoice**) на этом этапе его существования программист, скорее всего, выполнит рефакторинг кода проверки правил, чтобы вывести его на уровень прикладных операций (в данном случае — в систему инкассирования). Таким образом правила окажутся вообще отделенными от уровня предметной области, а в нем останутся только мертвые объекты данных, вообще не выражающие базовые закономерности и деловые регламенты прикладной модели. Такие правила должны оставаться на уровне предметной области, но их проверка неуместна в самом проверяемом объекте (в данном случае это **Счет-фактура**). К тому же, проверочные методы быстро обрастают всевозможным кодом контроля условий, отчего правила становится трудно читать.

Разработчики, придерживающиеся парадигмы логического программирования, пошли бы по другому пути. Они бы выразили правила в виде *предикатов*. Предикаты — это функции, выдающие значения "истина" или "ложь", которые можно комбинировать с помощью таких операций, как И и ИЛИ для выражения более сложных правил. С помощью предикатов можно было бы объявлять правила непосредственно и применять их к объекту типа **Счет-фактура**. Ах, если бы мы работали в парадигме логического программирования...

С учетом всего этого кое-кто попытался применить логические правила в системе, основанной на объектах. Некоторые из таких попыток были весьма искусными, другие же — наивными. Одни метили высоко, другие ограничивались малым. Некоторые дали результат, но многие были отброшены как провальные эксперименты. Результатом нескольких был срыв ряда проектов. Ясно только одно: несмотря на всю привлекательность идеи, полная реализация логики в объектах — очень непростое предприятие. (В конце концов, логическое программирование само по себе — целая парадигма моделирования и построения программной архитектуры.)

Реализация правил прикладной модели (деловых регламентов) часто не вписывается в обязанности никаких из лежащих на поверхности СУЩНОСТЕЙ **или** ОБЪЕКТОВ-ЗНАЧЕНИЙ**, а их разнообразие и обилие возможных комбинаций может исказить основной смысл объекта предметной области. Но "вынесение" правил из уровня предметной области — еще хуже, поскольку в таком случае код этого уровня перестает выражать модель.**

Логическое программирование предлагает концепцию отдельных комбинируемых объектов-правил, именуемых "предикатами", но полная реализация этой концепции с помощью объектов довольно громоздка. Она также имеет слишком общий характер, чтобы передавать смысл модели в коде так же хорошо, как более специализированные конструкции.

К счастью, для получения хорошего результата нам нет необходимости полностью реализовать парадигму логического программирования. Большинство наших правил сводится к нескольким частным случаям. Мы можем заимствовать понятие предиката и создать специализированные объекты, выдающие логическое (булево) значение. Те самые проверочные методы, которые у нас выходили из-под контроля, прекрасно "заживут" в виде отдельных объектов. Они будут представлять собой небольшие тесты на истинность, которые

можно выделить в отдельные ОБЪЕКТЫ-ЗНАЧЕНИЯ. Новый объект подобного рода будет оценивать другой объект, проверяя истинность применяемого к нему предиката.

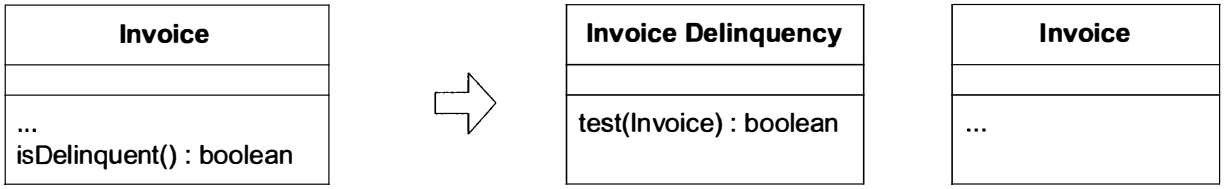


Рис. 9.12.

Другими словами, этот новый объект будет *спецификацией.* СПЕЦИФИКАЦИЯ (SPECIFICATION) задает условие-ограничение на состояние другого объекта, которое может возникнуть, а может и не возникнуть. Применений у нее много, но наиболее существенное по смыслу состоит в том, что СПЕЦИФИКАЦИЯ может проверять любой объект на соответствие заданным критериям.

Создайте набор явно определенных, предикатоподобных ОБЪЕКТОВ-ЗНАЧЕНИЙ **для решения специализированных задач.** СПЕЦИФИКАЦИЯ — **это предикат, который определяет, удовлетворяет объект некоторым критериям или нет.**

Многие СПЕЦИФИКАЦИИ являются простыми, узкоспециальными проверками, как в примере с неоплаченным счетом-фактурой. В случаях, когда предполагается реализация сложных регламентов, можно добавить возможность комбинирования простых спецификаций, как в случае комбинирования предикатов с помощью логических операций. (Техника реализации такой возможности будет рассмотрена в следующей главе.) Но фундаментальный шаблон остается одним и тем же, прокладывая путь от простых к более сложным моделям.

Случай неоплаченного счета-фактуры можно промоделировать с помощью СПЕЦИФИКАЦИИ, которая бы определяла, что такое "неоплаченный", могла бы проверить любой объект **`Счет-фактура`** и выдать заключение о его неоплаченности.

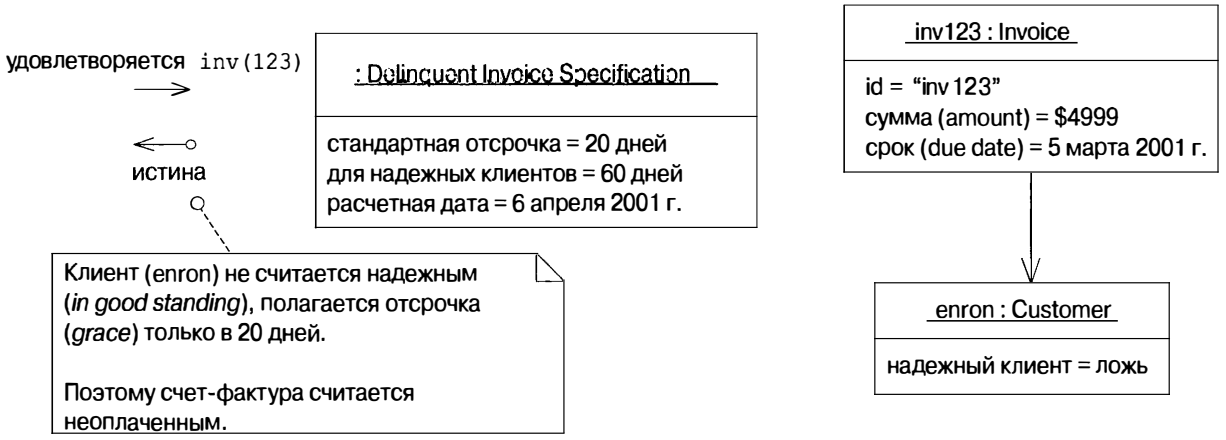


Рис. 9.13. Более сложный регламент для определения неоплаченности, выделенный в отдельную СПЕЦИФИКАЦИЮ

При наличии СПЕЦИФИКАЦИИ правило-регламент остается на уровне предметной области. Поскольку правило теперь является полноценным объектом, архитектура более четко отражает модель. ФАБРИКА может сконфигурировать СПЕЦИФИКАЦИЮ, используя информацию из других источников, — например, из банковского счета клиента или базы данных регламентов компании. Если предоставить доступ к этим источникам непосредственно

объекту **Счет-фактура**, это создаст между ними такую связь, которая непосредственно не относится к запросу на оплату (т.е. к непосредственной обязанности объекта). В этом случае необходимо создать объект **Спецификация неоплаченного счета-фактуры** (**Delinquent Invoice Specification**), использовать его для проверки тех или иных **Счетов-фактур**, а затем ликвидировать, чтобы осталась только конкретная дата проверки — это упрощение реализации. А СПЕЦИФИКАЦИИ можно предоставить любую необходимую для выполнения ее обязанностей информацию прямым и непосредственным способом.

* * *

Базовое понятие СПЕЦИФИКАЦИИ само по себе просто и помогает сосредоточиться на проблеме моделирования предметной области. Но для ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN) нужна еще и эффективная реализация, которая также выражает это понятие. Чтобы справиться с этим требованием, необходимо подробнее разобраться с новым архитектурным шаблоном. Шаблон — это не просто технический прием для вычерчивания UML-диаграмм; это решение проблемы программирования, которое к тому же следует принципам ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ.

Если вы примените шаблон правильно, перед вами раскроется целая стратегия решения определенного класса задач, относящихся к моделированию предметной области, и вы сможете черпать информацию из сокровищницы знаний, созданной годами поиска эффективных реализаций. В следующем разделе подробно описываются различные аспекты спецификаций, дается много вариантов реализации тех или иных возможностей. Архитектурный шаблон — это не книга готовых рецептов; он всего лишь позволяет начать решение проблемы с готовой базы опытного знания и дает язык, на котором можно обсуждать проблему.

При первом чтении можно ограничиться кратким знакомством с ключевыми понятиями. Позже, когда вы столкнетесь с конкретной необходимостью, можно будет вернуться и глубже проанализировать подробно изложенный практический опыт предшественников. А там уже недалеко и до решения собственной проблемы.

Применение и реализация спецификаций

Ценность СПЕЦИФИКАЦИЙ в значительной мере состоит в том, что они сводят воедино такие прикладные функции, которые могут показаться совершенно различными. Нам может понадобиться определить состояние некоторого объекта для одной или нескольких следующих целей.

1. Проверить пригодность объекта для удовлетворения какой-то потребности или достижения какой-то цели.
2. Выбрать объект из коллекции ему подобных (как в случае запроса на просроченные счета-фактуры).
3. Заказать создание нового объекта для определенных потребностей.

Эти три области применения — проверка пригодности, отбор по образцу, создание по заказу — на концептуальном уровне представляют собой одно и то же. Не имея такого шаблона, как СПЕЦИФИКАЦИЯ, одно и то же правило можно было бы реализовать “в разных обличьях” — не исключено, что и противоречащих друг другу. Так было бы потеряно принципиальное единство содержания. Но применение шаблона СПЕЦИФИКАЦИЯ позволяет использовать единообразную модель, пусть даже в разных формах реализации.

Проверка пригодности

Самое простое применение СПЕЦИФИКАЦИЙ — это проверка пригодности, и именно в нем само понятие демонстрируется наиболее непосредственно.

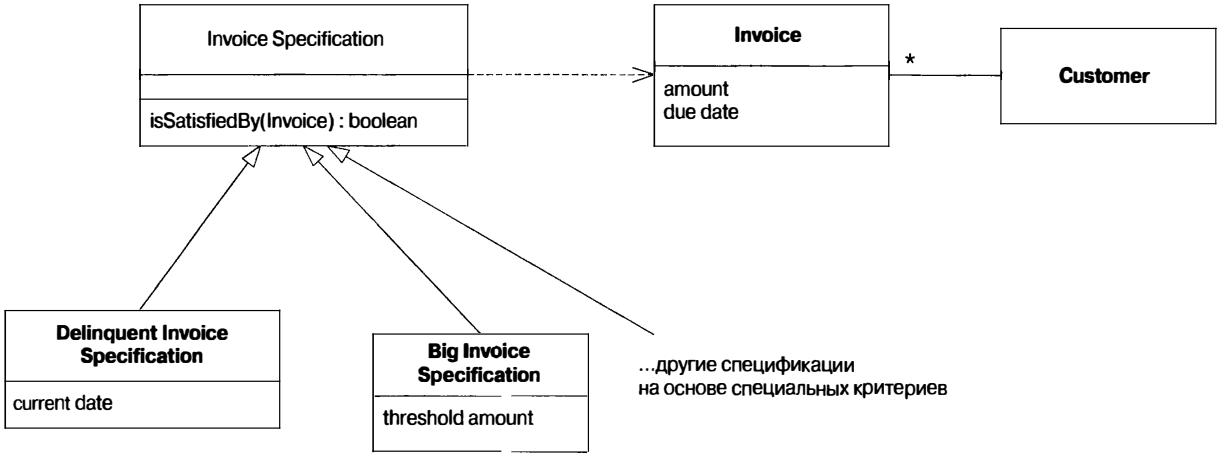


Рис. 9.14. Модель с применением СПЕЦИФИКАЦИИ для проверки пригодности

```
class DelinquentInvoiceSpecification extends
      InvoiceSpecification {
   private Date currentDate;
   // Экземпляр используется и удаляется в конкретный день

   public DelinquentInvoiceSpecification(Date currentDate) {
      this.currentDate = currentDate;
   }

   public boolean isSatisfiedBy(Invoice candidate) {
      int gracePeriod =
         candidate.customer().getPaymentGracePeriod();
      Date firmDeadline =
         DateUtility.addDaysToDate(candidate.dueDate(),
            gracePeriod);
      return currentDate.after(firmDeadline);
   }
}
```

Теперь предположим, что нужно выдать предупреждение, если торговый агент приводит нам клиента с просроченными счетами. Для этого достаточно написать метод в клиентском классе, что-нибудь в этом роде.

```
public boolean accountIsDelinquent(Customer customer) {
   Date today = new Date();
   Specification delinquentSpec =
      new DelinquentInvoiceSpecification(today);
   Iterator it = customer.getInvoices().iterator();
   while (it.hasNext()) {
      Invoice candidate = (Invoice) it.next();
      if (delinquentSpec.isSatisfiedBy(candidate)) return true;
   }
   return false;
}
```

Отбор по запросу

Проверка пригодности заключается в выяснении, соответствует ли некий конкретный объект заданным критериям — скорее всего, для того, чтобы по итогам проверки что-то предпринять. Но часто возникает и другая задача — выбрать подмножество из набора объектов по некоторым критериям. Здесь применимо все то же понятие СПЕЦИФИКАЦИИ, но с другими особенностями реализации.

Предположим, от программы требуется выдать список всех клиентов с неоплаченными **Счетами-фактурами (Invoices)**. Теоретически тут применима та же **Спецификация неоплаченного счета-фактуры (Delinquent Invoice Specification)**, которую мы определили раньше, но на практике придется несколько изменить реализацию. Чтобы продемонстрировать, что *идея* осталась той же, предположим для начала, что количество **Счетов-фактур** невелико и уже находится прямо в памяти. В этом случае пригодится упрощенная реализация, разработанная для проверки пригодности. В **Хранилище счетов-фактур (Invoice Repository)** можно включить обобщенный метод, который бы отбирал **Счета-фактуры** по СПЕЦИФИКАЦИИ.

```
public Set selectSatisfying(InvoiceSpecification spec) {

   Set results = new HashSet();
   Iterator it = invoices.iterator();
   while (it.hasNext()) {
      Invoice candidate = (Invoice) it.next();
      if (spec.isSatisfiedBy(candidate)) results.add(candidate);
   }

   return results;
}
```

Теперь в клиентском коде можно получить коллекцию всех неоплаченных **Счетов-фактур** с помощью одного-единственного оператора.

```
Set delinquentInvoices = invoiceRepository.selectSatisfying(
   new DelinquentInvoiceSpecification(currentDate));
```

В этой строке кода четко передается концептуальный смысл операции. Конечно, объекты **Счет-фактура**, скорее всего, находятся не в оперативной памяти; их вообще могут быть многие тысячи. В типичной коммерческой системе делопроизводства такая информация обычно хранится в реляционной базе данных. А как уже говорилось в предыдущих главах, в местах, где происходит пересечение с другими программными технологиями, есть шанс потерять концентрацию на предметной модели.

В реляционных СУБД имеются мощные поисковые механизмы. Как же нам воспользоваться их мощью, чтобы эффективно решить проблему, но не потерять модель СПЕЦИФИКАЦИИ? В подходе ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ требуется, чтобы модель "шла в ногу" с реализацией, но при этом разрешается свободно выбирать любую реализацию, которая честно передает смысл модели. К счастью для нас, язык SQL — это очень естественный способ писать СПЕЦИФИКАЦИИ.

Вот простой пример, в котором запрос инкапсулируется в том же классе, что и проверочное правило. В **Спецификацию счета-фактуры (Invoice Specification)** добавлен один метод, который реализован в подклассе **Спецификация неоплаченного счета-фактуры (Delinquent Invoice Specification)**.

```
public String asSQL() {
   return
      "SELECT * FROM INVOICE, CUSTOMER" +
      " WHERE INVOICE.CUST_ID = CUSTOMER.ID" +
      " AND INVOICE.DUE_DATE + CUSTOMER.GRACE_PERIOD" +
      " < " + SQLUtility.dateAsSQL(currentDate);
}
```

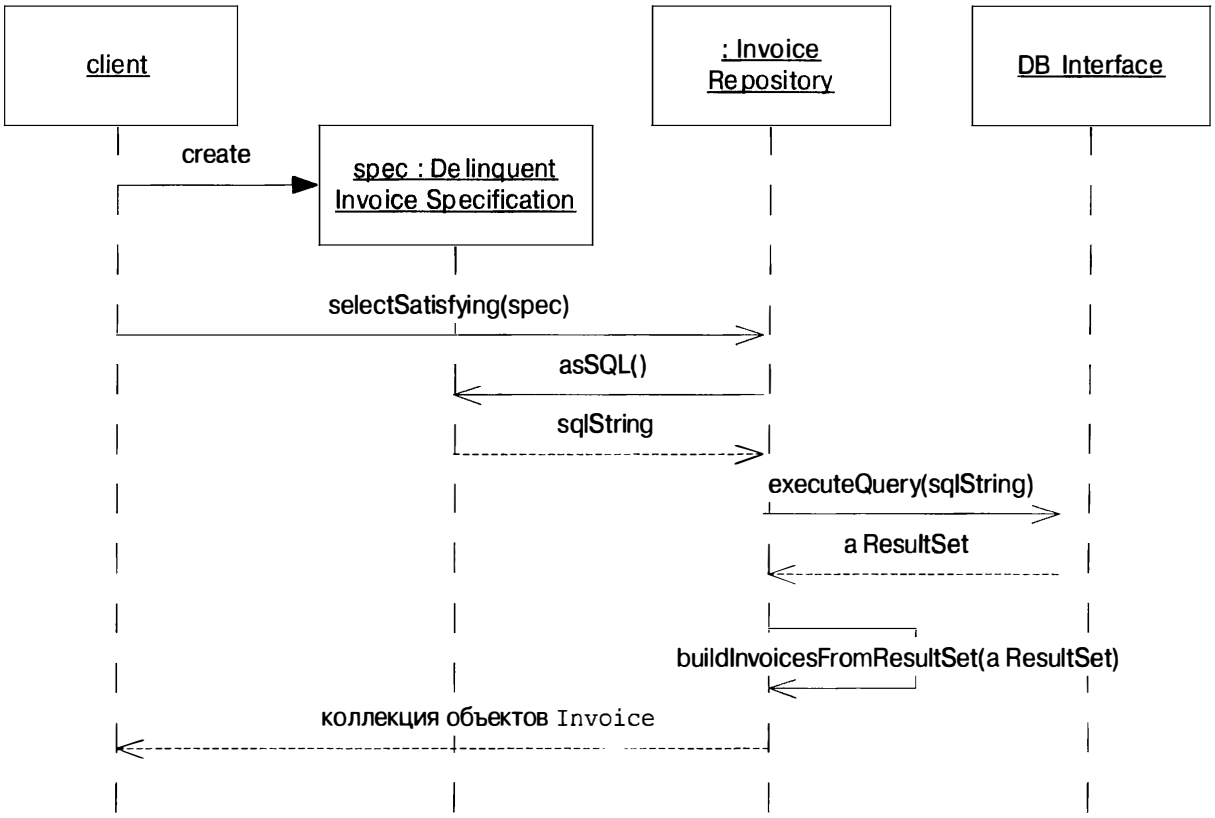


Рис. 9.15. Взаимодействие между ХРАНИЛИЩЕМ и СПЕЦИФИКАЦИЕЙ

СПЕЦИФИКАЦИИ легко находят общий язык с ХРАНИЛИЩАМИ стандартными механизмами для предоставления доступа по запросам к объектам предметной области, в частности для инкапсуляции интерфейса баз данных (рис. 9.15).

Но теперь в этой архитектуре есть ряд проблемных мест. Важнее всего то, что подробности структуры таблиц "проникли" на уровень предметной области, тогда как их следовало бы вывести в отдельный уровень отображения, устанавливающий соответствие между объектами предметной области и реляционными таблицами. Неявное дублирование этой информации именно здесь может помешать дальнейшей доработке и использованию объектов **Счет-фактура (Invoice)** и **Customer (Клиент)**, потому что любое изменение в их отображении теперь нужно отслеживать в нескольких местах. Но этот пример — просто иллюстрация к тому, как держать правило в одном месте. Некоторые среды для поддержки отображения между объектами и реляционными базами позволяют выразить запрос в терминах объектов и атрибутов модели, генерируя SQL-код самостоятельно на инфраструктурном уровне. Такими средствами можно убить сразу двух зайцев.

Но если инфраструктура не приходит на помощь, SQL-код можно выделить из смысловых объектов предметной области, добавив специальный запросный метод в **Хранилище счетов-фактур (Invoice Repository)**. Чтобы не помещать регламентное

правило в само ХРАНИЛИЩЕ, нужно постараться выразить запрос более общим способом — таким, который не содержит явно самого правила, но может комбинироваться и помещаться в контекст так, чтобы фактически построить это правило (в данном случае с использованием двойной отправки запроса).

```
public class InvoiceRepository {

   public Set selectWhereGracePeriodPast(Date aDate){
      // Это не правило, а просто специализированный запрос
      String sql = whereGracePeriodPast_SQL(aDate);
      ResultSet queryResultSet =
         SQLDatabaseInterface.instance().executeQuery(sql);
      return buildInvoicesFromResultSet(queryResultSet);
   }

   public String whereGracePeriodPast_SQL(Date aDate) {
     return
        "SELECT * FROM INVOICE, CUSTOMER" +
        " WHERE INVOICE.CUST_ID = CUSTOMER.ID" +
        " AND INVOICE.DUE_DATE + CUSTOMER.GRACE_PERIOD" +
        " < " + SQLUtility.dateAsSQL(aDate);
   }

   public Set selectSatisfying(InvoiceSpecification spec) {
      return spec.satisfyingElementsFrom(this);
   }
}
```

Метод `asSql()` в классе **Спецификация счета-фактуры (Invoice Specification)** заменен методом `satisfyingElementsFrom(InvoiceRepository)`, который в классе **Спецификация неоплаченного счета-фактуры (Delinquent Invoice Specification)** реализуется в таком виде.

```
public class DelinquentInvoiceSpecification {
   // Здесь идет основной код класса

   public Set satisfyingElementsFrom(
                     InvoiceRepository repository) {
      //Правило неоплаченности определяется так:
      // "на текущую дату срок оплаты истек"
      return repository.selectWhereGracePeriodPast(currentDate);
   }
}
```

Код SQL помещается в ХРАНИЛИЩЕ, а СПЕЦИФИКАЦИЯ отвечает за то, какой именно запрос должен использоваться. Правила не так уж изящно собраны в эту СПЕЦИФИКАЦИЮ, но, тем не менее, здесь присутствует ключевое определение, что такое неоплаченность счета (истечение срока его оплаты).

Теперь в ХРАНИЛИЩЕ имеется очень специализированный запрос, который, скорее всего, будет использоваться только в этом случае. Это приемлемо, но в зависимости от количества просроченных **Счетов-фактур** по сравнению с неоплаченными может оказаться достаточно эффективным промежуточное решение, в котором методы ХРАНИЛИЩА имеют более общий характер, а код СПЕЦИФИКАЦИИ более нагляден.

```
public class InvoiceRepository {

   public Set selectWhereDueDateIsBefore(Date aDate) {
      String sql = whereDueDateIsBefore_SQL(aDate);
      ResultSet queryResultSet =
         SQLDatabaseInterface.instance().executeQuery(sql);
      return buildInvoicesFromResultSet(queryResultSet);
   }

   public String whereDueDateIsBefore_SQL(Date aDate) {
      return
         "SELECT * FROM INVOICE" +
         " WHERE INVOICE.DUE_DATE" +
         " < " + SQLUtility.dateAsSQL(aDate);
   }

   public Set selectSatisfying(InvoiceSpecification spec) {
      return spec.satisfyingElementsFrom(this);
   }
}

public class DelinquentInvoiceSpecification {
   // Здесь идет основной код класса

   public Set satisfyingElementsFrom(
                           InvoiceRepository repository) {
      Collection pastDueInvoices =
         repository.selectWhereDueDateIsBefore(currentDate);

      Set delinquentInvoices = new HashSet();
      Iterator it = pastDueInvoices.iterator();
      while (it.hasNext()) {
         Invoice anInvoice = (Invoice) it.next();
         if (this.isSatisfiedBy(anInvoice))
            delinquentInvoices.add(anInvoice);
      }
      return delinquentInvoices;
   }
}
```

В этом коде достигается скачок быстродействия, поскольку одновременно извлекается больше **Счетов-фактур**, после чего отбор из них выполняется в оперативной памяти. Приемлемая ли это цена за лучшее разделение обязанностей — это зависит целиком от конкретных обстоятельств. Реализовать взаимодействия между СПЕЦИФИКАЦИЯМИ и ХРАНИЛИЩАМИ можно разными способами, используя специфические преимущества платформы разработки, но не трогая распределения основных обязанностей.

Иногда для улучшения быстродействия или, скорее, для повышения безопасности запросы могут быть реализованы на сервере как хранимые процедуры. В этом случае СПЕЦИФИКАЦИЯ может содержать только параметры, разрешенные в такой процедуре. Но при всем этом разнообразие реализаций ничего не меняет в самой модели. Выбирать конкретную программную реализацию можно совершенно свободно, если сама модель не накладывает специфических ограничений. Цена — только неудобство написания и модификации запросов.

Все вышеизложенное лишь слегка затронуло пласт проблем, которые связаны с организацией связи между базами данных и СПЕЦИФИКАЦИЯМИ, и я не буду даже пытаться охватить все соображения, которые при этом возникают. Я хочу лишь дать самое общее понятие о том, какой выбор приходится делать в таких случаях. Некоторые технические вопросы, связанные с разработкой ХРАНИЛИЩ со СПЕЦИФИКАЦИЯМИ, рассмотрены Ми (Mee) и Хайеттом (Hieatt) в книге [13].

Создание по заказу

Когда Пентагон желает получить новый самолет-истребитель, его сотрудники разрабатывают техническое задание — спецификацию. В спецификации может указываться, что истребитель должен достигать скорости в 2 числа Маха, иметь дальность полета 3000 километров, стоить не более 50 миллионов долларов и т.д. Но какой бы подробной ни была спецификация, это еще не проект самолета, и тем более не сам самолет. Авиастроительная компания берет такую спецификацию и создает на ее основе один или несколько проектов. Компании-конкуренты могут разработать несколько разных проектов, причем каждый из них предположительно соответствует исходной спецификации.

Многие компьютерные программы что-то генерируют в своей работе, и параметры этого чего-то должны задаваться каким-то образом. Когда вы вставляете картинку в документ, открытый в окне текстового редактора, она обтекается текстом. Вы задаете местоположение картинки и, возможно, также стиль ее обтекания. Точное положение каждого слова текста на странице затем вычисляется самим текстовым редактором таким образом, чтобы соблюдалась заданная вами “спецификация”.

Хотя это и не очевидно, здесь работает то же самое понятие СПЕЦИФИКАЦИИ, что и в случае проверки пригодности или отбора по запросу. Мы задаем критерии для объектов, которых еще нет в наличии в данный момент. Но программная реализация будет совершенно другой. Такая СПЕЦИФИКАЦИЯ не является фильтром для уже существующих объектов, как в случае отбора по запросу. Это и не проверка одного существующего объекта на соответствие каким-то критериям. На этот раз в соответствии со СПЕЦИФИКАЦИЕЙ будет создаваться или переконфигурироваться совершенно новый объект или набор объектов.

Без использования спецификации можно было бы написать генератор, включающий процедуры или набор команд для создания требуемых объектов. Код процедур и команд неявно определял бы функциональное поведение этого генератора.

Но вместо этого мы вводим для генератора такой интерфейс, который определяется в терминах описательной СПЕЦИФИКАЦИИ и задает явные ограничения для свойств продукции генератора. В этом подходе имеется ряд преимуществ.

- Реализация генератора отделена от его интерфейса. СПЕЦИФИКАЦИЯ определяет требования к результату, но не способ его достижения.
- Интерфейс передает заданные регламентные правила в явной форме, так что разработчики знают, чего ожидать от генератора, даже не понимая всех подробностей его работы. Между тем единственный способ описать поведение процедурно-определенного генератора — это проанализировать весь его код построчно или запустить серию тестов.
- Интерфейс обладает большей гибкостью или же ему можно придать большую гибкость, поскольку постановка задачи возложена на клиентский код, а от генератора требуется только выполнение требований, изложенных в СПЕЦИФИКАЦИИ.
- И последнее, но не по значимости: такой интерфейс легче тестировать, поскольку модель предлагает явный способ для определения входных данных генератора,

который одновременно является и способом проверки его выходных данных. Другими словами, та же самая СПЕЦИФИКАЦИЯ, которая передается в интерфейс генератора для накладывания ограничений на процесс генерирования данных, может затем использоваться в своей проверочной роли (если это поддерживается в программной реализации) для подтверждения правильности создания объекта. (Это пример шаблона УТВЕРЖДЕНИЕ, или ASSERTION, который рассматривается в главе 10.)

Создание по заказу может означать генерирование объекта "с нуля", но это может быть и переконфигурирование уже существующих объектов для подгонки под СПЕЦИФИКАЦИЮ.

Пример

Складирование химикатов

Имеется склад, на котором штабелями в больших контейнерах, аналогично товарным вагонам, хранятся различные химикаты. Некоторые из них безвредны и низкоактивны, а потому их можно складировать практически где угодно. Некоторые обладают летучестью, поэтому должны храниться в специальных вентилируемых контейнерах. Некоторые взрывоопасны, поэтому хранятся в специальных усиленных контейнерах. Имеются также правила относительно того, какие сочетания разрешены в одном контейнере.

Наша цель — написать программу, которая предлагала бы эффективное и безопасное размещение химикатов по контейнерам.

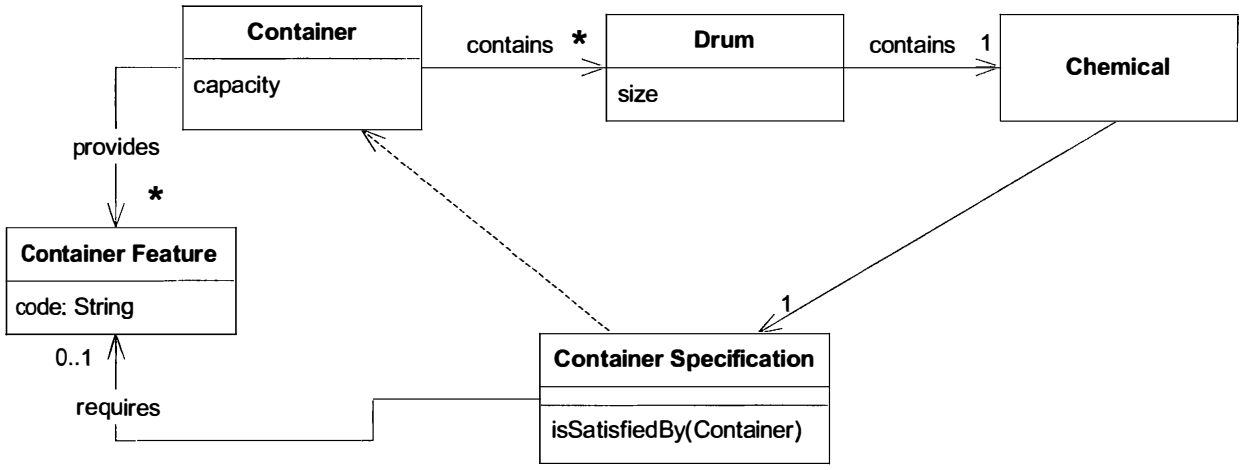


Рис. 9.16. Модель складского хранения

Можно было бы начать с написания процедуры, которая брала бы химикат и помещала его в контейнер, но давайте начнем с задачи проверки. Это побудит нас формулировать все правила явно и даст возможность протестировать финальную реализацию.

Каждый химикат будет иметь СПЕЦИФИКАЦИЮ необходимого для него контейнера.

Химикат	Спецификация контейнера
Тринитротолуол	Усиленный контейнер
Песок	
Биологические образцы	Не помещать в один контейнер со взрывчатыми веществами
Аммиак	Вентилируемый контейнер

Если все это оформить в виде **Спецификаций контейнера (Container Specifications)**, то можно будет взять конфигурацию заполненных контейнеров и проверить их на удовлетворение следующим условиям-ограничениям.

Описание контейнера	Содержимое	Соблюдена ли спецификация
Усиленный	10 кг тринитротолуола 250 кг песка	Да
	25 кг биологических образцов	Да
	Аммиак	Нет

Для проверки нужных **СвойствКонтейнера (ContainerFeatures)** понадобится реализовать специальный метод в **Спецификации контейнера** — `isSatisfied()`. Например, СПЕЦИФИКАЦИЯ, приложенная к взрывоопасному химикату, будет проверять свойство "усиленный" (`ARMORED`).

```
public class ContainerSpecification {
   private ContainerFeature requiredFeature;

   public ContainerSpecification(ContainerFeature required) {
      requiredFeature = required;
   }

   boolean isSatisfiedBy(Container aContainer){
      return aContainer.getFeatures().contains(requiredFeature);
   }
}
```

Вот пример клиентского кода для задания взрывчатого вещества.

```
tnt.setContainerSpecification(
      new ContainerSpecification(ARMORED));
```

Метод `isSafelyPacked()` в объекте **Контейнер (Container)** призван подтверждать, что **Контейнер** обладает всеми свойствами, указанными в спецификациях заполняющих его химикатов.

```
boolean isSafelyPacked(){
   Iterator it = contents.iterator();
   while (it.hasNext()) {
      Drum drum = (Drum) it.next();
      if (!drum.containerSpecification().isSatisfiedBy(this))
         return false;
   }
   return true;
}
```

На этом этапе можно было бы написать контролирующую программу, которая бы читала инвентарную базу данных и выдавала отчет о любых случаях небезопасного хранения.

```
Iterator it = containers.iterator();
while (it.hasNext()) {
   Container container = (Container) it.next();
   if (!container.isSafelyPacked())
```

```
        unsafeContainers.add(container);
}
```

Но это не та программа, которую нам заказывали. Владельцам склада полезно знать и о такой возможности, но все-таки на нас возложили разработку программы-организатора. У нас же пока есть только тест для такого организатора. Наше понимание предметной области и модели, основанной на СПЕЦИФИКАЦИЯХ, дает нам возможность определить четкий и простой интерфейс для СЛУЖБЫ, которая будет принимать коллекции **Бочек** (**Drums**) и **Контейнеров** и помещать их на склад в соответствии с правилами хранения.

```
public interface WarehousePacker {
   public void pack(Collection containersToFill,
      Collection drumsToPack) throws NoAnswerFoundException;

   /* ПРОВЕРКА: в конце метода pack() спецификация контейнера
   (ContainerSpecification) каждой бочки (Drum) должна
   удовлетворяться соответствующим контейнером (Container).
   Если нельзя найти полного решения, следует инициировать
   исключительную ситуацию. */
}
```

Теперь задача проектирования такого модуля-решателя для задачи оптимизации с наложенными ограничениями, который бы реализовал функции службы **Складировщика** (**Packer**), отделена от остальной части программы, и эти механизмы не будут загромождать ту часть архитектуры, которая непосредственно выражает модель. (См. о декларативном стиле проектирования архитектуры в главе 10, а о связных механизмах в главе 15.) Тем не менее правила, *регламентирующие* складирование, не извлечены из объектов предметной области.

Пример

Рабочий прототип организатора складирования

Написать алгоритм оптимизации для программы-организатора складирования не так-то просто. Для этой цели выделена небольшая группа программистов и специалистов по складскому хранению, но они еще даже не начали писать код. Между тем другая небольшая группа разрабатывает приложение, с помощью которого пользователи смогут извлекать инвентарную опись из базы данных, отправлять ее в **Складировщик** (**Packer**) и интерпретировать результаты. Они пытаются подстроить архитектуру под ожидаемый **Складировщик**, но все, что у них получается, — это воспроизвести дубликат интерфейса пользователя и написать кое-какой код для интеграции с базой данных. Они не могут предъявить пользователям полноценный рабочий интерфейс, чтобы получить от них обратную связь. По той же причине разработчики **Складировщика** также работают в вакууме.

Имея в своем распоряжении объекты предметной области и интерфейс СЛУЖБЫ, группа разработки приложения вдруг осознает, что могла бы написать упрощенную версию **Складировщика**, которая помогла бы продвинуть вперед параллельные процессы разработки и наладить обратную связь. Конечно, полноценно все заработает только в законченной и собранной воедино системе.

```
public class Container {
   private double capacity;
   private Set contents; // Бочки (Drums)
```

```
    public boolean hasSpaceFor(Drum aDrum) {
        return remainingSpace() >= aDrum.getSize();
    }

    public double remainingSpace() {
        double totalContentSize = 0.0;
        Iterator it = contents.iterator();
        while (it.hasNext()) {
            Drum aDrum = (Drum) it.next();
            totalContentSize = totalContentSize + aDrum.getSize();
        }
        return capacity - totalContentSize;
    }

    public boolean canAccommodate(Drum aDrum) {
        return hasSpaceFor(aDrum) &&
            aDrum.getContainerSpecification().isSatisfiedBy(this);
    }
}

public class PrototypePacker implements WarehousePacker {

    public void pack(Collection containers, Collection drums)
                                    throws NoAnswerFoundException {
        /* Этот метод выполняет ПРОВЕРКУ согласно описанию. Однако
           при инициировании исключительной ситуации содержимое
           контейнеров (Containers) может измениться.
           Возврат к прежнему содержимому следует реализовать на
           более высоком уровне. */

        Iterator it = drums.iterator();
        while (it.hasNext()) {
            Drum drum = (Drum) it.next();
            Container container =
                findContainerFor(containers, drum);
            container.add(drum);
        }
    }

    public Container findContainerFor(
                  Collection containers, Drum drum)
                  throws NoAnswerFoundException {
        Iterator it = containers.iterator();
        while (it.hasNext()) {
            Container container = (Container) it.next();
            if (container.canAccommodate(drum))
                return container;
        }
        throw new NoAnswerFoundException();
    }
}
```

Этот код, конечно, оставляет желать много лучшего. Он может, например, поместить песок в спецконтейнер, отчего место в таких контейнерах кончится раньше, чем будут складированы опасные химикаты. В нем полностью отсутствует оптимизация дохода. Впрочем, решения многих задач оптимизации далеки от идеала. Важно, что в текущей реализации полностью соблюдаются правила, сформулированные до сих пор.

Имея этот прототип, разработчики клиентского приложения могут двигаться дальше на полной скорости, включая и интеграцию с внешними системами. Группа разработчиков **Складировщика (Packer)** также получила обратную связь со специалистами, которые работают с прототипом и четче формулируют свои идеи, помогая прояснить требования и приоритеты. Эта группа решает доработать прототип для проверки собственных идей.

Теперь постоянно поддерживается соответствие между интерфейсом и самой последней версией архитектуры, проводится рефакторинг клиентского приложения, учитывается состояние некоторых объектов предметной области, и все это позволяет легко решать проблемы интеграции.

Как только будет готов главный, сложный **Складировщик**, интеграция не составит никакого труда, поскольку он был написан для хорошо знакомого интерфейса — тот самый интерфейс с теми самыми ПРОВЕРКАМИ, для которого писалось клиентское приложение, взаимодействовавшее с прототипом.

Преодоление "мертвых точек" с помощью прототипов

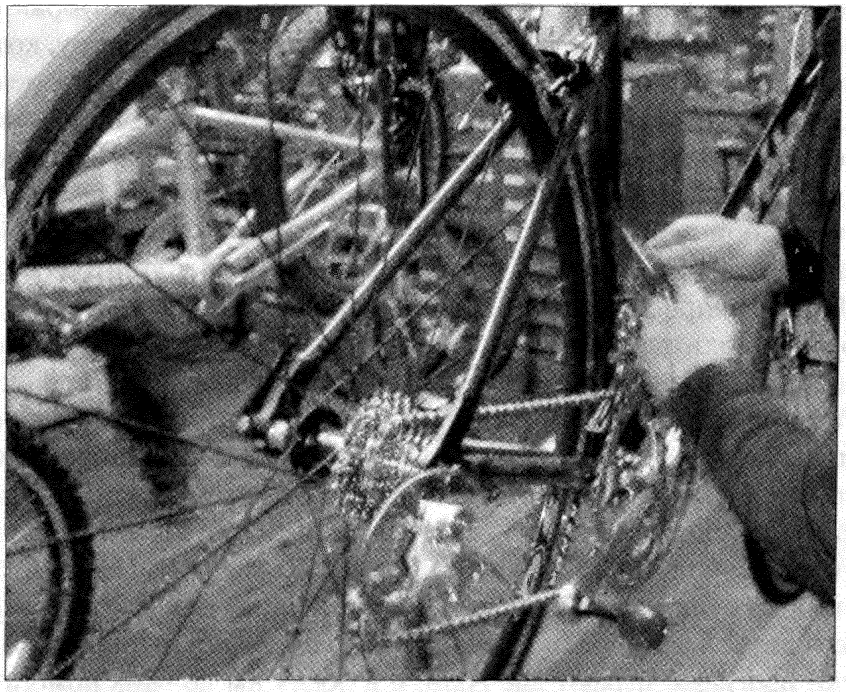
Одна группа должна ожидать, пока вторая напишет рабочий код, чтобы двигаться дальше. Обеим группам приходится ждать полной интеграции, чтобы протестировать свои компоненты или получить обратную связь от пользователей. В такого рода "мертвых точках" часто можно "сдвинуться с места", написав прототип ключевого компонента *на основе предметной модели*, пусть даже он и не удовлетворяет всем функциональным требованиям. Когда реализация независима от интерфейса, тогда можно гибко распараллелить работы по проекту и двигаться дальше, имея всего лишь какую-нибудь, более-менее работающую реализацию (прототип). Когда придет время, этот прототип можно будет заменить более эффективной реализацией. Между тем все остальные части системы получат на время разработки некий заменитель смежного компонента, с которым они смогут взаимодействовать.

Специалистам по алгоритмам оптимизации понадобилось несколько месяцев, чтобы все сделать как следует. Они немало почерпнули из обратной связи с пользователями, работавшими с прототипом, а у остальных частей системы тем временем было с чем взаимодействовать.

Итак, здесь мы видим пример использования "простейшего возможного решения, которое работает". Это стало возможным благодаря наличию достаточно сложной модели. Чтобы получить функционирующий прототип очень сложного компонента, бывает достаточно написать два-три десятка строк легко понятного кода. При подходе, в меньшей степени ориентированном на модель, все это было бы менее понятно, менее удобно в доработке (поскольку **Складировщик** был бы теснее привязан к остальной архитектуре), и в данном случае потребовало бы гораздо большего времени на разработку рабочего прототипа.

Глава 10

Гибкая архитектура



Конечная цель существования программ — служить их пользователям. Но прежде те же самые программы должны послужить их разработчикам. Особенно это верно для технологий разработки, в которых важное место занимает рефакторинг. В ходе эволюции программы разработчики реорганизуют и переписывают каждую ее часть. Они интегрируют объекты предметной области как с приложениями, так и с новыми объектами предметной области. Даже многие годы спустя программисты-доработчики изменяют и дописывают код. Людям *приходится* это делать. Но *захотят* ли они?

Если программное обеспечение, выполняющее сложные операции, не имеет хорошей архитектуры, его элементы трудно поддаются расчленению или слиянию в ходе рефакторинга. Как только программист теряет уверенность в том, что он может предсказать все последствия той или иной операции, начинается дублирование кода. Если элементы архитектуры монолитны и отдельные их части нельзя перекомбинировать в новое целое, тогда дублирование становится неизбежным. Классы и методы можно разбить на части для лучшей переносимости в новый код, но тогда становится тяжело уследить за тем, что делают разные мелкие их части. Когда у программы нет четкой архитектуры, программисты боятся даже заглядывать в хаотическое "месиво" ее кода, не говоря уже о том, чтобы вносить изменения, которые могли бы усугубить путаницу или что-нибудь испортить из-за наличия непредвиденной связи. В любой системе, за исключением самых примитивных, подобная уязвимость задает "потолок" сложности операционных алгоритмов, кото-

рые можно в этой системе реализовать. Рефакторинг и итерационное усовершенствование в такой системе невозможны.

Чтобы темпы реализации проекта увеличить в процессе разработки, а не замедлялись под тяжестью своего собственного богатого наследия, необходима такая архитектура, с которой приятно работать. Архитектура, приглашающая к изменениям. Гибкая архитектура.

Гибкая архитектура (*supple design*) — это логическое дополнение к углубленному моделированию. Как только вы "раскопали" неявные понятия и концепции, а потом перевели их в явные, перед вами оказывается "сырье". Серией итераций вы "выковываете" из этого "сырья" нечто полезное, одновременно формируя модель, которая просто и четко выражает ключевые цели и задачи проекта, и выстраивая архитектуру, которая позволит разработчику клиентской части приложения включить эту модель в реальную работу. Одновременная разработка архитектуры и кода приводит к скачкам в понимании проблемы, отчего совершенствуются и смысловые элементы модели. И снова мы приходим к циклу итераций и рефакторингу по модели, приводящему к более глубокой передаче смысла той прикладной деятельности, которая моделируется программой. Но какая архитектура является целью всего этого процесса? Какие эксперименты следует проводить на этом пути? На эти вопросы отвечает текущая глава.

Во имя гибкости архитектуры в программах было нагромождено множество ненужных конструкций. Лишние уровни абстрагирования и косвенных ссылок чаще мешают, чем помогают в этом деле. Посмотрите на архитектуру, которая действительно вдохновляет программистов, занимающихся ее доработкой, — и вы увидите, как правило, что-нибудь очень простое. Но простое — не значит легкое в исполнении. Чтобы создать такие элементы, которые можно собрать в сложные системы и при этом нетрудно понять, необходимо сочетать "преданность" ПРОЕКТИРОВАНИЮ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN) с достаточно строгим стилем архитектуры. Определенный навык проектирования нужен не только для создания чего-либо, но даже для *использования* готового.

Разработчики программ играют две роли, и в обеих им должна помогать архитектура. Один и тот же человек вполне может играть обе роли — даже переключаться с одной на другую через каждые несколько минут — но, тем не менее, его отношение к коду в этих ролях отличается. Одна из ролей — это разработчик клиентской части приложения, который вплетает объекты предметной области в код прикладных операций или другой код того же уровня предметной области, пользуясь возможностями архитектуры. Гибкая архитектура выражает углубленную модель, лежащую в ее основе, и раскрывает ее потенциал. Разработчик клиентского кода может гибко использовать минимальный набор слабо зависимых между собой понятий для того, чтобы выразить широкий диапазон рабочих сценариев, реализующихся в предметной области. Элементы архитектуры сочетаются друг с другом естественным образом и в итоге выдают результат надежный, легко предсказуемый, четко характеризуемый.

Но в равной степени архитектура должна служить и разработчику, занимающемуся ее доработкой. Чтобы демонстрировать открытость к изменениям, архитектура должна быть легко понятной и выражать *ту же самую* модель, на которую опирается разработчик клиентского кода. Она должна следовать очертаниям углубленной модели предметной области, чтобы большинство изменений "гнули" архитектуру только в тех местах, где она поддается изгибу. Влияние той или иной части кода должно быть очевидным, а последствия изменений соответственно легко предсказуемыми.

Ранние версии программной архитектуры обычно не обладают нужной гибкостью. Многие так и не достигают этого состояния из-за временных или бюджетных ограничений проекта. Я никогда не видел большой программы, которая бы обладала этим качеством в полной мере. Но когда главным препятствием на пути продвижения проекта явля-

ется сложность, “перековка” самых сложных и критических частей программы в гибкую архитектуру сразу показывает разницу между топким болотом доработки устаревшего кода и решительным прорывом потолка сложности.

Стандартных формул для проектирования такого программного обеспечения не существует. Но тем не менее я подобрал ряд архитектурных шаблонов, которые, по моему опыту, придают архитектуре гибкость в тех случаях, где их применение уместно. Эти шаблоны и примеры должны давать общее понятие о том, что такое гибкая архитектура и какое мышление нужно для ее построения.

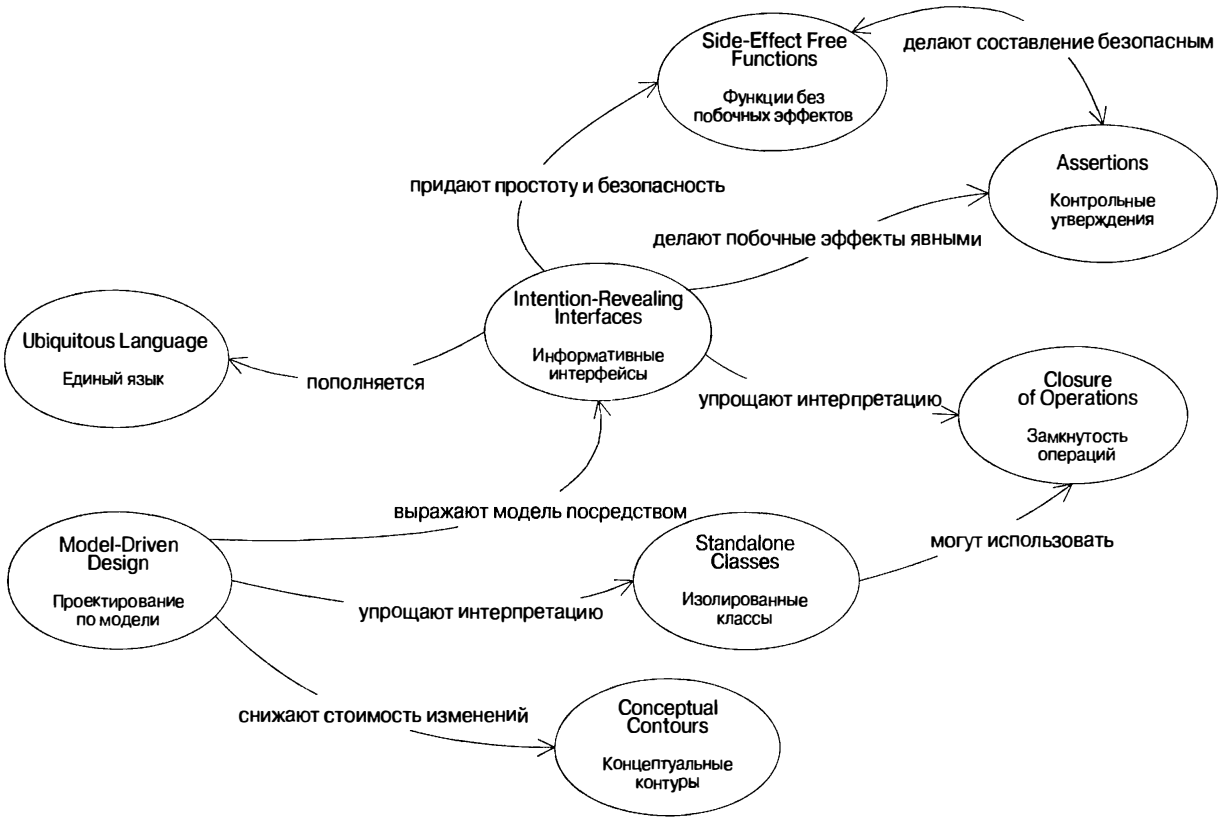


Рис. 10.1. Некоторые архитектурные шаблоны, помогающие построить гибкую архитектуру

Информативные интерфейсы

В предметно-ориентированном проектировании программ наша главная задача — думать о смысловой нагрузке операций из предметной области. Код, который фактически реализует эффект от некоторого правила, но не дает его явного определения, заставляет нас применять пошаговые процедуры, чтобы все-таки его выяснить. То же самое относится и к любой вычислительной операции, которую выполняет определенный фрагмент кода, при этом не открывая ее логический смысл. Не имея прямой привязки к модели, трудно понять, какой эффект имеет тот или иной код или предвидеть последствия вносимого изменения. В предыдущей главе рассматривалось моделирование правил и операций как явных объектов. Реализация таких объектов требует глубокого понимания мельчайших деталей операций и всевозможных тонкостей, заложенных в моделируемых правилах. Красота и сила объектов состоит в том, что они умеют инкапсулировать все это, чтобы клиентский код оставался простым и доступным для интерпретации в понятиях более высокого уровня.

Но если интерфейс не сообщает разработчику достаточной информации для того, чтобы использовать объект эффективно, ему придется самостоятельно залезть во "внутренности" и раскопать все детали. То же самое будет вынужден делать и читатель клиентского кода. В результате ценность инкапсуляции сильно падает. Мы всегда стараемся бороться с когнитивной перегрузкой: если голова разработчика клиентского кода загромождена подробностями о работе того или иного компонента, тонкости архитектуры собственно клиентского кода в ней уже не помещаются. Это справедливо даже в том случае, если один и тот же человек играет обе роли — и пишет, и использует написанный код. Даже если ему и не надо заново выяснять все детали, все равно есть предел количеству факторов, которые человек может рассматривать одновременно.

Если для правильного использования какого-нибудь компонента программисту приходится изучать его реализацию, от этого теряется смысл инкапсуляции. Если кто-то другой (не исходный разработчик) вынужден выяснять назначение объекта или смысл операции по их реализации, он может сделать правильные выводы о таком смысле только лишь случайно. И если вывод о назначении компонента сделан неправильно, то код может даже проработать какое-то время, но концептуальная основа архитектуры будет искажена, и два разработчика окажутся в роли антагонистов.

Чтобы понятия выражались в коде в явных формах классов или методов, элементам программы нужно давать имена, отражающие соответствующие понятия. Правильное именование классов и методов — это прекрасный способ улучшить коммуникацию в среде разработчиков, а также качество абстрагирования системы.

Кент Бек (Kent Beck) уже писал о том, как через имена методов передавать их назначение с помощью ИНФОРМАТИВНОГО СЕЛЕКТОРА (INTENTION-REVEALING SELECTOR) [4]. Все общедоступные (*public*) элементы архитектуры некоторого программного компонента образуют его интерфейс, и выбор имени для каждого из таких элементов — это возможность передать информацию о назначении данного компонента. Имена типов, имена методов, имена аргументов — все это вместе образует ИНФОРМАТИВНЫЙ ИНТЕРФЕЙС (INTENTION-REVEALING INTERFACE)

Давайте такие имена классам и операциям, чтобы они описывали их назначение и получаемый результат, но не способ выполнения ими своих функций. Это избавляет разработчика клиентского кода от необходимости понимать "внутреннюю кухню" этих объектов. Имена должны соответствовать терминам ЕДИНОГО ЯЗЫКА**, чтобы все члены рабочей группы понимали, о чем идет речь. Напишите тест для алгоритма, прежде чем писать сам алгоритм, чтобы ввести свое мышление в режим разработчика клиентского кода.**

Весь хитроумный механизм должен инкапсулироваться за абстрактными интерфейсами, которые информируют клиента о смысле и назначении своих функций, а не о средствах их реализации.

В *public*-интерфейсах уровня предметной области определяйте взаимосвязи и правила, но не способы, которыми они должны осуществляться. Описывайте суть событий и операций, но не то, как именно они должны происходить. Формулируйте уравнение, но не численный метод его решения. Ставьте вопрос, но не раскрывайте средств поиска ответа.

Пример

Рефакторинг: программа смешивания красок

Программа, предназначенная для обслуживания торговли красками, может показать покупателю результат смешивания стандартных красок. Вот первоначальная ее архитектура, включающая один-единственный класс предметной области.

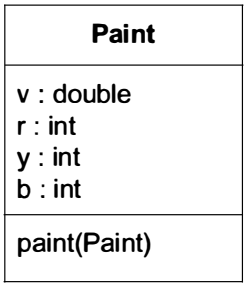


Рис. 10.2.

Единственный способ догадаться, что делает метод `paint(Paint)` — это прочитать код, так как его заголовок означает всего лишь *краска(Краска)*.

```
public void paint(Paint paint) {
   v = v + paint.getV(); //После смешивания объем суммируется
   // Опущено много строк сложного расчета смешивания цветов,
   // который заканчивается присваиванием новых значений
   // компонентов r (красного), b (синего) и y (желтого).
}
```

Итак, этот метод, похоже, смешивает две **Краски (Paints)**, в результате чего получается больший объем и новый смешанный цвет.

Чтобы взглянуть на проблему под другим углом, давайте напишем тест для этого метода. (Код совместим со средой тестирования JUnit.)

```
public void testPaint() {
    // Создаем чистую желтую краску объемом=100
    Paint yellow = new Paint(100.0, 0, 50, 0);
    // Создаем чистую синюю краску объемом=100
    Paint blue = new Paint(100.0, 0, 0, 50);

    // Смешиваем синюю краску с желтой
    yellow.paint(blue);

    // Должно получиться 200.0 единиц зеленой краски
    assertEquals(200.0, yellow.getV(), 0.01);
    assertEquals(25, yellow.getB());
    assertEquals(25, yellow.getY());
    assertEquals(0, yellow.getR());
}
```

Пройденный тест — это наша отправная точка. На этом этапе код отталкивает нас своей неинформативностью: он не сообщает, что он делает. Давайте перепишем тест так, чтобы он отражал *желаемый* способ использования объектов **Краска (Paint)** — как если бы мы писали клиентское приложение. Вначале этот тест окончится неудачей. Фактически он даже не будет скомпилирован. Мы пишем его только для того, чтобы исследовать интерфейс объекта **Краска** с точки зрения разработчика клиентского кода.

```
public void testPaint() {
    // Начинаем с чистой желтой краски объемом=100
    Paint ourPaint = new Paint(100.0, 0, 50, 0);
    // Берем чистую синюю краску объемом=100
    Paint blue = new Paint(100.0, 0, 0, 50);
```

```
    // Примешиваем синюю краску к желтой
    ourPaint.mixIn(blue);  // нашаКраска.примешать(синяя);

    // Должно получиться 200.0 единиц зеленой краски
    assertEquals(200.0, ourPaint.getVolume(), 0.01);
    assertEquals(25, ourPaint.getBlue());
    assertEquals(25, ourPaint.getYellow());
    assertEquals(0, ourPaint.getRed());
}
```

Не пожалеем времени для того, чтобы написать тест в нужном стиле "общения" с интересующими нас объектами. После этого выполним рефакторинг класса **Краска** (**Paint**), чтобы наконец пройти тест.

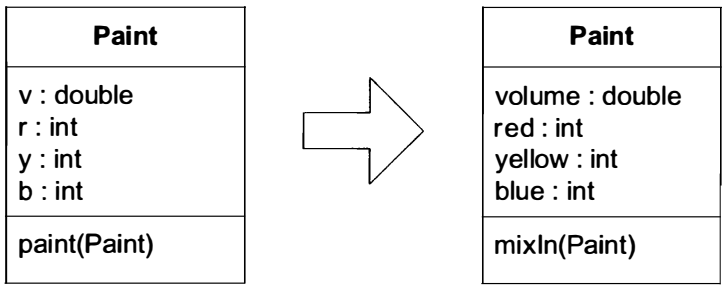


Рис. 10.3.

Новое имя метода, может, и не расскажет читателю все об эффекте "примешивания" другой **Краски** (для этого понадобятся УТВЕРЖДЕНИЯ, или ASSERTIONS, о которых вскоре будет идти речь). Но читатель получит достаточную подсказку, чтобы начать использовать этот класс, особенно при наличии примера из теста. А читатель клиентского кода сможет интерпретировать смысл, заложенный в код. В нескольких следующих примерах этой главы мы продолжим рефакторинг класса для дальнейшего улучшения его понятности и удобочитаемости.

* * *

Выделять в отдельные модули и скрывать за ИНФОРМАТИВНЫМИ ИНТЕРФЕЙСАМИ можно целые подобласти уровня предметной области. Использование такого приема для конкретизации проектных задач и управления сложностью больших систем будет более подробно рассматриваться в главе 15, при вводе СВЯЗНЫХ МЕХАНИЗМОВ (COHESIVE MECHANISMS) и ЕСТЕСТВЕННЫХ ПОДОБЛАСТЕЙ (GENERIC SUBDOMAINS).

В следующих двух архитектурных шаблонах мы попытаемся сделать последствия использования того или иного метода как можно более предсказуемыми. Сложные алгоритмы можно безопасно реализовать в ФУНКЦИЯХ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ (SIDE-EFFECT-FREE FUNCTIONS). Методы, изменяющие состояние системы, можно характеризовать с помощью УТВЕРЖДЕНИЙ (ASSERTIONS).

Функции без побочных эффектов

Все программные операции можно приблизительно разделить на две категории — команды и запросы. Запросы получают информацию от системы — иногда простым обращением к переменной, иногда через какие-то вычисления над данными. А вот команды (известные также как модификаторы) — это операции, которые вносят в систему какие-

то изменения (в самом простом случае — присваивают значение переменной). В повседневном языке *побочным эффектом* (*side effect*) называется некое непреднамеренное последствие того или иного действия. Но в программировании этим выражением обозначают любое влияние на состояние системы. Для наших целей давайте несколько сузим это понятие. Назовем побочным эффектом такое изменение в состоянии системы, которое влияет на ее будущие операции.

Почему для обозначения вполне намеренных изменений принят и применяется такой термин, как *побочный эффект*? Я думаю, это следствие практического опыта работы со сложными системами. Большинство операций зависит от других операций, а те операции инициируют третьи операции и т.д. Если эту вложенность операций не контролировать, становится очень тяжело предсказать все последствия вызова той или иной операции. Разработчик клиентского кода может и не осознавать его влияния на операции второго и третьего уровня вложенности — так они становятся побочными эффектами уже во всех смыслах. Элементы сложной архитектуры взаимодействуют между собой и другими способами, которые тоже способны порождать непредсказуемость. Выражение *побочный эффект* подчеркивает именно неизбежность такого взаимодействия.

Взаимное влияние нескольких правил или комбинаций вычислений бывает очень трудно предсказать. Программист, инициирующий какую-то операцию, должен понимать ее реализацию и реализацию всех вложенных в нее операций, чтобы предвидеть результат. Польза от абстрагирования интерфейсов невелика, если программистам приходится заглядывать за барьер этого абстрагирования. Не имея безопасного и предсказуемого абстрагирования операций, программисты вынуждены ограничивать комбинирование операций, тем самым накладывая слишком суровые ограничения на алгоритмическую сложность кода, которую можно было бы реализовать.

Операции, которые возвращают какой-то результат, не создавая побочных эффектов, называются *функциями*. Функцию можно вызывать много раз, каждый раз получая с ее помощью один и тот же результат. Функция может вызывать другие функции, не беспокоясь о глубине вложенности операций. Функции гораздо легче тестировать, чем операции с побочными эффектами. Поэтому применение функций снижает риск.

Очевидно, в большинстве программных систем нельзя избежать применения команд, но есть два способа смягчить проблему. Во-первых, можно держать команды и запросы строго разграниченными, оформлять их как разные операции. Сделайте так, чтобы методы, вызывающие изменения, не возвращали данные с уровня предметной области и были устроены как можно проще. Выполняйте все запросы и вычисления в методах, которые не вызывают никаких видимых побочных эффектов [19].

Во-вторых, часто можно построить альтернативные модели или архитектуры, в которых существующий объект вообще никогда не модифицируется. Вместо этого создается и возвращается новый ОБЪЕКТ-ЗНАЧЕНИЕ (VALUE OBJECT), представляющий результат вычислительной операции. Это распространенный прием, который будет продемонстрирован в следующем примере. ОБЪЕКТ-ЗНАЧЕНИЕ можно создать в ответ на запрос, передать куда следует, а потом забыть про него — в отличие от объекта-СУЩНОСТИ (ENTITY), цикл существования которого строго регламентируется и отслеживается.

ОБЪЕКТЫ-ЗНАЧЕНИЯ по своей природе неизменяемы, и это подразумевает, что *все* их операции — функции, если не считать инициализаций, происходящих только в момент создания объекта. ОБЪЕКТЫ-ЗНАЧЕНИЯ, как и функции, безопаснее в применении и проще в тестировании. Операцию, в которой вычисления или выполнение сложного алгоритма смешивается с изменением состояния, нужно факторизовать в две отдельные операции [12]. Но по определению такое выделение побочных эффектов в простые командные методы относится только к СУЩНОСТЯМ. *После* рефакторинга, отделяющего

модификацию от запроса, следует подумать о втором рефакторинге: вынесении сложных вычислений в ОБЪЕКТ-ЗНАЧЕНИЕ. Побочный эффект часто можно совсем ликвидировать, произведя на свет новый ОБЪЕКТ-ЗНАЧЕНИЕ вместо изменения какого-либо текущего состояния или передав всю ответственность за операцию в ОБЪЕКТ-ЗНАЧЕНИЕ.

Выносите как можно больше алгоритмической логики программы в функции, т.е. операции, которые возвращают результат, но не имеют видимых побочных эффектов. Четко выделяйте команды (методы, которые вносят модификации в наблюдаемые состояния объектов) в очень простые операции, которые не возвращают никакой информации из уровня предметной области. Дополнительно контролируйте побочные эффекты, вынося сложные вычисления в ОБЪЕКТЫ-ЗНАЧЕНИЯ**, если для таких операций имеются соответствующие концептуальные понятия.**

ФУНКЦИИ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ (SIDE-EFFECT-FREE FUNCTIONS), особенно инкапсулированные в неизменяемых ОБЪЕКТАХ-ЗНАЧЕНИЯХ, позволяют безопасно комбинировать операции. Когда такая функция предоставляется пользователю через ИНФОРМАТИВНЫЙ ИНТЕРФЕЙС, он может смело применять ее, не зная никаких подробностей ее реализации.

Пример

Дальнейший рефакторинг программы смешивания красок

Напоминаем, что речь идет о программе для торговли красками, которая может показать покупателю результат смешивания стандартных красок. Продолжая с того места, где мы закончили в предыдущем примере, вспомним наш единственный класс предметной области.

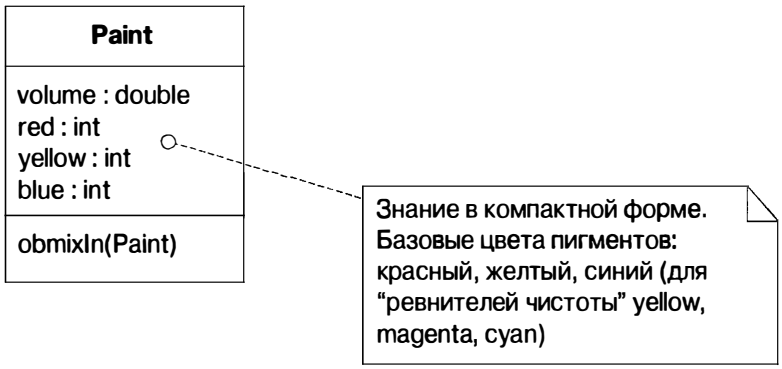


Рис. 10.4.

```
public void mixIn(Paint other) {
   volume = volume.plus(other.getVolume());
   // Много строк сложного расчета смешивания цветов,
   // который заканчивается присваиванием новых значений
   // компонентов r (красного), b (синего) и y (желтого).
}
```

В методе примешивания `mixIn()` много чего происходит, но в этой архитектуре строго соблюдается правило отделения модификации от запроса. Одна из проблем, которой мы займемся позже, заключается в том, что объем краски 2, аргумент метода `mixIn()`, остается в "подвешенном" состоянии. Объем краски 2 не изменяется в ходе этой операции, что кажется не совсем логичным в контексте этой концептуальной модели. Для исходных раз-

работчиков это не было проблемой, потому что, насколько можно судить, объект "краска 2" после операции их больше не интересовал, но все последствия побочных эффектов (или их отсутствие) предвидеть трудно. Мы вскоре вернемся к этому вопросу при рассмотрении УТВЕРЖДЕНИЙ (ASSERTIONS). А пока давайте займемся цветами.

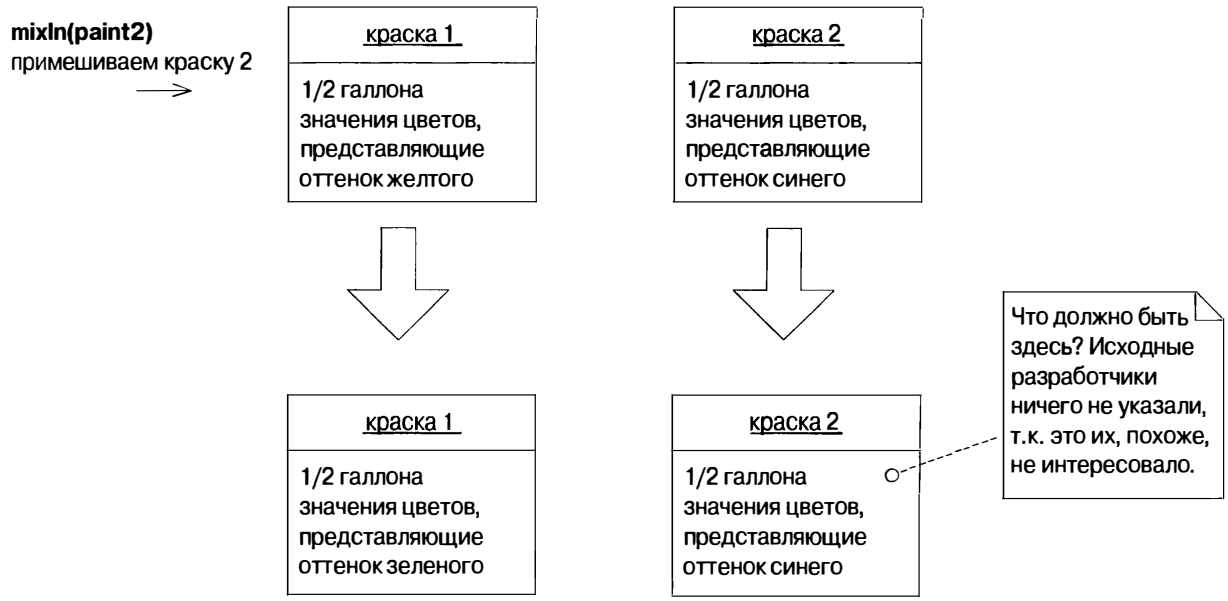


Рис. 10.5. Побочные эффекты метода `mixIn()`

Цвет в этой прикладной модели является важным понятием. Попробуем в виде эксперимента сделать его явным объектом. Как же назвать такой объект? Первым в голову приходит слово "Цвет", но проведенная ранее переработка знаний дала разработчикам важный вывод, что смешивание цветов для красок имеет более сложный характер, чем компонентов цветовой модели дисплея RGB. Имя должно отражать этот факт.

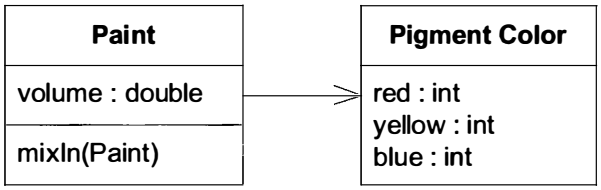


Рис. 10.6.

Выделение класса **Пигментный цвет (Pigment Color)** действительно делает архитектуру более информативной, чем раньше, но расчет выполняется так же — по-прежнему в методе примешивания `mixIn()`. Выделяя данные о цвете в отдельный объект, было бы неплохо вместе с ними выделить и нужные операции. Но прежде чем мы это сделаем, заметьте, что **Пигментный цвет** — это ОБЪЕКТ-ЗНАЧЕНИЕ. Поэтому его следует считать неизменяемым. При смешивании красок изменялся сам объект **Краска (Paint)**. Это была СУЩНОСТЬ с непрерывной историей существования. Но **Пигментный цвет**, представляющий определенный оттенок желтого цвета, всегда является именно им, и ничем другим. Смешивание не изменяет его, а дает новый объект **Пигментный цвет**, представляющий новый цвет.

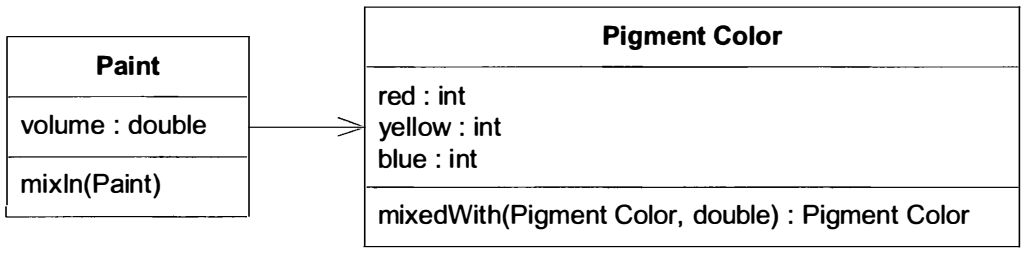


Рис. 10.7.

```
public class PigmentColor {

   public PigmentColor mixedWith(PigmentColor other,
                                 double ratio) {
      // Много строк сложного расчета смешивания цветов.
      // В результате создается новый объект PigmentColor
      // с новыми пропорциями красного, синего и желтого.
   }
}

public class Paint {

   public void mixIn(Paint other) {
      volume = volume + other.getVolume();
      double ratio = other.getVolume() / volume;
      pigmentColor =
         pigmentColor.mixedWith(other.pigmentColor(), ratio);
   }
}
```

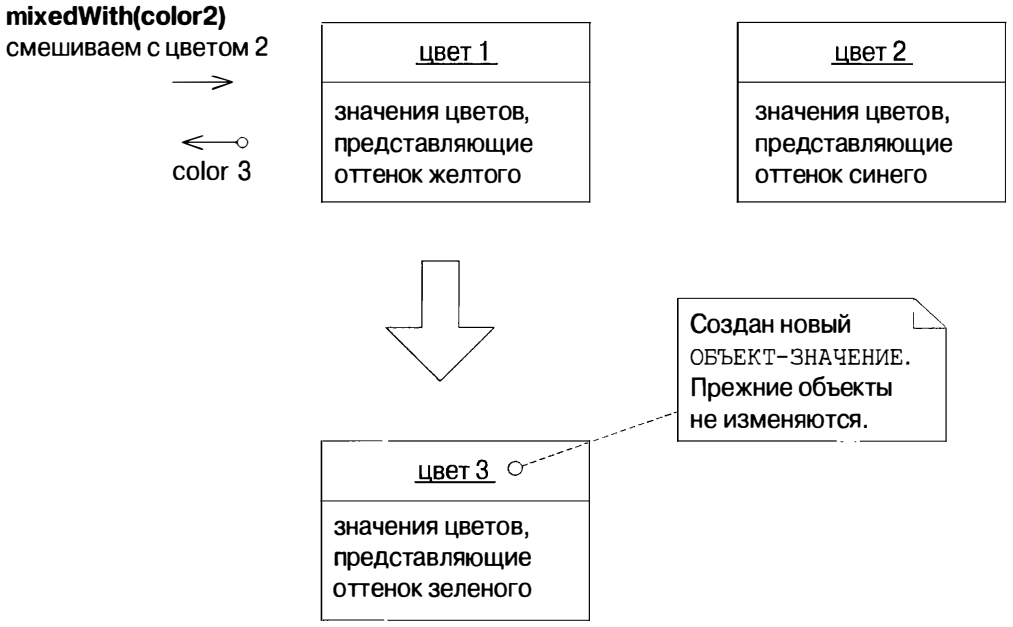


Рис. 10.8.

Теперь код модификации в классе **Краска (Paint)** упростился до предела. Новый класс **Пигментный цвет (Pigment Color)** заключает в себе знание предметной области и явно выражает его. Он предоставляет программисту-пользователю функцию без побочных эффектов, результат работы которой легко понятен, *прост в тестировании* и безопасен

в использовании или комбинировании с другими операциями. Благодаря высокому уровню безопасности сложный алгоритм смешивания цветов действительно инкапсулируется — разработчики, использующие этот класс, не обязаны что-либо знать о его реализации.

* * *

Утверждения

Выделение сложных вычислений в ФУНКЦИИ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ радикально упрощает проблему, но все-таки и после этого еще остается много команд в объектах-СУЩНОСТЯХ, которые порождают побочные эффекты. И все, кто пользуется такими объектами, должны понимать последствия этого. УТВЕРЖДЕНИЯ (ASSERTIONS) позволяют сделать побочные эффекты явными и упростить работу с ними.

* * *

Команду, не содержащую никаких сложных вычислений, сравнительно несложно понять, просто прочитав ее код. Но в архитектуре, где большие структуры собраны из меньших частей, команда может обращаться к другим командам. Разработчик, использующий команду высокого уровня, должен понимать последствия каждой вызываемой ею команды с более низких уровней. Вот вам и инкапсуляция! Поскольку интерфейсы объектов не накладывают никаких ограничений на побочные эффекты, два подкласса, реализующих один и тот же интерфейс, могут иметь разные побочные эффекты. Разработчик, использующий их, должен знать, где какой класс, чтобы правильно предвидеть последствия. Вот вам и абстрагирование с полиморфизмом!

Если побочные эффекты операций определены в их реализации только неявно, то в архитектурах с интенсивной передачей управления возникает сложное переплетение причин и следствий. Единственным способом понять программу становится трассировка ее выполнения по всем возможным ветвям. При этом теряется смысл инкапсуляции, а необходимость в трассировке выполнения делает бессмысленным также и абстрагирование.

Необходимо найти такой способ для понимания смысла тех или иных элементов архитектуры и последствий выполнения операций, который бы не требовал залезания к ним вовнутрь. ИНФОРМАТИВНЫЕ ИНТЕРФЕЙСЫ частично решают проблему, но неформальной передачи смысла и предназначения элементов не всегда бывает достаточно. Школа "контрактного проектирования" (*design by contract*) делает следующий шаг, организуя проверки утверждений относительно классов и методов, выполнение которых гарантирует разработчик. Этот стиль программирования подробно рассмотрен в книге [19]. Кратко говоря, есть "пост-условия", описывающие побочные эффекты операции — гарантированные последствия вызова метода. А есть "предусловия" — аналоги условий в контракте, которые необходимо соблюсти, чтобы пост-условие дало гарантированный результат. Инварианты классов — это и есть утверждения о состояния объекта в конце любой операции. Инварианты можно также объявлять для целых АГРЕГАТОВ, строго определяя правила сохранения целостности.

Все эти утверждения относятся к состояниям, а не процедурам, поэтому их легко анализировать. Инварианты классов помогают охарактеризовать смысл класса и упростить работу разработчика клиентского кода, делая поведение объектов более понятным и предсказуемым. Если вы доверяете гарантии, которую дает пост-условие, вам уже не надо беспокоиться о том, как именно работает метод. Все эффекты передачи управления уже должны быть учтены в контрольных утверждениях.

Формулируйте пост-условия для операций, а также инварианты для классов и АГРЕГАТОВ. **Если** УТВЕРЖДЕНИЯ **нельзя непосредственно запрограммировать в среде вашего языка, напишите модульные тесты для их проверки. Включите их в документацию или схемы, если это позволяет принятый стиль разработки проекта.**

Пытайтесь строить модели со связным набором понятий, которые заставляют разработчика формулировать необходимые контрольные УТВЕРЖДЕНИЯ**, ускоряя процесс обучения и снижая риск появления противоречий в коде.**

Хотя во многих современных объектно-ориентированных языках программирования УТВЕРЖДЕНИЯ напрямую не поддерживаются, мыслить в таких категориях все же полезно для архитектуры. Нехватку поддержки в языке можно частично компенсировать автоматизированными модульными тестами. УТВЕРЖДЕНИЯ формулируются в терминах состояний, а не процедур, поэтому тесты для них писать легко. При инициализации теста устанавливаются требуемые предусловия; затем, после выполнения теста, проверяется выполнение пост-условий.

Четко сформулированные инварианты, предусловия и пост-условия помогают разработчику понять последствия использования операции или объекта. Теоретически для работы годится любой непротиворечивый набор УТВЕРЖДЕНИЙ. Но процесс мышления у людей в головах не сводится к компилированию предикатов. Люди экстраполируют и интерполируют понятия моделей, поэтому важно найти такие модели, которые не только помогают писать программу, но и выглядят осмысленными.

Пример

Снова о смешивании красок

Напомню, что в предыдущем примере было не совсем ясно, что происходит с аргументом операции `mixIn(Paint)` в классе **`Paint` (Краска)**, и это вызывало беспокойство.

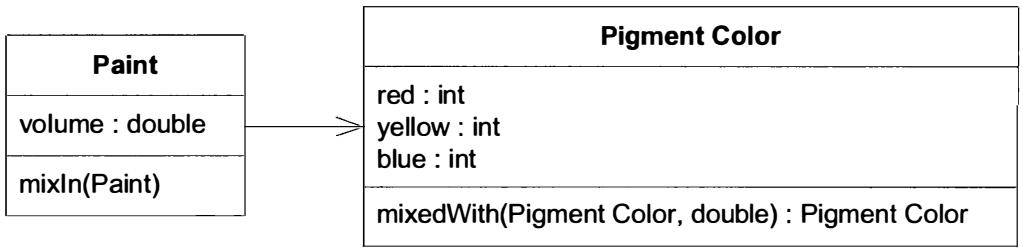


Рис. 10.9.

Объем объекта-получателя увеличивается на величину объема аргумента. Исходя из наших обыденных представлений о красках в физическом мире, в процессе смешивания из другой краски следует вычесть такой же объем, то есть осушить ее до нулевого объема или удалить совсем. В текущей реализации аргумент не модифицируется. К тому же внесение изменений в аргументы — это сам по себе побочный эффект довольно рискованного свойства.

Чтобы встать на твердую почву, давайте сформулируем пост-условие для метода `mixIn()` *как оно есть*.

```
После p1.mixIn(p2):
    p1.volume увеличивается на объем p2.volume
    p2.volume не изменяется
```

Беда в том, что программисты обязательно сделают здесь ошибку, потому что эти условия не соответствуют понятиям, которые мы предложили им для обдумывания. Топорным решением будет обнуление объема второй краски. Изменение аргумента — плохой стиль, но зато здесь все просто и интуитивно понятно. Можно сформулировать инвариант.

```
Общий объем краски не должен измениться от смешивания
```

Но погодите-ка! Пока программисты прорабатывали этот вариант, они поняли, что у исходных разработчиков была веская причина сделать все именно так, как было. По итогам вычислений программа *выдает список несмешанных красок, которые были добавлены в смесь*. В конце концов, перед программой стоит задача помочь пользователю определить, *из каких красок нужно составлять смесь*.

Итак, сделать модель логически непротиворечивой означает одновременно сделать ее непригодной для выполнения задач программы. Возникает дилемма. Что же, нам остается только задокументировать странное пост-условие и постараться компенсировать эту странность понятностью изложения. Не все в нашем мире интуитивно понятно, и иногда приходится поступать именно так. Но в данном случае противоречие, скорее всего, указывает на нехватку некоторых понятий. Попытаемся поискать новую модель.

Теперь все ясно

В наших поисках лучшей модели мы имеем преимущество перед исходными разработчиками, поскольку в промежутке мы уже произвели переработку знаний и рефакторинг по модели. Например, мы вычисляем цвет с помощью ФУНКЦИИ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ в ОБЪЕКТЕ-ЗНАЧЕНИИ. А это значит, что такое вычисление можно повторять сколько угодно раз по мере необходимости. Этим надо воспользоваться.

Кажется, класс **`Краска (Paint)`** имеет две разные базовые обязанности. Давайте попробуем разделить их.

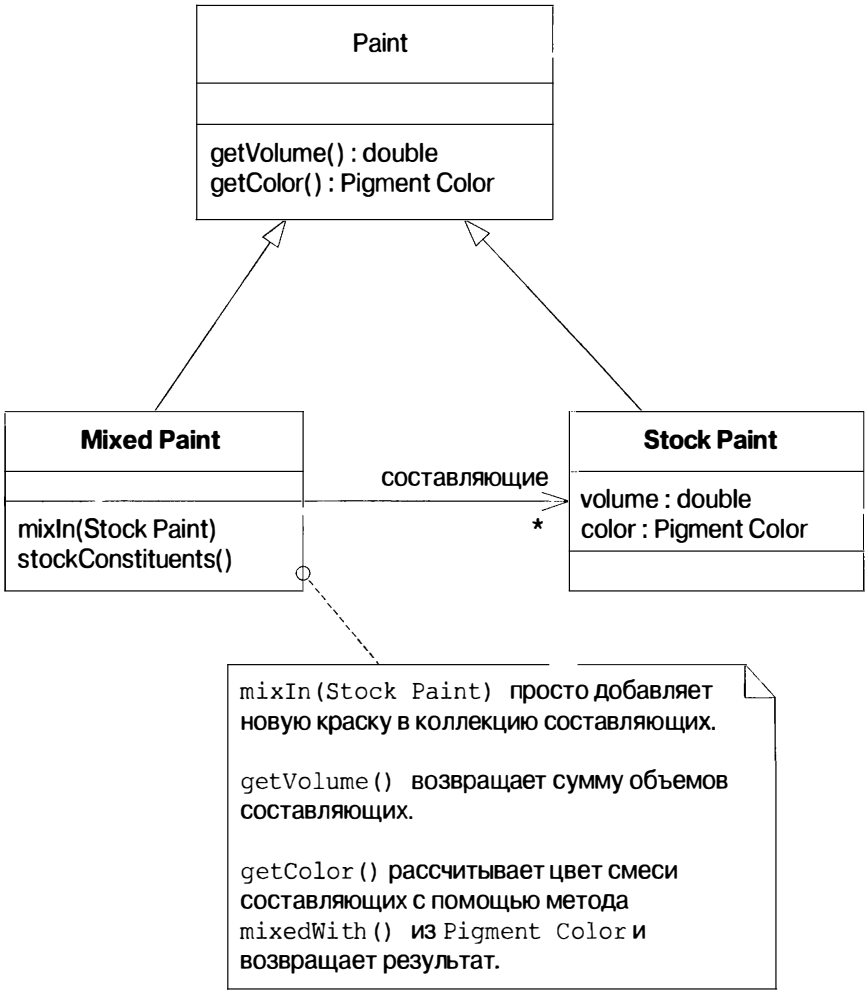


Рис. 10.10.

Теперь у нас только одна команда — `mixIn()`. Она просто добавляет объект в коллекцию; этот эффект очевиден из интуитивного представления о модели. Все остальные операции являются ФУНКЦИЯМИ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ.

Тестовый метод, который бы проверял одно из утверждений, перечисленных на рис. 10.10, мог бы выглядеть примерно так (с использованием среды тестирования JUnit).

```
public void testMixingVolume {
    PigmentColor yellow = new PigmentColor(0, 50, 0);
    PigmentColor blue = new PigmentColor(0, 0, 50);

    StockPaint paint1 = new StockPaint(1.0, yellow);
    StockPaint paint2 = new StockPaint(1.5, blue);
    MixedPaint mix = new MixedPaint();

    mix.mixIn(paint1);
    mix.mixIn(paint2);
    assertEquals(2.5, mix.getVolume(), 0.01);
}
```

Эта модель заключает в себе больше знаний и лучше описывает предметную область. Инварианты и пост-условия обладают четким смыслом, поэтому их легче использовать и дорабатывать.

---

* * *

Наглядность и описательность ИНФОРМАТИВНОГО ИНТЕРФЕЙСА (INTENTION-REVEALING INTERFACE) в сочетании с четкой предсказуемостью поведения, которую обеспечивают ФУНКЦИИ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ (SIDE-EFFECT-FREE FUNCTIONS) и УТВЕРЖДЕНИЯ (ASSERTIONS), способны сделать инкапсуляцию и абстрагирование вполне безопасными.

Следующая составляющая хорошей системы рекомбинируемых элементов — это эффективное разбиение, или декомпозиция.

Концептуальные контуры

Иногда программисты мелко "нарезают" функциональные возможности, чтобы сделать их комбинируемость более гибкой. Иногда они, наоборот, укрупняют их для инкапсуляции сложных алгоритмов. Бывает и так, что программисты добиваются единообразия в масштабе разбиения, пытаясь привести все классы и операции к одному порядку величины. Все это — грубые упрощения, которые плохо применимы в качестве общих правил. Но все-таки существует некоторый базовый набор задач, который мотивирует их применение.

Когда элементы модели или программной архитектуры слиты в одну монолитную конструкцию, их функциональные возможности дублируются. Внешний интерфейс не сообщает всего, что может оказаться нужным клиенту. Смысл элементов становится трудно понять, поскольку смешиваются в одно целое различные понятия.

С другой стороны, разбиение на классы и методы может привести к ненужному усложнению клиентского кода, потому что объектам-клиентам приходится знать, как сочетаются между собой мелкие программные фрагменты. И что еще хуже, нужное понятие может вообще потеряться среди всего этого. Половина атома урана — это уже не уран. Разумеется, имеет значение не то, каков размер фрагмента, а то, где проходит его граница.

Простые наборы стандартных рецептов тут не годятся. Но в большинстве предметных областей заложено глубокое внутреннее логическое единство, иначе они не представляли бы ценности. Нельзя сказать, что всякая предметная область однозначно непротиворечи-

ва и последовательна, и уж точно не отличаются этими качествами высказывания пользователей о предметных областях. Но где-то, на каком-то уровне, все-таки существует порядок и гармония, иначе моделирование не имело бы смысла. Эта внутренняя самосогласованность проявляется в том, что если мы находим модель, которая "звучит в унисон" с какой-то из частей предметной области, то она, скорее всего, будет хорошо согласована и с другими ее частями, которые мы откроем для себя позже. Иногда модель бывает не так-то просто приспособить к новым открытиям, и в этом случае приходится выполнять глубокий рефакторинг, надеясь, что *следующее* открытие окажется для нее более удобоваримым.

Это одна из причин, почему повторяющийся рефакторинг постепенно приводит к гибкой архитектуре. По мере адаптации кода к новому пониманию теоретических концепций или функциональных требований возникают КОНЦЕПТУАЛЬНЫЕ КОНТУРЫ (CONCEPTUAL CONTOURS).

"Близнецы-братья" — высокая внутренняя связность и низкая внешняя зависимость — играют роль во всех масштабах архитектуры: от отдельных методов, классов и МОДУЛЕЙ до крупномасштабных структур (см. главу 16). Эти два принципа применимы к понятиям в той же мере, что и к коду. Чтобы избежать сползания в чисто механистический взгляд на них, закаляйте ваше техническое мышление, периодически проверяя его интуитивным пониманием предметной области. Принимая любое решение, спрашивайте себя: "Что это — техническая уловка, основанная на каком-то наборе взаимосвязей в текущем коде и модели, или же отражение некоего смыслового контура в предметной области?".

Найдите концептуальную, смысловую единицу требуемой функциональности, и получившаяся архитектура станет одновременно гибкой и понятной. Например, если "сложение" двух объектов имеет логичный смысл в предметной области, то реализуйте методы именно на этом уровне. Не разбивайте метод сложения `add()` на два шага. Не переходите к следующему шагу в пределах одной операции. В слегка увеличенном масштабе каждый объект должен представлять собой одно законченное понятие, "ЦЕЛОСТНЫЙ ОБЪЕКТ-ЗНАЧЕНИЕ" (WHOLE VALUE[1]).

Аналогичные признаки позволяют выделить в любой предметной области такие подобласти, подробности устройства которых неинтересны будущим пользователям программы. Пользователи нашей гипотетической программы смешивания красок не добавляют в смесь красный или синий пигмент. Они смешивают готовые краски, в которых содержатся все три пигмента. Объединение в одно целое того, что не требует реорганизации или рассечения на части, позволяет избежать хаоса и разглядеть те элементы, которые действительно имеет смысл рекомбинировать. Если бы реальное оборудование наших пользователей позволяло добавлять отдельные пигменты, предметная область имела бы другой вид, и там можно было бы манипулировать пигментами. Химику, занимающемуся разработкой красок, тоже нужен совсем другой уровень контроля над процессом, и там нужно было бы провести другой анализ проблемы. В результате могла бы получиться гораздо более подробная модель создания краски, чем наш абстрактный "пигментный цвет". Но это не имеет никакого значения для участников проекта по разработке программы смешивания.

Разбивайте элементы архитектуры (операции, интерфейсы, классы и АГРЕГАТЫ**) на связные единицы, учитывая свое интуитивное понимание смысловых границ предметной области. Наблюдайте за направлениями изменений и осями стабильности в ходе последовательного рефакторинга, ищите** КОНЦЕПТУАЛЬНЫЕ КОНТУРЫ **(**CONCEPTUAL CONTOURS**), по которым происходит расслоение между ними. Прежде всего согласуйте**

[1] Это архитектурный шаблон, автор Уорд Каннингем (Ward Cunningham).

модель с теми логически последовательными и стройными аспектами предметной области, которые делают данную область жизнеспособной, практически ценной.

Цель этой деятельности — создать простой набор интерфейсов, которые можно логически сочетать между собой для составления осмысленных утверждений на ЕДИНОМ ЯЗЫКЕ, притом не отвлекаясь на техническое обслуживание маловажных аспектов. Обычно это получается в результате рефакторинга; заранее запланировать подобное очень трудно. Но в ходе чисто технического рефакторинга эта цель может и не быть достигнута; тут нужен рефакторинг на основе углубленной модели.

Даже если архитектура программы следует КОНЦЕПТУАЛЬНЫМ КОНТУРАМ, модификации и рефакторинг неизбежны. Когда успешный рефакторинг имеет тенденцию оставаться локальным, не затрагивая сразу много общих понятий модели, это показатель адекватности модели. Если же встречается требование к программе, которое заставляет вносить серьезные изменения в структуру объектов и методов, это уже сигнал: нашему пониманию предметной области не хватает глубины. Тем самым нам предоставляется возможность углубить модель и сделать архитектуру более гибкой.

Пример

Контуры в начислениях

В главе 9 мы занимались рефакторингом системы управления кредитами на основе углубляющихся знаний основных понятий из бухгалтерского учета.

Новая модель содержала всего на один объект больше, чем старая, но разграничение обязанностей изменилось очень сильно.

Графики (временны*е*), которые в классах **`Калькулятор (Calculator)`** реализовались через ветвление, были выделены в отдельные классы для разных видов сборов и процентов. С другой стороны, выплата сборов и процентов, которая раньше была разделена на отдельные процедуры, стала располагаться в одном месте.

Видя громкий резонанс, вызванный новыми явными понятиями, и связность иерархии **`Графиков начислений (Accrual Schedule)`**, разработчик стал полагать, что эта модель лучше отражает некоторые из КОНЦЕПТУАЛЬНЫХ КОНТУРОВ предметной области.

Изменение, которое разработчик смог уверенно предсказать, состояло в добавлении новых **`Графиков начислений`**. Эти требования уже, можно сказать, витали в воздухе. Поэтому, кроме упрощения и более четкого оформления существующих функций, разработчик выбрал модель, которая бы позволяла легко добавлять новые графики. Но нашел ли он КОНЦЕПТУАЛЬНЫЙ КОНТУР, который позволит изменять и расширять архитектуру предметной области по мере развития программы и ее практических приложений? Невозможно заранее гарантировать, что архитектура адекватно отреагирует на непредвиденные изменения. Но программист считал, что он повысил шансы на успех.

Непредвиденное изменение

По ходу проекта возникло следующее требование к программе: ввести подробные правила обработки досрочных и просроченных платежей. Изучая проблему, программист с удовольствием обнаружил, что практически те же правила применимы к платежам по процентам и платежам по сборам. Это означало, что новые элементы модели естественным образом связывались с единственным классом **`Платеж (Payment)`**.

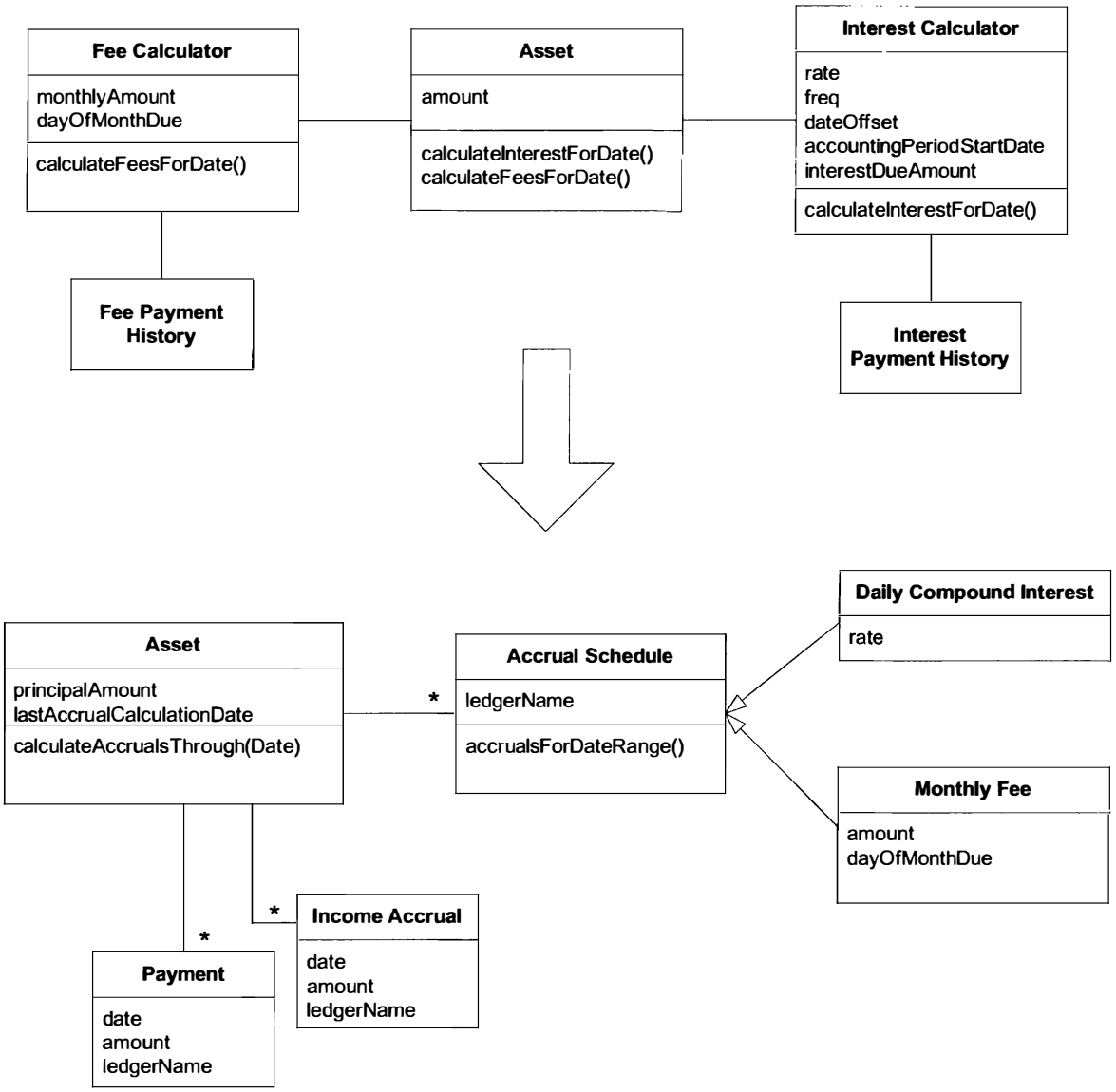


Рис. 10.11.

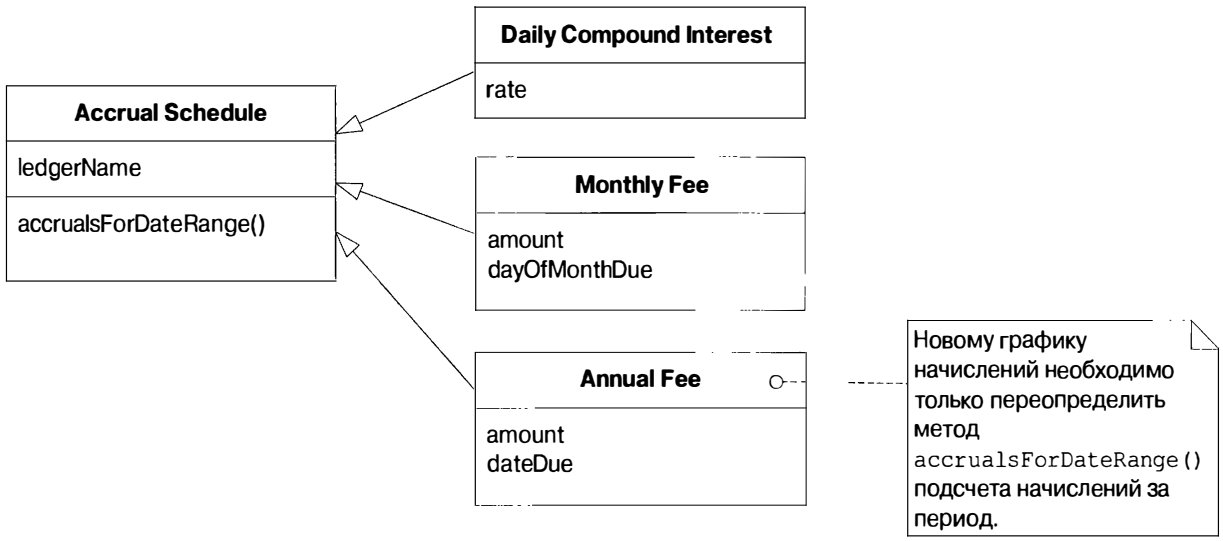


Рис. 10.12. Эта модель позволяет добавлять новые виды ***Графиков начислений*** *(****Accrual Schedule****)*

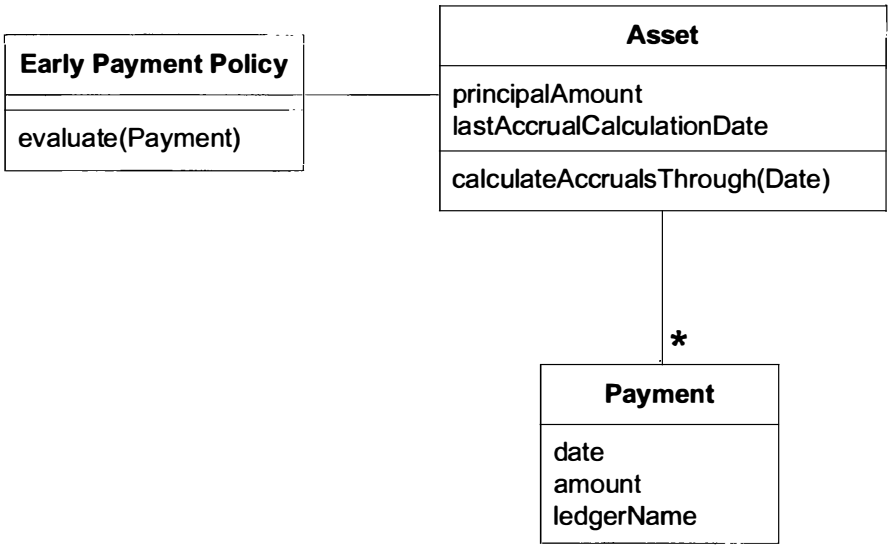


Рис. 10.13.

В старой архитектуре было бы неизбежно дублирование между двумя классами **Истории платежей (Payment History)**. (Кстати, эта трудность могла бы привести разработчика к выводу, что класс **Платеж (Payment)** должен использоваться совместно, и тогда он бы пришел другим путем к аналогичной модели.) Расширение прошло легко не потому, что программист предвидел это изменение. И не потому, что архитектура стала настолько универсальной, что могла бы воспринять любое мыслимое изменение. Все произошло потому, что предыдущий рефакторинг привел архитектуру в соответствие с фундаментальными понятиями предметной области.

* * *

ИНФОРМАТИВНЫЕ ИНТЕРФЕЙСЫ (INTENTION-REVEALING INTERFACES) позволяют клиентам представлять объекты как единицы смысла, а не просто рабочие механизмы. ФУНКЦИИ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ (SIDE-EFFECT-FREE FUNCTIONS) и УТВЕРЖДЕНИЯ (ASSERTIONS) позволяют без опаски пользоваться такими единицами и составлять из них сложные комбинации. Возникновение КОНЦЕПТУАЛЬНЫХ КОНТУРОВ (CONCEPTUAL CONTOURS) стабилизирует деление модели на части, а также делает смысловые единицы более наглядными и удобными в комбинациях.

И все-таки мы еще не избавились от угрозы смысловой перегрузки. Бывает так, что наличие взаимосвязей и взаимозависимостей принуждает нас думать о слишком многих вещах одновременно.

Изолированные классы

Наличие взаимосвязей и зависимостей между элементами делает модели и архитектуры трудными для понимания. Трудности возникают также в их тестировании и доработке. К тому же взаимосвязи имеют тенденцию накапливаться.

Каждая ассоциация, конечно, представляет собой зависимость. Для понимания того или иного класса необходимо понимать, с чем он связан. А то, с чем он связан, связано с чем-то еще, и оно тоже требует понимания. Тип каждого аргумента любого метода – это тоже зависимость и взаимосвязь. Возвращаемые значения — тоже не исключение.

Имея одну взаимосвязь, приходится думать одновременно о двух классах, а также о природе зависимости между ними. Имея две взаимосвязи, надо думать о каждом из

трех классов, о природе каждой зависимости одного класса от других и вообще обо всех возможных взаимоотношениях между ними. Если у них есть свои взаимосвязи с чем-то посторонним, это тоже приходится принимать во внимание. Имея три взаимосвязи... в общем, все заканчивается лавиной.

Ограничение паутины взаимосвязей входит в задачи как МОДУЛЕЙ, так и АГРЕГАТОВ. Как только высокосвязная предметная подобласть выделяется в отдельный модуль, от системы отрезается целый набор объектов, внутри которого количество взаимосвязанных понятий сравнительно невелико. Но даже если не впадать в фанатизм и не пытаться контролировать абсолютно все взаимосвязи в МОДУЛЕ, головной боли, тем не менее, будет хватать.

Даже в пределах одного МОДУЛЯ **сложность интерпретации архитектуры быстро возрастает при добавлении новых взаимосвязей. Это приводит к смысловой перегрузке и снижает потенциальную сложность архитектуры, которая была бы понятна разработчику. Наличие неявных понятий дает даже больший вклад в эту перегрузку, чем наличие явных ссылок между элементами.**

Для усовершенствования моделей их дистиллируют до тех пор, пока каждая оставшаяся связь между понятиями не начнет представлять нечто фундаментальное для смысла этих понятий. Для какого-нибудь важного подмножества предметной области количество взаимосвязей можно уменьшить вообще до нуля. В результате получившийся класс можно полностью понять "из самого себя", располагая в дополнение только несколькими общеизвестными примитивами и понятиями из стандартного словаря.

В любой среде программирования есть несколько вещей настолько базовых, что они никогда не выходят из головы у программиста. Например, в программировании на Java существует несколько таких примитивов и элементов стандартных библиотек, как числа, строки, коллекции. С практической точки зрения оттого, что программист помнит понятие "целого числа", интелектуальная нагрузка на его мозг не слишком возрастает. Но если не считать таких случаев, каждое дополнительное понятие, которое нужно удерживать в уме, чтобы понимать объект, дает вклад в интеллектуальную перегрузку.

Неявные понятия, как осознаваемые, так и неосознаваемые, имеют не меньшее влияние, чем явные ссылки. Хотя, как правило, мы можем игнорировать взаимосвязи с такими примитивными значениями, как числа или строки, нельзя игнорировать *то, что они представляют*. В первом примере программы смешивания красок объект **`Краска (Paint)`** содержал три *public*-значения, представлявших целочисленные коды красного, желтого и синего компонентов. Создание объекта **`Пигментный цвет (Pigment Color)`** не увеличивает количество понятий, участвующих во взаимосвязях, но делает уже существующие более явными и понятными. С другой стороны, и операция `size()` объекта **`Коллекция (Collection)`** возвращает целое число (`int`), представляющее собой просто количество — т.е. то, для чего целые числа и существуют. Здесь тоже не скрыто никаких новых понятий.

Всякая взаимосвязь и зависимость должна находиться под подозрением, пока не доказано, что она принципиально важна для понятия, скрытого за объектом. Такое "расследование" должно начинаться с выделения (*факторизации*) собственно понятий модели. Потребуется пристальное внимание к каждой конкретной ассоциации и операции. Проектные и модельные решения, вполне возможно, будут приводить к отсечению одной зависимости за другой — до полного их уничтожения.

Низкая внешняя зависимость — это качество, фундаментальное для проектирования архитектуры объектов. Если это возможно, устраняйте любую зависимость, как только возможно. Убирайте *все* другие понятия из картины. Тогда класс станет полно-

стью самодостаточным, его можно будет изучать и понимать отдельно от других. Любой такой самодостаточный класс существенно облегчает бремя понимания МОДУЛЯ.

Зависимость от других классов внутри одного и того же модуля значительно менее вредоносна, чем внешние взаимосвязи. Даже более того — если два объекта имеют естественную тесную связь друг с другом, большое количество операций с одной и той же парой объектов может фактически прояснить природу их взаимосвязи. Цель, собственно, состоит не в том, чтобы устранить вообще все взаимосвязи, а в том, чтобы устранить все несущественные. Если можно убрать не все взаимосвязи, а только некоторые, то устранение каждой из таковых позволяет программисту сосредоточиться на оставшихся концептуальных зависимостях.

Старайтесь выделить наиболее сложные и запутанные вычисления в ИЗОЛИРОВАННЫЕ КЛАССЫ(STANDALONE CLASSES) — возможно, путем моделирования ОБЪЕКТОВ-ЗНАЧЕНИЙ, содержащихся внутри более зависимых классов.

Понятие краски неразрывно связано с понятием цвета. Но цвет, даже цвет пигмента, можно рассматривать и без краски. Делая эти два понятия явными и дистиллируя отношение между ними, мы создаем такую одностороннюю ассоциацию, которая передает важное утверждение, причем класс **`Пигментный цвет`**(**`Pigment Color`**), в котором заключается большая часть сложных вычислений, можно изучать и *тестировать* автономно.

* * *

Низкая внешняя зависимость — это основной способ уменьшения смысловой перегрузки. ИЗОЛИРОВАННЫЙ КЛАСС — это граничный случай низкой внешней зависимости.

Устранение взаимосвязей не должно означать профанацию модели путем принудительного сведения всех ее элементов к самым примитивным. Последний архитектурный шаблон в этой главе, ЗАМКНУТОСТЬ ОПЕРАЦИЙ (CLOSURE OF OPERATIONS), дает пример того, как устранить лишние взаимосвязи при сохранении богатого и интеллектуального интерфейса...

Замкнутость операций

> *Если мы возьмем два действительных числа и перемножим их, то получим другое действительное число. [Действительные числа — это все рациональные числа и все иррациональные числа.] Это верно в любом случае, поэтому мы говорим, что действительные числа "замкнуты относительно операции умножения" — эта операция не может вывести нас за пределы множества. Комбинируя два элемента из множества, получаем результат, также принадлежащий этому множеству.*
>
> *The Math Forum, Drexel University*

Взаимосвязи, т.е. проявления зависимости, будут существовать всегда. И в этом нет ничего плохого, когда зависимость играет фундаментальную роль для некоторого понятия. Если упростить интерфейсы до того, что в них останутся ссылки только на примитивы, от этого пострадают функциональные возможности. Но тем не менее в интерфейсах встречается много ненужных зависимостей и даже целых лишних понятий.

Самые интересные объекты делают такое, что нельзя охарактеризовать одними только примитивами.

Один из распространенных приемов совершенствования архитектуры основан на достижении ЗАМКНУТОСТИ ОПЕРАЦИЙ (CLOSURE OF OPERATIONS). Это название пришло из самой тонкой и совершенной системы понятий из математики. Возьмем соотноше-

ние 1 + 1 = 2. Операция сложения замкнута по отношению к множеству действительных чисел. Математики упорно избегают введения ненужных, лишних понятий, а свойство замкнутости позволяет им определить операцию, не привлекая никаких других понятий. Мы так привыкли к чистоте и строгости математики, что не всегда осознаем, насколько мощными являются ее небольшие уловки. Но эта широко используется и в проектировании программ. Так, основное применение языка XSLT — преобразование одного документа XML в другой. Эта операция XSLT замкнута относительно множества документов XML. Свойство замкнутости сильно упрощает интерпретирование операции, а также облегчает комбинирование или последовательную реализацию нескольких операций.

Там, где это уместно, определяйте операцию(-и) с тем же типом возвращаемого значения, что и тип аргумента(-ов). Если реализующий ее объект имеет состояние, используемое в вычислениях, то этот объект фактически является аргументом операции, поэтому аргумент(ы) и возвращаемое значение должны быть того же типа, что и реализующий объект. Такая операция замкнута относительно множества экземпляров этого типа. Замкнутая операция обеспечивает интерфейс высокого уровня без какой-либо зависимости от других понятий.

Данный архитектурный шаблон чаще всего применяется в операциях ОБЪЕКТОВ-ЗНАЧЕНИЙ. Поскольку цикл существования объекта-СУЩНОСТИ играет важную роль в предметной области, нельзя просто слепить новую СУЩНОСТЬ на скорую руку, чтобы ответить на задаваемый вопрос. Существуют и операции, замкнутые относительно типа СУЩНОСТИ. Так, можно спросить у объекта **Работник (Employee)**, кто является его непосредственным начальником, и получить в ответ тоже объект **Работник**. Но в целом СУЩНОСТИ обычно не принадлежат к тем объектам, которые возникают как результат некоей вычислительной операции. Поэтому, как правило, нужные возможности следует изыскивать в ОБЪЕКТАХ-ЗНАЧЕНИЯХ.

Операция может быть замкнута относительно абстрактного типа, и в этом случае аргументы могут относиться к разным конкретным классам. В конце концов, сложение ведь замкнуто относительно действительных чисел, а они могут быть рациональными или иррациональными.

В ходе экспериментирования с методами для уменьшения взаимозависимости и повышения внутренней связности этот шаблон до определенной степени возникает сам собой. Аргумент соответствует реализатору, но тип возвращаемого значения отличается или же тип возвращаемого значения соответствует получателю, но тип аргумента — другой. Такие операции не замкнуты, но и они обладают некоторыми преимуществами ЗАМКНУТОСТИ. Если дополнительный тип — примитив или базовый библиотечный класс, он освобождает от хлопот почти так же хорошо, как ЗАМКНУТОСТЬ.

В предыдущем примере операция `mixedWith()` класса **Пигментный цвет (Pigment Color)** была замкнута относительно этого класса; по тексту книги разбросано еще несколько аналогичных примеров. Ниже приведен пример, показывающий всю полезность этой идеи даже в том случае, когда не достигается полноценная ЗАМКНУТОСТЬ.

Пример

Выбор из коллекций

Если нужно выбрать подмножество элементов из **Коллекции** в языке Java, следует запросить у нее **Итератор (Iterator)**. Далее с помощью него происходит итерационный перебор элементов с проверкой каждого из них, причем подходящие элементы могут накапливаться в новой **Коллекции**.

```
Set employees = (some Set of Employee objects);
Set lowPaidEmployees = new HashSet();
Iterator it = employees.iterator();
while (it.hasNext()) {
   Employee anEmployee = it.next();
   if (anEmployee.salary() < 40000)
      lowPaidEmployees.add(anEmployee);
}
```

С концептуальной точки зрения здесь из множества просто выбирается подмножество. Зачем для этого нужно лишнее понятие **Итератора** и вся эта техническая сложность? В языке Smalltalk я бы вызвал операцию выбора (`select`) из **Коллекции**, передав критерий-проверку в качестве аргумента. В результате получилась бы новая **Коллекция**, содержащая именно те элементы, которые прошли проверку.

```
employees := (some Set of Employee objects).
lowPaidEmployees := employees select:
         [:anEmployee | anEmployee salary < 40000].
```

Коллекции языка Smalltalk предлагают и другие подобные ФУНКЦИИ, возвращающие производные **Коллекции**, которые могут принадлежать к нескольким конкретным классам. Эти операции не замкнуты, поскольку в качестве аргумента они принимают "блок". Но блок — это базовый библиотечный тип в Smalltalk, так что память программиста это не перегружает. Возвращаемое значение соответствует типу реализующего объекта, поэтому их можно выстроить в цепочку, как последовательность фильтров, причем все это легко как писать, так и читать. При этом не вводятся лишние понятия, которые несущественны для задачи отбора подмножеств.

* * *

Шаблоны, представленные в этой главе, иллюстрируют общий стиль программирования и способ мышления при разработке программной архитектуры. Если стараться сделать программу понятной, наглядной, информативной, предсказуемой в поведении, то абстракция и инкапсуляция станут по-настоящему эффективны. Модели следует факторизовать так, чтобы выделенные в них объекты были просты в использовании и понимании, но тем не менее имели гибкие и развитые интерфейсы.

Чтобы успешно применять такие приемы при проектировании архитектуры, требуется довольно высокая квалификация. Без хороших навыков архитектурного проектирования программ иногда не обойтись даже при написании клиентского кода. Эффективность подхода ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN) сильно зависит от качества детализированной архитектуры и технических решений при ее реализации. Достаточно всего лишь нескольких дезориентированных программистов, чтобы проект покатился куда-то в сторону, а не к намеченной цели.

Итак, для программистов, которые готовы развивать свои навыки моделирования и проектирования, изложенные выше шаблоны и отражаемый ими способ мышления открывают путь к успешной разработке, доработке и переработке программного обеспечения, позволяющий создавать программные продукты любой сложности.

Декларативная архитектура

С помощью контрольных УТВЕРЖДЕНИЙ (ASSERTIONS) архитектуру программы можно значительно улучшить, даже учитывая сравнительно неформальный характер этого способа тестирования. Но в программах, написанных вручную, реальных гарантий дать

нельзя. Проверка УТВЕРЖДЕНИЙ может ничего не дать хотя бы по той причине, что в коде могут присутствовать побочные эффекты. Пусть даже наша архитектура и спроектирована строго ПО МОДЕЛИ, в дополнение к ней мы все равно пишем процедуры, имитирующие эффект концептуальных взаимосвязей. Много времени затрачивается на то, чтобы "склепать" код, который фактически не добавляет в программу ни смысла, ни операций. Это и скучно, и чревато ошибками, а обилие такого кода "затуманивает" смысл самой модели. (Некоторые языки в этом отношении лучше других, но все равно любой из них заставляет нас делать много нудной работы.) ИНФОРМАТИВНЫЕ ИНТЕРФЕЙСЫ (INTENTION-REVEALING INTERFACES) и другие архитектурные шаблоны из этой главы помогают частично решить проблему, но они никогда не придают обычным объектно-ориентированным программам формальную строгость.

Имеется целый ряд соображений и мотивов, приводящих нас к *декларативной архитектуре* (*declarative design*). Этот термин означает много разного для разных людей, но обычно под ним подразумевают способ написания программы или какой-то ее части в виде чего-то вроде выполняемой спецификации. Программа фактически управляется очень точным описанием ее свойств. Это можно проделать в самых разных формах через механизм отражения или в ходе компиляции с помощью автоматического генерирования кода (при этом на основе декларации создается обычный код). Такая методика позволяет другому программисту воспринимать декларацию именно так, как она написана. Это полная и абсолютная гарантия.

Генерирование работающей программы из декларации свойств модели — это нечто вроде Святого Грааля для идеологии ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ[2]. На практике, впрочем, в этом подходе имеются свои ловушки. Вот, например, две конкретные проблемы, с которыми я не раз сталкивался на практике.

- Декларативный язык оказывается недостаточно выразительным, чтобы запрограммировать все нужные операции; он является всего лишь архитектурной средой, в которой очень трудно расширить программу за пределы ее автоматизированной части.
- Средства генерирования кода препятствуют итерационному циклу разработки: они вставляют автоматически созданный код в написанный вручную таким образом, что повторное генерирование действует на код разрушительно.

Многие попытки декларативного построения архитектуры поневоле заканчивались примитивизацией модели и программы. Программисты, попавшись в ловушку ограничений созданной ими архитектурной среды, урезали все, что могли, чтобы в результате получить хоть *что-нибудь* работающее.

Еще один перспективный подход в декларативной архитектуре — это логическое программирование (*rule-based programming*) с механизмом выводов и базой правил. К сожалению, и эта идея может провалиться из-за некоторых тонкостей.

Хотя программа, основанная на совокупности логических правил, в принципе декларативна, в большинстве таких систем имеются "управляющие предикаты", добавленные туда для возможности настройки производительности. Этот управляющий код вносит побочные эффекты, и в результате поведение программы уже не полностью определяется декларированными правилами. Добавление, удаление и переупорядочение правил может привести к непредсказуемым и неправильным результатам. Поэтому программист

---

[2] Это надо понимать в том смысле, что такой способ давно ищут, о нем мечтают, но пока так и не нашли. — *Примеч. перев.*

в логической парадигме, как и программист объектно-ориентированный, должен тщательно следить за очевидностью и предсказуемостью поведения кода.

Смысл многих декларативных методик искажается, если разработчики намеренно или ненамеренно обходят их принципы. Это может случиться, если система трудна в использовании или накладывает слишком много ограничений. Чтобы пользоваться преимуществами декларативности, каждый должен следовать правилам, задаваемым архитектурной средой.

Наиболее ценное достижение, которое я видел на практике — это автоматизация особо рутинных и подверженных ошибкам аспектов архитектуры с помощью узкоспециальной среды разработки. Это поддержка непрерывности существования объектов, а также отображение между объектами и реляционными базами данных. Лучшие из этих достижений позволили разгрузить программистов от нудной рутины, оставив им полную свободу в проектировании архитектуры программы.

Специализированные предметные языки

Интересный подход, который иногда бывает декларативным — это разработка специализированного предметного языка (*domain-specific language*) для задачи. В этом стиле программирования клиентский код пишется на языке программирования, специально приспособленном к конкретной модели конкретной предметной области. Например, язык системы организации грузоперевозок может содержать такие термины, как *груз* и *маршрут*, вместе с необходимым синтаксисом. Написанная на таком языке программа может транслироваться на обычный объектно-ориентированный язык, в котором реализацию терминов из специализированного языка обеспечивают библиотеки классов.

Программы на таком языке могут быть чрезвычайно выразительными, находясь в прямой связи с ЕДИНЫМ ЯЗЫКОМ. Это очень интересная концепция, но у предметно-специализированных языков есть и недостатки в части знакомых мне подходов, основанных на объектно-ориентированной технологии.

Для уточнения модели разработчик должен иметь возможность модифицировать свой язык. При этом могут изменяться как декларации грамматических правил, так и другие средства языка. Может потребоваться также модификация библиотек классов, лежащих в его основе. Я обеими руками за изучение передовых технологий и архитектурных концепций, но приходится трезво оценивать квалификацию конкретных групп разработчиков, а также квалификацию будущих доработчиков. Кроме того, само по себе ценно качество идеального соответствия между приложением и моделью, написанными на одном и том же языке. Еще один недостаток состоит в трудности рефакторинга клиентского кода под уточненную модель и ассоциированный с нею специализированный предметный язык. Конечно, для проблемы рефакторинга может найтись и чисто техническое решение.

Этот прием может быть полезен в работе с очень зрелыми, проработанными моделями, где клиентский код, возможно, пишется другой группой разработчиков. Правда, при такой организации работы может возникнуть опаснейший разрыв между искусными технарями-разработчиками среды и технически невежественными авторами собственно приложения, но это вовсе не обязательно случится.

В языке Scheme нечто подобное входит в стандартный стиль программирования, и выразительность специализированного предметного языка может достигаться без разрыва в организации системы.

С чистого листа

Возможно, для специализированных предметных языков может лучше подойти другая парадигма, нежели объектная. В языке Scheme, представляющем "функциональное программирование", нечто подобное входит в стандартный стиль программирования, и выразительность специализированного предметного языка может достигаться без разрыва в организации системы.

Декларативный стиль архитектуры

Как только в архитектуре вашей программы появляются ИНФОРМАТИВНЫЕ ИНТЕРФЕЙСЫ (INTENTION-REVEALING INTERFACES), ФУНКЦИИ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ (SIDE-EFFECT-FREE FUNCTIONS) и УТВЕРЖДЕНИЯ (ASSERTIONS), вы уже одной ногой вступаете на декларативную территорию. Многие преимущества декларативной архитектуры начинают проявляться тогда, когда у вас появляются хорошо сочетаемые элементы, четко передающие свой смысл, побочные эффекты которых известны, очевидны либо отсутствуют.

Гибко спроектированное приложение часто позволяет использовать в клиентском коде декларативный *стиль* архитектуры. Чтобы проиллюстрировать эту мысль, в следующем разделе сводятся вместе некоторые из шаблонов этой главы, и это позволяет сделать СПЕЦИФИКАЦИЮ более гибкой и декларативной.

Расширение спецификаций в декларативном стиле

В главе 9 рассматривалось фундаментальное понятие СПЕЦИФИКАЦИИ (SPECIFICATION), роль, которую она может играть в программе, и некоторые особенности ее реализации. Теперь давайте рассмотрим несколько уловок, которые могут оказаться полезными в некоторых ситуациях, где присутствуют сложные регламентные правила.

СПЕЦИФИКАЦИЯ — это разновидность строгого формального утверждения, предиката. У предикатов есть различные полезные свойства, которыми мы избирательно воспользуемся.

Комбинирование спецификаций в логических операциях

Используя СПЕЦИФИКАЦИИ, часто встречаешься с необходимостью комбинировать их в те или иные сочетания. Как только что было сказано, спецификация — это разновидность предиката, а предикаты можно сочетать и модифицировать с помощью операций "И", "ИЛИ" и "НЕ". Эти логические операции замкнуты относительно множества предикатов, так что комбинации СПЕЦИФИКАЦИЙ демонстрируют ЗАМКНУТОСТЬ ОПЕРАЦИЙ.

СПЕЦИФИКАЦИИ содержат в себе важные обобщенные возможности, и для их полного раскрытия полезно создать абстрактный класс или интерфейс, который может использоваться для работы со СПЕЦИФИКАЦИЯМИ любого рода. Аргументы методов в нем должны представлять собой абстрактные классы высокого уровня.

```
public interface Specification {
   boolean isSatisfiedBy(Object candidate);
}
```

Такая абстракция требует применения предохранительного оператора в начале метода, но в остальном никак не влияет на функциональность. Например, вот как можно было бы модифицировать класс **Спецификация контейнера (Container Specification)** из главы 9.

```
public class ContainerSpecification implements Specification {
   private ContainerFeature requiredFeature;

   public ContainerSpecification(ContainerFeature required) {
      requiredFeature = required;
   }

   boolean isSatisfiedBy(Object candidate){
      if (!candidate instanceof Container) return false;
      return
(Container)candidate.getFeatures().contains(requiredFeature);
   }
}
```

Теперь расширим интерфейс **Спецификации (Specification)**, добавив три новые операции.

```
public interface Specification {
   boolean isSatisfiedBy(Object candidate);

   Specification and(Specification other);
   Specification or(Specification other);
   Specification not();
}
```

Напомним, что одни **Спецификации контейнеров** требовали вентилируемых (*ventilated*) **Контейнеров**, а другие — усиленных (*armored*). Химикат, являющийся одновременно и летучим, и взрывоопасным, вероятно, должен удовлетворять *обеим* этим **Спецификациям**. Это легко записать, используя новые методы.

```
Specification ventilated = new ContainerSpecification(VENTILATED);
Specification armored = new ContainerSpecification(ARMORED);

Specification both = ventilated.and(armored);
```

Эта декларация объявляет новый объект **Спецификация** с заданными свойствами. Если делать все по-старому, такое сочетание потребовало бы более сложной **Спецификации контейнера** и при этом имело бы узкоспециальное назначение.

Предположим, у нас имеется несколько разных видов вентилируемых **Контейнеров**. Для некоторых химикатов не имеет значения, в каком именно они хранятся; такие вещества можно поместить в любой из контейнеров.

```
Specification ventilatedType1 =
      new ContainerSpecification(VENTILATED_TYPE_1);
Specification ventilatedType2 =
      new ContainerSpecification(VENTILATED_TYPE_2);

Specification either = ventilatedType1.or(ventilatedType2);
```

Если считать излишеством помещение песка в специализированный контейнер, его можно запретить, введя СПЕЦИФИКАЦИЮ “дешевого” (“легкого”) контейнера без специальных возможностей.

```
Specification cheap = (ventilated.not()).and(armored.not());
```

Такое ограничение позволило бы избежать некоторых неоптимальных решений в прототипе программы-складировщика, который рассматривался в главе 9.

Возможность строить сложные спецификации из простых элементов улучшает выразительность кода. А комбинации элементов записываются во вполне декларативном стиле.

Сами операции комбинирования могут быть простыми или сложными в реализации в зависимости от того, как именно реализованы СПЕЦИФИКАЦИИ. Далее показана очень простая реализация, которая в одних ситуациях была бы неэффективна, а в других — вполне работоспособна. Здесь она приведена *только как наглядный пример*. На самом деле способов реализации великое множество, как и в случае любого другого архитектурного шаблона.

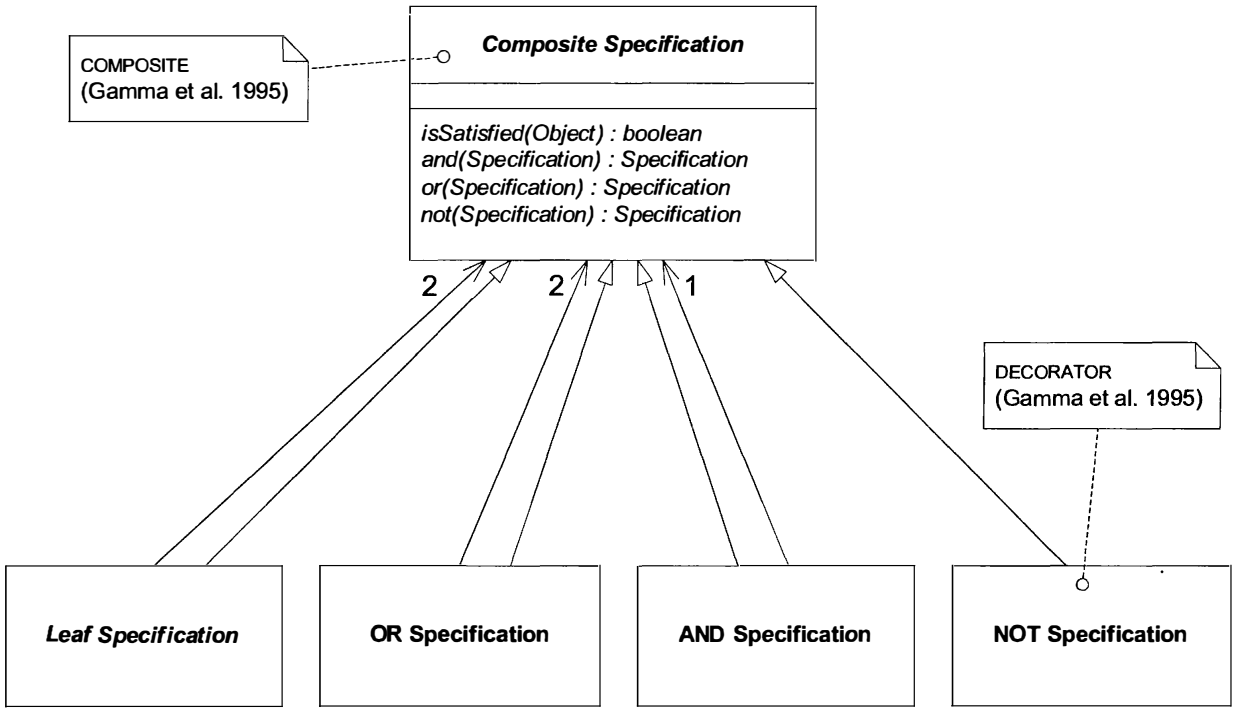


Рис. 10.14. Составная структура СПЕЦИФИКАЦИИ

```
public abstract class AbstractSpecification implements
      Specification {
   public Specification and(Specification other) {
      return new AndSpecification(this, other);
   }
   public Specification or(Specification other) {
      return new OrSpecification(this, other);
   }
   public Specification not() {
      return new NotSpecification(this);
   }
}

public class AndSpecification extends AbstractSpecification {
   Specification one;
   Specification other;
   public AndSpecification(Specification x, Specification y) {
      one = x;
      other = y;
   }
   public boolean isSatisfiedBy(Object candidate) {
      return one.isSatisfiedBy(candidate) &&
         other.isSatisfiedBy(candidate);
```

```
    }
}

public class OrSpecification extends AbstractSpecification {
    Specification one;
    Specification other;
    public OrSpecification(Specification x, Specification y) {
        one = x;
        other = y;
    }
    public boolean isSatisfiedBy(Object candidate) {
        return one.isSatisfiedBy(candidate) ||
        other.isSatisfiedBy(candidate);
    }
}

public class NotSpecification extends AbstractSpecification {
    Specification wrapped;

    public NotSpecification(Specification x) {
        wrapped = x;
    }
    public boolean isSatisfiedBy(Object candidate) {
        return !wrapped.isSatisfiedBy(candidate);
    }
}
```

Этот код был специально сделан как можно проще для удобства чтения в книге. Как уже говорилось выше, во многих случаях он неэффективен. Но возможны и другие варианты реализации, в которых минимизировалось бы количество участвующих объектов, или оптимизировалось быстродействие, или соблюдалась бы совместимость с какой-нибудь экзотической технологией, используемой в проекте. Важно лишь помнить о модели, содержащей ключевые понятия предметной области, и о том, что реализация должна придерживаться этой модели. При этом остается достаточно места для маневра, чтобы решить проблемы быстродействия.

Кроме того, слишком широкая общность во многих случаях и не нужна. В частности, операция И имеет гораздо более широкую применимость, чем две другие, и создает меньше проблем в реализации. Смело реализуйте одну только операцию И, если большего вам не требуется.

В самом начале книги, в главе 2, программисты явно не реализовали операцию "удовлетворяется" (*satisfied by*) своей СПЕЦИФИКАЦИИ. До того момента СПЕЦИФИКАЦИЯ использовалась только для создания объектов по заказу. Но даже при этом абстрагирование было соблюдено, и добавление новых функций оставалось сравнительно несложным. Применение того или иного архитектурного шаблона не означает, что вы обязаны реализовать все его особенности и свойства, даже ненужные. Их можно добавить потом, по мере доработки — лишь бы важные понятия не потерялись из виду.

Пример

Альтернативная реализация составной спецификации

Некоторые исполняемые среды не слишком хорошо справляются с большими количествами очень мелких объектов. Как-то я работал над проектом с объектной базой данных, где требовалось выдавать объектный идентификатор каждому объекту, а затем отслежи-

вать его. Каждый объект отнимал много дополнительных ресурсов памяти и процессора, и общее адресное пространство оказалось ограничивающим фактором. В нескольких ключевых точках архитектуры предметной области я применил СПЕЦИФИКАЦИИ, что показалось мне удачным решением. Но я воспользовался несколько более сложной версией той реализации, которая описана в этой главе, и это оказалось ошибкой. В системе возникли миллионы мелкомасштабных объектов, что вызвало серьезное замедление ее быстродействия.

Далее приведен пример альтернативной реализации, в которой составная СПЕЦИФИКАЦИЯ кодируется в виде строки или массива с логическим выражением, интерпретируемым в ходе выполнения программы.

(Не волнуйтесь, если не понимаете, как это реализовать. Самое важное сейчас — понять, что существует много способов реализовать СПЕЦИФИКАЦИЮ с помощью логических операций и что если простой способ не работает в конкретной ситуации, то всегда есть выбор.)

Содержимое стека спецификации для "дешевого контейнера"

Верх	**`AndSpecificationOperator`**`(Fly Weight)` [*Операция-И* (*Легкий*)]
	`NotSpecificationOperator``(Fly Weight)` [*Операция-Не* (*Легкий*)]
	`Armored` (*Усиленный*)
	`NotSpecificationOperator` [*Операция-Не*]
	`Ventilated` [*Вентилируемый*]

Когда нужно проверить контейнер-кандидат, приходится интерпретировать эту структуру. Это можно проделать, извлекая каждый элемент из стека и либо анализируя его значение, либо извлекая следующее в зависимости от того, чего требует операция. В конце концов получится следующее.

```
and(not(armored), not(ventilated))
```

Здесь есть свои "плюсы" (+) и "минусы" (–):

+ участвует мало объектов;

+ эффективно используется память;

– требуется высокая квалификация программистов.

Вам придется найти такую реализацию, компромиссы которой подходили бы к вашему случаю. Один и тот же шаблон, одна и та же модель могут лежать в основе самых разных программных реализаций.

Сужение

В этом последнем из описанных здесь средств редко возникает необходимость, да и реализовать его не так-то просто, но время от времени с его помощью удается решить серьезную проблему. Оно также проливает свет на смысл СПЕЦИФИКАЦИИ.

Снова рассмотрим программу-складировщик химикатов. Напомним, что каждый **`Химикат`** (**`Chemical`**) имеет свою **`Спецификацию контейнера`** (**`Container Specification`**),

а служба **Складировщик** (`Packer`) гарантирует удовлетворение всех спецификаций при помещении **Бочек** (`Drums`) в **Контейнеры** (`Containers`). И все идет хорошо... пока кто-нибудь не изменит нормативные требования.

Каждые несколько месяцев выходит в свет новый документ с нормами и правилами, и нашим пользователям хотелось бы иметь возможность получать от программы список видов химикатов, на которые стали накладываться более жесткие требования.

Конечно, можно было бы дать частичный ответ (причем такой, который тоже мог бы понадобиться пользователям), подвергнув проверке по новым СПЕЦИФИКАЦИЯМ каждую **Бочку** в инвентарной описи и выбрав те из них, которые больше не удовлетворяют требованиям. Так пользователи узнали бы, какие **Бочки** в существующем размещении им нужно переместить.

Но *просили*-то они совсем другое: список всех химикатов, требования к обращению с которыми ужесточились. Может быть, на складе сейчас таких химикатов нет вообще или же они случайно помещены в более строгий контейнер, чем нужно. В любом из этих случаев они не будут упомянуты в отчете.

Давайте введем новую операцию для прямого сравнения двух СПЕЦИФИКАЦИЙ.

```
boolean subsumes(Specification other);
```

Более строгая спецификация является и более узкой по смыслу, чем менее строгая, а потому может занять ее место, не нарушая ни одного из ранее соблюдаемых требований.

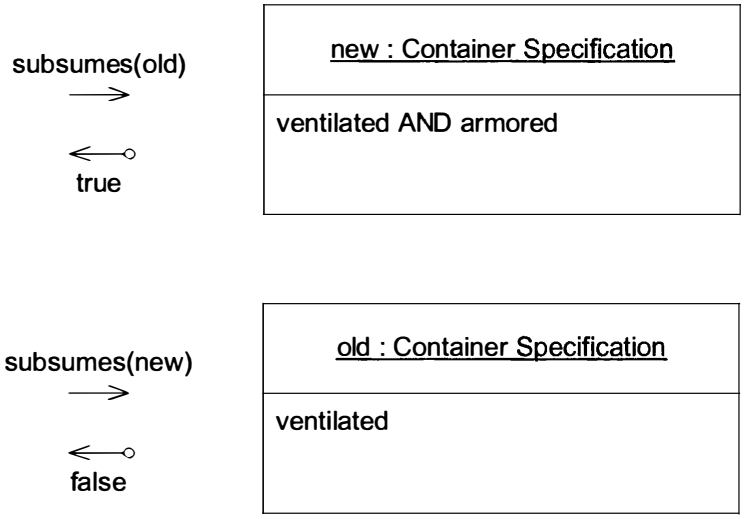


Рис. 10.15. СПЕЦИФИКАЦИЯ для контейнера, содержащего бензин, стала более строгой

На языке СПЕЦИФИКАЦИЙ мы бы сказали, что новая спецификация есть *сужение* (*subsumption*) старой, потому что любой контейнер, удовлетворяющий новой спецификации, также удовлетворяет и старой.

Если каждую из СПЕЦИФИКАЦИЙ рассматривать как предикат, сужение будет эквивалентом логического следования (импликации). В стандартной записи $A \rightarrow B$ означает, что из утверждения A следует утверждение B, т.е. если A истинно, то и B также истинно.

Применим эту логику к нашему распределению химикатов по контейнерам. Когда СПЕЦИФИКАЦИЯ изменяется, мы хотим знать, удовлетворяет ли новый предложенный вариант всем условиям старого.

```
Новая спецификация → Старая спецификация
```

Другими словами, если новая спецификация истинна, то истинна и старая. Доказательство логического следования в общем виде очень трудно, а вот частные случаи могут оказаться попроще. Например, некоторые параметризованные СПЕЦИФИКАЦИИ могут задавать свои собственные правила сужения.

```
public class MinimumAgeSpecification {
   int threshold;

   public boolean isSatisfiedBy(Person candidate) {
      return candidate.getAge() >= threshold;
   }

   public boolean subsumes(MinimumAgeSpecification other) {
      return threshold >= other.getThreshold();
   }
}
```

Тест для среды JUnit может содержать такие строки.

```
drivingAge = new MinimumAgeSpecification(16);
votingAge = new MinimumAgeSpecification(18);
assertTrue(votingAge.subsumes(drivingAge));
```

Еще один практический частный случай, подходящий для решения проблемы **Спецификации контейнера** — это интерфейс СПЕЦИФИКАЦИИ, в котором комбинируются сужение и одна логическая операция И.

```
public interface Specification {
   boolean isSatisfiedBy(Object candidate);
   Specification and(Specification other);
   boolean subsumes(Specification other);
}
```

Доказать импликацию при наличии только операции И легко.

А И В → *А*

Можно взять и более сложный случай.

А И В И С → *А И В*

Итак, если **Составная спецификация (Composite Specification)** умеет собирать вместе все СПЕЦИФИКАЦИИ-листья (в терминологии древовидных структур), соединенные операцией И, то достаточно проверить, что сужающая спецификация содержит все листья, которые охватываются сужаемой, и, может быть, еще некоторые — ее листья являются надмножеством множества листьев сужаемой спецификации.

```
public boolean subsumes(Specification other) {
  if (other instanceof CompositeSpecification) {
     Collection otherLeaves =
        (CompositeSpecification) other.leafSpecifications();
     Iterator it = otherLeaves.iterator();
     while (it.hasNext()) {
        if (!leafSpecifications().contains(it.next()))
           return false;
     }
  } else {
```

```
      if (!leafSpecifications().contains(other))
         return false;
   }
   return true;
}
```

Эту процедуру можно усовершенствовать, чтобы можно было сравнивать тщательно отобранные параметризованные СПЕЦИФИКАЦИИ-листья и учитывать некоторые другие осложнения. К сожалению, если присутствуют операции ИЛИ и НЕ, эти доказательства становятся намного сложнее. Во многих ситуациях такой сложности лучше избегать, делая конкретный выбор: отказаться либо от некоторых операций, либо от сужения. Если же ни без того, ни без другого не обойтись, серьезно подумайте, настолько ли велик ожидаемый эффект, чтобы оправдать трудности реализации.

Аристотель о спецификациях

Все люди смертны	`Specification manSpec = new ManSpecification();`
	`Specification mortalSpec = new MortalSpecification();`
	`assert manSpec.subsumes(mortalSpec);`

Аристотель — человек	`Man aristotle = new Man();` `assert manSpec.isSatisfiedBy(aristotle);`
Следовательно, Аристотель смертен	`assert mortalSpec.isSatisfiedBy(aristotle);`

Углы атаки

В этой главе было представлено немало разных приемов, помогающих прояснить смысл кода, сделать очевидными последствия его выполнения, уменьшить зависимость между элементами модели. Но проектирование архитектуры программ в таком стиле все равно остается непростым делом. Нельзя взять огромную, сложную программную систему и сказать: “Давайте-ка сделаем ее гибкой”. Нужно правильно поставить цели. Ниже рассказано о нескольких общих подходах, а затем приведен большой пример, где показано, как сочетать архитектурные шаблоны и на их основе строить большую систему.

Выделение подобластей

Невозможно справиться со всей архитектурой одним махом. Двигайтесь шаг за шагом. Некоторые аспекты системы сами подскажут вам правильные подходы; их можно будет сразу выделить и реализовать в нужном виде. Может оказаться, что какая-то часть программы -- это специализированная математика; отделите ее. Если изменение состояний в программе регулируется сложными правилами, выделите эту часть в отдельную модель или простую архитектурную среду, где можно удобно задавать такие правила. После каждого такого шага не только обретает завершенность новый модуль, но и уменьшается (и становится понятнее) оставшаяся часть работы. А эту часть можно написать в декларативном стиле, в виде деклараций на языке специализированной математики или регламентирующей среды либо в любом другом виде, задаваемом сутью предметной подобласти.

Полезнее нанести решающий удар в одной области и тем самым сделать часть архитектуры действительно гибкой. О выборе подобластей и работе с ними подробнее говорится в главе 15.

Использование сложившихся формальных систем

Созданием строгой системы понятий "с нуля" приходится заниматься не так уж часто. Бывает, что в ходе проекта удается разыскать и усовершенствовать нечто в этом роде. Но ведь использовать и адаптировать к своим потребностям можно и такие системы понятий, которые давно существуют в данной предметной области или какой-нибудь другой. Некоторые из этих систем совершенствовались и дистиллировались в течение столетий. Например, во многих коммерческих системах применяется бухгалтерский учет. В бухгалтерском учете существует хорошо проработанная система объектов-СУЩНОСТЕЙ и правил, которые способствуют легкой адаптации к углубленной модели и гибкой архитектуре.

Существует много подобных формализованных систем понятий, но лично мне больше всего по вкусу математика. Удивительно, сколько пользы может принести введение в модель некоей системы арифметических действий и правил. Во многих предметных областях математика присутствует в том или ином виде. Поищите ее. Копайте как можно глубже. Специализированная математика строго формулируется, подчиняется четким правилам, доступна для понимания. В завершение этой главы приведу один из примеров, который мне встретился на практике — "кредитную математику".

Пример

Интеграция шаблонов: кредитная математика

В главе 8 рассказывалось, как удалось добиться качественного скачка в модели системы для обслуживания синдицированных кредитов. Сейчас мы рассмотрим некоторые подробности, сосредоточившись всего на одной особенности архитектуры, аналогичной той, которая применялась в упоминавшемся проекте.

Одно из технических требований к приложению состояло в том, что когда заемщик выплачивает основную сумму кредита, эти средства по умолчанию должны распределяться согласно долям кредиторов в кредите.

Первоначальная схема распределения средств

По мере рефакторинга код будет становиться понятнее, так что не стоит долго задерживаться на этой, первоначальной, версии.

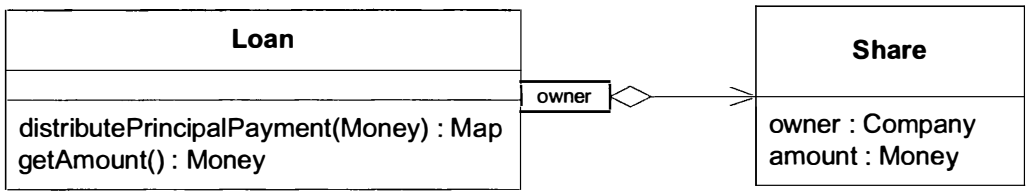


Рис. 10.16.

```
public class Loan {
   private Map shares;

   //Методы доступа, конструкторы и простейшие методы опущены

   public Map distributePrincipalPayment(double paymentAmount) {
      Map paymentShares = new HashMap();
      Map loanShares = getShares();
```

```
        double total = getAmount();
        Iterator it = loanShares.keySet().iterator();
        while(it.hasNext()) {
            Object owner = it.next();
            double initialLoanShareAmount = getShareAmount(owner);
            double paymentShareAmount =
                initialLoanShareAmount / total * paymentAmount;
            Share paymentShare =
                new Share(owner, paymentShareAmount);
            paymentShares.put(owner, paymentShare);

            double newLoanShareAmount =
                initialLoanShareAmount - paymentShareAmount;
            Share newLoanShare =
                new Share(owner, newLoanShareAmount);
            loanShares.put(owner, newLoanShare);
        }
        return paymentShares;
    }

    public double getAmount() {
        Map loanShares = getShares();
        double total = 0.0;
        Iterator it = loanShares.keySet().iterator();
        while(it.hasNext()) {
            Share loanShare = (Share) loanShares.get(it.next());
            total = total + loanShare.getAmount();
        }
        return total;
    }
}
```

Разделение команд и функций без побочных эффектов

В этой архитектуре уже присутствуют ИНФОРМАТИВНЫЕ ИНТЕРФЕЙСЫ (INTENTION-REVEALING INTERFACES). Но метод `distributePrincipalPayment()` делает опасную вещь: он и вычисляет доли в распределении, и модифицирует объект **Кредит (Loan)**. Давайте выполним рефакторинг и отделим запрос от модификации.

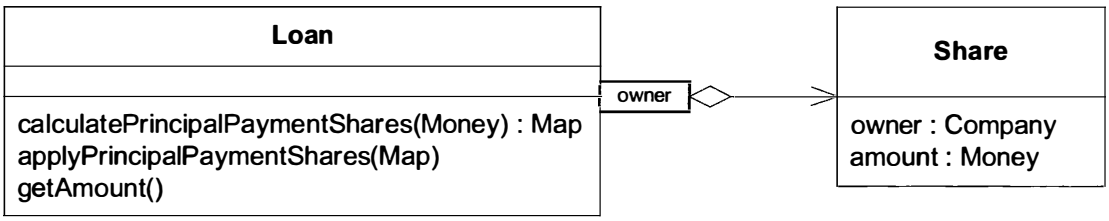


Рис. 10.17.

```
public void applyPrincipalPaymentShares(Map paymentShares) {
    Map loanShares = getShares();
    Iterator it = paymentShares.keySet().iterator();
    while(it.hasNext()) {
```

```
      Object lender = it.next();
      Share paymentShare = (Share) paymentShares.get(lender);
      Share loanShare = (Share) loanShares.get(lender);
      double newLoanShareAmount = loanShare.getAmount() -
         paymentShare.getAmount();
      Share newLoanShare = new Share(lender, newLoanShareAmount);
      loanShares.put(lender, newLoanShare);
   }
}

public Map calculatePrincipalPaymentShares(double paymentAmount) {
   Map paymentShares = new HashMap();
   Map loanShares = getShares();
   double total = getAmount();
   Iterator it = loanShares.keySet().iterator();
   while(it.hasNext()) {
      Object lender = it.next();
      Share loanShare = (Share) loanShares.get(lender);
      double paymentShareAmount =
         loanShare.getAmount() / total * paymentAmount;
      Share paymentShare = new Share(lender, paymentShareAmount);
      paymentShares.put(lender, paymentShare);
   }
   return paymentShares;
}
```

Теперь клиентский код будет выглядеть так.

```
Map distribution =
   aLoan.calculatePrincipalPaymentShares(paymentAmount);
aLoan.applyPrincipalPaymentShares(distribution);
```

Что ж, неплохо. Функции инкапсулируют значительную часть сложных вычислений, скрытых за информативными интерфейсами. Но все-таки код начинает увеличиваться, когда мы добавляем методы `applyDrawdown()` для выборки средств, `calculateFeePaymentShares()` для расчета долей в полученной плате за пользование кредитом и т.п. Каждое расширение усложняет и утяжеляет код. На этом этапе дробление кода, пожалуй, еще крупновато. Обычно вычислительные методы дополнительно разбиваются на процедуры. Это и само по себе может оказаться полезно. Но следует учесть, что нам в конечном счете нужно провести именно естественные, концептуальные границы, и тем самым углубить модель. Любые необходимые варианты всегда можно составить из элементов архитектуры, разбиение которых проведено по концептуальным границам.

Выведение неявных понятий в явные

Итак, у нас уже достаточно информации, чтобы начать "нащупывать" новую модель. Объекты **Доля (Share)** в этой реализации будут пассивными, и манипулировать ими мы будем сложными, низкоуровневыми способами. Причина состоит в том, что большая часть правил и вычислений, касающихся долей, относится не к отдельным долям, а к их группам. В модели пропущено важное понятие о том, как именно доли связаны друг с другом — это части одного целого. Если сделать это понятие явным, правила и вычисления можно будет формулировать более сжато и точно.

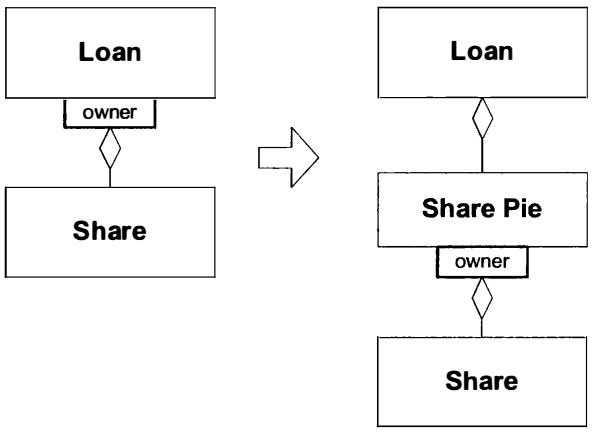


Рис. 10.18.

Здесь **Распределение долей (Share Pie)** полностью описывает распределение **Кредита (Loan)** между кредиторами. Все вычисления, касающиеся вопросов распределения, можно делегировать объекту **Распределение долей**.

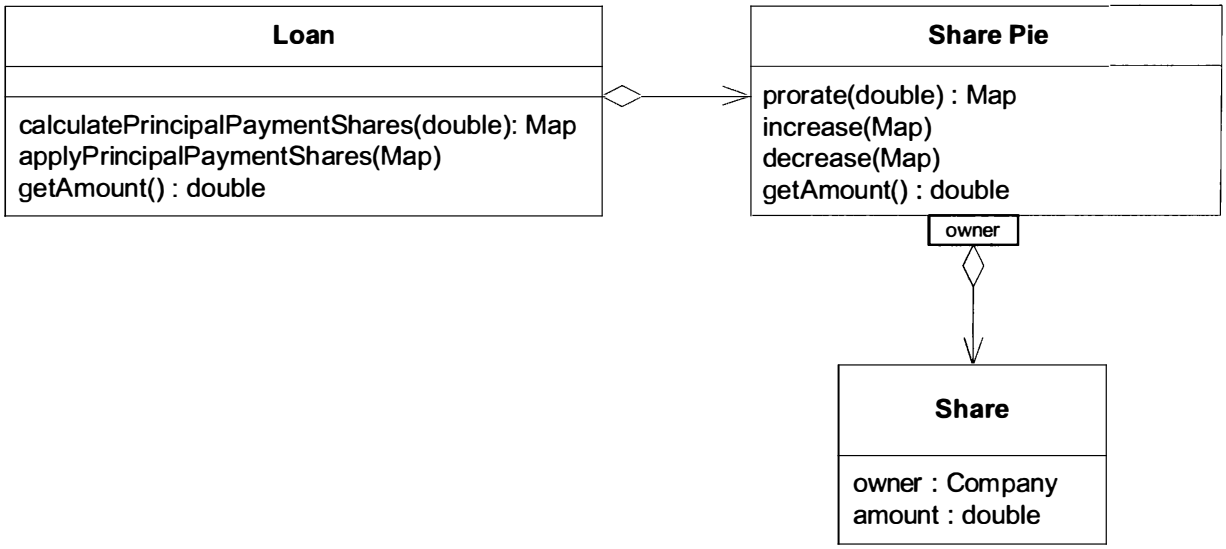


Рис. 10.19.

```
public class Loan {
   private SharePie shares;

   //Методы доступа, конструкторы и очевидные методы опущены

   public Map calculatePrincipalPaymentDistribution(
                                          double paymentAmount) {
      return getShares().prorated(paymentAmount);
   }

   public void applyPrincipalPayment(Map paymentShares) {
      shares.decrease(paymentShares);
   }
}
```

Здесь **Кредит (Loan)** упрощен, а вычисления по **Долям (Shares)** сосредоточены в ОБЪЕКТЕ-ЗНАЧЕНИИ, специально предназначенном для выполнения этих обязанностей. И все-таки вычисления пока еще не стали более универсальными или простыми в использовании.

Распределение долей как объект-значение: цепочка выводов

Часто практический опыт по реализации новой архитектуры дает толчок к новым выводам относительно самой модели. В нашем случае тесная зависимость между **Кредитом (Loan)** и **Распределением долей (Share Pie)** несколько затмевает взаимосвязь между **Распределением долей** и самими **Долями (Shares)**. Что будет, если сделать **Распределение долей** ОБЪЕКТОМ-ЗНАЧЕНИЕМ?

Методы `increase(Map)` и `decrease(Map)` станут невозможны, поскольку **Распределение долей** станет неизменяемым объектом. Чтобы изменить что-нибудь в **Распределении долей**, его придется заменить целиком. Поэтому можно использовать операцию наподобие `addSharesMap()`, которая бы возвращала целиком новое, большее **Распределение долей**.

Проделаем оставшиеся шаги К ЗАМКНУТОСТИ ОПЕРАЦИЙ (CLOSURE OF OPERATIONS). Вместо "увеличения" доли в **Распределении долей** или добавления к нему дополнительной **Доли** будем просто складывать два **Распределения долей** и получать новое — большее.

Частично можно замкнуть операцию пропорционального раздела `prorate()` над **Распределением долей**, просто изменив тип возвращаемого значения. Переименовав ее в `prorated()`, мы изменяем желаемое на свершившийся факт и подчеркиваем отсутствие побочных эффектов. "Кредитная математика" начинает приобретать форму, пусть пока она и состоит всего из четырех операций.

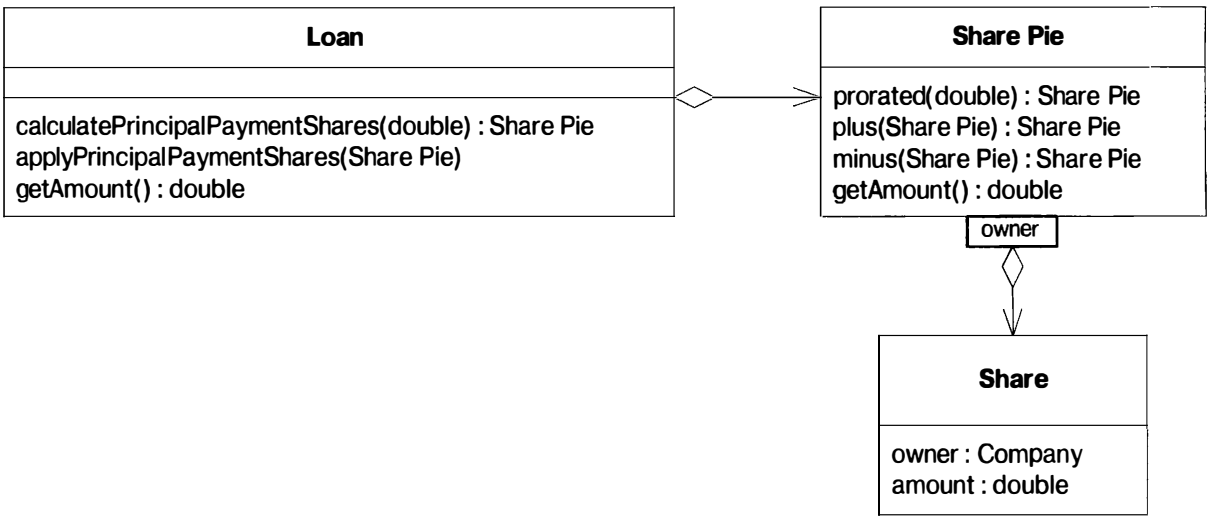


Рис. 10.20.

Можно составить несколько хорошо определенных УТВЕРЖДЕНИЙ относительно наших новых ОБЪЕКТОВ-ЗНАЧЕНИЙ — **Распределений долей**. Каждый метод при этом имеет четкий смысл.

```
public class SharePie {
   private Map shares = new HashMap();
```

```
   //Методы доступа и другие очевидные методы опущены

   public double getAmount() {
      double total = 0.0;
      Iterator it = shares.keySet().iterator();
//
//       Целое равно сумме частей
//
      while(it.hasNext()) {
         Share loanShare = getShare(it.next());
         total = total + loanShare.getAmount();
      }
      return total;
   }

   public SharePie minus(SharePie otherShares) {
      SharePie result = new SharePie();
      Set owners = new HashSet();
      owners.addAll(getOwners());
      owners.addAll(otherShares.getOwners());
//
//       Разница между двумя Распределениями (Pies) -
//       это разница между долями каждого из собственников
//
      Iterator it = owners.iterator();
      while(it.hasNext()) {
         Object owner = it.next();
         double resultShareAmount = getShareAmount(owner) -
               otherShares.getShareAmount(owner);
         result.add(owner, resultShareAmount);
      }
      return result;
   }

//
//       Сумма двух Распределений (Pies) -
//       это сумма долей каждого собственника
//
   public SharePie plus(SharePie otherShares) {
      //Аналогично реализации minus()
   }

   public SharePie prorated(double amountToProrate) {
      SharePie proration = new SharePie();
      double basis = getAmount();
//
//       Денежная сумма может распределяться
//       пропорционально между всеми участниками
//
      Iterator it = shares.keySet().iterator();
      while(it.hasNext()) {
         Object owner = it.next();
         Share share = getShare(owner);
         double proratedShareAmount =
```

```
                share.getAmount() / basis * amountToProrate;
            proration.add(owner, proratedShareAmount);
        }
        return proration;
    }
}
```

Гибкость новой архитектуры

На этом этапе методы самого важного класса **Кредит (Loan)** упростились до самого простого вида.

```
public class Loan {
   private SharePie shares;

   //Методы доступа, конструкторы и прочие очевидные
   //методы опущены

   public SharePie calculatePrincipalPaymentDistribution(
                                              double paymentAmount) {
      return shares.prorated(paymentAmount);
   }

   public void applyPrincipalPayment(SharePie paymentShares) {
      setShares(shares.minus(paymentShares));
   }
```

Каждый из этих коротких методов сам декларирует свой *фактический смысл*. Выплата по основной сумме кредита означает, что уплаченная сумма вычитается из кредита, одна доля за другой. Распределение выплаты основной суммы происходит путем ее пропорционального деления между участниками-кредиторами. Структура класса **Распределение долей (Share Pie)** позволяет нам использовать декларативный стиль в коде класса **Кредит (Loan)**. Получается такой код, который читается как концептуальное определение коммерческой транзакции, а не некий сложный расчет.

Другие виды транзакций (слишком сложные, чтобы говорить о них раньше) теперь можно легко записать в декларативном стиле. Например, выборка средств (*drawdown*) по кредиту распределяется между кредиторами пропорционально их долям в **Предприятии** (**Facility**). Новая выборка добавляется к текущей сумме задолженности по **Кредиту**. На нашем новом языке предметной области это выглядит так.

```
public class Facility {
   private SharePie shares;
   . . .
   public SharePie calculateDrawdownDefaultDistribution(
                                             double drawdownAmount) {
      return shares.prorated(drawdownAmount);
   }
}

public class Loan {
   . . .
   public void applyDrawdown(SharePie drawdownShares) {
      setShares(shares.plus(drawdownShares));
   }
}
```

Чтобы определить отклонение каждого из кредиторов от взятых им на себя обязательств по долевому вкладу в кредит, надо взять теоретическое распределение текущей задолженности по **Кредиту** и вычесть его из фактических долей кредиторов в нем.

```
SharePie originalAgreement =
   aFacility.getShares().prorated(aLoan.getAmount());
SharePie actual = aLoan.getShares();
SharePie deviation = actual.minus(originalAgreement);
```

Легкость в сочетании элементов и информативность этого кода достигается за счет определенных характеристик, присущих структуре объекта **Распределение долей (Share Pie)**.

- *Сложные расчетные алгоритмы инкапсулированы в специализированные* ОБЪЕКТЫ-ЗНАЧЕНИЯ, *содержащие* ФУНКЦИИ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ. Большая часть сложных вычислительных операций спрятана в этих неизменяемых объектах. Поскольку **Распределения долей** являются ОБЪЕКТАМИ-ЗНАЧЕНИЯМИ, математические операции могут создавать новые экземпляры таких объектов, которыми можно свободно заменять устаревшие.

  Ни один из методов **Распределения долей** ничего не меняет в существующем объекте. Это позволяет свободно пользоваться методами `plus()`, `minus()` и `prorate()` в промежуточных вычислениях, сочетая их как угодно и ожидая от них предсказуемого по их именам результата, но ничего более. На основе этих же методов теперь можно развивать аналитические возможности. (Ранее их можно было вызывать только после создания фактического распределения, поскольку при каждом вызове данные изменялись.)

- *Операции, изменяющие состояние, просты и характеризуются* УТВЕРЖДЕНИЯМИ (ASSERTIONS). Высокий уровень абстрагирования в "кредитной математике" позволяет записывать инварианты транзакций кратко и в декларативном стиле. Например, отклонение от обязательств равно фактическому распределению долей минус выплаченная сумма **Кредита (Loan)**, разделенная в пропорции на основе **Распределения долей (Share Pie)** в **Предприятии (Facility)**.

- *Понятия модели сделаны взаимно независимыми; в операции вовлекается минимум данных разных типов.* Некоторые методы в **Распределении долей (Share Pie)** демонстрируют ЗАМКНУТОСТЬ ОПЕРАЦИЙ (методы сложения или вычитания замкнуты относительно таких **Распределений**). В других методах в качестве аргументов или возвращаемых значений используются просто суммы денег. Эти операции не замкнуты, но дополнительной концептуальной нагрузки не несут. **Распределение долей (Share Pie)** тесно взаимодействует только с одним классом — **Долей (Share)**. В итоге класс **Распределение долей** стал самодостаточным, легко понятным, легко тестируемым и столь же легко комбинируемым в декларативные транзакции. Все эти "черты" унаследованы от математического формализма.

- *Знакомый характер формализации делает рабочий протокол легко понятным.* Теоретически можно было бы построить совершенно оригинальный протокол обращения с долями в кредите, взяв за основу финансовую терминологию. В принципе так можно было бы даже построить достаточно гибкую структуру. Но в такой архитектуре было бы два недостатка. Во-первых, ее пришлось бы творчески разрабатывать, изобретать, что само по себе задача трудная и неопределенная. Во-вторых, ее

пришлось бы изучать с нуля всякому, кто захотел бы иметь с ней дело. Люди же, сталкивающиеся с "кредитной математикой", узнают в ней уже знакомую им систему, которая не вводит их в заблуждение благодаря тому, что архитектура тщательно выстроена в соответствии с правилами арифметики.

Отделив ту часть проблемы, которая соответствует математическому формализму, мы получили гибкую архитектуру **Долей (Shares)**, которая позволяет далее дистиллировать ключевые методы **Кредита (Loan)** и **Предприятия (Facility)**. (См. в главе 15 раздел, посвященный СМЫСЛОВОМУ ЯДРУ, или CORE DOMAIN.)

Гибкость архитектуры оказывает огромное влияние на способность программы адаптироваться к изменениям и усложнениям. Как показали примеры из этой главы, она возникает чаще всего благодаря весьма детальному моделированию и тщательному выбору проектных архитектурных решений. Эффект от всего этого может выйти далеко за рамки конкретной проблемы моделирования и проектирования. В главе 15 будет рассматриваться стратегическое значение гибкой архитектуры как одного из нескольких инструментов дистилляции модели предметной области, позволяющих превратить большой и сложный проект в нечто обозримое, поддающееся обработке.

Глава 11

Применение аналитических шаблонов

У глубленные модели и гибкие архитектуры не даются в руки так сразу. Прогресс наступает после интенсивного изучения предметной области, активных дискуссий, множества проб и ошибок. Впрочем, иногда в этом деле можно рассчитывать на постороннюю помощь.

Когда опытный разработчик, изучающий предметную область, видит знакомый круг обязанностей или знакомую систему взаимосвязей, у него в памяти может всплыть решение аналогичной задачи, полученное ранее. Какие модели при этом пробовались и какие из них оказались рабочими? Какие трудности возникли при реализации и как их преодолели? Вдруг оказывается, что пробы и ошибки из предыдущего опыта имеют прямое отношение к новой ситуации. Некоторые из шаблонов, описывающих такие ситуации, хорошо документированы и достаточно распространены, что позволяет остальным разработчикам пользоваться накопленным опытом.

По сравнению с фундаментальными шаблонами структурных элементов программ, представленными в части II, и принципами проектирования гибкой архитектуры из главы 10, эти шаблоны относятся к более высокому уровню и более специализированы; в них для представления некоторой концепции комбинируются несколько объектов. Благодаря им можно "проскочить" этап проб и ошибок, сразу начав с модели достаточно выразительной, хорошо реализуемой и отвечающей тонким неочевидным требованиям, на изучение которых может уйти много усилий. Начиная с этой отправной точки, мы выполняем рефакторинг и экспериментируем, потому что это еще не готовые стандартные решения с конвейера.

Мартин Фаулер (Martin Fowler) в книге *Analysis Patterns: Reusable Object Models* ("Аналитические шаблоны: объектные модели многократного использования") определяет такие шаблоны следующим образом [11].

> *Аналитические шаблоны*[1] *— это группы понятий или концепций, представляющие часто используемую конструкцию (или логическое построение) в моделировании прикладной деятельности. Эта конструкция может относиться как к одной, так и сразу к нескольким предметным областям.*

Представленные М. Фаулером шаблоны возникли из опыта в данной области, поэтому они вполне практичны, если их правильно применить к ситуации. Эти шаблоны предоставляют тому, кому предстоит работать с непростой предметной областью, это — ценные "отправные точки" для запуска итерационного процесса разработки. Само название

---

[1] Возможно, это и не самый удачный перевод. Иначе это выражение можно толковать как "схема, образец анализа". Анализ в данной книге понимается узко — как аналитическая часть разработки программы, изучение проблемы с целью построения модели. — *Примеч. перев.*

этого вида шаблонов подчеркивает их характер: это не готовые технические решения, а общие указания по разработке модели для конкретной области.

К сожалению, название не отражает другого факта: реализация этих шаблонов требует обширных дискуссий и большой работы над кодом. Фаулер понимает, чем грозит увлечение анализом без связи с конкретной архитектурой. Вот интересный пример, в котором этот автор мыслит даже дальше установки приложения на его рабочей платформе – его заботит долгосрочное влияние тех или иных вариаций модели на техническую поддержку работающей системы.

> *Когда мы внедряем новую практику [в бухгалтерском учете], мы создаем сеть новых примеров для того или иного правила проводки через бухгалтерские книги. Мы делаем это без перекомпиляции и пересборки системы — она по-прежнему эксплуатируется в рабочем режиме. В некоторых случаях мы не можем обойтись без введения нового подтипа правила проводки, но это очень редкая оказия.*

В зрелом, давно разрабатываемом проекте, из опыта работы с приложением часто можно понять, какие именно варианты модели в нем реализованы, какие проектные решения приняты. В таких случаях апробируется много разных реализаций всевозможных компонентов, некоторые из них будут работать в финальной версии и даже доживут до стадии доработки в ходе эксплуатации. При наличии этого опыта можно избежать многих проблем. Аналитические шаблоны способны помочь в переносе этого опыта из других проектов. При работе с ними наличие углубленных знаний о модели сочетается с интенсивными дискуссиями по проектным решениям и последствиям тех или иных вариантов реализации. Если обсуждать идеи модели вне этого контекста, они становятся сложнее в применении, и возникает риск опасного разрыва между анализом и проектированием, а это полная противоположность ПРОЕКТИРОВАНИЮ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN).

Сам принцип устройства и применения аналитических шаблонов лучше пояснить на примере, а не в абстрактных описаниях. В этой главе приводятся два примера того, как разработчики пользуются небольшой, но представительной выборкой моделей из [11]. Аналитические шаблоны будут описаны ровно в том объеме, который нужен для понимания примеров. Здесь не предпринимается попытка каталогизировать шаблоны этого типа или даже полностью описать те, которые выбраны для примеров. Мы ставим перед собой цель проиллюстрировать их интеграцию в процесс предметно-ориентированного проектирования программ.

Пример

Получение процентного дохода по счетам

В главе 10 было рассказано о нескольких способах, которыми разработчик может добиться углубления своей модели, созданной для реализации узкоспециализированного финансового приложения. Здесь предлагается еще один сценарий. В этот раз программисту предстоит искать полезные идеи в книге М. Фаулера (M. Fowler) *Analysis Patterns* (“Аналитические шаблоны”).

Вкратце говоря, у нас есть приложение, следящее за состоянием выданных кредитов и других финансовых активов с процентным доходом, рассчитывающее начисляемые проценты и сборы и отслеживающее поступление платежей от заемщиков. По ночам запускается пакетный процесс, который собирает все эти цифры и передает их в ранее написанную систему бухгалтерского учета, указывая при этом, по какой книге следует про-

вести каждый платеж. Существующая архитектура работает, но она неудобна в использовании, плохо поддается изменениям и не слишком информативна.

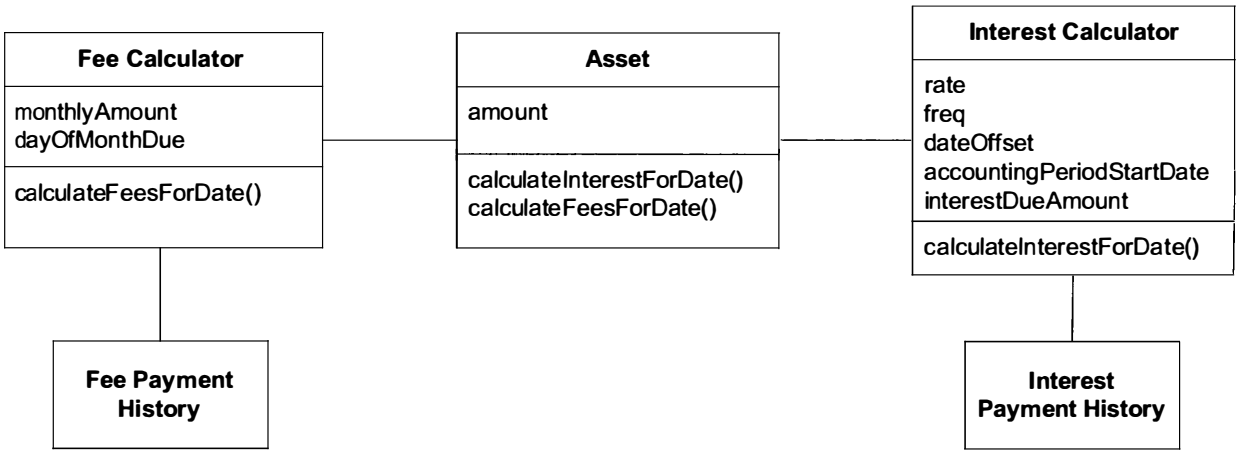


Рис. 11.1. Первоначальная диаграмма классов

Разработчик принимает решение прочитать главу 6 из вышеупомянутой книги под названием "Inventory and Accounting" ("Управление запасами и бухгалтерский учет"). Ниже дается резюме той части, которая показалась ему наиболее близкой к поставленной задаче.

Модели бухгалтерского учета

Коммерческие приложения всевозможных разновидностей часто имеют дело со *счетами* (*accounts*), на которые зачисляются ценности — обычно деньги. Во многих случаях отслеживать только общую сумму на счету бывает недостаточно — важно также контролировать и учитывать каждое изменение этой суммы. Именно это требование лежит в основе самых элементарных моделей бухучета.

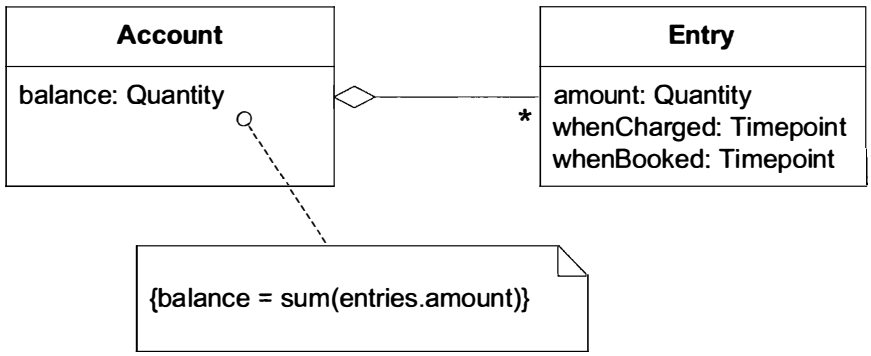


Рис. 11.2. Элементарная модель бухгалтерского учета

Счет можно пополнить добавлением *Записи* (`Entry`) в бухгалтерскую книгу. Можно списать сумму со счета, добавив отрицательную **Запись** (**`Entry`**). Эти **Записи** никогда не удаляются из книг, так что сохраняется полная история изменений. Бухгалтерский баланс представляет собой совокупный эффект всех **Записей**. Баланс можно по требованию рассчитать или выдать наличными — это проектное решение инкапсулируется интерфейсом **Счет** (**`Account`**).

Фундаментальным принципом бухгалтерского учета является *закон сохранения*. Деньги не появляются из ниоткуда и не исчезают без следа. Они только переходят с одного **Счета** на другой.

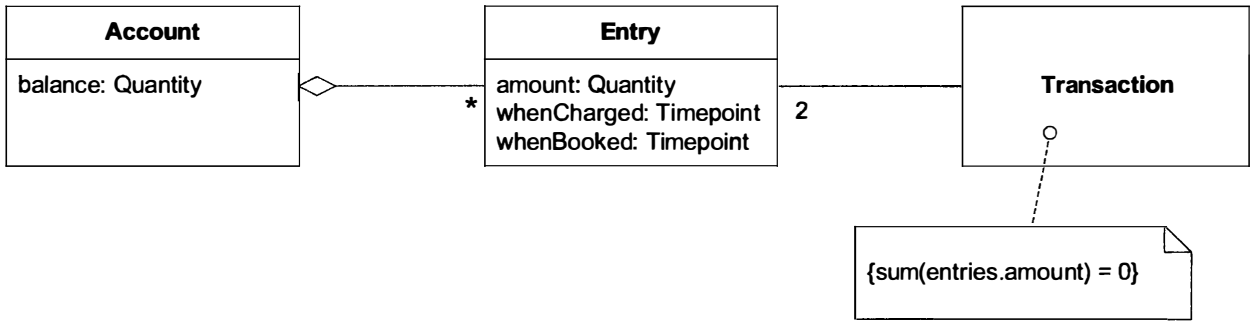


Рис. 11.3. Модель с транзакцией

Этот закон по сути выражает и известная концепция *двойной бухгалтерской записи* (двойной итальянской бухгалтерии): у каждого кредита есть соответствующий дебет. Конечно, этот закон сохранения, как и другие, применим только к замкнутым системам, содержащим в себе все источники и стоки. Для многих простых приложений такой строгости не требуется.

В книге М. Фаулера предлагаются более сложные и проработанные виды таких моделей, а также много говорится об их относительных преимуществах и недостатках.

Прочитав все это, программист **(Прогр. 1)** приходит к некоторым новым умозаключениям. Он[2] показывает главу из книги коллеге **(Прогр. 2)**, который работал с ним над реализацией алгоритмов вычисления процентов и написал запускаемый каждую ночь пакетный процесс. Вместе они набросали модификации к модели, включив в нее некоторые элементы, о которых прочитали.

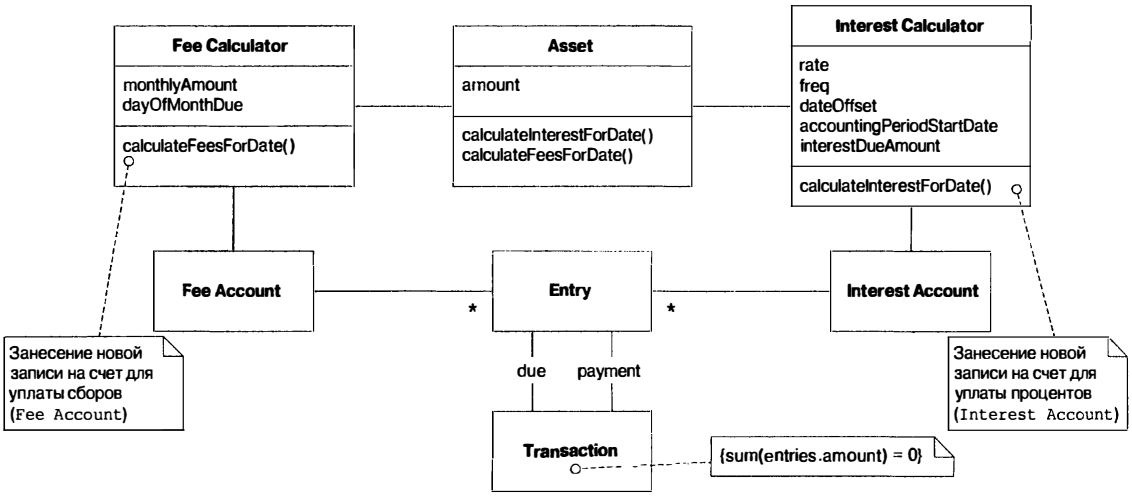


Рис. 11.4. Предложение по новой модели

Затем для обсуждения новых идей относительно модели они приглашают специалиста по предметной области **(Спец.)**.

---

[2] И снова у автора фигурирует "она". Но нам достаточно иметь в виду, что в русском языке мужской род включает женский, и этим программистом может быть кто угодно. — *Примеч. перев.*

Прогр. 1. В этой новой модели мы делаем новую **Запись (Entry)** для проводки по **Процентному счету (Interest Account)**, а не просто изменяем величину суммы процентного дохода к поступлению на счет (`interestDueAmount`). Затем для поддержания баланса вносится еще одна **Запись**, касающаяся выполненного платежа.

Спец. Получается, теперь мы сможем видеть всю историю процентных начислений так же, как и историю платежей? Мы давно хотели получить такую возможность.

Прогр. 2. По-моему, мы не вполне правильно используем данную "**Транзакцию**". В определении говорится о переводе средств с одного **Счета (Account)** на другой, а не о балансе двух записей на одном и том же **Счету**.

Прогр. 1. Кстати, да. Меня тоже беспокоило, почему в книге так настойчиво подчеркивается, что транзакция создается вся сразу. Фактические платежи по процентам могут выполняться на несколько дней позже.

Спец. Эти платежи совсем не обязательно запаздывают. Существует очень гибкая практика их получения.

Прогр. 1. То есть, это может быть тупиковый вариант. Я тут подумал, что стоит определить некоторые неявные понятия. Так, будет информативнее, если именно **Калькулятор процентов (Interest Calculator)** будет создавать объекты-**Записи (Entry)**. А **Транзакция**, по-моему, "изящно" связывает вычисляемые проценты с платежами по ним.

Спец. Зачем нам связывать начисления с платежами? Это отдельные проводки бухгалтерской системы. Самое главное — баланс на **Счету**. Он, да еще набор отдельных **Записей** — вот все, что нам нужно.

Прогр. 2. Вы имеете в виду, что не следите, проведена ли выплата процентов?

Спец. Нет, следим, конечно. Но все не так просто, как нарисованная вами схема "одно начисление — один платеж".

Прогр. 2. Вообще-то, если избавиться от этой взаимосвязи и больше о ней не беспокоиться, это многое бы упростило.

Прогр. 1. Ладно, а что если так? [*Берет копию старой диаграммы классов и начинает набрасывать изменения.*] Кстати, вы несколько раз употребили слово "начисления" (*accruals*). Не могли бы вы объяснить подробнее?

Спец. Конечно. Начисление — это действие по учету прихода или расхода средств вне зависимости от того, когда именно деньги переходят из рук в руки. Так что мы начисляем проценты каждый день, а платеж выполняется один раз в конце месяца (к примеру).

Прогр. 1. Да, действительно, такое слово нам не помешает. И как же это выглядит?

Прогр. 1. Теперь можно избавиться от всех осложнений, которые возникали в калькуляторе от привязки к платежам. К тому же мы ввели термин *начисления*, который лучше передает суть дела.

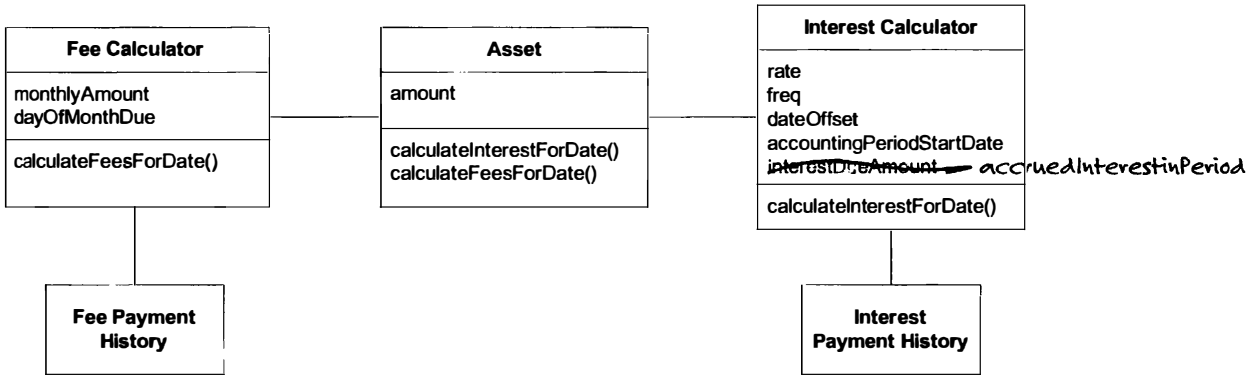


Рис. 11.5. Исходная диаграмма классов, где начисления отделены от платежей

Спец. Так что же, у нас теперь не будет объекта **Счет (Account)**? А я так ждал возможности увидеть в нем все вместе: и начисления, и платежи, и общий баланс.

Прогр. 1. Что, правда?! Тогда, может быть, подойдет это. [*Берет другую диаграмму и рисует.*]

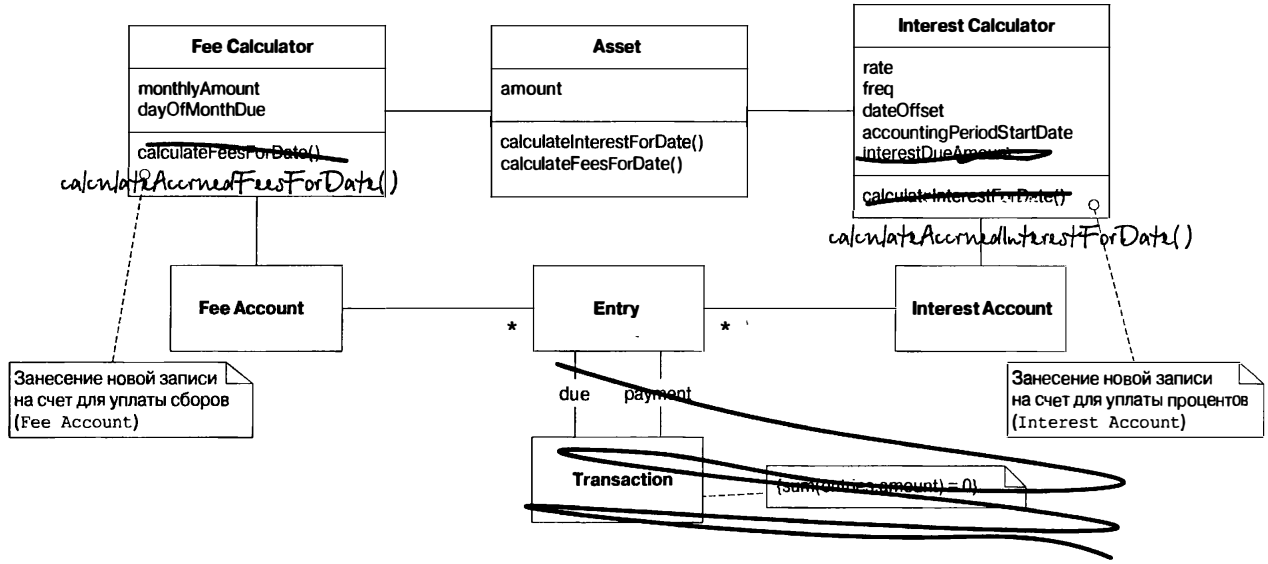


Рис. 11.6. Диаграмма на основе работы со счетами без ***Транзакции***

Спец. А вот это весьма неплохо!

Прогр. 2. Пакетный сценарий легко будет изменить так, чтобы использовались эти новые объекты.

Прогр. 1. Чтобы запустить новый **Калькулятор процентов (Interest Calculator)**, потребуется несколько дней. Нужно поменять довольно многое в тестах, но, в конце концов, они станут более наглядными.

И двое программистов взялись за рефакторинг на основе новой модели. По ходу работы с кодом и "подтягивания" архитектуры они приходили к новым выводам, которые помогали улучшить модель.

Записи (Entries) были подразделены на **Платежи (Payment)** и **Начисления (Accrual)**, поскольку тщательное исследование показало, что у этих категорий записей в приложении несколько разные обязанности. К тому же и то, и другое — важные понятия предметной области. С другой стороны, между **Записями** нет никакой принципиальной или функциональной разницы в том смысле, относятся ли они к сборам или процентам — они просто соответствуют тому или иному **Счету**.

К сожалению, получилось так, что разработчики вынуждены были отказаться от этой последней абстракции из соображений реализации. Данные хранились в реляционной таблице, и стандарты проекта требовали, чтобы эта таблица поддавалась интерпретации без запуска программы. Это означало, что записи о сборах и о процентах должны были храниться в отдельных таблицах. Единственный способ, каким программисты могли это реализовать, используя собственный механизм объектно-реляционного отображения, — это ввести частные подклассы **Платеж сбора (Fee Payment)**, **Платеж процентов (Interest Payment)** и т.д. Будь у них другая инфраструктура, этого неуклюжего расширения можно было бы избежать.

Я придумал этот поворот сюжета в почти целиком выдуманной истории для того, чтобы придать ей вкус реальности, которая постоянно нас окружает. Нам приходится

идти на рассчитанные компромиссы, а затем двигаться дальше, не позволяя этим компромиссам заставить нас свернуть с пути ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN).

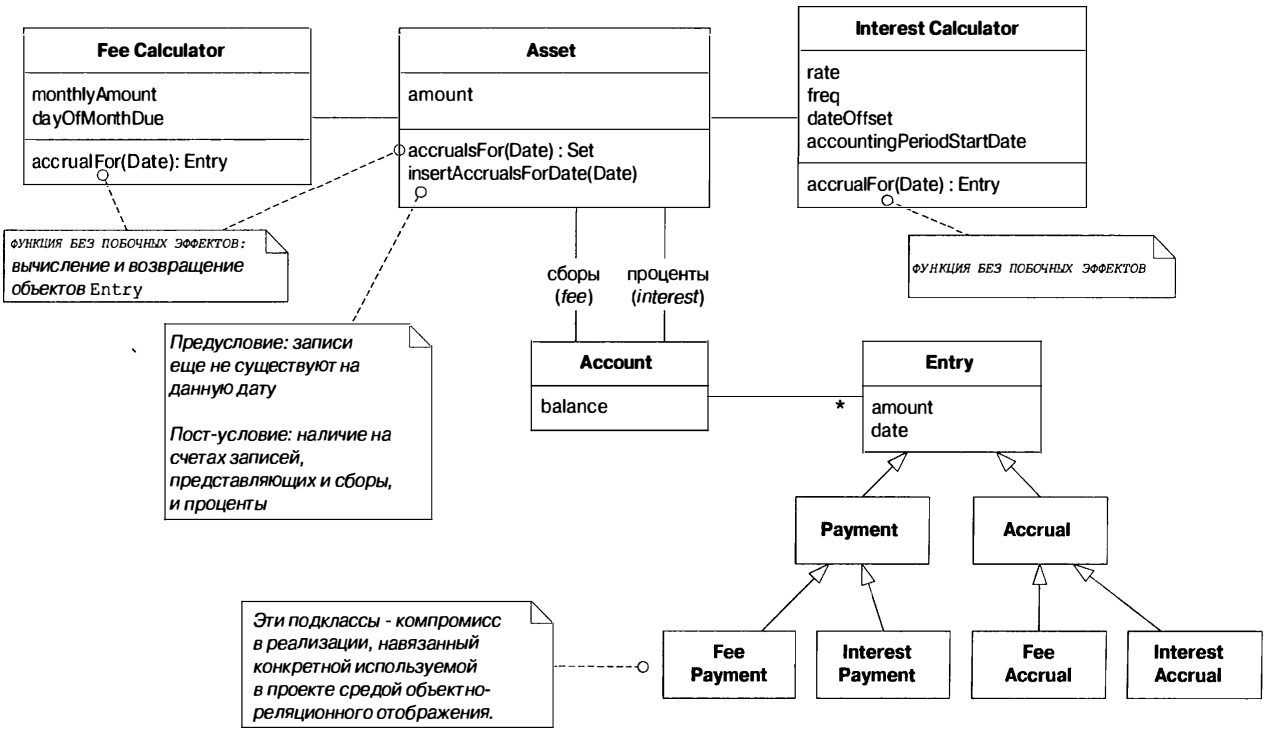


Рис. 11.7. Диаграмма классов после реализации

Новая архитектура оказалась значительно проще для анализа и тестирования, потому что большая часть ее сложных операций была вынесена в ФУНКЦИИ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ. Оставшаяся командно-директивная часть состоит из самого простого кода (потому что вызывает много ФУНКЦИЙ) и характеризуется контрольными УТВЕРЖДЕНИЯМИ (ASSERTIONS).

Бывают такие фрагменты кода, из которых, на первый взгляд, вряд ли возможно сделать что-нибудь хорошее на основе модели предметной области. Возможно, они начинались с очень простого кода, который затем дорабатывался механически и приобретал вид не столько прикладного алгоритма из предметной области, сколько запутанной операционной части программы. Для выявления этих "пятен" и бывают полезны аналитические шаблоны.

В следующем примере программист сделал новые и полезные выводы о таком "черном ящике", как ночной пакетный сценарий, который раньше вообще не считался предметно-ориентированным.

Пример (продолжение)

Изучаем пакетный сценарий

После нескольких недель работы модель на основе объекта **`Счет (Account)`** стала, наконец, вырисовываться. Как это часто бывает, из-за наглядности новой архитектуры стали более очевидными другие недостатки. Один из разработчиков (**Программист 2**), который адаптировал ночной сценарий для взаимодействия с новой архитектурой, начал

видеть связь между работой этого сценария и некоторыми понятиями из книги об аналитических шаблонах. Ниже кратко описаны те понятия, которые он счел наиболее важными в данной ситуации.

Правила бухгалтерской проводки

В системах бухгалтерского учета часто предлагается много вариантов представления одной и той же финансовой информации. На одном счету может вестись учет дохода, тогда как на другом — расчетного налога на этот доход. Если от системы требуется автоматически обновлять налоговый счет, то реализация этих двух счетов становится в высшей степени взаимосвязанной. Существуют системы, в которых большинство бухгалтерских записей (проводок) выполняется по таким регламентам; в них сеть взаимосвязей неизбежно превращается в кашу. Даже в более скромных по масштабам системах выполнять все эти перекрестные проводки бывает непросто. Первый шаг к тому, чтобы обуздать паутину взаимосвязей — это сформулировать регламентные правила явно, введя для этого новый объект.

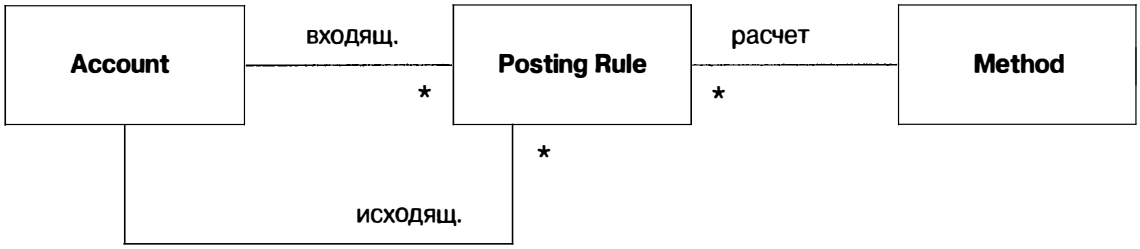


Рис. 11.8. Диаграмма классов для правила проводки

Правило проводки (**Posting Rule**) включается в работу новой **Записью** (**Entry**) на ее "входящем" счету. Затем оно порождает новую **Запись** на основе ее собственного расчетного **Метода** (**Method**) и добавляет ее на "исходящий" **Счет** (**Account**). В системе учета заработной платы добавление **Записи** на счет фонда зарплаты, например, может активизировать **Правило проводки**, которое начисляет 30% расчетного подоходного налога и помещает эту цифру в виде **Записи** на **Счет**, предназначенный для вычета налогов.

Выполнение правил проводки

Правило проводки (**Posting Rule**) устанавливает концептуальную связь между различными **Счетами** (**Accounts**), но если бы на этом аналитический шаблон и закончился, следовать ему было бы нелегко. Одна из главных сложностей при проектировании взаимосвязей — это управление обновлениями, в том числе расчет времени для них. Фаулер рассматривает три варианта.

1. "*Немедленный запуск*" — наиболее очевидный, но на практике не слишком удачный. Как только **Запись** (**Entry**) поступает на **Счет**, она немедленно активизирует **Правило проводки** (**Posting Rule**), и все обновления выполняются сразу же.
2. "*Запуск со* **Счета**" — вариант, при котором обработка задерживается. В какой-то момент объекту **Счет** посылается сообщение, и он активизирует свои **Правила проводки**, чтобы обработать все **Записи**, добавленные после предыдущего запуска.
3. И наконец, "*независимый запуск* **Правила проводки**" инициируется внешним агентом, который отдает распоряжение, когда подключать нужное правило. **Правило**

проводки отвечает за то, чтобы найти все **Записи**, поступившие на "входящие" **Счета** с последнего запуска.

Разные режимы запуска можно смешивать в одной системе, но каждому конкретному набору регламентных правил нужно иметь одну четко определенную точку активизации, а также полномочия по нахождению **Записей** "входящих" **Счетов**.

Добавление трех режимов запуска в ЕДИНЫЙ ЯЗЫК так же важно для успешной реализации шаблона, как и сами определения объектов модели. Этим устраняется неоднозначность в принятии решений, и оно сводится к четко определенному набору возможных вариантов. Знание трех указанных режимов позволяет увидеть нетривиальную проблему и обогащает словарь нужными терминами для осмысленной дискуссии.

Программисту 2 необходимо было озвучить его новые идеи в устной дискуссии. Он встретился с коллегой (**Программистом 1**), который в основном и отвечал за моделирование начислений.

Прогр. 2. В какой-то момент пакетный сценарий превратился в нечто наподобие той избы, из которой никогда не выносят сор. В операциях сценария неявно заключена алгоритмическая логика предметной области, и он становится все сложнее и сложнее. Я уже давно мечтаю перепроектировать этот сценарий на основе модели, выделить в нем уровень предметной области, сделать сам сценарий простым уровнем над предметной областью. Но я так и не смог понять, какова, собственно, сама эта модель. Кажется, что все сводится к набору неких процедур, которые в виде объектов не имеют особого смысла. Читая главу по **Правилам проводки** (**Posting Rule**) в книге об аналитических шаблонах, я пришел к некоторым идеям. Вот что я придумал. [*Передает рисунок.*]

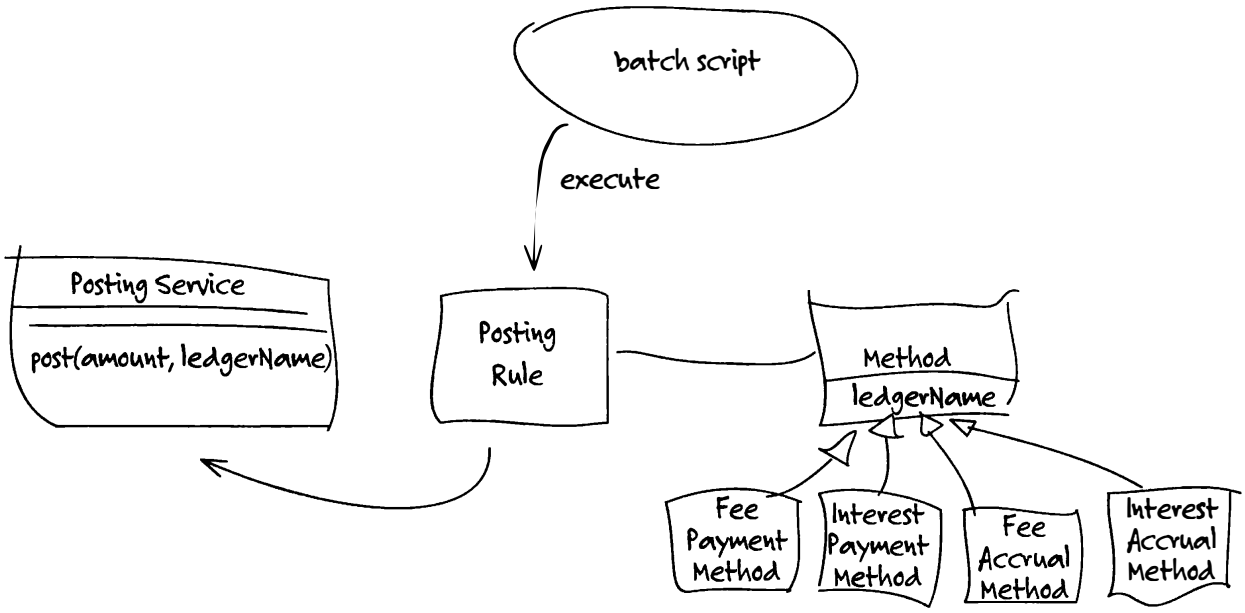


Рис. 11.9. Попытка реализовать ***Правила проводки*** *(**Posting Rules**) в пакетном сценарии*

Прогр. 1. Что такое эта **Служба проводки** (**Posting Service**)?

Прогр. 2. Это ФАСАДНЫЙ МЕТОД (FACADE), который представляет прикладной программный интерфейс (API) программы бухучета в виде СЛУЖБЫ (SERVICE). Я фактически создал его уже довольно давно, чтобы упростить код сценария, и он предоставил мне также ИНФОРМАТИВНЫЙ ИНТЕРФЕЙС (INTENTION-REVEALING INTERFACE) для проводок через старую систему.

Прогр. 1. Интересно. Так какой же вариант запуска обновления вы планируете для этих **Правил проводки**?

Прогр. 2. Вот до этого я еще не дошел.

Прогр. 1. "Немедленный запуск" подошел бы для **Начислений (Accruals)**, поскольку пакетный сценарий, по сути, приказывает **Активу (Asset)** добавить их, но для **Платежей (Payments)**, вводимых в течение дня, он не годится.

Прогр. 2. По-моему, так тесно привязывать метод расчета к сценарию нам не нужно. Если нам когда-нибудь понадобится запустить вычисление процентов в другое время, от этого все перепутается, да и с концептуальной точки зрения это неправильно.

Прогр. 1. Похоже, в нашем случае уместен независимый запуск **Правила проводки**. Сценарий отдает приказ о выполнении каждому **Правилу проводки**, оно запускается, разыскивает новые **Записи** и делает, что ему положено. В общем, так вы это и нарисовали.

Прогр. 2. Итак, мы избавляемся от большого количества взаимосвязей в архитектуре пакетного сценария, и он при этом сохраняет управление. Кажется, это правильно.

Прогр. 1. Мне по-прежнему не все ясно во взаимодействии этих объектов со **Счетами (Accounts)** и **Записями (Entries)**.

Прогр. 2. Мне тоже. В примерах из книги создается прямая связь между **Счетами** и **Правилами проводки (Posting Rules)**. В принципе это логично, но нам это, по-моему, не подойдет. Нам придется каждый раз создавать эти объекты по данным, и мы вынуждены будем выяснять, какое правило здесь применимо, чтобы корректно ассоциировать его. Между тем именно объект **Актив (Asset)** знает, что содержится на каждом **Счету**, и соответственно, какое правило должно применяться. Так что там насчет остального?

Прогр. 1. Ненавижу придираться, но, по-моему, мы неправильно пользуемся "**Методом**" ("**Method**"). Мне кажется, его концепция состоит в том, что **Метод** вычисляет проводимую сумму — например, высчитывает 20% налога из дохода. Но в нашем случае это просто: проводится всегда полная сумма. Я думаю, само **Правило проводки** должно "знать", по какому счету выполнять проводку, что соответствует нашему "имени бухгалтерской книги".

Прогр. 2. Ага. То есть, если возложить на **Правило проводки** обязанность знать нужную книгу, то **Метод** нам, скорее всего, не понадобится вообще.

Получается, что все эти дела с выбором нужной книги становятся все сложнее и сложнее. Это уже сочетание вида дохода (проценты или сборы) с "классом актива" (категорией, которая присваивается каждому **Активу** по роду деятельности). Надеюсь, что именно в этом месте нам удастся получить какую-то помощь от новой модели.

Прогр. 1. Ладно, сосредоточимся на этом. **Правило проводки** отвечает за выбор бухгалтерской книги по атрибутам **Счета**. Пока что можно напрямую оформить выбор класса актива и различие между процентами и сборами. В дальнейшем у нас будет ОБЪЕКТНАЯ МОДЕЛЬ, которую можно усовершенствовать для работы с более сложными случаями.

Прогр. 2. Мне нужно еще подумать над этим. Я "прокручу" все это в голове и перечитаю шаблоны, а потом предприму еще одну попытку. Можно поговорить с вами снова, завтра после обеда?

В течение следующих нескольких дней двое программистов разработали модель и выполнили рефакторинг кода, так что сценарий стал просто перебирать **Активы (Assets)**, посылая каждому несколько четких и понятных сообщений, а затем выполняя транзакцию базы данных. Вся сложность перешла на уровень предметной области, где объектная модель позволила добиться большей наглядности и абстрагирования операций.

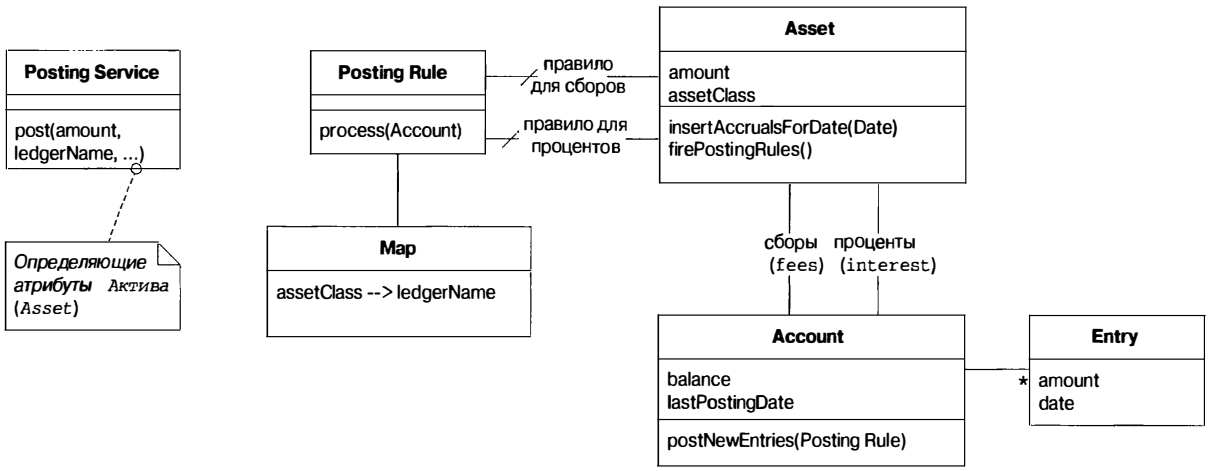


Рис. 11.10. Диаграмма классов с ***Правилами проводки (Posting Rules)***

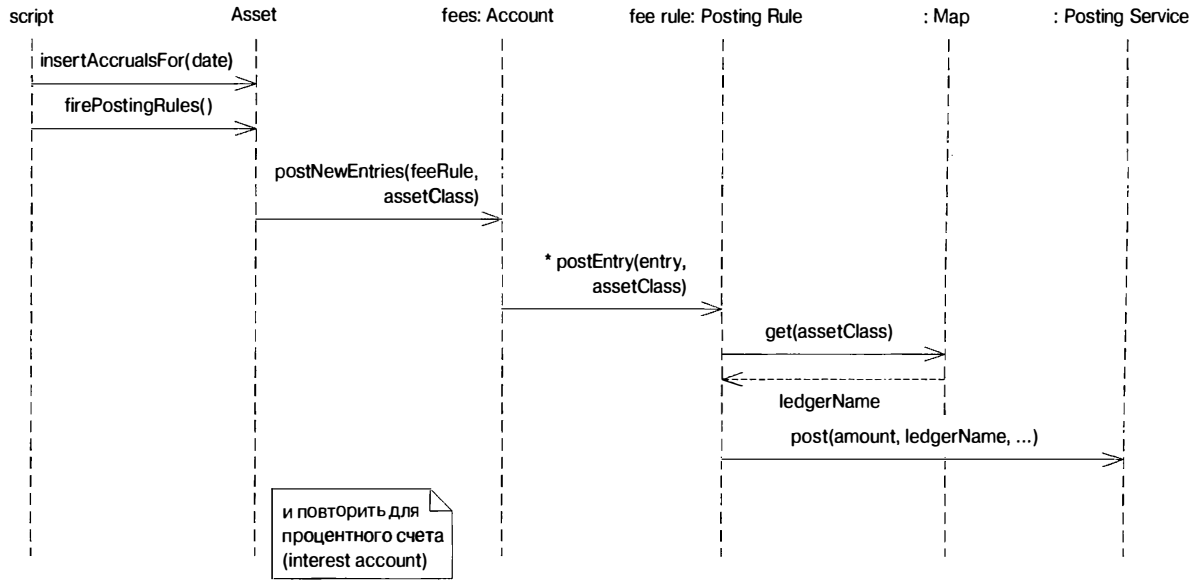


Рис. 11.11. Циклограмма, на которой показано выполнение правил

Разработчики существенно отошли от подробностей устройства моделей, описанных в книге об аналитических шаблонах, но им все же казалось, что они сохранили саму суть. Правда, их несколько смущало привлечение **Актива (Asset)** к выбору **Правила проводки (Posting Rule)**. Они выбрали этот способ, потому что **Актив** знает разновидность каждого **Счета** (для сборов или для процентов), а также представляет собой естественную точку доступа для сценария. Если ассоциировать объект правила непосредственно со **Счетом (Account)**, это потребовало бы сотрудничества с объектом **Актив (Asset)** при каждой инициализации экземпляров объектов (т.е. при каждом запуске сценария). Вместо этого объект **Актив** разыскивает два нужных правила через их объекты-ОДИНОЧКИ (SINGLETON) и передает им соответствующий **Счет**. В таком виде код стал более четко выражать суть дела, так что программисты сошлись на этом практичном решении.

Оба разработчика полагали, что с концептуальной точки зрения было бы лучше ассоциировать **Правила проводки (Posting Rules)** только со **Счетами (Accounts)**, оставив **Активу** его основную работу — формирование **Начислений (Accruals)**. Они надеялись, что последующий рефакторинг и углубленное понимание модели снова приведет их к этому подходу и покажет способ, как выполнить это четкое разделение, не теряя наглядности кода.

Аналитические шаблоны как источник знания

Если вам даже и повезло иметь аналитический шаблон для вашего случая, он вряд ли окажется готовым решением конкретной задачи. И все же он предлагает ценные "путеводные подсказки", а также хорошо абстрагированный словарь. Кроме того, из аналитического шаблона можно почерпнуть сведения о последствиях его реализации, которые могут сэкономить лишние усилия.

Все это "горючее" поступает в машину переработки знаний и рефакторинга по модели, стимулируя качественную разработку. Результат часто напоминает по форме то, что задокументировано в аналитическом шаблоне, но с адаптацией к обстоятельствам. Иногда результат даже не имеет ярко выраженного сходства с самим аналитическим шаблоном, а, скорее, стимулируется выводами и знаниями, полученными из шаблона.

Есть одна разновидность модификации шаблона, которой следует избегать. Используя термин из хорошо известного аналитического шаблона, старайтесь оставить нетронутыми его фундаментальные понятия, пусть даже изменения и кажутся совсем поверхностными. Для этого есть две причины. Во-первых, в шаблоне может заключаться такое понимание проблемы, которое поможет вам избежать неприятностей. Во-вторых, что важнее, ваш ЕДИНЫЙ ЯЗЫК совершенствуется, когда в него добавляются термины либо широко распространенные, либо хотя бы хорошо описанные. Если определения в вашей модели изменяются в ходе ее естественной эволюции, не забудьте внести изменения и в имена с названиями.

Об объектных моделях написано довольно много. Некоторые из них специализированы под один вид приложений в одной отрасли знания, а другие могут быть довольно общими. В большинстве из них содержится зерно некоей идеи, но только в некоторых передан ход рассуждений, стоящий за проектными решениями, и последствия этих решений, а именно это — самое важное в аналитических шаблонах. Было бы неплохо иметь побольше таких усовершенствованных аналитических шаблонов — это помогло бы нам не изобретать велосипед снова и снова. Я буду очень удивлен, если когда-нибудь появится универсальный каталог шаблонов, но по отдельным отраслям знания такие каталоги вполне могут сформироваться. И шаблоны из некоторых предметных областей, где пересекается множество приложений, вполне могут получить широкое распространение.

Данная разновидность повторного использования организованных знаний совершенно отличается от переноса готового кода посредством сред или компонентов. Сходство состоит разве что в том, что и там, и там можно найти рациональное зерно совсем не очевидной идеи. Модель, даже в виде обобщенной архитектурной среды, это единое работающее целое, тогда как аналитический шаблон — это набор фрагментов конструктора. В аналитических шаблонах упор делается на наиболее критические и трудные решения, подчеркиваются имеющиеся альтернативы и варианты выбора. В них также рассматриваются долговременные последствия, которые могут дорого обойтись, если всякий раз разбираться с ними заново.

Глава 12

Шаблоны и модель

Архитектурные шаблоны, или образцы, рассмотренные до сих пор в этой книге, специально предназначены для решения проблем в модели предметной области, следуя контексту ПРОЕКТИРОВАНИЯ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN). Однако, честно говоря, большинство опубликованных до сей поры шаблонов касаются больше технической стороны дела. В чем же разница между архитектурным шаблоном (*design pattern*) и шаблоном уровня предметной области (*domain pattern*)? Для начала следует выслушать авторов основополагающей книги в этой области, *Design Patterns*.

> *Понятие о том, что собою представляет и чего не представляет шаблон, индивидуально и зависит от точки зрения. То, что для одного — полноценный шаблон, другому может показаться примитивным структурным "кирпичиком" программы. В этой книге мы занимаемся шаблонами на определенном уровне абстракции. Говоря об архитектурных шаблонах, мы не имеем в виду такие архитектурные элементы, как связанные списки или хэш-таблицы, которые можно оформить в виде классов и переносить, куда потребуется, в готовом виде. Но это и не сложные специализированные архитектуры для целого приложения или подсистемы. Архитектурные шаблоны из этой книги представляют собой описания способов взаимодействия между объектами и классами, адаптированные для решения общей задачи архитектурного проектирования в конкретном контексте [14].*

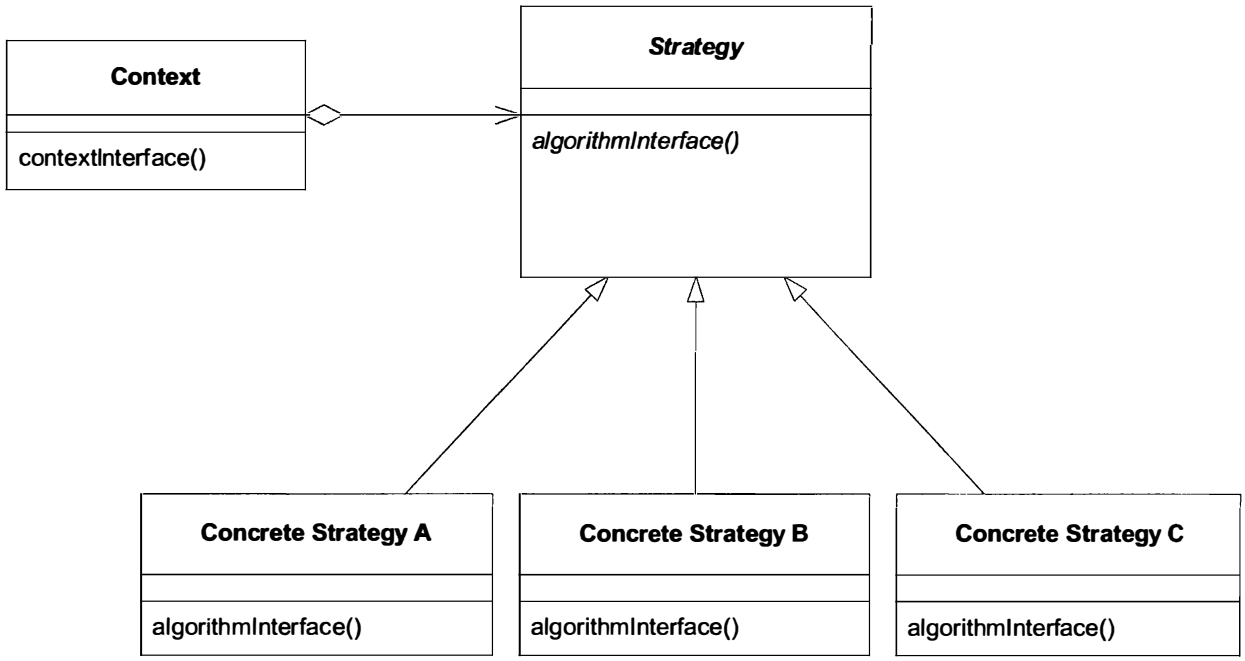
Некоторые, но не все, шаблоны из книги *Design Patterns* могут использоваться в качестве шаблонов уровня предметной области. Но чтобы добиться этого, нужно сместить акценты. В упомянутой книге представлен каталог архитектурных элементов, которые проявили себя в решении распространенных проблем из целого ряда разных контекстов. Причины использования этих шаблонов, как и сами шаблоны, изложены с чисто технической точки зрения. Но некоторое подмножество этих элементов может применяться в более широком контексте моделирования и архитектурного проектирования уровней предметных областей, поскольку такие элементы соответствуют общим понятиям, возникающим во многих областях знания.

В дополнение к шаблонам из книги *Design Patterns* за много лет накопилось и опубликовано множество других технических шаблонов. Некоторые из них соответствуют углубленным понятиям, возникающим в разных предметных областях. Хорошо, когда есть возможность пользоваться готовыми результатами. Но чтобы применить такие шаблоны в предметно-ориентированном проектировании программ, необходимо рассматривать их сразу на двух уровнях. На одном уровне это технические архитектурные образцы для написания корректного кода. На другом — это концептуальные структуры в модели предметной области.

Чтобы показать, как шаблон, возникший в качестве архитектурного, может применяться в модели предметной области, а также прояснить разницу между техническим шаблоном и концептуальным шаблоном предметной области, мы воспользуемся конкретными при-

мерами из книги *Design Patterns*. На примере шаблонов COMPOSITE (КОМПОЗИТ) и STRATEGY (СТРАТЕГИЯ)[1] будет продемонстрировано, как можно применить классический архитектурный шаблон в предметной области, посмотрев на него под другим углом...

Стратегия



Определите семейство алгоритмов, инкапсулируйте каждый из них и сделайте их взаимозаменяемыми. СТРАТЕГИЯ[2] позволяет изменять алгоритм, не влияя на клиентов, обращающихся к нему [14]

Модели предметных областей могут содержать процессы, для которых нет технической мотивации, но зато они имеют прямой смысл в соответствующей прикладной деятельности. Если нужно обеспечить наличие альтернативных процессов, сложность выбора подходящего процесса накладывается на сложность собственно реализации нескольких процессов сразу, и все это может выйти из-под контроля.

При моделировании процессов мы часто понимаем, что существует далеко не один законный способ сделать требуемое. Начиная описывать все возможные варианты, мы можем сделать определение процесса громоздким и слишком запутанным. Фактические алгоритмические альтернативы, между которыми мы выбираем, "затемняются", смешиваясь с остальными операционными аспектами программы.

Хотелось бы отделить все эти вариации от основной концепции процесса. В этом случае как основной процесс, так и возможные варианты стали бы понятнее. На решение именно этой проблемы и нацелен шаблон СТРАТЕГИЯ (STRATEGY), давно устоявшийся в сообществе проектировщиков программного обеспечения. Хотя основной упор в нем сделан на техническую сторону дела, здесь он применяется как концепция в модели и от-

---

[1] В английском языке это слово звучит несколько менее "высокопарно" и с меньшим размахом, чем по-русски — скорее, как система правил, рекомендаций, норм, принципов. — *Примеч. перев.*

[2] В названии раздела фигурирует также синоним этого термина — *policy*, дословно "политика", а по смыслу набор критериев и требований. — *Примеч. перев.*

ражается в программной реализации. Потребность в отделении изменчивой, многовариантной части процесса от более стабильной его составляющей присутствует и здесь.

Выделите изменчивую часть процесса в отдельный объект-"стратегию" вашей модели. Отделите правило от операций, которыми оно управляет. Реализуйте правило или подставляемый процесс, следуя архитектурному шаблону СТРАТЕГИЯ **(**STRATEGY**). Несколько версий такого объекта представляют различные способы, которыми можно реализовать процесс.**

Обычно СТРАТЕГИЮ понимают лишь как технический шаблон, суть которого составляет возможность подставлять разные алгоритмы. Но при его использовании в качестве шаблона предметной области основной интерес представляет его способность выражать понятие, концепцию -- обычно это процесс или нормативное правило.

Пример

Процедуры определения маршрутов

В **Службу маршрутизации (Routing Service)** передается **Спецификация маршрута (Route Specification)**, и там конструируется подробный **Путь следования (Itinerary)**, удовлетворяющий данной СПЕЦИФИКАЦИИ. Данная СЛУЖБА является механизмом оптимизации, который можно настроить на поиск либо самого быстрого, либо самого дешевого маршрута.

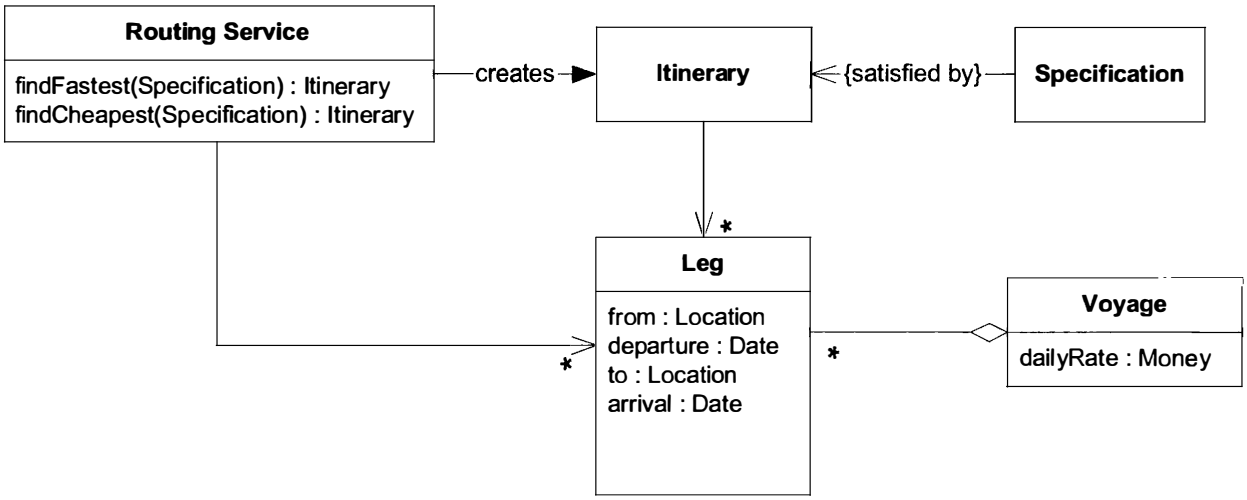


Рис. 12.1. Для реализации вариантов в интерфейсе СЛУЖБЫ требуется логика условного ветвления

Здесь все вроде бы выглядит нормально, но если пристальнее взглянуть на код маршрутизации, то окажется, что условное ветвление, т.е. выбор между самым быстрым и самым дешевым маршрутом, встречается повсеместно, на каждом шаге расчета. Если же нужно будет добавить еще варианты выбора между маршрутами на основе более тонких критериев, дело еще больше запутается.

Одно из возможных решений — выделить все параметры настройки в СТРАТЕГИИ. Тогда они будут представлены в явном виде и передаваться в **Службу маршрутизации (Routing Service)** как единый параметр.

Теперь **Служба маршрутизации** обрабатывает все запросы к ней единообразно, без условного ветвления, разыскивая последовательность **Участков пути (Legs)** с низким весом (*magnitude*), который вычисляется в **Стратегии выбора веса участка (Leg Magnitude Policy)**.

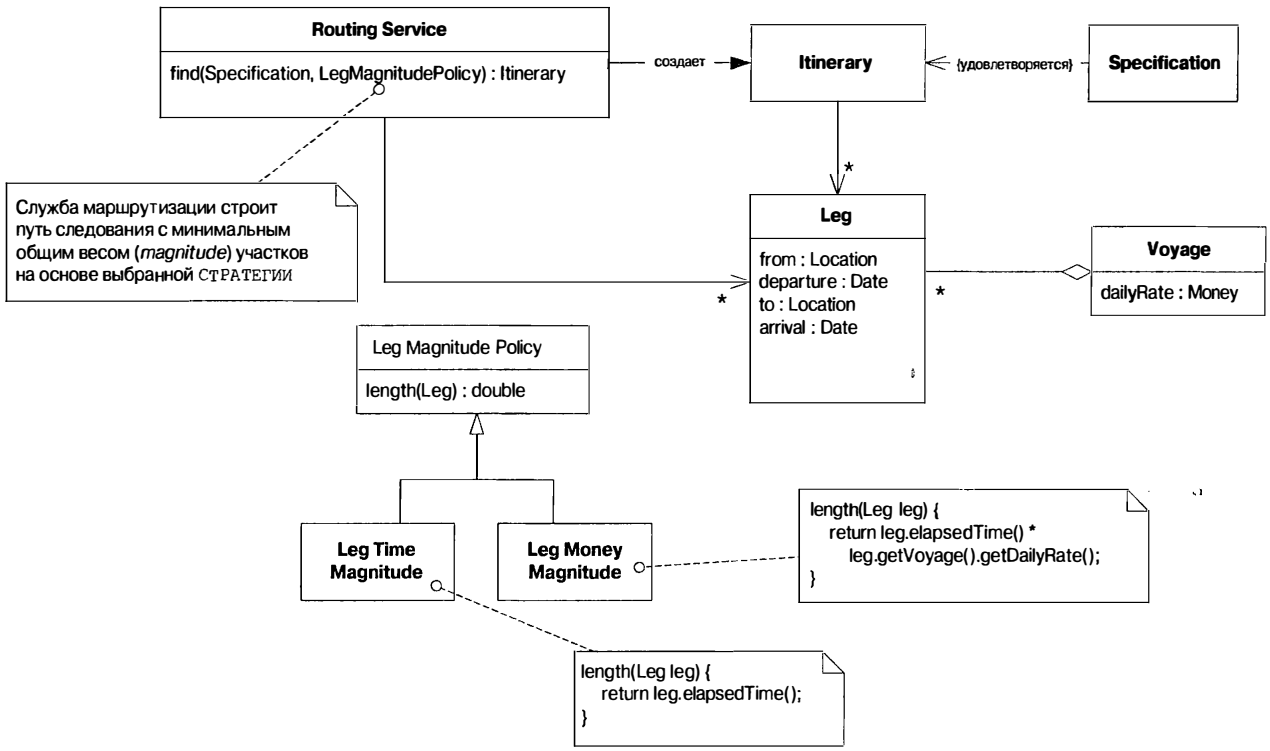


Рис. 12.2. Передача вариантов выбора через параметры в виде единой СТРАТЕГИИ (STRATEGY или POLICY)

В этой архитектуре есть преимущества, которые служат мотивом для применения шаблона СТРАТЕГИЯ в книге *Design Patterns*. Если говорить о гибкости и универсальности операций приложения, то работу **Службы маршрутизации (Routing Service)** теперь можно направлять и расширять как угодно путем добавления новой **Стратегии выбора веса участка (Leg Magnitude Policy)**. СТРАТЕГИИ, показанные на рис. 12.2 (самый быстрый или самый дешевый маршрут), — это только наиболее очевидные варианты. Вполне возможны промежуточные варианты, сочетающие скорость и низкую стоимость. Можно ввести и совершенно другие факторы — скажем, приоритет доставки груза на собственном транспорте компании перед субподрядами на доставку транспортом других фирм. Все эти модификации можно было бы реализовать и без СТРАТЕГИЙ, но тогда соответствующая логика вплеталась бы во внутреннее устройство **Службы маршрутизации (Routing Service)**, запутывая ее интерфейс. А отделение всего этого делает код нагляднее и проще в тестировании.

Фундаментальное правило данной предметной области, основа для предпочтения одного **Участка пути (Leg)** перед другим при построении **Пути следования (Itinerary)**, теперь выражено в явной и четкой форме. Оно передает знание о том, что основой для маршрутизации являются конкретные атрибуты (возможно, производные) отдельных участков пути, сведенные к числовым значениям. Благодаря этому можно описать работу **Службы маршрутизации (Routing Service)** на языке предметной области одним простым предложением: **Служба маршрутизации** выбирает **Путь следования** с минимальным общим весом **Участков пути** на основе выбранной СТРАТЕГИИ.

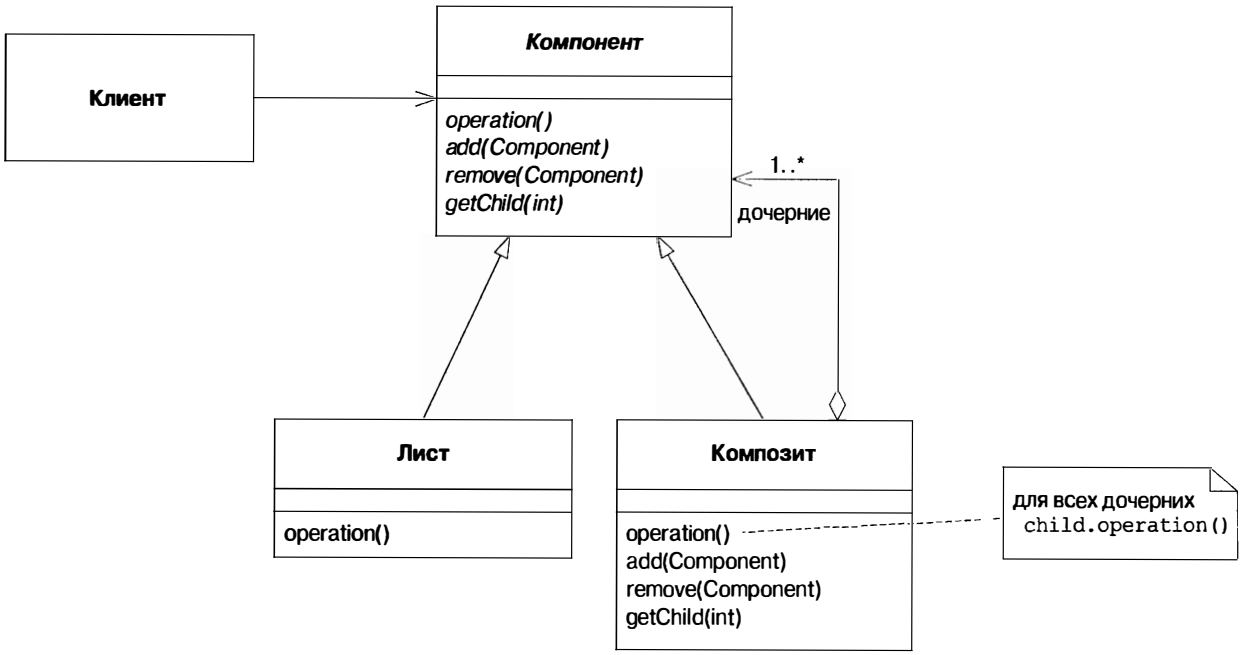
Примечание. Здесь подразумевается, что **Служба маршрутизации** в ходе поиска **Пути следования** сама вычисляет значения веса для **Участка пути**. Этот подход концептуально несложен, и на его основе можно построить прототипную программную реализацию, но в целом он может оказаться неэкономичным и потому неприемлемым. С этим приложением мы еще встретимся в главе 14, где тот же интерфейс будет сочетаться с совершенно другой реализацией **Службы маршрутизации (Routing Service)**.

* * *

Чтобы использовать технический архитектурный шаблон на уровне предметной области, необходимо ввести дополнительную мотивацию, еще один уровень смысла. Если шаблон СТРАТЕГИЯ соответствует реальной стратегии, т.е. процедуре выбора по критериям, принятой в прикладной деятельности, то он уже выходит за рамки обычного полезного приема программирования (хотя и это само по себе неплохо).

Тут в полной мере "включаются" *последствия* применения архитектурного шаблона. Например, в книге [14] подчеркнуто, что клиент должен быть осведомлен о наличии разных СТРАТЕГИЙ, а это уже дополнительная проблема в моделировании. Есть и чисто техническая проблема реализации: наличие СТРАТЕГИЙ может увеличить количество объектов в приложении. Если это важно, то дополнительные затраты можно снизить, реализовав СТРАТЕГИИ как объекты без состояния, которые могут совместно использоваться разными контекстами. Здесь применимы разнообразные методики реализации, подробно рассмотренные в упомянутой книге. Но главное, что мы не отказываемся от использования СТРАТЕГИЙ. Поэтому, хотя мотивы наши слегка изменились и это влияет на выбор решения в той или иной ситуации, в целом, опыт и знания, сосредоточенные в архитектурном шаблоне, остаются в нашем распоряжении.

Композит



Компонуйте объекты в древовидные структуры, представляя иерархии "от части к целому". КОМПОЗИТ (COMPOSITE)[3] позволяет клиентам единообразно воспринимать как отдельные объекты, так и их комбинации [14]

Часто при моделировании сложных предметных областей нам встречаются важные объекты, составленные из частей, которые в свою очередь составлены из частей, а те тоже составлены из частей и т.д. — иногда вложение может иметь произвольную глубину. В некоторых областях знания уровни вложенности принципиально отличаются, но ино-

[3] Материал, структурно состоящий из нескольких фракций (возможно, на нескольких уровнях организации), но работающий как единое целое. — *Примеч. перев.*

гда бывает так, что любую часть целого можно считать примерно тем же, что и целое, только в меньшем масштабе.

Если в модели не отражена родственная связь между вложенными контейнерами, то на каждом уровне иерархии приходится дублировать одни и те же операции, и тогда вложенность является жесткой (например, контейнеры обычно не могут содержать другие контейнеры своего уровня, и количество уровней фиксировано). Клиентам приходится взаимодействовать с разными уровнями иерархии через разные интерфейсы, хотя разницы, принципиальной для клиента, между ними может и не быть. Затруднительной является рекурсия по иерархии для сбора агрегированной информации.

Применяя любой из архитектурных шаблонов на уровне предметной области, прежде всего, стоит задуматься о том, насколько хорошо идея шаблона соответствует понятию из этой области. Допустим, нам удобно рекурсивно передвигаться по ассоциированным объектам, но действительно ли они образуют иерархию "от части к целому"? Удалось ли нам найти абстракцию, в которой все части на самом деле принадлежат к одному концептуальному типу? Если да, то КОМПОЗИТ сделает эти аспекты модели более четкими и ясными, позволяя в то же время проникнуть в глубины тщательно продуманной архитектуры и в технические соображения, лежащие в основе данного шаблона.

Определите абстрактный тип, заключающий в себе все члены КОМПОЗИТА**. Для возвращения агрегированной информации о содержимом контейнеров в них реализуются специальные методы. Узлы-"листья" реализуют эти методы в соответствии со значениями своих собственных атрибутов. Клиенты имеют дело с абстрактным типом и не обязаны отличать контейнеры от "листьев" дерева.**

В структурном отношении это сравнительно очевидный шаблон, но архитекторы, как правило, не особо стараются выделить и оформить в шаблоне операционный уровень. КОМПОЗИТ (COMPOSITE) предлагает одинаковое поведение объектов на каждом структурном уровне. Как к малым, так и к большим частям можно обращаться с запросами, ответы на которые будут отражать их состав. Эта строгая симметрия является ключевой особенностью, придающей шаблону его силу.

Пример

Маршруты доставки, составленные из маршрутов

Полный маршрут доставки груза имеет сложный состав. Прежде всего контейнер с грузом следует доставить автотранспортом на конечную железнодорожную станцию, затем перевезти в порт, затем переправить на судне в другой порт (возможно, с перегрузкой на другие суда) и, наконец, довезти по суше на другой конец маршрута.

Группа разработчиков приложения построила объектную модель. Она описывает всю цепочку произвольной длины, составленную из участков, которые вместе образуют маршрут.

Используя эту модель, разработчики могут создавать объекты **`Маршрут (Route)`** на основании запросов-заказов. У них есть возможность включать **`Участки (Legs)`** в операционный план пошаговой манипуляции грузом. И тут они кое-что обнаружат.

Разработчики всегда представляли себе маршрут в виде цепочки однородных, принципиально не отличающихся участков.

Но оказалось, что специалисты по перевозкам воспринимают маршрут как последовательность из пяти логических сегментов.

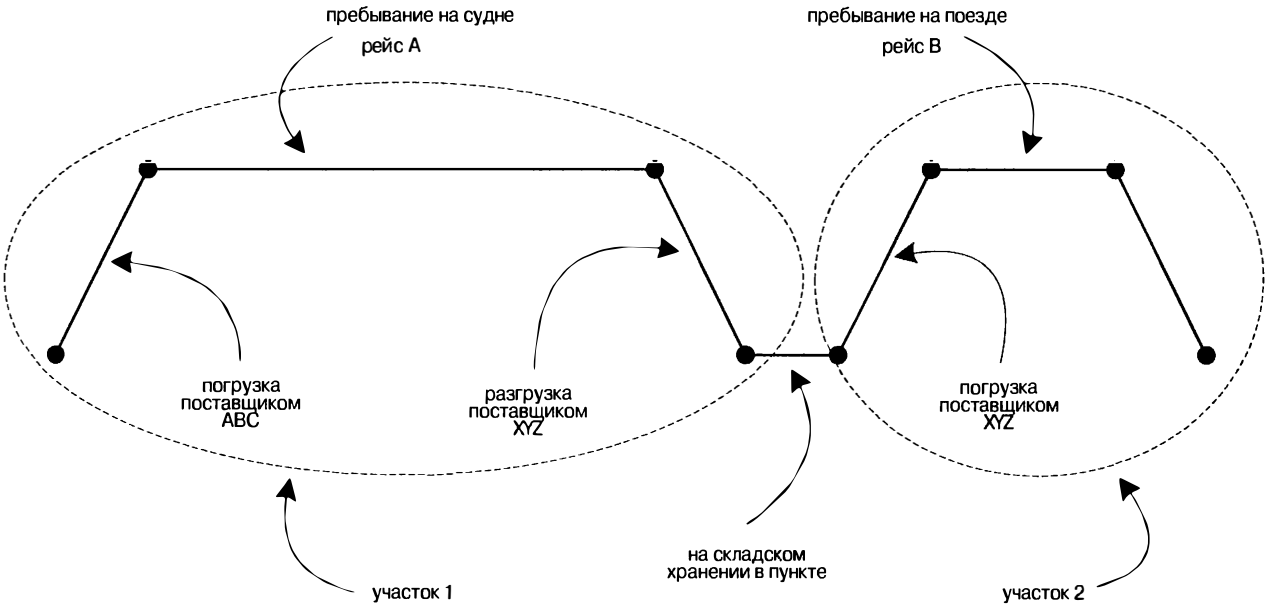


Рис. 12.3. Схема "маршрута", составленного из "участков"

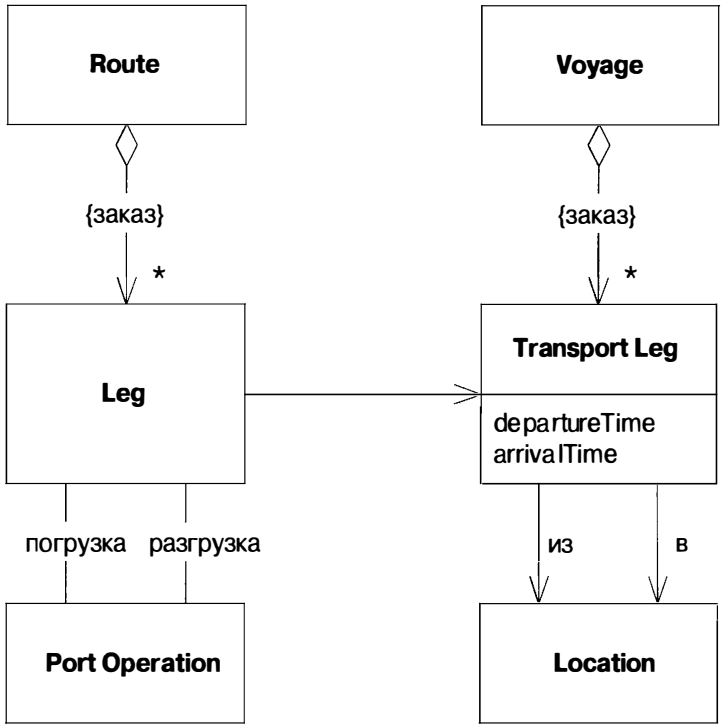


Рис. 12.4. Диаграмма классов ***Маршрута*** *(***Route***), составленного из* ***Участков*** *(***Legs***)*

Рис. 12.5. Концепция маршрута с точки зрения разработчиков

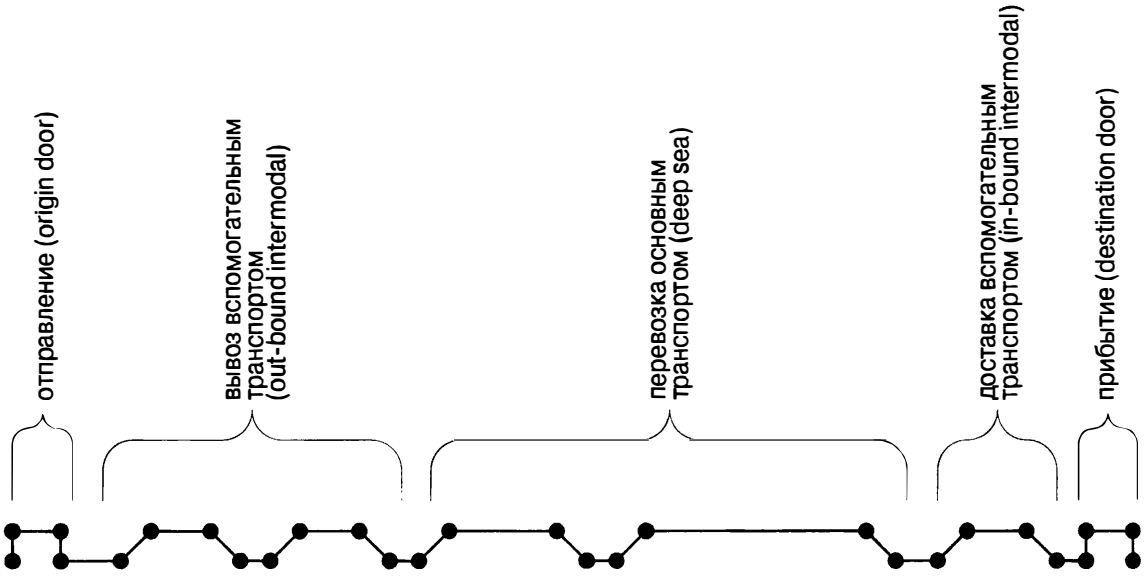


Рис. 12.6. Концепция маршрута с точки зрения специалистов

Кроме всего прочего, эти подмаршруты могут еще и планироваться в разное время разными людьми, так что их следует считать существенно различными. Если же присмотреться еще пристальнее, то этапы "отправления" и "прибытия" совершенно отличаются по характеру от остальных участков пути, поскольку в них используется наем местных перевозчиков или даже вывоз/доставка собственным транспортом клиента. На остальных же участках перевозка подчиняется сложному расписанию железнодорожного и водного транспорта.

Объектная модель, в которой все это учтено, изрядно усложняется.

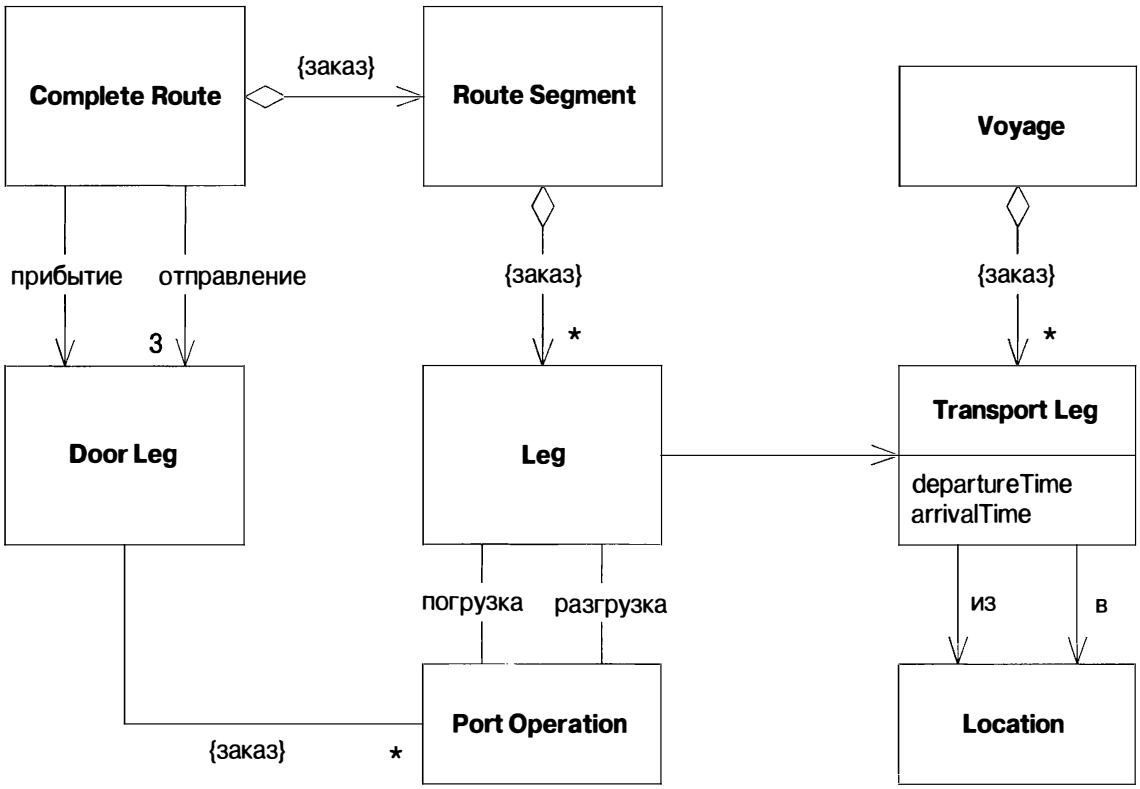


Рис. 12.7. Сложная диаграмма классов для маршрута

В структурном отношении эта модель не так уж плоха, но единообразие выполнения операционного плана уже утрачено, что сильно усложнит код или даже описание операций. Начинают "всплывать" и другие осложнения. Любое отслеживание маршрута требует множества коллекций объектов разного типа.

И тут на сцене появляется КОМПОЗИТ. Некоторым клиентам было бы удобно воспринимать различные уровни в этой системе единообразно — как маршруты, составленные из маршрутов. С концептуальной точки зрения это разумно. Каждый уровень **Маршрута** (**Route**) — это перемещение контейнера из одного пункта в другой, вплоть до уровня отдельных участков пути (рис. 12.8).

Надо сказать, что статическая диаграмма классов не сообщает нам так много о взаимодействии и соотношении между участками отправления, прибытия и т.д., как предыдущая схема. Но модель — это нечто большее, чем статическая диаграмма классов. Информацию о сборке в единое целое мы донесем через другие схемы (рис. 12.9) и через код, который теперь сильно упростится. Данная модель передает глубокую внутреннюю связь между всеми этими разновидностями **Маршрутов** (**Routes**). Составление операционного плана, равно как и другие операции по отслеживанию маршрутов, опять упростилось.

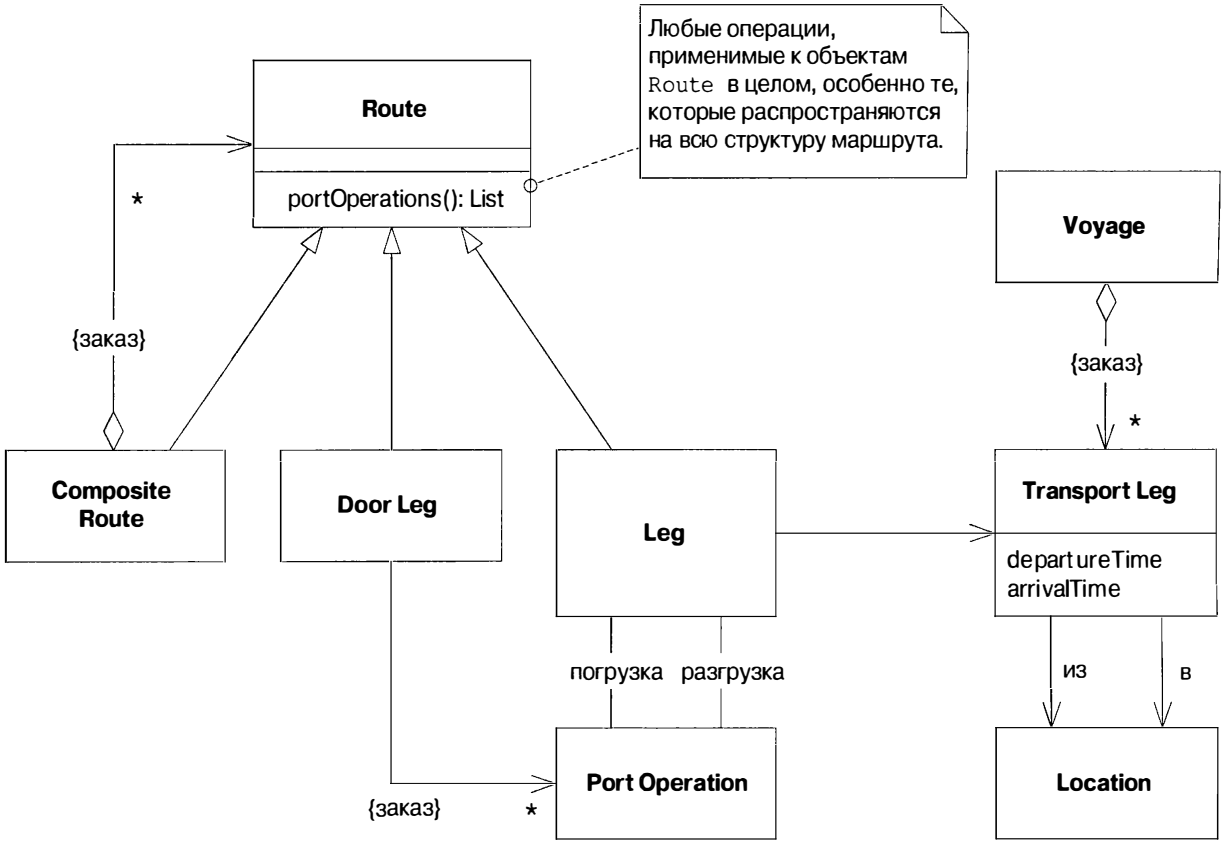


Рис. 12.8. Диаграмма классов с применением КОМПОЗИТА (COMPOSITE)

Если маршруты составляются из других маршрутов, соединяемых друг с другом концами с целью доставки груза из одного пункта в другой, это позволяет достичь в программе любой желаемой степени детализации. Можно отрезать конец маршрута и приставить к нему другой, можно вложить маршруты на какую угодно глубину, можно ввести в модель полезные параметры самого разного свойства.

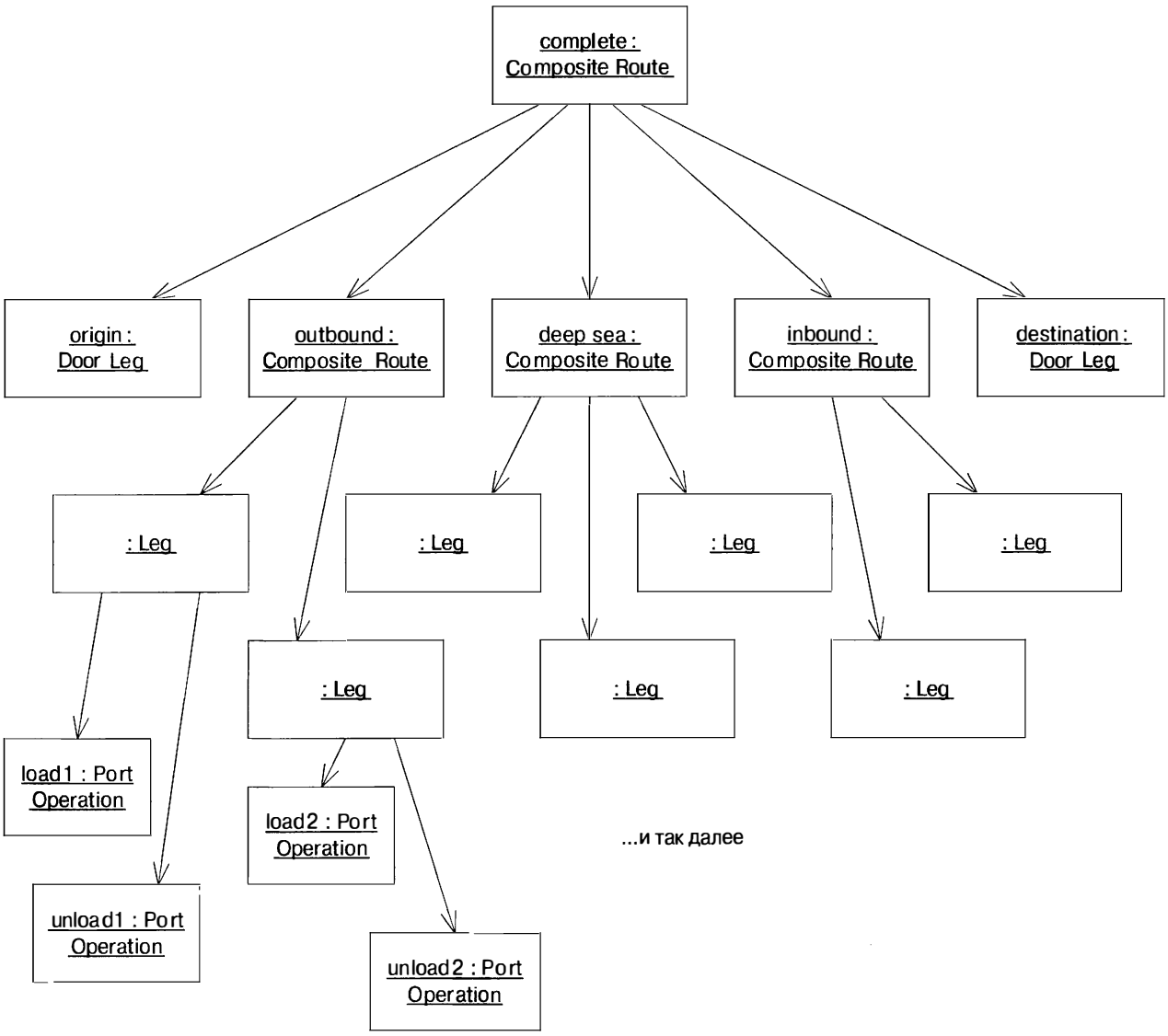


Рис. 12.9. Объекты, представляющие полный ***Маршрут*** *(****Route****)*

Конечно, все эти вещи нам пока не нужны. Прежде чем мы углубились в сегменты маршрутов, участки отправления и прибытия, мы прекрасно обходились и без композита. Архитектурный шаблон вообще следует применять только тогда, когда в нем возникает потребность.

* * *

Почему не “мелкий объект” (flyweight)?

Раньше, в главе 5, уже упоминался шаблон МЕЛКИЙ ОБЪЕКТ (FLYWEIGHT), и может показаться, что и этот шаблон применим к моделям предметных областей. Но на самом деле МЕЛКИЙ ОБЪЕКТ — это образцовый пример архитектурного шаблона, *неприменимого* на уровне предметной области.

Если некий ограниченный набор ОБЪЕКТОВ-ЗНАЧЕНИЙ (VALUE OBJECTS) используется многократно (как в примере с электрическими выводами в проекте дома), то имеет смысл реализовать их в виде МЕЛКИХ ОБЪЕКТОВ. Это вариант *программной реализации*, возможный для ОБЪЕКТОВ-ЗНАЧЕНИЙ (VALUE OBJECTS), но не для СУЩНОСТЕЙ (ENTITIES). Это полная противоположность КОМПОЗИТУ (COMPOSITE), в котором одни концептуальные

объекты составляются из других концептуальных объектов. В последнем случае шаблон относится как к модели, так и к ее реализации, а это и есть характерная особенность шаблона уровня предметной области.

Я не собираюсь здесь составлять список архитектурных шаблонов, которые могут использоваться в качестве шаблонов уровня предметной области. Хотя трудно себе представить, например, использование шаблона INTERPRETER (ИНТЕРПРЕТАТОР) на уровне предметной области, все же не рискну заявить, что такой предметной области совсем не может быть. Единственное требование к шаблону состоит в том, чтобы он сообщал нечто концептуальное о предметной области, а не был бы просто техническим решением технической проблемы.

ГЛАВА 13

Углубляющий рефакторинг

Углубляющий рефакторинг[1] — это многогранный процесс. Чтобы свести воедино некоторые основные моменты, остановимся на минутку и оглянемся на пройденное. Акцент нужно сделать на следующих трех принципах.

1. "Жить" в предметной области.
2. Постоянно смотреть на вещи под разными углами.
3. Поддерживать непрекращающийся диалог со специалистами.

Поиски глубокого понимания предметной области всегда расширяют контекст для рефакторинга.

Согласно классическому сценарию рефакторинга, один-два программиста сидят за клавиатурой, ищут фрагменты кода, которые можно усовершенствовать, и сразу же вносят в них изменения (конечно, проверяя результаты модульными тестами). Такое нужно практиковать постоянно, однако это еще не все.

В предыдущих пяти главах был изложен более широкий взгляд на рефакторинг, который призван накладываться на обычную методику микрорефакторинга.

Инициирование

Углубляющий рефакторинг может начинаться разными способами. Это может быть ответ на проблему в коде — какую-то сложность или неудобство. Вместо того чтобы внести в код стандартные в таких случаях изменения, разработчики считают, что суть проблемы заключается в модели предметной области. Возможно, пропущено важное понятие, а может быть, какая-то взаимосвязь организована неправильно.

Если отойти от обычного понимания рефакторинга, такая догадка может возникнуть даже тогда, когда с кодом, на первый взгляд, все в порядке, но язык модели кажется недостаточно согласованным со специалистами в предметной области или же новые требования к программе не удовлетворяются естественным образом. Рефакторинг может стать и результатом обучения, когда разработчик, выработавший углубленное понимание предмета, видит возможность построить более ясную или полезную модель.

Часто самая трудная и неопределенная задача состоит как раз в том, чтобы увидеть "очаг" проблемы. Когда это сделано, программисты могут целенаправленно подобрать

---

[1] В оригинале автор пишет "refactoring toward deeper insight", что примерно означает "рефакторинг в направлении более глубокого понимания сути". К сожалению, вряд ли можно найти достаточно емкий и краткий русский перевод. Поэтому в книге употребляются выражения "рефакторинг по модели" и "углубляющий рефакторинг", примерно описывающие авторский замысел с разных сторон. — *Примеч. перев.*

элементы новой модели, провести "мозговой штурм" совместно с коллегами и специалистами, извлечь пользу из систематизированного знания в виде аналитических или архитектурных шаблонов.

Исследовательские группы

Каким бы ни был источник неудовлетворенности, следующий шаг — это поиск усовершенствования, которое бы придало модели естественную и четкую коммуникативность. Не исключено, что для этого потребуется самое небольшое изменение, которое лежит на поверхности и вносится в течение нескольких часов. Такие изменения напоминают традиционный рефакторинг. А вот поиск новой модели вполне может отнять больше времени и потребовать труда многих людей.

Инициаторы изменений выбирают пару-тройку программистов, которые владеют адекватным проблеме образом мыслей, знакомы с предметной областью или имеют хороший навык моделирования. Если в деле есть тонкости, обязательно привлекается специалист в предметной области. Группа из четырех-пяти человек идет в конференц-зал или кафе и там ведет "мозговой штурм" длительностью от получаса до полутора часов. Они рисуют UML-диаграммы, пытаются пройти рабочие сценарии с применением объектов. Они добиваются того, чтобы специалист понял модели и счел их полезными. Когда наконец в процессе возникает нечто удовлетворительное, они возвращаются и пишут код. Или же они решают подождать и подумать над материалом в течение нескольких дней, возвращаются на рабочие места и берутся за другие дела. Несколько дней спустя группа собирается снова с той же целью. В этот раз они уже более уверены в себе, "переварили" свои идеи и пришли к кое-каким заключениям. Они возвращаются к компьютерам и пишут код для новой архитектуры.

Есть несколько ключевых принципов, позволяющих сделать такую работу плодотворной.

- *Самоорганизация.* Для исследования проблемы в архитектуре программы можно на ходу собрать совсем небольшую группу, которая поработает несколько дней и самораспустится. Нет необходимости создавать долговременные и сложные организационные структуры.
- *Время и масштаб*. Более-менее осмысленную архитектуру можно создать за два-три совещания, проведенные в течение нескольких дней. Затягивать дело не имеет смысла. Если вы застряли, может быть, вы взваливаете на себя слишком много. Возьмитесь за менее масштабный аспект архитектуры и сосредоточьтесь на нем.
- *Единый язык.* Привлечение других членов группы — в частности, специалистов по предмету — к "мозговому штурму" создает возможность усовершенствоваться в ЕДИНОМ ЯЗЫКЕ (UBIQUITOUS LANGUAGE). Конечным результатом этих усилий становится усовершенствование самого этого ЯЗЫКА, который программисты берут на вооружение и формализуют в коде.

В предыдущих главах книги было приведено несколько диалогов, в которых программисты со специалистами пытались найти более удачные модели. Полноценный сеанс "мозгового штурма" динамичен, не имеет четкой структуры и весьма продуктивен.

Предыдущие наработки

Не всегда приходится заново изобретать велосипед. В процессе "мозгового штурма" в поисках пропущенного понятия или лучшей модели потенциально возможно заимствование идей из любых источников, которые затем комбинируются с имеющимися знаниями и "перемалываются" до тех пор, пока не будут найдены ответы на текущие вопросы.

Можно брать идеи прямо из книг и других источников знаний непосредственно по предметной области. Специалисты в этой области, может быть, и не создали ни одной модели, пригодной для разработки программы, но, тем не менее, они организовали понятия их системы и нашли ряд полезных абстракций. Если пустить процесс переработки знаний по этому пути, можно быстро получить качественный результат, который к тому же покажется знакомым специалисту.

Иногда можно почерпнуть пользу из опыта предшественников, выраженного в форме аналитического шаблона. Получение такой информации в некоторой степени родственно чтению литературы о предметной области, но в данном случае акценты смещены именно в сторону разработки программ, и знания основаны непосредственно на опыте внедрения программного обеспечения. В аналитических шаблонах могут содержаться тонкие понятия модели, и они помогут вам избежать многих ошибок. Но это не готовые рецепты "а-ля поваренная книга", а "сырье" для процесса переработки знаний.

По мере сборки частей в единое целое приходится параллельно иметь дело с проблематикой модели и программной архитектуры. И в этом случае не всегда приходится изобретать все заново. На уровне предметной области часто применимы архитектурные шаблоны, которые соответствуют как потребностям программной реализации, так и идеям модели.

Аналогично, если в некоторой части предметной области уместно применение общепринятого формализма, — например, арифметики или логики предикатов — то эту часть всегда можно выделить в отдельную и адаптировать для нее правила формальной системы. Отсюда возникают очень строгие и наглядные модели.

Архитектура для разработчиков

Программное обеспечение существует не только для пользователей, но также и для разработчиков. Разработчикам приходится интегрировать свой код с другими частями системы. Они вносят в него изменения снова и снова в итерационном процессе. Углубляющий рефакторинг одновременно и приводит программу к гибкой архитектуре, и облегчается именно такой архитектурой.

Гибкая архитектура (*supple design*) сама передает заложенные в нее цели. Благодаря ей легко предсказать, что произойдет в результате выполнения кода, а следовательно, и что произойдет в результате изменений кода. Гибкая архитектура помогает избежать умственного перенапряжения, в основном за счет снижения количества взаимосвязей и побочных эффектов. Она основывается на углубленной модели предметной области, которая доведена до самого мелкого масштаба в местах, особенно критических для пользователя. Это позволяет добиться гибкости там, где изменения наиболее часты, и простоты в остальных местах.

Расчет времени

Если вы ждете момента, когда изменение в программе будет безупречно обоснованно, то вы ждете уже слишком долго. Ваш проект уже влечет за собой материальные потери, а откладываемые изменения будет внести еще тяжелее, поскольку код становится все сложнее, все теснее привязывается к другому коду.

Постоянный рефакторинг уже признан наиболее передовым подходом, *best practice*, к разработке программ, но большинство групп разработчиков все еще относится к нему настороженно. Они осознают риск изменений в коде и стоимость времени программистов, требуемого на эти изменения. Но труднее осознать риск, заключающийся в сохранении неудобной архитектуры, и стоимость усилий по обходу ее неудобств. От разработчиков, которые хо-

тят выполнить рефакторинг, часто требуют обосновать их желание. Это кажется разумным, но на самом деле такое требование делает и без того трудное дело практически невозможным, на корню пресекая рефакторинг или загоняя его в подполье. Разработка программного обеспечения не является настолько предсказуемым делом, чтобы преимущества внесения изменений или стоимость отсутствия этих изменений можно было точно подсчитать.

Углубляющий рефакторинг должен стать органичной частью непрерывных исследований проблематики предметной области, повышения квалификации программистов и совместной работы их со специалистами. Поэтому смело идите на рефакторинг, если:

- архитектура не отражает текущих представлений группы о предметной области;
- важные понятия присутствуют в архитектуре в неявном виде (а вы видите способ сделать их явными);
- есть возможность сделать важную часть программной архитектуры более гибкой в работе.

Впрочем, эта энергичная позиция не может оправдать совсем уж любое изменение в любое время. Не занимайтесь рефакторингом за день до выпуска или сдачи программного продукта. Не внедряйте "гибких архитектур", которые демонстрируют техническую виртуозность, но не передают самой сути предметной области. Не вводите "углубленную модель", в необходимости которой вы не можете убедить специалиста, пусть даже вам она кажется очень красивой. Не возводите ничего в абсолют, но всегда жертвуйте комфортом в пользу рефакторинга.

Кризис как потенциальная возможность

Более столетия после Чарлза Дарвина стандартная модель эволюции заключалась в том, что виды изменялись постепенно, иногда устойчиво, с течением времени. Но внезапно в 1970-е годы эту модель вытеснила модель точечного, или кусочного, равновесия (*punctuated equilibrium*). В новом взгляде на эволюцию длительные периоды постепенных изменений или постоянства сменялись сравнительно кратковременными всплесками быстрых изменений. Затем все приходило в новое равновесие. В разработке программного обеспечения есть сознательная направленность, которой не хватает эволюции (хотя по некоторым проектам этого не скажешь), но и она следует такому же ритму.

Классические определения рефакторинга так и дышат стабильностью. Углубляющий рефакторинг обычно не обладает этой особенностью. Период постепенного усовершенствования модели может внезапно привести вас к таким выводам, которые "потрясут" все основы. Эти прорывы и скачки случаются не каждый день, но из них проистекает значительная часть тех изменений, которые приводят к углубленной модели и гибкой архитектуре.

Подобные ситуации часто кажутся не потенциальными возможностями, а кризисами. Раз — и модель оказывается очевидно неадекватной. В круге выражаемых ею понятий зияет дыра, в какой-то критической области она совершенно непонятна, а некоторые ее утверждения могут быть откровенно ложными.

Это означает, что группа разработчиков достигла нового уровня понимания. С новой высоты прежняя модель выглядит убогой, но зато видна перспектива построить гораздо лучшую.

Углубляющий рефакторинг — процесс бесконечный. Обнаруживаются и выводятся на явный уровень неявные концепции. Тем или иным элементам программной архитектуры придается гибкость, временами основанная на декларативном стиле. Разработка внезапно приходит на грань прорыва и порождает углубленную модель, а затем вновь начинается медленное, постепенное усовершенствование.

IV

Стратегическое проектирование

Когда сложность системы возрастает до того, что ее уже невозможно знать на уровне отдельных объектов, появляется потребность в специальных приемах для восприятия больших моделей и манипулирования ими. В этой части книги представлены принципы, позволяющие распространять процесс моделирования на очень сложные и крупномасштабные предметные области. Большинство решений такого рода должно приниматься на уровне группы разработки или даже в переговорах между группами. В этих решениях часто пересекаются как архитектурные, так и политические соображения.

В идеале корпоративная программная система, если она задумана с размахом, стремится превратиться в тесно интегрированную систему, охватывающую всю деятельность в данной отрасли знания. Но модель сразу всей предметной области, скорее всего, окажется слишком большой и сложной как в управлении, так и в понимании единого целого для практически любой организации. Систему следует разбивать на меньшие части как с концептуальной точки зрения, так и для программной реализации. Сложность состоит в том, чтобы добиться такой модульности, *не потеряв преимуществ интеграции* — отдельные части системы должны работать совместно, способствуя координированию различных прикладных операций. Монолитная и всеохватывающая модель предметной области была бы слишком громоздкой, местами дублирующейся и полной неочевидных противоречий. Но набор небольших отдельных подсистем, соединенных через неединообразные интерфейсы, "не потянет" решение задач в корпоративном масштабе. Кроме того, в каждой точке сопряжения будут постоянно возникать проблемы совместимости и единообразия. Но ловушек обеих крайностей можно избежать, если планировать архитектуру систематически и постепенно.

Даже и в этом масштабе предметно-ориентированное проектирование программ (*domain-driven design*) не порождает модели, совершенно оторванные от реализации. Каждое проектное решение должно иметь прямое влияние на разработку системы, иначе непонятно, зачем его вообще принимать. Конкретные архитектурные решения должны основываться на принципиальных стратегических решениях, чтобы уменьшить взаимозависимость отдельных частей и улучшить наглядность работы системы, не теряя синергизма и критически важного взаимодействия. Все это должно продвигать модель к отражению концептуального ядра системы, ее "видению". *И все это нужно делать, не тормозя проект.* Тому, как достичь этих целей, и посвящена часть IV этой книги, где рассматриваются три обширные темы: контекстность, дистилляция, крупномасштабная структура.

Контекстность представляет собой наименее очевидный из трех принципов, но одновременно наиболее фундаментальный. Удачная модель, будь она большая или маленькая, должна быть логически непротиворечивой и единообразной, лишенной противоречий и перекрывающихся определений. Корпоративные системы часто интегрируют подсистемы из различных источников или содержат приложения столь различные, что мало какие элементы предметной области воспринимаются ими в одном свете. Может быть, требование унифицировать модель во всех, даже категорически несоизмеримых, частях системы чрезмерно и невыполнимо. Но моделировщик все же имеет возможность избежать дальнейшего вырождения модели. Для этого нужно явно определить ОГРАНИЧЕННЫЙ КОНТЕКСТ (BOUNDED CONTEXT), внутри которого применима модель, а затем при необходимости определить его взаимосвязи с другими контекстами.

Дистилляция позволяет сосредоточить внимание на том, что нужно. Часто много усилий тратится на второстепенные, периферийные вопросы. Но всеобъемлющая модель целой предметной области должна подчеркнуть наиболее значительные, особые аспекты системы, и при этом иметь такую структуру, чтобы придать им максимальную силу. Некоторые вспомогательные компоненты, конечно, бывают критически важны, но их следует ставить на положенное им место. Подобная ориентация моделирования не только позволяет направить усилия на самые важные части системы, но и предотвращает поте-

рю видения всей системы. Стратегическая дистилляция может сделать большую модель понятнее и нагляднее. А это позволяет спроектировать архитектуру СМЫСЛОВОГО ЯДРА (CORE DOMAIN) системы с наибольшей пользой для дела.

Картину завершает крупномасштабная структура. В очень сложной модели легко потерять лес за деревьями. Проблему частично решает дистилляция, помогая сосредоточиться на главной, центральной части и представить остальные элементы во вспомогательных ролях. Но взаимосвязи между ними все равно могут остаться слишком сложными, если отсутствует "главная тема", если не применяются архитектурные формы и шаблоны масштаба целой системы. В нашей книге будет дан обзор нескольких подходов к построению крупномасштабной структуры, а затем мы углубимся в детали одного из соответствующих шаблонов, УРОВНИ РАЗДЕЛЕНИЯ ОБЯЗАННОСТЕЙ (RESPONSIBILITY LAYERS), для изучения последствий применения такой структуры. Будут рассматриваться только частные примеры; исчерпывающего каталога читатель здесь не найдет. Придумывать новые структуры или развивать существующие следует по мере необходимости, эволюционно. Какая-то из этих структур окажется способной привнести в архитектуру такое единообразие, которое ускорит разработку и улучшит интеграцию.

Три перечисленных принципа, полезные даже по отдельности, но особенно мощные при совместном использовании, помогают строить хорошие архитектуры даже в достаточно хаотической системе, которую полностью никто не понимает. Крупномасштабная структура придает согласованность трудносовместимым фрагментам и помогает им стать одним целым. Структуризация и дистилляция позволяют воспринимать и понимать сложные взаимосвязи между отдельными частями, при этом удерживая в поле зрения всю картину. ОГРАНИЧЕННЫЕ КОНТЕКСТЫ (BOUNDED CONTEXTS) дают возможность выполнять работу одновременно в разных частях системы без повреждения модели или ее нечаянного фрагментирования. Добавление всех этих понятий в ЕДИНЫЙ ЯЗЫК (UBIQUITOUS LANGUAGE) группы разработчиков поможет им выработать собственные принципиальные решения.

Глава 14

Поддержание целостности модели

Как-то я работал в проекте, в котором новую большую систему параллельно разрабатывали несколько рабочих групп. В один прекрасный день группа, работавшая над модулем для выписывания клиентам счетов-фактур, собралась реализовать объект, который они назвали **Charge (Платеж)**. И тут оказалось, что другая группа уже сконструировала такой объект. Первая группа решила прилежно перенести его к себе. Оказалось, что в нем нет "кода назначения платежа" (*expense code*), и его добавили. Там уже был нужный им атрибут "проводимая сумма" (*posted amount*), и хотя они планировали назвать его "сумма к оплате" (*amount due*), но изменили решение — в конце концов, так ли важно конкретное название? Добавив еще несколько методов и ассоциаций, разработчики получили нечто, показавшееся им приемлемым, и не стали разбираться подробнее, что там внутри. Им пришлось проигнорировать ряд ненужных им ассоциаций, и их модуль все-таки заработал.

Через несколько дней возникли загадочные проблемы в приложении-модуле оплаты счетов, для которого изначально был написан объект **Charge (Платеж)**. Всплыли странные **Платежи**, которых никто не вводил и которые не имели никакого смысла. Программа начала аварийно завершаться при обращении к некоторым функциям — в частности, выдаче налогового отчета по текущему месяцу. Выяснилось, что аварийное завершение происходило при вызове функции, которая вычисляла общую сумму необлагаемых налогом вычетов из всех платежей текущего месяца. Загадочные записи не содержали никаких значений в поле "необлагаемый процент", хотя система проверки в приложении ввода данных не только требовала его наличия, но и сама вставляла туда значение по умолчанию.

Проблема состояла в том, что две группы имели две *разные модели*, но не понимали этого. При этом в проекте отсутствовали какие-либо процедуры для выявления этого факта. Каждая из групп делала предположения о природе платежа, полезные в их собственных контекстах (выставление счетов клиентам и выплаты поставщикам соответственно). Но когда написанный ими код сводился воедино без разрешения имеющихся противоречий, итоговая программа получалась ненадежной.

Если бы разработчики осознали, как фактически обстоят дела в проекте, они могли бы принять сознательное решение по исправлению ситуации. Возможно, для этого пришлось бы вместе выработать общую модель, а затем написать набор автоматизированных тестов, чтобы избежать будущих сюрпризов. А может быть, разработчики договорились бы разработать отдельные модели и не вмешиваться в код друг друга. В любом случае такая работа начинается с согласования границ, внутри которых применяется та или иная модель.

Что же они сделали, узнав о существовании проблемы? Они разработали отдельные классы **Платеж клиента (Customer Charge)** и **Платеж поставщику (Supplier Charge)**, определив каждый в соответствии с целями и задачами соответствующей

группы. Решив непосредственную проблему, они вернулись к тому режиму работы, который практиковали и раньше.

Хотя мы редко задумываемся об этом, самые фундаментальные требования к модели состоят в том, чтобы она была самосогласованной, внутренне непротиворечивой; чтобы ее термины всегда имели одно и то же значение; и чтобы в модели не было противоречивых правил. Самосогласованность модели, при которой каждый термин всегда имеет однозначный смысл и ни одно правило не противоречит другому, называется *унификацией*. Если логической самосогласованности нет, модель как таковая не имеет смысла. В идеальном мире у нас была бы одна модель, охватывающая всю предметную область, в которой работает корпоративное приложение. Эта модель была бы унифицированной, не имела бы внутренних противоречий или пересекающихся определений терминов. Любое логическое утверждение о предметной области было бы непротиворечивым.

Но мир больших систем отнюдь не является идеальным. Поддержание такого уровня унификации больше отнимает усилий, чем даст полезного эффекта. Необходимо разрешить разработку разных моделей для разных частей системы, но следует правильно и тщательно выбрать, какие части системы разделить и какие между ними установить отношения. Нужно найти способы поддерживать строгую унификацию между критически важными частями модели. Все это не возникает само собой, от одних только благих намерений. Результат дает только сознательное принятие проектных решений и организацию некоторых специальных процессов и процедур. **Полная унификация модели предметной области для большой системы либо невозможна, либо неоправданно затратна**.

Иногда люди пытаются бороться с этой реальностью. Большинство видит ту цену, которую приходится платить из-за ограниченной интегрированности и неудобств коммуникации между несколькими моделями. Кроме того, наличие нескольких моделей субъективно кажется некрасивым. Это сопротивление использованию нескольких моделей иногда вызывает к жизни амбициозные попытки унифицировать все программное обеспечение в рамках одного проекта под эгидой единой модели. Я знаю этот грех и за собой. Но рассмотрим же, чем это чревато.

1. Слишком много уже имеющегося кода придется одновременно заменять на новый.
2. Большие проекты могут затормозиться из-за того, что дополнительная нагрузка по управлению ими превзойдет имеющиеся возможности.
3. Приложениям со специализированными требованиями могут быть навязаны такие модели, которые не полностью соответствуют их задачам, поэтому реализацию важных операций придется выносить куда-то в другое место.
4. И напротив, попытка удовлетворить всех одной моделью добавит в нее столько вариантов и параметров, что ею станет трудно пользоваться.

Более того, расхождения в моделях могут вызываться не только техническими соображениями, но и политическим размежеванием или разногласиями в управленческих предпочтениях. Возникновение разных моделей может быть и результатом организации группы и процесса разработки. Поэтому даже если никакие технические факторы не мешают полной интеграции, в проекте все равно может появиться набор разных моделей.

Уже зная, что невозможно поддерживать унифицированную модель для всего проектного предприятия, тем не менее, не следует отдаваться воле случая. Четкую и единую картину происходящего вполне можно построить, если принять ряд превентив-

ных решений по частичной унификации и прагматично рассмотреть вопрос о том, что унифицировать не следует. Сделав это, дальше можно озаботиться тем, чтобы унифицируемые части системы такими и остались, тогда как не унифицируемые не вызвали бы путаницы и не нанесли вреда.

Нам нужен способ обозначить границы и взаимоотношения между разными моделями. Необходимо сознательно выбрать стратегию и последовательно придерживаться ее.

В этой главе предлагаются приемы распознавания, описания и выбора границ модели вместе с ее взаимосвязями с другими моделями. Все начинается с построения карты той территории, которая отведена проекту. Диапазон применимости каждой модели задает ОГРАНИЧЕННЫЙ КОНТЕКСТ (BOUNDED CONTEXT), тогда как общий вид всех контекстов проекта и отношений между ними предоставляет КАРТА КОНТЕКСТОВ (CONTEXT MAP). Такое снижение многозначности и двусмысленности само по себе изменяет ход проекта, но оно не всегда бывает обязательным. Как только КОНТЕКСТ ОГРАНИЧЕН, унификацию модели обеспечит процесс НЕПРЕРЫВНОЙ ИНТЕГРАЦИИ (CONTINUOUS INTEGRATION).

Затем, отталкиваясь от этого стабильного положения, мы можем начать двигаться в сторону более эффективных стратегий для ОГРАНИЧЕННЫХ КОНТЕКСТОВ и построения отношений между ними — от тесно связанных друг с другом контекстов с ОБЩИМИ ЯДРАМИ (SHARED KERNELS) до весьма косвенно соотносящихся моделей, для которых характерно ОТДЕЛЬНОЕ СУЩЕСТВОВАНИЕ (SEPARATE WAYS).

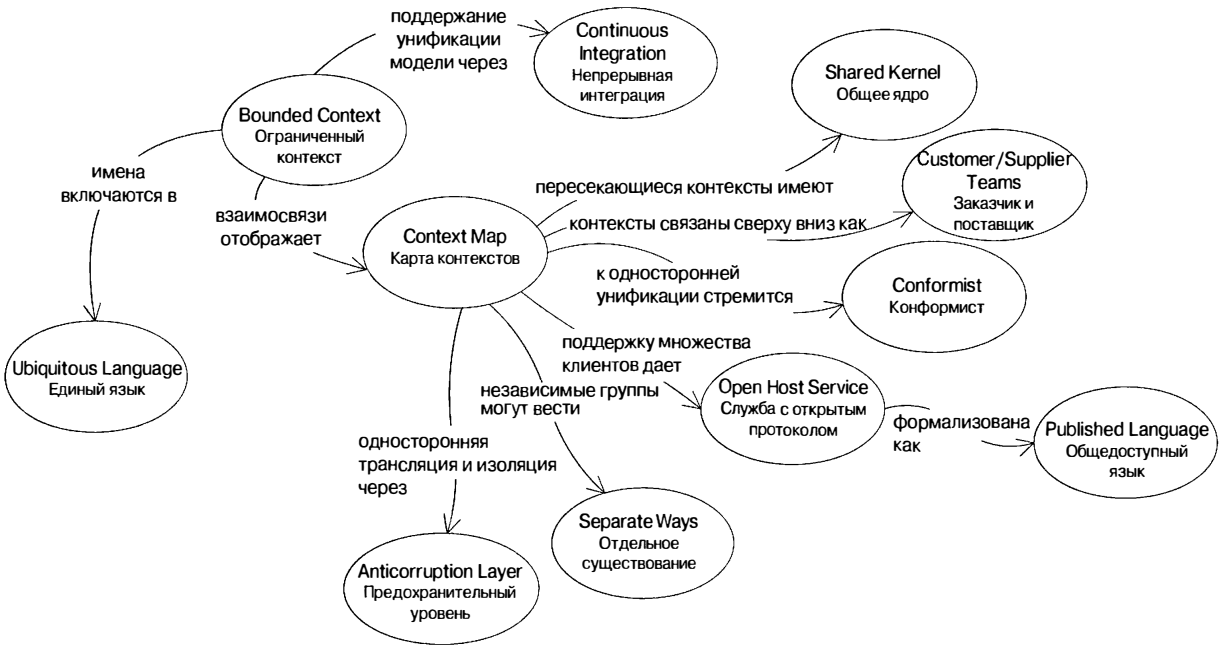
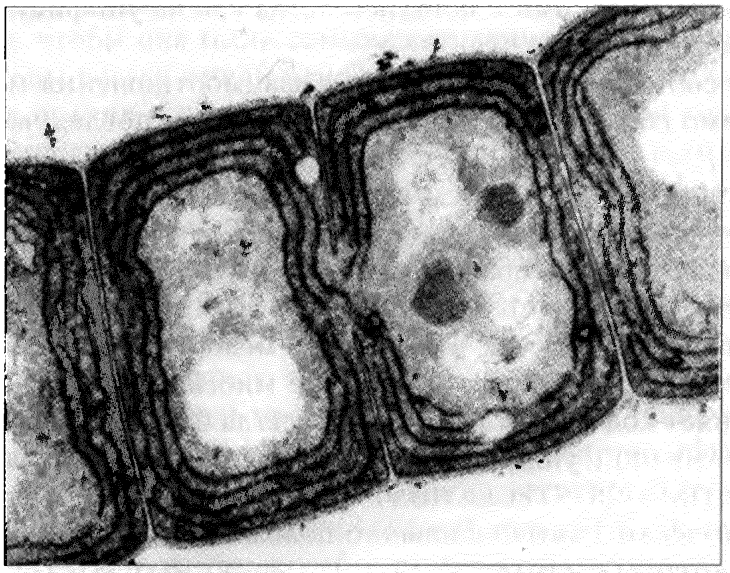


Рис. 14.1. Карта-схема шаблонов для поддержания целостности модели

Ограниченный контекст



Клетки существуют потому, что их стенки-мембраны определяют, что должно находиться внутри, что — снаружи, а что может через них проходить

В больших проектах сосуществует сразу много моделей, и во многих случаях это вполне нормально. Разные модели применимы в разных контекстах. Например, вашу новую программу может понадобиться интегрировать с внешней системой, над которой ваша группа не имеет контроля. В такой ситуации наличие разных контекстов, в одном из которых разрабатываемая модель непригодна, не вызывает сомнений. Но другие ситуации могут оказаться не столь очевидными. В истории, с которой начинается эта глава, две группы разработчиков занимались разными функциями одной и той же новой системы. Работали ли они по одной модели? В их намерения входило совместное использование по крайней мере некоторых своих наработок, но нигде не была проведена граница между тем, чем они собирались и чем не собирались делиться с коллегами. У них также не было готовой процедуры совместного пользования моделью или оперативного выявления расхождений. Они поняли, что разошлись в разные стороны, только тогда, когда поведение системы внезапно стало непредсказуемым.

Даже одна и та же группа может придти к необходимости нескольких моделей. В группе может нарушиться коммуникация, и из-за этого возникнут слегка противоречивые интерпретации модели. В более старом коде обычно отражены и более ранние понятия модели, еще отличающиеся от текущих.

Всем знаком случай, когда формат данных в другой системе отличается и требует их преобразования. Но это только механический аспект проблемы. Более фундаментальное различие кроется в моделях, на которых основаны две разные системы. Когда расхождение возникает не с внешней системой, а внутри одной и той же базы кода, распознать его бывает еще труднее. А это случается в *любом* крупном проекте.

В любом крупном проекте возникает необходимость работать с несколькими моделями. Но при комбинировании в одно целое кода, разработанного на основе сильно отличающихся моделей, программа становится ненадежной, трудной в понимании, склонной к сбоям. Запутывается коммуникация между членами группы. Часто бывает непонятно, в каком контексте модель *не должна* применяться.

Неспособность организовать все просто и надежно в конце концов обнаруживается, когда получившийся код не хочет работать правильно. Однако проблема коренится еще и в способах организации рабочих групп, взаимодействии между их членами. Поэтому чтобы прояснить контекст модели, нам нужно рассмотреть одновременно и проект, и его конечный результат (код, структуры баз данных и т.д.).

Модель применяется в определенном *контексте.* Контекстом может быть определенная часть кода или же работа конкретной рабочей группы. Для модели, которая родилась в ходе "мозгового штурма", контекст можно ограничить содержанием конкретной беседы с группой. Контекст любой модели, использованной в примере этой книги, — это глава книги и последующие ее разделы, где эта модель фигурирует снова. Вообще, контекст модели — это некий набор условий, которые должны выполняться, чтобы можно было утверждать, что термины модели имеют конкретный четкий смысл.

Чтобы приступить к решению проблемы множественных моделей, нам необходимо явно определить область действия конкретной модели как замкнутую и ограниченную часть программной системы, внутри которой применяется и может максимально унифицироваться одиночная модель. Это определение следует согласовать с группой разработки.

Явно определите контекст, в котором применима модель. Явным образом установите границы в соответствии с организационной структурой группы, особенностями операций в разных частях приложения, организацией баз кода и данных. Строго следите за самосогласованностью модели в установленных границах, не отвлекайтесь на внешние по отношению к ним проблемы.

Ограниченные контексты — это не модули

Эти два понятия часто путают, но на самом деле это два разных шаблона с разной мотивацией их применения. Действительно, когда два разных набора объектов явно образуют разные модели, их почти всегда помещают в разные МОДУЛИ. Это позволяет организовать разные пространства имен (что существенно для разных КОНТЕКСТОВ) и создает некоторое разграничение.

Но МОДУЛИ также используются и для организации элементов одной модели; они не обязательно выражают намерение разграничить КОНТЕКСТЫ. Отдельные пространства имен, создаваемые МОДУЛЯМИ внутри ОГРАНИЧЕННЫХ КОНТЕКСТОВ, фактически препятствуют обнаружению случайной фрагментации модели.

ОГРАНИЧЕННЫЙ КОНТЕКСТ (BOUNDED CONTEXT) ставит пределы применимости той или иной модели, чтобы разработчики четко понимали (и разделяли это понимание между собой), в чем следует поддерживать единообразие и согласованность, и как соотносить это с другими КОНТЕКСТАМИ. Внутри данного КОНТЕКСТА старайтесь поддерживать модель в логически унифицированном виде, но не беспокойтесь о ее применимости вне установленных границ. В других контекстах применимы другие модели, с другой терминологией, понятиями и правилами, а также с другими диалектами ЕДИНОГО ЯЗЫКА. Проведя четкую границу, вы сможете сохранить чистоту модели и, следовательно, ее мощь там, где она применима. Одновременно вы сможете избежать путаницы при переносе внимания на другие КОНТЕКСТЫ. Интеграция через границы неизбежно потребует какой-то разновидности трансляции, и этот вопрос можно будет проанализировать.

Пример

Контекст заказа доставки грузов

Компания по доставке грузов открыла свой внутренний проект по разработке нового приложения для заказа доставки. Это приложение должно строиться на основе объектной модели. Каков ОГРАНИЧЕННЫЙ КОНТЕКСТ (BOUNDED CONTEXT), внутри которого эта модель будет применяться? Для ответа на этот вопрос надо посмотреть, что же происходит в проекте. Помните, что это взгляд на проект *как он есть*, а не каким он должен быть в идеале.

Одна из групп проекта работает над самим приложением заказа грузов. Эта группа не должна модифицировать объекты модели, но приложение, которое они пишут, должно отображать эти объекты и манипулировать ими. Данная группа разработчиков представляет собой потребителя модели. Модель является действующей внутри этого приложения (основного потребителя). Ее границы задает именно это приложение для заказа доставки грузов.

Заполненные заказы передаются в ранее написанную систему отслеживания перевозки грузов. Заранее, еще до разработки, было решено, что новая модель будет отличаться от старой, поэтому старая система находится вне ее границ. Необходимая трансляция (перевод понятий) между новой моделью и старой системой возлагается на группу сопровождения старой системы. Механизм трансляции не управляется моделью и не проектируется по ней. Он не находится в данном ОГРАНИЧЕННОМ КОНТЕКСТЕ. (На самом деле он является частью границы, и об этом будет сказано при обсуждении КАРТЫ КОНТЕКСТОВ, CONTEXT MAP.) Это хорошо, что трансляция находится вне КОНТЕКСТА (не основана на модели). Было бы непрактично просить группу сопровождения старой системы каким-либо образом пользоваться моделью, поскольку основные обязанности разработчиков лежат вне ее КОНТЕКСТА.

Группа, отвечающая за модель, ведает всем циклом существования каждого объекта, включая и постоянное хранение/восстановление. Поскольку именно на членов этой группы возложено управление структурой базы данных, они специально сделали объектно-реляционное отображение максимально удобным для своих целей и поддерживают его в таком виде. Другими словами, структура базы данных определяется моделью и поэтому находится в ее смысловых границах.

Еще одна группа работает над моделью и приложением для составления расписания рейсов грузовых судов. Группа заказов и группа расписания были организованы одновременно, и обе группы намеревались построить одну унифицированную систему. Две группы в какой-то мере координировали свои усилия и время от времени "делились объектами", но все это делалось несистематически. Они *не работали* в пределах одного ОГРАНИЧЕННОГО КОНТЕКСТА. В этом есть риск, поскольку сами они не считают, что работают по разным моделям. В ходе интеграции их разработок обязательно возникнут проблемы, если только они не разработают специальные процедуры для решения этой проблемы. (Хорошим вариантом может оказаться ОБЩЕЕ ЯДРО, SHARED KERNEL, которое рассматривается дальше в этой главе.) Но первым шагом должно стать принятие ситуации *как она есть*. Группы не находятся в одном КОНТЕКСТЕ и должны избегать делиться кодом, пока ситуация не изменится.

Данный ОГРАНИЧЕННЫЙ КОНТЕКСТ состоит из всех аспектов системы, которые управляются этой моделью: объекты модели, структура базы данных, в которой эти объекты сохраняются, а также само приложение заказа грузов. В этом КОНТЕКСТЕ в основном работают две группы: группа моделирования и группа разработки приложения. Со старой системой отслеживания грузов необходимо обмениваться информацией, и на группу сопровождения этой системы возложена обязанность обеспечивать трансляцию на грани-

це. Группа моделирования же обязана оказывать им помощь. Между моделью заказа доставки и моделью расписания рейсов нет четко определенной взаимосвязи, и определение этой связи должно стать одной из первых задач обеих групп. А тем временем им нужно крайне осторожно делиться кодом или данными.

Какую же пользу принесло определение данного ОГРАНИЧЕННОГО КОНТЕКСТА? Группам, работающих в этом КОНТЕКСТЕ, оно принесло ясность. Две группы теперь понимают, что должны придерживаться одной модели. Они принимают проектные решения с учетом этого и тщательно следят за возможными расхождениями. Внешним группам определение КОНТЕКСТА принесло свободу. Они не должны теперь чувствовать себя в подвешенном состоянии, не работая по той же модели, но чувствуя себя обязанными это делать. Но наиболее ощутимое достижение в данном случае — это, пожалуй, осознание риска неформального обмена информацией между группой моделирования заказов и группой моделирования расписания. Чтобы избежать проблем, группам придется взвесить преимущества и недостатки такого обмена и разработать процедуры, которые бы сделали его действенным. Этого не будет, пока все не поймут, где находятся границы контекстов модели.

* * *

Сами по себе границы являются особыми местами. Взаимоотношения между ОГРАНИЧЕННЫМ КОНТЕКСТОМ (BOUNDED CONTEXT) и его соседями требуют внимания и осторожности. Расчертить территорию и построить крупномасштабную картину КОНТЕКСТОВ и связей между ними позволяет КАРТА КОНТЕКСТОВ (CONTEXT MAP), а еще несколько шаблонов определяют характер различных взаимоотношений между КОНТЕКСТАМИ. Единство же модели внутри ограниченного контекста поддерживается при помощи процесса НЕПРЕРЫВНОЙ ИНТЕГРАЦИИ (CONTINUOUS INTEGRATION).

Но прежде чем заняться всем этим, мы хотим знать: что бывает, когда нарушается унификация модели? Как распознать концептуальные дефекты смысла в ней?

Распознавание дефектов внутри ограниченного контекста

Временно не обнаруженные различия моделей могут проявляться в виде многих различных симптомов. Наиболее очевидные из них — несоответствие запрограммированных интерфейсов. На более тонком уровне верным знаком является непредсказуемое поведение программы. "Выловить" такого рода проблемы позволяет процесс НЕПРЕРЫВНОЙ ИНТЕГРАЦИИ (CONTINUOUS INTEGRATION) с автоматизированными тестами. А вот заблаговременным предупреждением может служить путаница в языке.

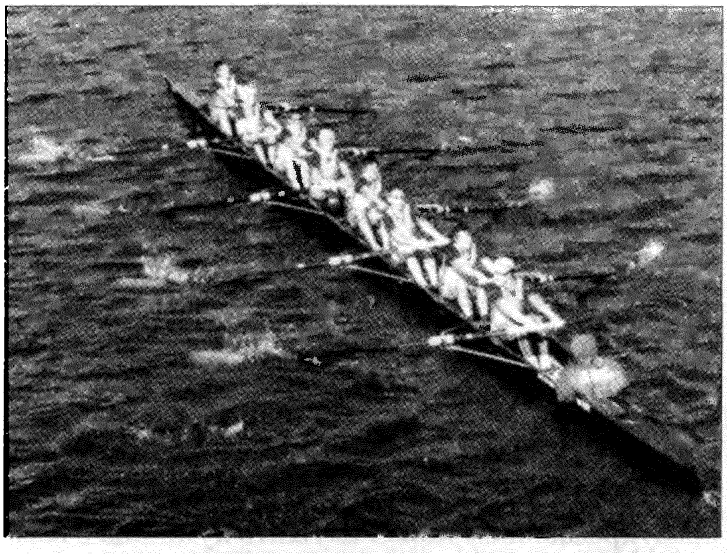
При комбинировании элементов различающихся моделей возникает две категории проблем: *дублирующиеся понятия* (*концепции*) и *ложные родственники*. Дублирование понятий означает, что в модели есть два элемента (и две соответствующих программных реализации), которые на самом деле представляют одно и то же понятие. Всякий раз, когда информация об этом понятии обновляется, ее приходится обновлять в двух местах. Как только новые знания приводят к изменениям в одном из объектов, второй тоже нужно заново проанализировать и изменить. Только вот повторный анализ в реальности не происходит, так что в результате возникают два варианта одного и того же понятия, которые следуют разным правилам и даже содержат разные данные. Помимо этого, разработчикам приходится осваивать не один, а два способа делать одно и то же, плюс все нужные способы синхронизации между ними.

"Ложные родственники" встречаются несколько реже, но зато они более коварны. В этом случае двое людей, пользующихся одним и тем же термином (или программным объектом), думают, что имеют в виду одно и то же, а на самом деле это не так. Типичный

пример приведен в начале главы (две разные прикладные операции используют объект **Платеж**), но конфликт может быть еще менее заметен, если два определения действительно относятся к одному и тому же аспекту предметной области, но осмыслены и составлены несколько по-разному. Присутствие "ложных родственников" приводит к тому, что группы разработчиков вмешиваются в код друг друга, в базах данных появляются странные противоречия, а в общении внутри групп возникает путаница. Термин "ложные родственники" обычно применяется к естественным человеческим языкам[1]. Так, если англоязычный человек изучает испанский язык, он часто ошибается в употреблении слова *embarazada*, думая по его внешнему виду, что оно означает *embarrassed* в женском роде (смущенная, пристыженная, сбитая с толку), тогда как на самом деле это "беременная". Ай, как неудобно получается!

Обнаружив такую проблему, группе разработчиков придется что-то решать. Возможно, понадобится временно "отозвать" модель из работы и усовершенствовать рабочие процедуры, чтобы избежать фрагментирования. Но группы могут тащить модель каждая в свою сторону, вызывая тем самым ее фрагментирование, и по каким-то веским причинам. Тогда не исключено, что им лучше позволить вести разработку независимо. Решение таких вопросов и составляет суть шаблонов, которые мы рассмотрим дальше в этой главе.

Непрерывная интеграция



Определив ОГРАНИЧЕННЫЙ КОНТЕКСТ (BOUNDED CONTEXT), следует поддерживать его в хорошем состоянии.

* * *

Когда в одном и том же ОГРАНИЧЕННОМ КОНТЕКСТЕ **работает одновременно много людей, модель имеет тенденцию фрагментироваться, распадаться на части. Чем больше группа разработчиков, тем больше и проблема, но всерьез столкнуться с трудностями могут всего три-четыре человека. Но разбиение системы на еще меньшие**

[1] В русском языке это понятие (*false cognates*) принято называть "ложными друзьями переводчика". Оно обозначает слова чужого языка, которые на слух кажутся очевидными по смыслу. На самом же деле их смысл совершенно отличается. Например, *angina* — это стенокардия, а не ангина, а *talon* — это коготь, а не талон. Но в контексте книги уместно говорить именно "родственники". — *Примеч. перев.*

КОНТЕКСТЫ **в конце концов приводит к тому, что в ней теряется полезный уровень интеграции и связности.**

Иногда программисты не вполне понимают назначение объекта или взаимосвязи, промоделированных кем-то другим, и изменяют их так, что они становятся непригодными для первоначального назначения. Иногда они не осознают, что понятие, над которым они работают, уже существует в другой части модели, так что они просто дублируют и само понятие, и его операции (причем неточно). Бывает, что разработчики даже знают об этих других выражениях нужных понятий, но боятся связываться с ними, чтобы не поломать то, что работает, и в итоге продолжают дублировать понятия и функции.

Очень тяжело поддерживать коммуникацию между разработчиками на таком уровне, чтобы иметь возможность разработать унифицированную систему произвольного масштаба. Необходимо иметь способы улучшения коммуникации и снижения сложности. Но надо также застраховаться и от слишком осторожного поведения, — например, многократного дублирования функций программистами только из страха испортить работающий код.

Именно в такой среде уместен подход экстремального программирования (*Extreme Programming, XP*). Многие приемы XP нацелены на решение именно специфической проблемы поддержания связной архитектуры при том, что ее постоянно изменяет множество людей. В чистом виде XP — прекрасная методика поддержания целостности модели в пределах одного ОГРАНИЧЕННОГО КОНТЕКСТА. Однако несмотря на то, используется подход XP или нет, все равно важно иметь в проекте какую-то процедуру НЕПРЕРЫВНОЙ ИНТЕГРАЦИИ (CONTINUOUS INTEGRATION).

НЕПРЕРЫВНАЯ ИНТЕГРАЦИЯ означает, что вся работа в пределах контекста сливается воедино и приводится в согласованный вид достаточно часто, чтобы даже при возникновении смысловых дефектов они выявлялись и устранялись достаточно быстро. НЕПРЕРЫВНАЯ ИНТЕГРАЦИЯ, как и все остальное в предметно-ориентированном проектировании, работает на двух уровнях: (1) интеграция понятий модели и (2) интеграция программной реализации.

Понятия и концепции интегрируются путем постоянной коммуникации между членами рабочей группы. Такая группа должна культивировать совместное понимание непрерывно меняющейся модели. Тут помогает много приемов, но самый фундаментальный из них — постоянная доработка ЕДИНОГО ЯЗЫКА проекта. Между тем все написанное программистами интегрируется воедино систематически проводимой процедурой сборки-компиляции-тестирования, которая позволяет выявлять дефекты в модели на ранних стадиях. Для интеграции проекта используется много процедур, но большинство самых эффективных обладает следующими характеристиками:

- пошаговая, легко воспроизводимая технология сборки и компиляции;
- наличие наборов автоматизированных тестов;
- правила, определяющие временные рамки (достаточно непродолжительные) для существования неинтегрированных в общую систему изменений.

Другой стороной медали в эффективных процедурах интеграции является *концептуальная* интеграция, хотя ее не часто определяют формально:

- постоянная практика в ЕДИНОМ ЯЗЫКЕ проекта при обсуждении модели и приложения.

В большинстве *Agile*-проектов как минимум раз в день выполняется сборка изменений кода, внесенных разными программистами. Частоту можно приспособить к конкретному темпу разработки, лишь бы любое еще не интегрированное изменение было

учтено еще до того, как другие участники проекта выполнят существенный объем несовместимой с ним работы.

В ПРОЕКТИРОВАНИИ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN) интегрирование понятий и концепций в единое целое облегчает и сглаживает интеграцию программной реализации. В свою очередь качество интеграции кода служит проверкой и гарантией единообразия модели, а также детектором смысловых дефектов.

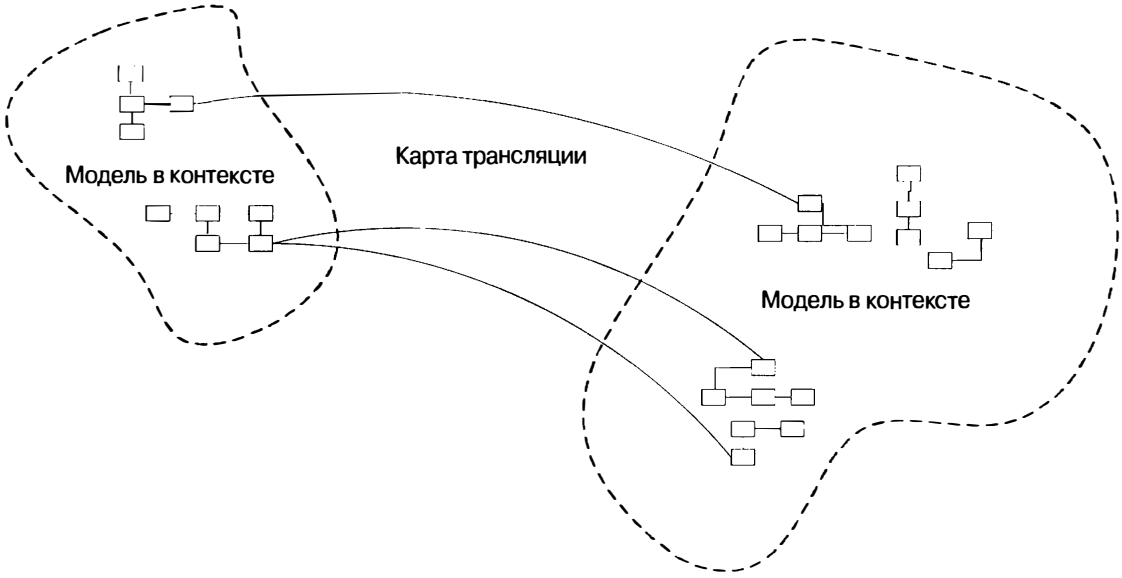
Организуйте процедуру частой сборки всего кода и других результатов работы по проекту, с автоматизированными тестами, позволяющими быстро обнаружить места фрагментации. Упорно практикуйтесь в ЕДИНОМ ЯЗЫКЕ**, чтобы выстроить единое понимание модели по мере эволюции понятий в представлениях разных людей.**

И последнее: не делайте больше работы, чем это необходимо. НЕПРЕРЫВНАЯ ИНТЕГРАЦИЯ существенно важна только в пределах ОГРАНИЧЕННОГО КОНТЕКСТА. Вопросы проектирования архитектуры с участием близлежащих КОНТЕКСТОВ, включая трансляцию понятий, не обязательно решать в таком же темпе.

* * *

НЕПРЕРЫВНАЯ ИНТЕГРАЦИЯ (CONTINUOUS INTEGRATION) применима внутри любого отдельного ОГРАНИЧЕННОГО КОНТЕКСТА (BOUNDED CONTEXT), превосходящего по размеру работу двух человек. Она нацелена на поддержание целостности одиночной модели этого контекста. Но когда сосуществуют несколько ОГРАНИЧЕННЫХ КОНТЕКСТОВ, приходится решать, каковы же взаимосвязи между ними, и проектировать соответствующие интерфейсы...

Карта контекстов



Любой отдельный ОГРАНИЧЕННЫЙ КОНТЕКСТ (BOUNDED CONTEXT) сам по себе не дает глобального видения. КОНТЕКСТЫ других моделей могут все еще оставаться в восприятии расплывчатыми и изменчивыми.

* * *

Члены других групп разработчиков могут быть не в курсе установленных границ ОГРАНИЧЕННОГО КОНТЕКСТА **и по незнанию внесут такие изменения, которые сделают границу нечеткой или усложнят взаимосвязи. Если между разными контекста-**

ми необходимо наличие связей, эти контексты имеют тенденцию "просачиваться" друг в друга.

Перенос кода из одного ОГРАНИЧЕННОГО КОНТЕКСТА в другой — это рискованное дело. Интеграция функциональности и данных должна выполняться через трансляцию. Чтобы уменьшить путаницу, следует определить отношения между разными контекстами и построить глобальный обзор контекстов для всего проекта.

КАРТА КОНТЕКСТОВ (CONTEXT MAP) — это средство, находящееся на стыке управления проектом и архитектурного проектирования. Естественный ход событий — совпадение границ контекстов с организационным делением группы. Люди, работающие вместе, естественным образом разделяют общий контекст модели. Члены разных групп или люди, не общающиеся между собой даже в составе одной группы, расходятся в различные контексты. Может иметь значение и физическое пространство офиса: так, члены группы, сидящие в разных концах здания, — не говоря уже о разных городах — наверняка разойдутся в контекстах, если не приложить особые усилия по интеграции. Большинство руководителей проектов интуитивно понимают эти факторы и организуют рабочие группы более или менее вокруг программных подсистем. Но простого наличия взаимосвязи между организацией группы, моделью и архитектурой программы еще недостаточно. Как разработчики, так и управленцы должны иметь четкое видение складывающегося концептуального подразделения модели и архитектуры.

Определите все модели, используемые в проекте, и задайте для каждой свой ОГРАНИЧЕННЫЙ КОНТЕКСТ**. Учитывайте и неявные модели подсистем, не являющихся объектно-ориентированными. Дайте каждому** ОГРАНИЧЕННОМУ КОНТЕКСТУ **имя и включите эти имена в** ЕДИНЫЙ ЯЗЫК **проекта.**

Опишите точки соприкосновения между моделями, явно задавая трансляцию для любого способа коммуникации и выделяя любые совместно используемые ресурсы.

Постройте карту *уже существующей* территории. Преобразованиями займетесь потом.

Внутри каждого ОГРАНИЧЕННОГО КОНТЕКСТА у вас будет свой самодостаточный диалект ЕДИНОГО ЯЗЫКА. Имена ОГРАНИЧЕННЫХ КОНТЕКСТОВ сами тоже войдут в ЕДИНЫЙ ЯЗЫК, чтобы можно было недвусмысленно выражаться относительно модели любой из частей архитектуры, ясно задавая контекст разговора.

КАРТУ КОНТЕКСТОВ не обязательно документировать в какой-либо конкретной форме. Я считаю, что достаточную выразительность и наглядность имеют схемы и диаграммы, приведенные в этой главе. Кто-то другой может предпочесть текстовое представление или графические схемы другого типа. В каких-то ситуациях достаточно будет простого обсуждения между участниками группы. Детализацию можно варьировать по потребностям. Но какую бы форму ни приняла карта, она должна быть известна и понятна всем в проекте. Она должна задавать четкие имена для всех ОГРАНИЧЕННЫХ КОНТЕКСТОВ, а также ясно обозначать их точки соприкосновения и общий характер.

* * *

Взаимосвязи и отношения между ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ принимают много разных форм в зависимости как от организации, так и от архитектуры проекта. Позже в этой главе будут представлены различные шаблоны взаимосвязей между КОНТЕКСТАМИ, эффективные в разных ситуациях и предоставляющие терминологию для описания связей из ваших собственных КАРТ. Учитывая, что КАРТА КОНТЕКСТОВ всегда представляет ситуацию *такой, какой она есть*, обнаруживаемые взаимосвязи вначале могут и не соответствовать упомянутым шаблонам. Если близость между ними достаточна, можно взять имя шаблона, но не навязывать его силой. Просто опишите такую взаимосвязь, какую вы хотите. Позже можно будет начать двигаться к более стандартным отношениям.

Итак, что же делать, если вы обнаружили смысловой дефект — все взаимосвязи в модели установлены, но при этом она содержит противоречия? Сосредоточьтесь на карте и заставьте себя закончить полное описание. Затем, имея точный общий вид, займитесь местами проявления противоречий. Исправив небольшой дефект, можно ввести специальные процедуры для поддержания целостности. Если какая-то связь не вполне ясна, можно выбрать близкий по смыслу шаблон, а затем "продвигаться" к нему. Ваша первая задача на повестке дня — построить четкую КАРТУ КОНТЕКСТОВ, а это может включать в себя решение некоторых реальных проблем, с которыми вы сталкиваетесь. Но не позволяйте "мелкому ремонту" перейти в глобальную реорганизацию. Пока у вас нет четкой КАРТЫ КОНТЕКСТОВ, которая распределяет всю вашу работу по ОГРАНИЧЕННЫМ КОНТЕКСТАМ и задает явные отношения между всеми взаимосвязанными моделями, исправляйте только самые вопиющие противоречия.

Как только у вас появится полная КАРТА КОНТЕКСТОВ, вы сами увидите места, в которые нужно внести изменения. Можно будет изменять как организацию групп, так и архитектуру системы. Помните: не вносите изменение в КАРТУ, пока оно не сделано в реальности.

Пример

Два контекста в программе доставки грузов

Мы снова возвращаемся к системе управления доставкой грузов. Одной из основных возможностей программы должна быть автоматическая маршрутизация грузов в момент заказа их доставки. Соответствующая модель представляет собой примерно следующее.

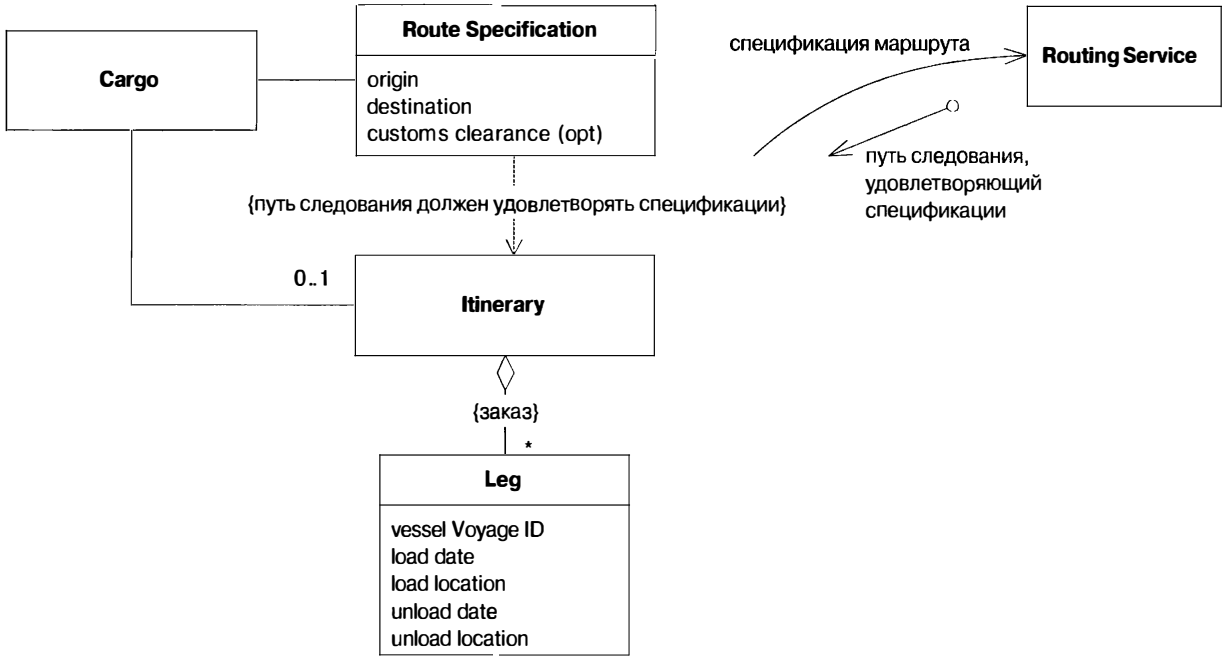


Рис. 14.2.

Служба маршрутизации (Routing Service) — это СЛУЖБА (SERVICE), инкапсулирующая механизм, который стоит за ИНФОРМАТИВНЫМ ИНТЕРФЕЙСОМ (INTENTION-REVEALING INTERFACE) и состоит из ФУНКЦИЙ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ (SIDE-EFFECT-FREE FUNCTIONS). Результаты работы этих функций характеризуются КОНТРОЛЬНЫМИ УТВЕРЖДЕНИЯМИ (ASSERTIONS).

1. В интерфейсе объявлено, что когда передается **Спецификация маршрута (Route Specification)**, должен возвращаться **Путь следования (Itinerary)**.
2. КОНТРОЛЬНОЕ УТВЕРЖДЕНИЕ гласит, что возвращаемый **Путь следования** должен удовлетворять переданной **Спецификации маршрута**.

Здесь ничего не говорится о том, *как* выполнить эту очень сложную задачу. Давайте заглянем за кулисы, чтобы увидеть механизм.

Первоначально, работая в проекте, на основе которого построен этот пример, я был чрезмерно догматичен по отношению к внутренней реализации **Службы маршрутизации (Routing Service)**. Я хотел, чтобы фактическая маршрутизация выполнялась с использованием расширенной модели предметной области, в которой бы представлялись рейсы судов, непосредственно привязанные к **Участкам (Legs) Пути следования (Itinerary)**. Но группа, работавшая над маршрутизацией, сообщила, что решение следует реализовать в виде оптимизируемой транспортной сети, где каждый участок рейса представлялся бы элементом
зволяло привлечь хорошо известные, стандартные алгоритмы. Они настаивали на создании специальной модели для операций доставки груза в этом случае.

Группа была совершенно права в том, что касалось вычислительных аспектов процедуры маршрутизации в тогдашнем ее виде, и за неимением более удачных идей я сдался. По сути, мы создали два отдельных ОГРАНИЧЕННЫХ КОНТЕКСТА, в каждом из которых операции доставки груза были организованы концептуально по-разному (рис. 14.3).

Наше требование заключалось в том, чтобы запрос к **Службе маршрутизации (Routing Service)** перевести (транслировать) в термины, понятные **Службе трассировки по транспортной сети (Network Traversal Service)**, затем получить от нее результат и транслировать его в форму, ожидаемую от **Службы маршрутизации**.

Это означает, что нет необходимости устанавливать полное соответствие между двумя моделями, а достаточно определить только две трансляции.

Спецификация маршрута (Route Specification) → **Список (List)** кодов местоположений

Список номеров **Узлов (Nodes)** → **Путь следования (Itinerary)**

Для этого нужно выяснить назначение элемента в одной модели и найти способ выразить его через понятия другой.

Что касается первой трансляции (**Спецификация маршрута** → **Список**), необходимо подумать, какой смысл имеет последовательность местоположений в списке. Первый пункт списка — это начало пути; затем маршрут должен пройти по очереди через все пункты, пока не достигнет последнего в списке. Итак, место отправления и место назначения являются первым и последним пунктами списка, а в середине находятся пункты растаможивания (если таковые есть).

(К счастью, две группы использовали одни и те же коды местоположений, так что не нужно вводить трансляцию хотя бы на этом уровне.)

Следует отметить, что обратная трансляция является неоднозначной, потому что на вход трассировки по транспортной сети можно подавать любое количество промежуточных пунктов, не обязательно пунктов фактического растаможивания. К счастью, это не проблема, потому что в этом направлении выполнять трансляцию не надо. Но все же следует быть в курсе, почему некоторые трансляции невозможны.

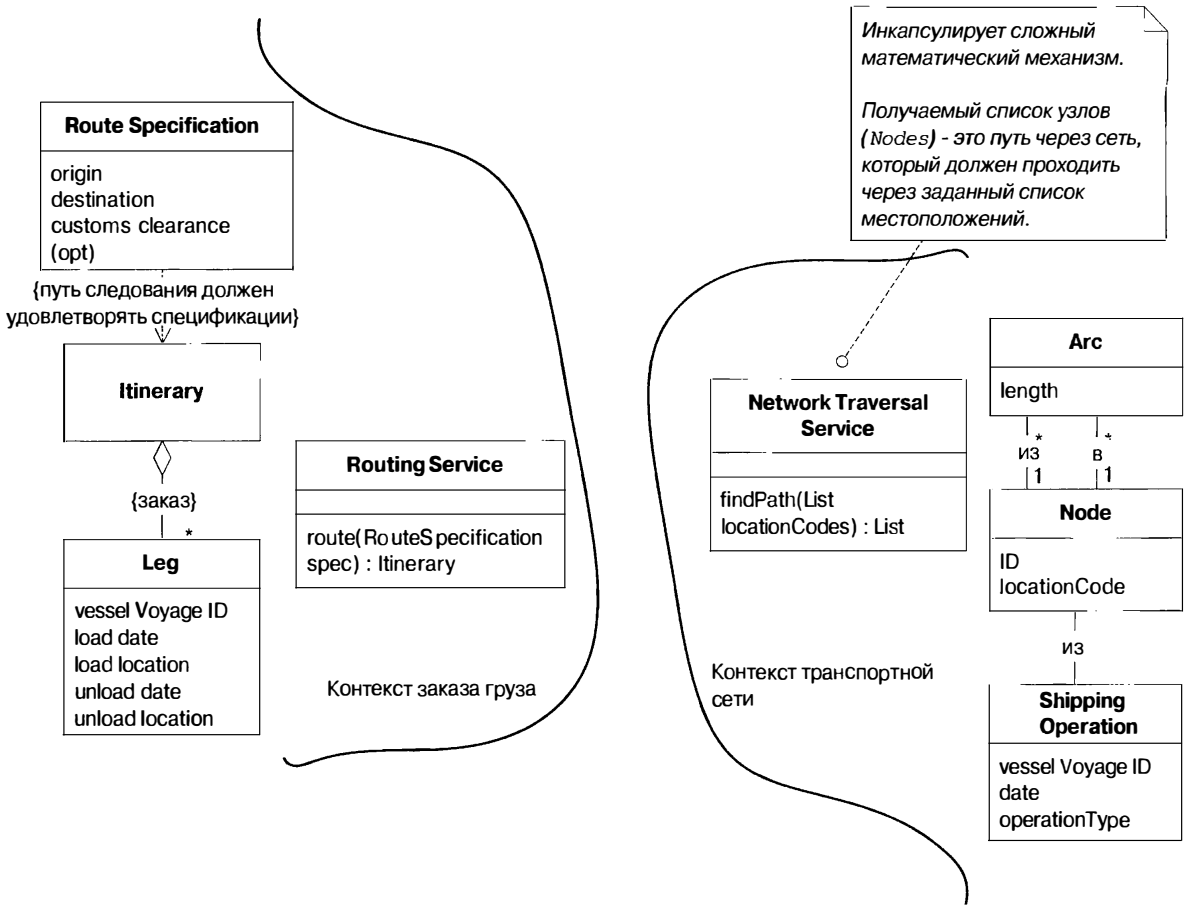


Рис. 14.3. Два ОГРАНИЧЕННЫХ КОНТЕКСТА дают возможность применять эффективные алгоритмы маршрутизации

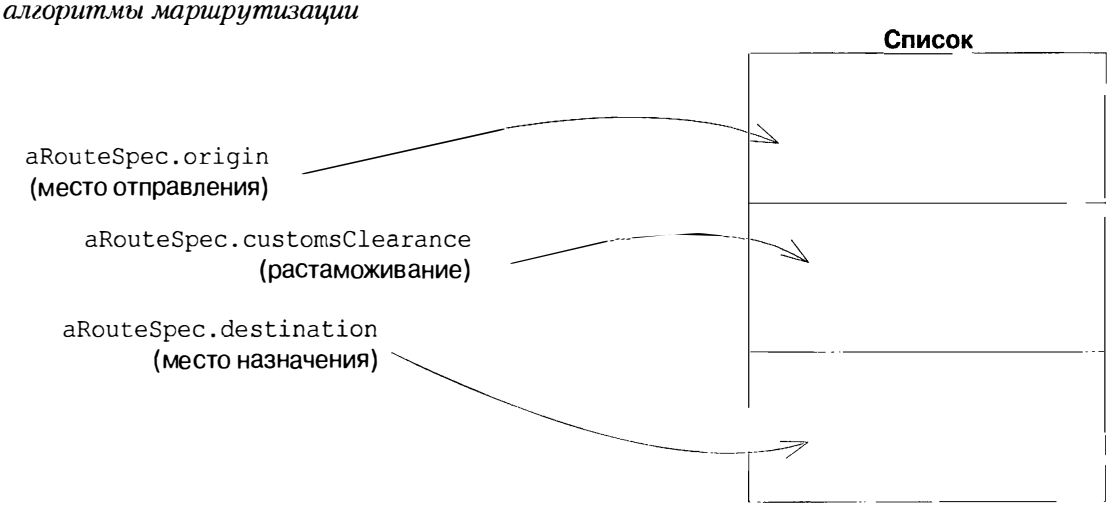


Рис. 14.4. Трансляция запроса к ***Службе трассировки по транспортной сети (Network Traversal Service)***

Теперь давайте транслируем результат (**Список** номеров **Узлов** → **Путь следования**). Предполагаем, что для поиска объектов **Узел** (**Node**) и **Перевозка** (**Shipping Operation**) по полученным нами идентификационным номерам **Узлов** мы можем воспользоваться ХРАНИЛИЩЕМ. Итак, как же **Узлы** (**Nodes**) соответствуют **Участкам** (**Legs**)? На основе атрибута `operationType` мы можем разбить список **Узлов** на пары "пункт отправления пункт прибытия". Каждую пару затем можно поставить в соответствие одному участку.

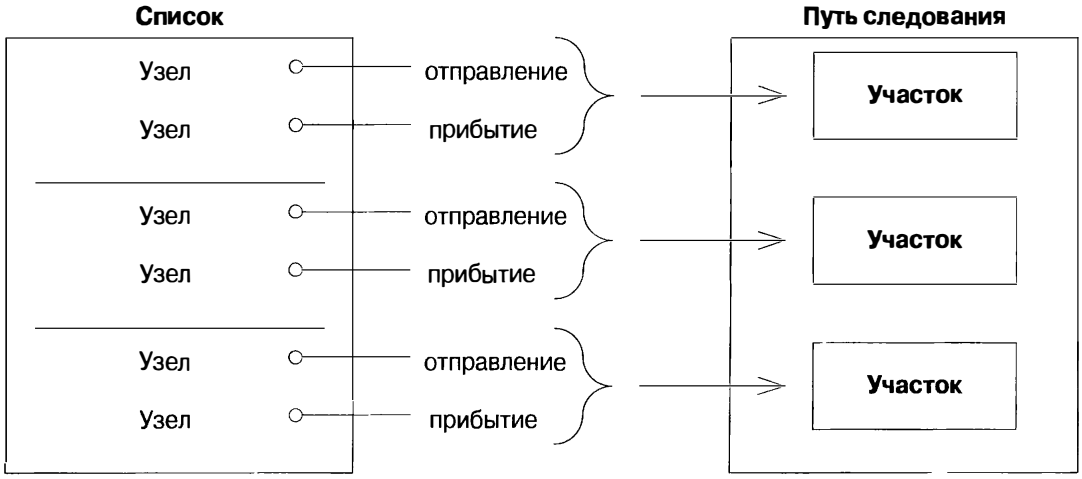


Рис. 14.5. Трансляция маршрута, найденного ***Службой трассировки по транспортной сети*** *(**Network Traversal Service**)*

Атрибуты каждой пары **Узлов** отображаются следующим образом.

```
departureNode.shippingOperation.vesselVoyageId →
                                    leg.vesselVoyageId
departureNode.shippingOperation.date → leg.loadDate
departureNode.locationCode → leg.loadLocationCode
arrivalNode.shippingOperation.date → leg.unloadDate
arrivalNode.locationCode → leg.unloadLocationCode
```

Это и есть концептуальная карта трансляции между двумя моделями. Теперь нужно реализовать в программе что-то, что будет выполнять эту трансляцию само. В простом случае наподобие этого для такой цели обычно создается специальный объект, а потом находится или создается еще один объект, который будет оказывать эту услугу остальной части подсистемы.

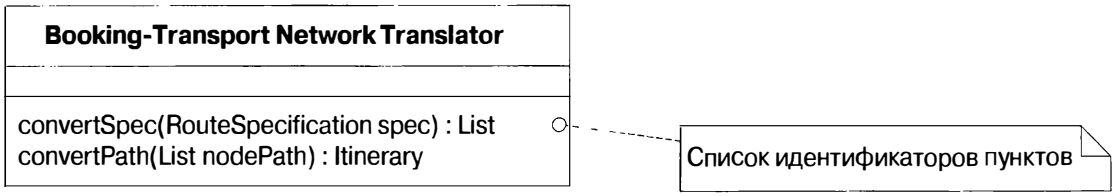


Рис. 14.6. Двусторонний транслятор

Над этим объектом следует работать сразу двум группам. Спроектировать объект надо так, чтобы его было легко тестировать модульными тестами. Особенно удачно было бы организовать совместную работу групп над набором тестов. Во всем остальном они могут идти собственными, отдельными путями.

Реализация **Службы маршрутизации (Routing Service)** теперь сводится к делегированию полномочий транслятору и службе трассировки по сети. Одна операция этой службы теперь может выглядеть так.

```
public Itinerary route(RouteSpecification spec) {
   Booking_TransportNetwork_Translator translator =
      new Booking_TransportNetwork_Translator();

   List constraintLocations =
      translator.convertConstraints(spec);
```

```
    // Получение доступа к службе NetworkTraversalService
    List pathNodes =
       traversalService.findPath(constraintLocations);

    Itinerary result = translator.convert(pathNodes);
    return result;
}
```

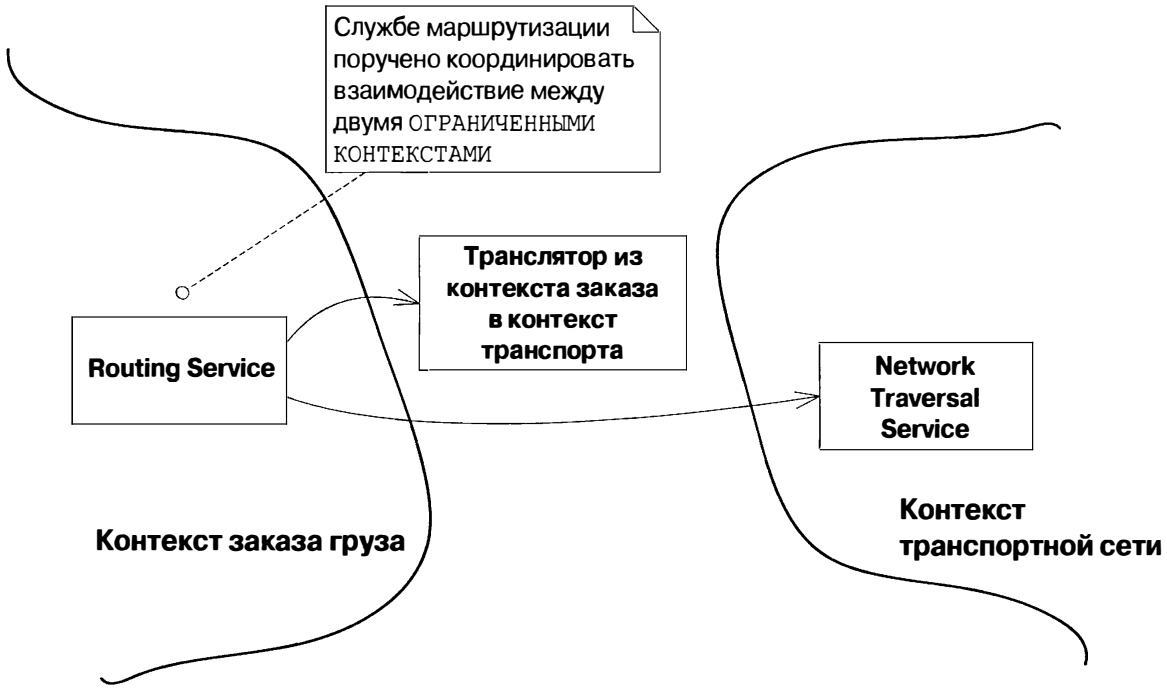


Рис. 14.7.

Неплохо. ОГРАНИЧЕННЫЕ КОНТЕКСТЫ помогли сохранить каждую из моделей в относительно чистом виде, позволили каждой из групп работать в значительной мере независимо, и если первоначальные допущения были правильными, то контексты выполнили свою задачу. (Мы еще вернемся к ним в этой главе.)

Интерфейс между двумя контекстами довольно невелик. Интерфейс **Службы маршрутизации (Routing Service)** изолирует остальную часть архитектуры КОНТЕКСТА заказа грузов от событий, происходящих в мире маршрутизации. Этот интерфейс легко тестировать, потому что состоит он из ФУНКЦИЙ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ (SIDE-EFFECT-FREE FUNCTIONS). Один из секретов комфортного сосуществования с другими КОНТЕКСТАМИ состоит в том, чтобы иметь для всех интерфейсов эффективные наборы тестов. Как говорил президент Рейган во время переговоров о сокращении вооружений, "доверяй, но проверяй"[2].

Построить набор автоматизированных тестов, которые передают **Спецификацию маршрута (Route Specification)** в **Службу маршрутизации (Routing Service)**, а затем проверить возвращаемый **Путь следования (Itinerary)**, не должно составить труда.

[2] Рональд Рейган перевел старинную русскую поговорку, которая точно описала происходящее с обеими переговаривающимися сторонами и это еще одна метафора, выражающая суть отношений между контекстами.

Контексты моделей существуют всегда, но без сознательного внимания к ним они могут пересекаться и дрейфовать. Определив ОГРАНИЧЕННЫЕ КОНТЕКСТЫ (BOUNDED CONTEXTS) и КАРТУ КОНТЕКСТОВ (CONTEXT MAP), ваша рабочая группа сможет начать управлять процессом унификации моделей и организацией связи между теми из них, которые должны остаться различными.

Тестирование в границах контекста

Особое значение имеет тестирование точек соприкосновения между ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ. Тесты помогают прояснить тонкости трансляции и компенсировать ухудшение коммуникации, которое обычно происходит на границе. Тестирование может служить системой раннего предупреждения, особенно полезной в случаях, когда вы зависите от подробностей устройства модели, над которой не имеете власти.

Организация и документирование карт контекстов

Здесь следует соблюсти только два важных принципа.

1. ОГРАНИЧЕННЫЕ КОНТЕКСТЫ должны иметь имена, чтобы на них можно было ссылаться при обсуждении. Эти имена должны войти в ЕДИНЫЙ ЯЗЫК группы разработчиков.
2. Все разработчики должны знать, где пролегают границы, и уметь распознать КОНТЕКСТ любого фрагмента кода в любой ситуации.

Второе требование можно удовлетворить многими способами в зависимости от уровня культуры в рабочей группе. Как только ОГРАНИЧЕННЫЕ КОНТЕКСТЫ определены, естественным образом происходит распределение кода разных КОНТЕКСТОВ по разным МОДУЛЯМ. Это порождает проблему определения и учета того, какой МОДУЛЬ принадлежит к какому КОНТЕКСТУ. Для ее решения можно задать правила имен или любой другой простой механизм, предотвращающий путаницу.

В той же степени важно так описать концептуальные границы, чтобы все разработчики понимали их одинаково. В качестве таких описаний мне нравятся неформальные схемы, показанные в вышеприведенном примере. Можно составить более строгие схемы или текстовые описания, перечислив все пакеты в каждом КОНТЕКСТЕ вместе с точками соприкосновения и механизмами связи и трансляции. Некоторым группам будет легче работать так, а другим будет достаточно устных договоренностей и активных обсуждений.

В любом случае, если имена контекстов добавляются в ЕДИНЫЙ ЯЗЫК, то и ввести КАРТЫ КОНТЕКСТОВ в обсуждение имеет смысл. Не говорите так: “То, что делает группа Джорджа, изменилось, поэтому и нам надо изменить то у нас, что к нему привязано”. Выражайтесь так: “Модель *Транспортной сети* меняется, поэтому нам следует внести изменения в *транслятор* для контекста *Заказа грузов*”.

Взаимосвязи между ограниченными контекстами

В рассматриваемых далее шаблонах предлагаются подходы и методики для установления взаимосвязей между двумя моделями, которые совместно могут покрыть всю корпоративную предметную область. Эти шаблоны выполняют сразу две задачи: задают цели для успешной организации процесса разработки и предлагают словарь для описания той организации, которая уже существует в настоящее время.

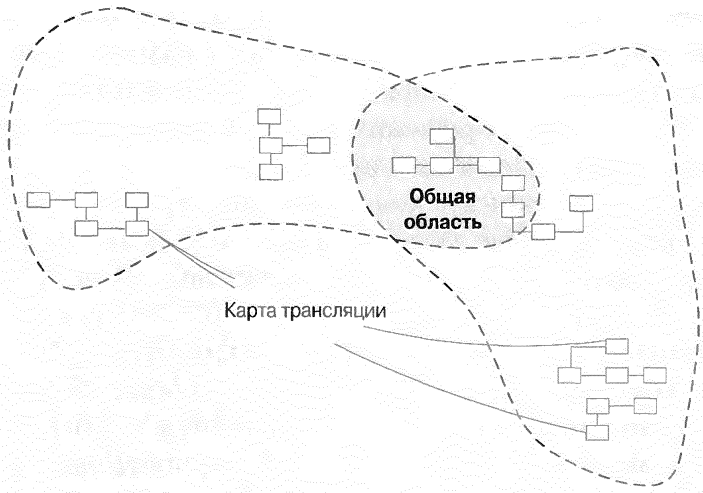
Существующая взаимосвязь может как по чистой случайности, так и по замыслу оказаться близкой к одному из этих шаблонов, и тогда ее можно описать в соответствующих

терминах, не забывая указать отличия и вариации. Затем постепенно, небольшими модификациями, можно приблизить эту взаимосвязь к выбранному шаблону.

С другой стороны, может оказаться и так, что исходная взаимосвязь слишком неопределенна или усложнена. Может потребоваться некоторая реорганизация, просто для того чтобы построить четкую КАРТУ КОНТЕКСТОВ. В этой ситуации или в любой другой, где возникает мысль о реорганизации, описываемые здесь шаблоны предлагают набор вариантов, пригодных в тех или иных обстоятельствах. Варьироваться может уровень вашего контроля над каждой из моделей, объем и способ сотрудничества между рабочими группами, степень интеграции функциональности и данных.

Набор шаблонов, предлагаемых далее, охватывает наиболее распространенные и важные случаи, а в других случаях из них можно почерпнуть общий подход. Высококвалифицированные разработчики, в активном сотрудничестве создающие тесно интегрированный продукт, способны построить большую унифицированную модель. Необходимость удовлетворить потребности разных сообществ пользователей или ограниченные возможности по координации в группе могут потребовать связей типа ОБЩЕГО ЯДРА (SHARED KERNEL) или ЗАКАЗЧИК-ПОСТАВЩИК (CUSTOMER/SUPPLIER). Иногда, если пристально присмотреться к требованиям, то оказывается, что интеграция не так уж и нужна и что для данных конкретных систем более целесообразно ОТДЕЛЬНОЕ СУЩЕСТВОВАНИЕ (SEPARATE WAYS). Большинство проектов приходится в какой-то степени интегрировать со старыми и внешними системами, что приводит к применению ПРЕДОХРАНИТЕЛЬНЫХ УРОВНЕЙ (ANTICORRUPTION LAYERS) и СЛУЖБ С ОТКРЫТЫМ ПРОТОКОЛОМ (OPEN HOST SERVICES).

Общее ядро



Когда интеграция подсистем в функциональном отношении ограничена, затраты на НЕПРЕРЫВНУЮ ИНТЕГРАЦИЮ (CONTINUOUS INTEGRATION) могут оказаться слишком большими, особенно если у рабочих групп не хватает квалификации или организованности, чтобы ее поддерживать, или же если единая группа разработчиков слишком велика и неудобна в управлении. Тогда можно определить отдельные ОГРАНИЧЕННЫЕ КОНТЕКСТЫ (BOUNDED CONTEXTS) и сформировать несколько отдельных рабочих групп.

* * *

Нескоординированные группы, работающие над близко связанными приложениями, могут поначалу взять хороший темп, но затем результаты их работы могут не состыковаться. Тогда они потратят больше времени на трансляционные уровни и реорганизацию кода, чем с самого начала потратили бы на НЕПРЕРЫВНУЮ ИНТЕГРАЦИЮ, **к тому же дублируя усилия и утратив преимущества обладания** ЕДИНЫМ ЯЗЫКОМ.

Во многих проектах я видел, как инфраструктурный уровень использовался совместно группами, которые в остальном работали независимо. Аналогично можно работать и в пределах предметной области. Полная синхронизация всей модели и базы кода может потребовать слишком много усилий, а вот тщательно отобранное ее подмножество может сохранить многие преимущества при снижении затрат.

Выберите некоторое подмножество модели предметной области, с которым две группы согласны работать совместно. Разумеется, к этой части модели прилагается соответствующая часть базы кода или архитектуры базы данных. Эти совместно используемые ресурсы имеют особый статус в проекте, и вносить в них изменения нельзя без согласия другой стороны.

Выполняйте интеграцию функциональной системы достаточно часто, но не так часто, как НЕПРЕРЫВНУЮ ИНТЕГРАЦИЮ **внутри групп. При каждой такой интеграции запускайте тесты обеих групп.**

Так создается разумный баланс. ОБЩЕЕ ЯДРО (SHARED KERNEL) нельзя изменять так часто, как другие части архитектуры. Принятие решений не может обходиться без консультаций с другой группой. Наборы автоматизированных тестов следует интегрировать в одно целое, потому что при внесении изменений должны выполняться все тесты обеих групп. Обычно рабочие группы вносят изменения в разные копии ЯДРА, выполняя интеграцию между собой через некоторые интервалы. (Например, для группы, выполняющей НЕПРЕРЫВНУЮ ИНТЕГРАЦИЮ ежедневно или чаще, нормальный ритм слияния изменений в ОБЩЕМ ЯДРЕ — еженедельный.) Но вне зависимости от графика интеграции кода, чем чаще группы обсуждают свои изменения, тем лучше.

* * *

ОБЩЕЕ ЯДРО (SHARED KERNEL) часто представляет собой СМЫСЛОВОЕ ЯДРО предметной области (CORE DOMAIN), набор ЕСТЕСТВЕННЫХ ПОДОБЛАСТЕЙ (GENERIC SUBDOMAINS) или то и другое одновременно (см. главу 15). Но в принципе оно может представлять собой любую часть модели, которая требуется каждой из двух рабочих групп. Здесь ставится цель уменьшить дублирование работы (но не полностью устранить его, как это было бы в случае одного ОГРАНИЧЕННОГО КОНТЕКСТА) и сделать интеграцию между двумя подсистемами сравнительно простой.

Группы “заказчик-поставщик”

Часто бывает так, что одна подсистема фактически подает что-то на вход другой, снабжает ее данными. “Нижний” компонент выполняет анализ или другие функции, ко-

торые мало что передают в "верхний" компонент, так что все зависимости и взаимосвязи идут в одном направлении[3]. Две подсистемы обычно предназначены для очень разных сообществ пользователей, которые занимаются разной деятельностью, моделируемой разными способами. Различными могут быть и средства разработки, так что код таких приложений невозможно переносить, объединять, совместно использовать.

* * *

Верхняя и нижняя подсистемы естественным образом разделяются на два различных ОГРАНИЧЕННЫХ КОНТЕКСТА (BOUNDED CONTEXTS). Особенно верно это для случаев, когда два компонента требуют разных навыков или реализуются разными программными средствами. Трансляция в одну сторону проще, чем в обе. Но в зависимости от взаимоотношений между двумя группами могут возникать и проблемы.

Свобода действий верхней группы может оказаться стесненной, если нижняя группа имеет право вето на изменения или если процедуры запроса изменений слишком громоздки. Верхняя группа даже может приостановить свою деятельность, беспокоясь о судьбе нижней подсистемы. Но и нижняя группа тоже может оказаться беспомощной, целиком во власти приоритетов верхней группы.

Внизу нуждаются в поставках сверху, но те, кто вверху, не отвечают за продукцию, выдаваемую снизу. Определить, что и как повлияет на другую группу разработчиков — довольно трудоемкая задача, учитывая и человеческую натуру, и фактор времени, и пр. Всем становится легче жить, если отношения между группами формализованы. Можно организовать процедуры для балансирования потребностей двух сообществ пользователей и составить график работ по функциям, которые нужны внизу.

В проектах, выполняемых по методике экстремального программирования, уже есть механизм точно для этой цели: процесс итерационного планирования. Все, что нужно сделать, — это определить отношения между двумя группами с точки зрения процесса планирования. Представители нижней группы могут выступать в качестве представителей пользователя: присутствовать на сеансах планирования, непосредственно обсуждать предстоящие задачи как полноправные "заказчики" продукции, сопоставлять преимущества и недостатки решений. Так возникает итерационный план для группы-поставщика, содержащий задачи, как первоочередные для нижней группы, так и отложенные, по которым пока не ожидается результата.

Если используется отличная от экстремального программирования методология, можно взять какую-нибудь аналогичную методику для балансирования потребностей разных пользователей и усовершенствовать ее так, чтобы учесть потребности приложения нижней группы.

Определите четкие отношения "заказчик-поставщик" между двумя группами. На сеансах планирования нижняя группа должна играть роль заказчика по отношению к верхней. Обсуждайте и распределяйте задачи согласно требованиям нижней группы, чтобы каждый в группе имел представление об обязательствах и временном графике работы.

Совместно разработайте автоматизированные приемочные тесты для проверки ожидаемого интерфейса. Добавьте эти тесты в набор тестов верхней рабочей группы, чтобы они выполнялись в рамках происходящей у них непрерывной интеграции. Такое тестирование позволит верхней группе спокойно вносить изменения, не боясь побочных эффектов внизу.

---

[3] Имеется в виду "верхний" и "нижний" по течению реки или потока: *upstream* и *downstream* соответственно. *Примеч. перев.*

В ходе итерационного процесса члены нижней группы разработчиков должны быть так же "доступны" верхней группе для контакта, как обычные заказчики для оперативного ответа на вопросы и решения проблем.

Автоматизация приемочных тестов — важнейшая часть отношений такого вида. Даже в проекте с самым тесным сотрудничеством, где заказчик может определить и сообщить поставщику все нужные взаимосвязи, а поставщик изо всех сил пытается донести до заказчика сделанные изменения, без тестов все равно случаются сюрпризы. Эти сюрпризы мешают работе нижних разработчиков, а верхних заставляют срочно вносить исправления, не предусмотренные графиком. Вместо этого нужно сделать так, чтобы группа-заказчик, совместно с группой-поставщиком, разработала автоматические приемочные тесты, проверяющие на правильность требуемый интерфейс. Верхняя группа должна выполнять эти тесты в составе своего стандартного набора тестов. Любые изменения в этих тестах требуют согласования с другой группой, поскольку изменения в тестах требуют изменений в интерфейсе.

Отношения "заказчик-поставщик" также возникают между проектами в отдельных компаниях — в тех ситуациях, когда какой-то один заказчик очень важен для дела поставщика. Здесь хвост начинает вилять собакой: влиятельный заказчик может выдвигать такие требования, которые важны для успеха верхнего проекта, но они также могут нарушить процесс разработки верхнего проекта. Обе стороны выигрывают от формализации процесса отклика на требования, потому что соотношение затрат и выгоды еще труднее разглядеть в отношениях с внешними партнерами, чем внутри одного IT-проекта.

В этом стратегическом шаблоне есть два ключевых момента.

1. Отношения заказчика и поставщика подразумевают, что потребности заказчика однозначно стоят на первом месте. Но поскольку нижняя группа разработчиков не является единственным заказчиком, требования всех заказчиков приходится согласовывать в процессе переговоров. Тем не менее, именно за ними остается приоритет. Вместо таких отношений часто можно встретить отношения типа "бедный родственник", когда нижняя группа должна выпрашивать у верхней то, что ей нужно.
2. Должен существовать набор автоматизированных тестов, который позволяет верхней группе вносить изменения в код, не боясь что-нибудь испортить в работе нижней, а также дает возможность нижней группе сосредоточиться на своих задачах, а не постоянно контролировать деятельность верхней.

В эстафетном забеге лидер не может все время оглядываться назад и проверять, все ли идет нормально. Он должен доверять бегуну, несущему палочку, чтобы ее передача произошла вовремя, иначе вся команда безнадежно отстанет.

Пример

Анализ доходности при заказе грузов

Вернемся к нашему безотказному примеру с доставкой грузов. Для анализа всех заказов, проходящих через фирму, и максимизации дохода организована специализированная группа. В ходе работы она может, например, выяснить, что на судах есть пустое пространство, и порекомендовать принимать больше лишних заказов. Или же они решат, что суда слишком рано заполняются объемистыми грузами, заставляя компанию отказывать в срочных перевозках особо ценных товаров. В этом случае они могут порекомендовать специально оставлять место для таких грузов или поднять оптовые цены на обычный фрахт.

Для подобного анализа специалисты используют свои собственные сложные модели. Для вычислительной реализации они пользуются хранилищем данных (*data warehouse*) со средствами построения аналитических моделей. И им требуется очень много информации из приложения оформления заказов (*Booking*).

С самого начала было ясно, что это два отдельных ОГРАНИЧЕННЫХ КОНТЕКСТА (BOUNDED CONTEXTS), потому что в них используются разные средства программной реализации и, что еще важнее, разные модели предметной области. Каковы же должны быть отношения и связи между ними?

Может показаться логичным ввести ОБЩЕЕ ЯДРО (SHARED KERNEL), потому что в анализе доходности потребуется какое-то подмножество модели заказа, к тому же в их собственной модели имеются пересекающиеся понятия грузов, цен и т.п. Но ОБЩЕЕ ЯДРО малопригодно в случаях, когда используются разные технологии реализации. Кроме того, запросы у группы анализа доходности очень специфические, они все время экспериментируют со своими моделями и вводят альтернативные. Может быть, им будет лучше транслировать все необходимое из КОНТЕКСТА приложения заказа в свой собственный. (С другой стороны, если бы они смогли воспользоваться ОБЩИМ ЯДРОМ, бремя трансляции значительно облегчилось бы. Конечно, потребовалось бы реализовать модель заново и транслировать данные в новую реализацию, но если модель одна и та же, этот переход не должен оказаться слишком сложным.)

Приложение заказа грузов никак не зависит от анализа доходности, потому что никакая автоматическая коррекция деловой стратегии здесь не предусмотрена. Специалисты-люди принимают решения и сообщают их соответствующим работникам и программным системам. Так что у нас четко присутствует отношение “сверху-вниз”. Внизу от “верхов” требуется следующее.

1. Кое-какие данные, ненужные ни для каких операций заказа/резервирования.
2. Определенная стабильность в структуре базы данных (или, по крайней мере, надежная система оповещения об изменениях) или же наличие утилиты экспортирования.

К счастью, руководитель проекта по разработке приложения заказа грузов заинтересован помочь группе анализа доходности. Здесь могла бы возникнуть проблема, поскольку операционный отдел, который выполняет повседневную работу по резервированию заказов, подотчетен не тому вице-президенту, которому подчинена группа анализа доходности. Но более высокое начальство выказывает заинтересованность в таком анализе. Поэтому данный проект организован таким образом, чтобы руководители обеих групп были подотчетны одному и тому же ответственному лицу.

Итак, все требования для применения шаблона, именуемого ГРУППЫ РАЗРАБОТЧИКОВ “ЗАКАЗЧИК-ПОСТАВЩИК” (CUSTOMER/SUPPLIER DEVELOPER TEAMS), удовлетворены.

Я видел в нескольких местах, как события развивались именно по этому сценарию, т.е. разработчики аналитической и операционной подсистем вырабатывали именно отношения “заказчик-поставщик”. Когда члены верхней группы разработчиков считали, что их задача — обслуживать заказчика, все шло неплохо. Почти всегда взаимосвязи носили неформальный характер и в значительной степени зависели от личных отношений руководителей двух проектов.

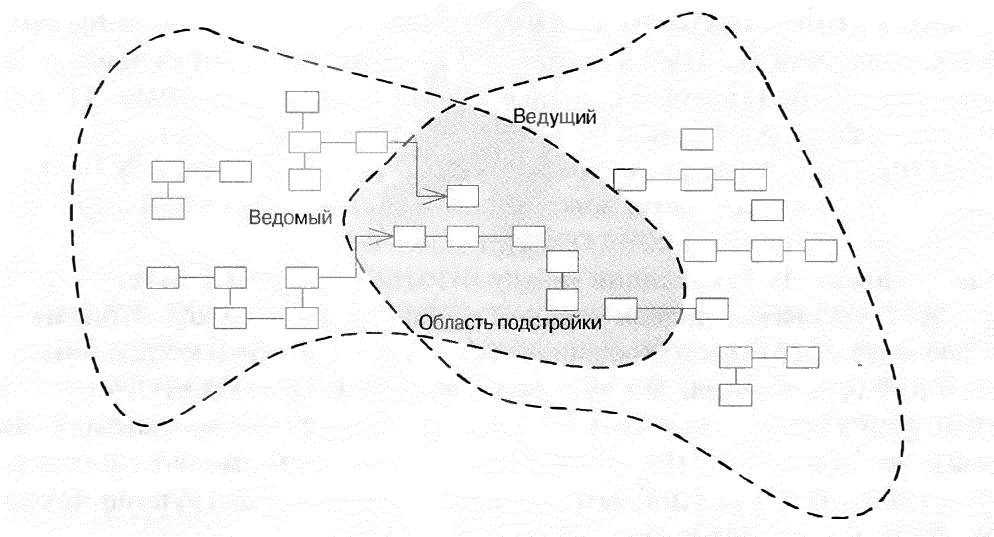
В одном проекте по экстремальному программированию я видел формализацию таких отношений в том смысле, что на каждой итерации представители нижней группы разыгрывали “игру планирования” в роли заказчиков, устраивая совещания совместно с обычными заказчиками (потребителями прикладных функций программы), и обсуждали с ними, какие задачи нужно решить на очередном этапе итерации. Этот проект вы-

полнялся в небольшой компании, так что ближайший общий начальник находился не очень высоко, в пределах досягаемости. И все получилось хорошо.

* * *

Группы "заказчик-поставщик" хорошо работают в тех случаях, когда обе группы находятся под единым руководством (т.е., в конце концов, преследуют одни и те же цели), или же тогда, когда они относятся к разным организациям, но сами эти организации фактически находятся в таких взаимоотношениях. Если же мотивировать верхнюю группу печем, то ситуация резко отличается...

Конформист



Когда две группы разработчиков с отношениями типа "верх-низ"[4] не имеют эффективного руководства из одного источника, такой типовой образец сотрудничества, как заказчик и поставщик, неприменим. Если наивно попытаться воспользоваться им, нижняя группа окажется в беде. Это может случиться в большой компании, где две группы находятся далеко друг от друга в управленческой иерархии или где общий куратор проектов безразличен к отношениям между группами. Возможно подобное также и между группами в разных компаниях, если деятельность заказчика не представляет особой, индивидуальной ценности для поставщика. Возможно, у поставщика есть много мелких заказчиков или же поставщик меняет направление своей деятельности и старые клиенты его больше не интересуют. Поставщик может иметь плохо налаженную систему управления. Он вообще может уйти из своего бизнеса. Как бы там ни было, группа-заказчик вдруг оказывается наедине со своими проблемами.

Когда две группы разработчиков состоят в отношениях "верх-низ" и у "верха" нет мотивации удовлетворять потребности "низа", нижняя группа оказывается беспомощной. Из альтруизма разработчики верхней группы могут выдавать обещания, но они вряд ли будут выполнены. Вера в добрые намерения побуждает нижнюю группу строить планы на основе обещанных ресурсов, которые никогда не поступят. Проект внизу будет заморожен до тех пор, пока группа не научится обходиться тем, что у нее есть. А интерфейсу, специально "скроенному" под нужды нижней группы разработки, уже не суждено будет появиться на свет.

[4] Вверху (*upstream*) и внизу (*downstream*) "по течению", соответственно. Имеется в виду, как и ранее, односторонний поток данных и кода от верхней группы к нижней. — *Примеч. перев.*

Из этой ситуации есть три возможных выхода. Один — это вообще отказаться от помощи сверху. Этот вариант следует рассмотреть реалистично, не предполагая, что верхняя группа обязательно станет удовлетворять потребности нижней. Иногда мы преувеличиваем ценность или недооцениваем цену такой зависимости. Если нижняя группа решает перерезать свой поводок, то обе группы начинают вести ОТДЕЛЬНОЕ СУЩЕСТВОВАНИЕ (SEPARATE WAYS), о чем будет речь идти далее в этой главе.

Иногда ценность использования программ сверху настолько велика, что зависимость от них приходится поддерживать (или, допустим, принимается политическое решение, которое разработчики изменить не в силах). В этом случае остаются два выхода, выбор между которыми зависит от качества и стиля архитектуры, предлагаемой верхними разработчиками. Если с ней очень трудно работать, например, из-за недостаточной инкапсуляции, неудобного абстрагирования или моделирования в парадигме, которую другая группа не может использовать, то нижним разработчикам все равно придется строить свою собственную модель. Им придется нести полную ответственность за создание трансляционного уровня, который, скорее всего, окажется сложным. (См. ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ, ANTICORRUPTION LAYER, позже в этой главе.)

С другой стороны, если архитектура неплоха по качеству и стиль более-менее совместим, то имеет смысл, может быть, вовсе отказаться от независимой модели. В этих обстоятельствах уместен шаблон КОНФОРМИСТ (CONFORMIST).

Устраните сложность трансляции между ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ **(**BOUNDED CONTEXTS**), придерживаясь модели верхней группы разработчиков. Конечно, это ставит архитекторов нижней группы в стесненные обстоятельства, да и модель может оказаться неидеальной для приложения. Но все-таки действия в духе** КОНФОРМИЗМА **позволяют очень сильно упростить интеграцию. К тому же с группой-поставщиком можно будет разговаривать на** ЕДИНОМ ЯЗЫКЕ**. Управляет-то процессом именно поставщик, поэтому есть смысл сделать общение удобным для него. И тогда его альтруизма может оказаться достаточно, чтобы он поделился нужной информацией.**

Такое решение углубляет зависимость от верхней группы и накладывает ограничения на приложение в виде возможностей только верхней модели — плюс чисто аддитивные улучшения. Эмоционально это очень непривлекательный вариант, почему мы и выбираем его менее часто, чем следовало бы.

Следовать за кем-то не всегда плохо

Используя готовый компонент с обширным интерфейсом, обычно приходится подстраиваться (*conform*) под заложенную в него неявную модель, т.е. действовать как КОНФОРМИСТ. Компонент и приложение, очевидно, являются разными ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ (BOUNDED CONTEXTS) с точки зрения организации и управления, поэтому для незначительных изменений формата могут потребоваться какие-то адаптеры-переходники, но модель должна быть эквивалентной. В противном случае нужно поставить под сомнение ценность использования данного компонента. Если от него есть польза, значит, в его архитектуре содержится какое-то переработанное знание. В своей узкой сфере он может содержать значительно больше, чем вы об этом знаете. Ваша модель, предположительно, простирается за пределы сферы действия этого компонента, и ваши собственные концепции будут развиваться там соответственно. Но в точках контакта ваша модель представляет собой КОНФОРМИСТА (CONFORMIST), следующего за ведущей моделью — моделью компонента. Может даже случиться, что вам насильно навяжут лучшую архитектуру, чем ваша собственная.

Если ваш интерфейс с компонентом невелик по объему, то унификация моделей не так важна, и вполне можно обойтись трансляцией. Но если интерфейс велик и интеграция существенно важна, то обычно имеет смысл следовать за лидером.

Если такие компромиссы неприемлемы, но зависимость от верхней группы неизбежна, то еще остается второй вариант: изолировать себя, насколько это возможно, с помощью ПРЕДОХРАНИТЕЛЬНОГО УРОВНЯ (ANTICORRUPTION LAYER). Это агрессивный подход к реализации карты трансляции, о котором мы поговорим позже.

* * *

Шаблон КОНФОРМИСТ напоминает ОБЩЕЕ ЯДРО (SHARED KERNEL) тем, что у них есть: перекрывающаяся область, в которой модель одна и та же; области, в которых ваша модель аддитивно расширена; области, где вторая модель никак на вас не влияет. Разница между шаблонами состоит в процессе принятия решений и ходе разработки. Если ОБЩЕЕ ЯДРО — это сотрудничество между двумя тесно скоординированными группами, то КОНФОРМИСТ представляет интеграцию с группой, которая фактически не заинтересована в сотрудничестве.

Мы рассмотрели целый спектр вариантов сотрудничества при интеграции между ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ (BOUNDED CONTEXTS): от достаточно тесного двустороннего сотрудничества типа ОБЩЕГО ЯДРА (SHARED KERNEL) или ГРУПП РАЗРАБОТЧИКОВ "ЗАКАЗЧИК-ПОСТАВЩИК" (CUSTOMER/SUPPLIER DEVELOPER TEAMS) до односторонней работы КОНФОРМИСТА (CONFORMIST). Теперь мы сделаем последний шаг к еще более пессимистическому взгляду на взаимоотношения и предположим, что с другой стороны нас не ожидает ни полезная архитектура, ни желание сотрудничать...

Предохранительный уровень

Новые системы почти всегда приходится интегрировать со старыми или сторонними системами, у которых есть свои собственные модели. Трансляционные уровни для них могут быть простыми и даже "изящными", если они соединяют хорошо спроектированные ОГРАНИЧЕННЫЕ КОНТЕКСТЫ (BOUNDED CONTEXTS) сотрудничающих групп разработчиков. Но если с другой стороны границы начинаются "протечки", трансляционный уровень может стать линией обороны.

* * *

Когда строится новая система, которая должна иметь обширный интерфейс с другой системой, трудность соотнесения одной модели с другой может в конце концов начисто перечеркнуть всякий смысл новой модели, так как ее придется подстраивать под модель второй системы прямо на ходу. Модели старых, уже не поддерживаемых, систем обычно посредственны, но даже какое-нибудь высококачественное исключение может не удовлетворять потребности текущего проекта. Тем не менее интеграция с такими системами может иметь ценность, а иногда она выдвигается как категорическое требование.

Избегать вообще всякой интеграции — это не выход. Мне приходилось работать с такими проектами, где участники с энтузиазмом брались за полную замену всего старого кода. Но это слишком большой объем работы, чтобы вот так в нее бросаться. Кроме того, интегрирование с существующими системами — это полезная разновидность повторного использования программного кода. В больших проектах одна подсистема часто должна взаимодействовать с несколькими другими, независимо разработанными подсистемами через какие-то интерфейсы. Те подсистемы отражают предметную область проблемы по-другому. При комбинировании систем, основанных на разных моделях, потребность новой системы в принятии семантики других систем может привести к порче собственной модели новой системы. Даже если та, другая система хорошо спроектирована, она все равно не основана *на той же модели*, что ее клиент. К тому же другая система нечасто бывает хорошо спроектированной.

Взаимодействие с внешней системой через интерфейс не так-то просто наладить. Например, инфраструктурный уровень должен обеспечить средства коммуникации с другой системой, которая может работать на другой платформе или по другим протоколам. Типы данных другой системы приходится транслировать в типы нашей собственной. Но

о чем часто забывают, так это о том, что в другой системе наверняка используется другая концептуальная модель предметной области.

Кажется, всем очевидно, что если взять данные из одной системы и неправильно интерпретировать их в другой, от этого возникнут ошибки. Может быть, даже пострадает и окажется испорченной база данных. Но тем не менее эта проблема продолжает возникать, потому что мы думаем, будто переносим между системами только примитивные данные, имеющие однозначный смысл и одинаково понимаемые с обеих сторон. Эта точка зрения обычно неверна. Специфика связи тех или иных данных со своей системой может вызывать тонкие, но важные различия в их смысле в разных системах. Но пусть примитивные элементы данных даже и означают строго одно и то же в разных системах. Все равно это ошибка — позволять интерфейсу к другой системе работать на таком низком уровне. Низкоуровневый интерфейс делает модель другой системы бессильной там, где она могла бы помочь в объяснении смысла данных, ограничений на их значения, взаимосвязей. При этом новая система нагружается лишней работой по интерпретации примитивных данных, не выраженных через их собственную модель.

Необходимо обеспечить трансляцию между частями системы, основанными на разных моделях, и не дать моделям пострадать от невоспринимаемых ими элементов внешних моделей.

Создайте “изолирующий слой”, который бы предоставлял клиентам нужные функции в понятиях их собственной модели предметной области. Такой уровень будет общаться с другими системами через их существующие интерфейсы, что потребует лишь незначительной или вовсе никакой модификации этих других систем. Внутри же уровня будет идти необходимая трансляция в обе стороны между двумя моделями.

* * *

Эти обсуждения разных механизмов связи между двумя системами напоминают о таких проблемах, как перенос данных из одной программы в другую или с одного сервера на другой. Вскоре мы рассмотрим использование технических механизмов коммуникации. Но это все мелочь по сравнению с ПРЕДОХРАНИТЕЛЬНЫМ УРОВНЕМ (ANTICORRUPTION LAYER), который не представляет собой механизма пересылки сообщений от одной системы к др гой. Скорее это механизм, который транслирует концептуальные объекты и операции из одной модели и протокола в другую модель и протокол.

ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ может сам по себе быть немалой программной разработкой. Ниже представлены некоторые соображения, которыми надо руководствоваться при его планировании и проектировании.

Проектирование интерфейса предохранительного уровня

Открытый интерфейс ПРЕДОХРАНИТЕЛЬНОГО УРОВНЯ (ANTICORRUPTION LAYER) обычно построен как набор СЛУЖБ (SERVICES), хотя иногда он может принимать вид объекта-СУЩНОСТИ (ENTITY). Построение целого нового уровня для того, чтобы обеспечить трансляцию между семантикой двух разных систем, дает нам возможность заново абстрагировать операции другой системы и предложить как ее услуги, так и данные нашей системе в согласованном с нашей моделью виде. В нашей модели, может быть, даже не всегда есть смысл представлять внешнюю систему в виде одного компонента. Бывает лучше представить ее в виде нескольких СЛУЖБ, а иногда и СУЩНОСТЕЙ, каждая из которых имеет четко определенные обязанности в терминах нашей модели.

Реализация предохранительного уровня

Один из способов организации архитектуры ПРЕДОХРАНИТЕЛЬНОГО УРОВНЯ (ANTICORRUPTION LAYER) — представить его в виде набора ФАСАДНЫХ ОБЪЕКТОВ (FACADES), АДАПТЕРОВ (ADAPTERS) [14] и трансляторов, а также механизмов коммуникации и переноса, обычно необходимых для связи между системами.

Часто приходится интегрироваться с системами, имеющими большие, сложные, беспорядочные интерфейсы. Это проблема программной реализации; она не вызвана напрямую различиями в концептуальных моделях, которые делают необходимым применение ПРЕДОХРАНИТЕЛЬНЫХ УРОВНЕЙ. Но именно при попытке создания таких уровней эта проблема и возникает. Трансляция из одной модели в другую (особенно если одна из них нечеткая) — сама по себе достаточно сложная работа, а тут приходится иметь дело еще и с "невразумительным" интерфейсом подсистемы. К счастью, именно для этого существуют ФАСАДНЫЕ ОБЪЕКТЫ (FACADES).

ФАСАДНЫЙ ОБЪЕКТ — это альтернативный интерфейс некоторой подсистемы, который упрощает обращение к ней со стороны клиента и вообще облегчает ее использование. Нам известно в точности, какие функции другой системы мы хотим использовать, вот мы и пишем ФАСАДНЫЙ ОБЪЕКТ, который упрощает и спрямляет доступ к ним, скрывая все остальное. ФАСАДНЫЙ ОБЪЕКТ *не изменяет* модель представляемой им системы. Он должен быть написан в строгом соответствии с ее моделью, иначе мы в лучшем случае разбросаем ответственность за трансляцию на много объектов и перегрузим сам ФАСАДНЫЙ ОБЪЕКТ, а в худшем — построим еще одну модель, которая не относится ни к другой системе, ни к нашему собственному ОГРАНИЧЕННОМУ КОНТЕКСТУ. Между тем ФАСАДНЫЙ ОБЪЕКТ относится к ОГРАНИЧЕННОМУ КОНТЕКСТУ другой системы. Это ее представитель, дружественно настроенный и специально приспособленный к нашим потребностям.

АДАПТЕР (ADAPTER) представляет собой объект-оболочку, позволяющую клиенту пользоваться не тем протоколом, который понятен реализатору операции, а каким-нибудь другим. Когда клиент посылает сообщение АДАПТЕРУ, оно преобразуется в семантически эквивалентное сообщение и пересылается "адаптанту". Ответ оттуда опять преобразуется и пересылается назад. Термин "адаптер" (что по сути означает "переходник") используется здесь несколько произвольно; в [14] специально подчеркивалось, что с его помощью оборачиваемый в оболочку объект делают соответствующим стандартному интерфейсу, которого ожидает клиент. А мы тут выбираем пользоваться адаптированным интерфейсом, и при этом не исключаем, что "адаптант" — вообще не объект. Наша цель — организовать трансляцию между двумя моделями, но я думаю, что "дух" АДАПТЕРА при этом сохраняется.

Для каждой определяемой нами СЛУЖБЫ (SERVICE) необходим АДАПТЕР, который поддерживает интерфейс этой СЛУЖБЫ и умеет делать эквивалентные запросы к другой системе или ее ФАСАДНОМУ ОБЪЕКТУ.

Остается еще такой элемент, как транслятор. Задача АДАПТЕРА состоит в том, чтобы делать запросы. Фактическое преобразование концептуальных объектов или данных — это совсем другая сложная задача, которую можно передать отдельному объекту, отчего обе задачи станут понятнее. Транслятор может быть небольшим объектом, создаваемым и инициализируемым по необходимости. Ему не нужно собственное состояние, и его не надо распределять, поскольку он должен находиться там же, где и обслуживаемый(-е) им АДАПТЕР(Ы).

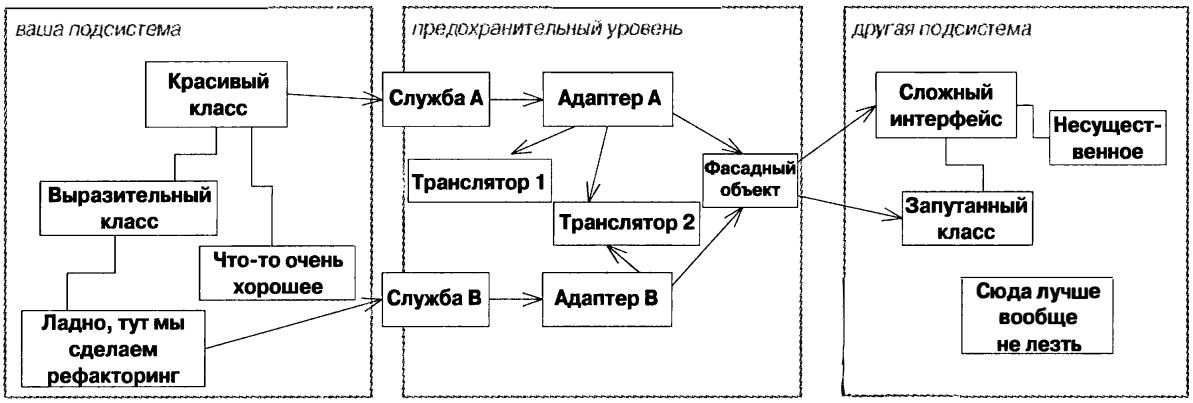


Рис. 14.8. Структура ПРЕДОХРАНИТЕЛЬНОГО УРОВНЯ (ANTICORRUPTION LAYER)

Вот из таких основных элементов строится ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ (ANTICORRUPTION LAYER). Есть и еще несколько соображений по этому поводу.

- Как правило, операцию инициирует проектируемая система (наша подсистема). Именно это подразумевается на рис. 14.8. Но бывают и случаи, когда другой подсистеме может понадобиться сделать какой-то запрос к нашей подсистеме или известить ее о каком-нибудь событии. ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ может быть двусторонним: определять СЛУЖБЫ с собственными АДАПТЕРАМИ на обоих интерфейсах, с возможностью использования одних и тех же трансляторов симметрично в обе стороны. Реализация ПРЕДОХРАНИТЕЛЬНОГО УРОВНЯ обычно не требует изменений в другой системе. Но в этом случае кое-какие изменения все же могут понадобиться, чтобы другая система вызывала службы ПРЕДОХРАНИТЕЛЬНОГО УРОВНЯ.
- Обычно для связи между двумя подсистемами, которые могут находиться даже на разных серверах, требуется какой-то коммуникационный механизм. В этом случае приходится решать, где разместить средства коммуникации. Если к другой подсистеме нет доступа, эти связующие звенья придется поместить между ФАСАДНЫМ ОБЪЕКТОМ и другой подсистемой. Но если ФАСАДНЫЙ ОБЪЕКТ можно интегрировать с другой подсистемой напрямую, то хороший вариант — поместить связующее коммуникационное звено между АДАПТЕРОМ и ФАСАДНЫМ ОБЪЕКТОМ, потому что протокол ФАСАДНОГО ОБЪЕКТА должен быть проще, чем то, что он скрывает. Бывают также случаи, когда весь ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ может размещаться вместе с другой подсистемой, располагая механизмы коммуникации или распределения между вашей подсистемой и СЛУЖБАМИ, образующими интерфейс ПРЕДОХРАНИТЕЛЬНОГО УРОВНЯ. Эти решения по реализации и установке приложений нужно принимать сугубо прагматически; они не влияют на концептуальную роль ПРЕДОХРАНИТЕЛЬНОГО УРОВНЯ.
- Если у вас все-таки есть доступ к другой подсистеме, то иногда работу может облегчить некоторый ее рефакторинг. В частности, можно попытаться написать более явные интерфейсы для тех функций, которыми вы будете пользоваться, и начать эту работу с автоматизированных тестов, если возможно.
- Если требования к интеграции очень строги, стоимость трансляции сильно возрастает. Тогда в модели проектируемой подсистемы придется выбирать такие решения, чтобы приблизить ее к внешней системе и тем самым облегчить трансляцию. Это нужно делать осторожно, не подвергая опасности целостность модели, и притом избирательно, только в тех случаях, когда трансляция представляет особые трудности. Если такой подход кажется самым естественным решением для значительной и важной части проблемы, то подумайте, не стоит ли применить в подсистеме архитектуру КОНФОРМИСТ и убрать трансляцию вовсе.

- Если другая подсистема устроена просто или имеет четкий интерфейс, то ФАСАДНЫЙ ОБЪЕКТ может не понадобиться.
- В ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ можно добавить некоторые рабочие функции, если они *специфичны именно для взаимосвязи между двумя подсистемами*. На ум приходят такие две полезные возможности, как отслеживание использования внешней системы или трассировка для отладки обращений к интерфейсу.

Помните, что ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ — это средство связи между двумя ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ. Обычно в таком случае мы представляем себе систему, написанную кем-то другим; мы не полностью понимаем ее и практически не контролируем. Но это не единственная ситуация, в которой бывает нужна "прокладка" между системами. Бывает так, что имеет смысл соединить через ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ даже две системы своей собственной разработки, если в основе их лежат разные модели. Предположительно, в таком случае вы имеете полную власть над обеими сторонами и можете обойтись простым уровнем трансляции. Все же, если два ОГРАНИЧЕННЫХ КОНТЕКСТА ведут ОТДЕЛЬНОЕ СУЩЕСТВОВАНИЕ, но между ними требуется некоторая интеграция, ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ может уменьшить возникающие трения.

Пример

Старое приложение заказа грузов

Чтобы первая версия системы была небольшой и вышла в свет быстро, мы напишем минимально необходимое приложение, которое позволяет задать параметры поставки груза и передать их в старую систему резервирования через трансляционный уровень, поддерживающий оформление заказа и разные вспомогательные операции. Поскольку наш трансляционный уровень строится специально для того, чтобы защитить новую модель от влияния устаревшей архитектуры, он является именно ПРЕДОХРАНИТЕЛЬНЫМ УРОВНЕМ (ANTICORRUPTION LAYER).

В первоначальном варианте ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ принимает объекты, представляющие поставку груза, преобразует их, передает в старую систему и делает заказ. Затем он получает подтверждение и транслирует его обратно в объект подтверждения из новой архитектуры. Такая изоляция позволит нам разработать новое приложение практически независимо от старого, хотя в трансляцию придется вложить много усилий.

С выходом каждой новой версии новая система может либо брать на себя все больше функций старой, либо добавлять новые, не заменяя имеющиеся — это уж как будет решено. Подобная гибкость, а также возможность непрерывно пользоваться комбинированной системой, при этом продолжая вносить изменения, вероятно, оправдывает расходы на построение ПРЕДОХРАНИТЕЛЬНОГО УРОВНЯ.

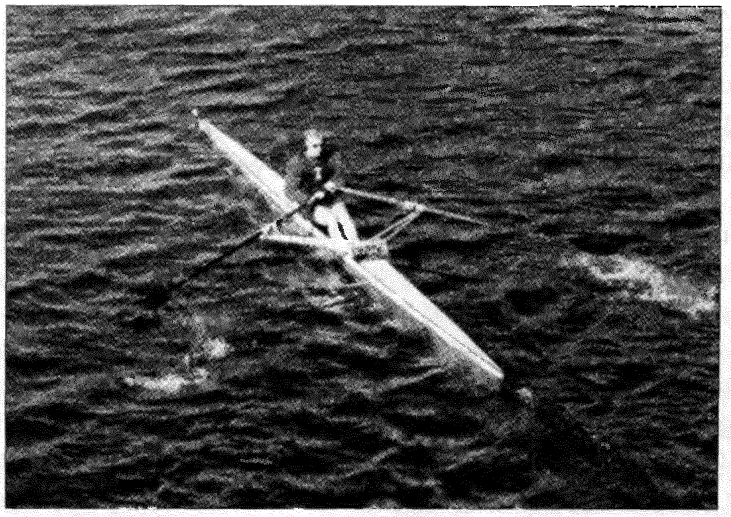
Поучительная история

Чтобы защитить свои границы от набегов воинственных кочевых племен, древние китайцы построили Великую стену. Она не выполнила свою задачу как непроницаемый барьер, но позволила урегулировать торговлю с соседями и стать препятствием для врагов. Две тысячи лет она определяла границу и отгораживала аграрную китайскую цивилизацию от царящего снаружи хаоса, чем и помогла ей сформироваться.

Хотя Великая китайская стена и способствовала становлению китайской самобытной культуры, все же ее строительство потребовало огромных расходов и разорило как минимум одну династию. Преимущества стратегии изоляции следует соизмерять с ее ценой. Бывает, что нужно проявить прагматизм и внести в модель тщательно выверенные исправления, чтобы состыковать ее с внешними системами.

Любая интеграция требует затрат: и полноценная НЕПРЕРЫВНАЯ ИНТЕГРАЦИЯ (CONTINUOUS INTEGRATION) внутри одного ОГРАНИЧЕННОГО КОНТЕКСТА (BOUNDED CONTEXT), и несколько менее тесная связь через ОБЩЕЕ ЯДРО (SHARED KERNEL) или в ГРУППАХ "ЗАКАЗЧИК-ПОСТАВЩИК" (CUSTOMER/SUPPLIER DEVELOPMENT TEAMS), и односторонний КОНФОРМИСТ (CONFORMIST), и перестраховочный ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ (ANTI-CORRUPTION LAYER). Интеграция может быть очень ценной и полезной, но она всегда дорого стоит. Поэтому не помешает уверенность в том, что она действительно необходима...

Отдельное существование



Требования к приложению надо нещадно разделять по областям действия. Два набора функций, между которыми нет обязательной, неизбежной взаимосвязи, вполне можно отделить друг от друга.

* * *

Интеграция всегда обходится дорого, а отдача от нее не всегда велика.

Интеграция не только требует обычных затрат на координирование групп разработчиков, но еще и заставляет идти на компромиссы. Простая специализированная модель, которая могла бы послужить конкретной цели, "вынуждена уступить дорогу" более абстрактной модели, способной справиться с любыми ситуациями. Возможно, какая-нибудь совершенно другая технология могла бы обеспечить нужные функции очень легко, но ее было бы трудно интегрировать. А может быть, с какой-то группой так трудно работать, что любые попытки других разработчиков сотрудничать с ними кончаются неудачей.

Во многих обстоятельствах существенной отдачи от интеграции получить не удается. Если две функциональные части системы не нуждаются в операциях друг друга или взаимодействии между объектами, с которыми соприкасаются они обе, или совместном использовании данных, то интеграция даже через трансляционный уровень может оказаться необязательной. Функциональные возможности не обязаны интегрироваться в одно целое только потому, что они совместно используются в каком-то случае.

Объявите ОГРАНИЧЕННЫЙ КОНТЕКСТ **(**BOUNDED CONTEXT**) никак не связанным с другими контекстами. Это позволит разработчикам найти простые узкоспециализированные решения в данном ограниченном пространстве.**

Нужные функции могут быть объединены на уровне ПО среднего уровня (*middleware*) или на уровне интерфейса, но совместного использования алгоритмической

логики не будет, и пересылка данных между трансляционными уровнями сведется к минимуму — хотя лучше к нулю.

Пример

"Урезание" проекта по обслуживанию страхования

Одна группа взялась за разработку новой программы по управлению страховыми претензиями, которая бы интегрировала в единую систему все, что нужно агенту по обслуживанию клиентов или оценщику страховых убытков. Год напряженных усилий ни к чему не привел. Их победило сочетание "мандража"[5] и слишком тщательное заблаговременное выстраивание инфраструктуры — в результате им оказалось практически нечего предъявить нетерпеливому начальству. Если же говорить серьезнее, разработчики не рассчитали масштаба и размаха того, что они пытались сделать.

Новый руководитель проекта загнал всех в комнату на неделю и заставил писать новый план. Вначале было составлено техническое задание, т.е. список требований. Разработчики попытались оценить их трудность и расставить по важности, после чего самые трудные и маловажные были безжалостно выброшены. В оставшемся списке стали наводить порядок. За неделю было принято много остроумных решений, но в конце концов важным оказалось только одно. В какой-то момент наступило понимание, что *есть такие функции программы, которые мало что выигрывают от интеграции.* Например, оценщикам ущерба требовался доступ к некоторым существующим базам данных, а их текущий способ доступа был очень неудобным. Но *хотя пользователям эти данные были нужны, никакие остальные функции предложенной программной системы ими не пользовались.*

Члены группы предложили самые разные способы для облегчения доступа к данным. В одном варианте ключевой отчет экспортировался в HTML и выкладывался во внутренней сети. В другом варианте оценщикам предлагался специальный запрос к базе данных, написанный с использованием одного стандартного пакета программ. Все эти функции можно было интегрировать, организовав ссылки на странице в сети или добавив кнопки на рабочий стол пользователя.

Группа начала серию небольших проектов, интеграция между которыми ограничивалась запуском из общего меню. Несколько ценных функций были реализованы буквально за сутки. Избавившись от бремени этих дополнительных возможностей, разработчики остались с дистилированным набором требований, который дал им на время надежду, что когда-нибудь будет готово и основное приложение.

Все могло бы так и произойти, но, к сожалению, группа скатилась к старым привычкам и снова впала в ступор. В конце концов от них только и осталось ценного, что те несколько маленьких приложений, которые первыми получили ОТДЕЛЬНОЕ СУЩЕСТВОВАНИЕ.

* * *

Принятие решения об ОТДЕЛЬНОМ СУЩЕСТВОВАНИИ сразу же исключает некоторые варианты выбора. Хотя при непрерывном рефакторинге рано или поздно отменяется любое решение, все-таки трудно интегрировать в одно целое модели, развивавшиеся в полной изоляции друг от друга. Если интеграция со временем оказывается нужной, то приходится создавать трансляционные уровни, иногда очень сложные. Впрочем, с этим любом случае приходится сталкиваться.

[5] Автор употребляет звучную игру слов *analysis paralysis*, которая дословно означает "паралич анализа", а по сути то же самое: невозможность двинуться с места из-за пустых страхов. *Примеч. перев.*

Теперь давайте вернемся к интеграционным отношениям и рассмотрим способы выведения интеграции на более масштабный уровень...

Службы с открытым протоколом

Как правило, в любом ОГРАНИЧЕННОМ КОНТЕКСТЕ (BOUNDED CONTEXT) определяется по одному трансляционному уровню для каждого компонента за пределами КОНТЕКСТА, с которым необходимо интегрироваться. Если интеграция делается "на один раз", то подход с добавлением одного трансляционного уровня для каждой внешней системы позволяет предохранить свою модель с минимальными издержками. Но если ваша подсистема вдруг становится широко востребованной, может потребоваться более гибкий подход.

* * *

Если подсистему необходимо интегрировать со многими другими, необходимость построения отдельного транслятора для каждой из других систем становится тяжким бременем. Все больше компонентов приходится дорабатывать, а потом беспокоиться о возрастающем количестве изменений.

Бывает, что разработчики делают одно и то же много раз, снова и снова. Если с подсистемой требуется какая-то интеграция, то не исключено, что ее можно описать в виде набора СЛУЖБ (SERVICES), отвечающих стандартным потребностям других подсистем.

Намного труднее спроектировать протокол настолько четкий и ясный, чтобы его поняли и использовали сразу несколько групп разработчиков. Эта работа окупается только тогда, когда ресурсы подсистемы можно представить в виде связного набора подсистем, и при этом требуется много раз выполнять интеграцию. В таких обстоятельствах есть большая разница между сопровождением и возможностью дальше разрабатывать систему.

Определите протокол, который бы предоставил доступ к вашей подсистеме как к набору СЛУЖБ (SERVICES). Сделайте протокол открытым, чтобы все желающие интегрироваться смогли бы им воспользоваться. Совершенствуйте и расширяйте протокол для учета новых интеграционных требований, кроме тех случаев, когда у отдельных групп возникают особо экзотические запросы. В таких случаях можно дописать одноразовый транслятор в дополнение к протоколу, чтобы основной общий протокол оставался простым и единообразным.

* * *

Такая формализация коммуникации подразумевает, что у разных моделей имеется некий общий словарь — основа для интерфейсов соответствующих СЛУЖБ. В результате сторонние подсистемы оказываются привязанными к модели с ОТКРЫТЫМ ПРОТОКОЛОМ (OPEN HOST), и это вынуждает их разработчиков изучать конкретный диалект, используемый группой разработки протокола. В некоторых случаях снизить зависимость и облегчить взаимопонимание можно, если применить в качестве промежуточной модели достаточно известный и ОБЩЕДОСТУПНЫЙ ЯЗЫК (PUBLISHED LANGUAGE)...

Общедоступный язык

Трансляция между моделями двух разных ОГРАНИЧЕННЫХ КОНТЕКСТОВ (BOUNDED CONTEXTS) требует наличия какого-то общего языка.

* * *

Если двум разным моделям предметных областей необходимо сосуществовать и обмениваться информацией, процесс трансляции сам по себе может усложниться, плохо

поддаваться документированию и даже пониманию. Если мы строим новую систему, то обычно полагаем, что наша новая модель — самая лучшая из имеющихся, поэтому и ориентируемся на прямую трансляцию в ее понятия. Но иногда мы дорабатываем набор старых систем, пытаясь интегрировать их. Выбор одной из двух одинаково беспорядочных моделей — это не более чем выбор меньшего из двух зол.

А вот другая ситуация. Если два предприятия (или две отрасли, или два вида деятельности) должны обмениваться информацией между собой, как им это делать? Предполагать, что одна сторона примет модель предметной области другой, во-первых, нереалистично, а во-вторых, может быть, и нежелательно для обеих сторон. Модель предметной области разрабатывают, для того чтобы она решала задачи своих пользователей. Она может содержать такие средства, которые без нужды усложнят коммуникацию с другой системой. Также, если модель одного из приложений используется как среда коммуникации, ее нельзя будет легко изменить под какие-либо новые потребности. Наоборот, ей придется быть как можно стабильнее, чтобы выполнять свою роль средства коммуникации.

Прямая трансляция из существующей модели предметной области или в нее может оказаться неудачным решением. Такие модели могут быть слишком сложными или плохо факторизуемыми. Они также бывают недокументированными. Если одна используется в качестве языка обмена данными, то она фактически замораживается в развитии и больше не сможет отвечать новым потребностям, возникающим в ходе разработки.

Для многосторонней интеграции СЛУЖБА С ОТКРЫТЫМ ПРОТОКОЛОМ (OPEN HOST SERVICE) использует стандартизированный протокол. В ней для коммуникации между системами применяется модель предметной области, даже если она внутри самих систем и не задействована. Теперь нужно сделать шаг вперед и вывести язык этой модели на общедоступный уровень или же найти такой язык, который уже общедоступен. Под общедоступностью следует понимать наличие свободного доступа к нему со стороны сообщества, заинтересованного в пользовании им, а также документированность, достаточную для того, чтобы совместить его независимые интерпретации.

В недавние годы мир электронной коммерции с энтузиазмом воспринял новую технологию: язык Extensible Markup Language (XML), обещавший значительно облегчить обмен данными[6]. Бесценное свойство этого языка состоит в том, что посредством *определения типа документа* (*Document Type Definition*, *DTD*) или шаблонных схем XML можно ввести формальную спецификацию специализированного профессионального языка, на который могут транслироваться (переводиться) данные. Уже начали создаваться различные отраслевые группы разработчиков, ставящие своей целью создание единого стандартного DTD для своей отрасли — чтобы, скажем, химическую формулу или генетический код можно было свободно передавать заинтересованным сторонам. Такие группы фактически создали совместную модель предметной области в виде определения ее языка.

Пользуйтесь хорошо документированным, известным всем сторонам языком, который может представить необходимую информацию о предметной области в виде общей среды коммуникации, так что на этот язык и с него можно будет при необходимости переводить (транслировать).

Такой язык не обязательно создавать с нуля. Много лет назад меня подрядила на работу компания, в программном продукте которой, написанном на Smalltalk, данные хранились в СУБД DB2. Компания хотела расширить возможности сбыта своего программного обеспечения, продавая его пользователям без лицензии на DB2. Поэтому они наняли меня создать интерфейс к Btrieve — менее тяжеловесной СУБД, лицензия на пользование исполнительным механизмом которой предоставлялась бесплатно. Btrieve — не полностью реляци-

[6] Это обещание было выполнено. Язык XML не подвел и продолжает расширять область своего применения. 3-е издание данной книги вышло в 2004 году. *Примеч. перев.*

онная СУБД, но мой клиент все равно пользовался только небольшой частью возможностей DB2 и как раз попадал в наименьший общий знаменатель двух СУБД. Программисты компании надстроили над DB2 кое-какие абстракции, реализующие хранение объектов. Я решил воспользоваться их результатами как интерфейсом для моего компонента Btrieve.

Этот подход сработал. Программа четко интегрировалась с системой моего клиента. Однако отсутствие формального технического задания или документации на абстракции постоянно существующих объектов в архитектуре клиента потребовало большой работы по составлению технического задания на новый компонент. Невелик был и шанс перенести этот компонент куда-нибудь с целью перевода других приложений с DB2 на Btrieve. Новое программное обеспечение еще глубже закрепило модель поддержания постоянного существования объектов, принятую в компании, что затруднило ее потенциальный рефакторинг.

Лучше для них было бы определить, какое подмножество интерфейса DB2 фактически используется компанией, и реализовать его поддержку. Интерфейс DB2 состоит из языка SQL и нескольких закрытых протоколов. Он очень сложен, но четко специализирован и подробно документирован. Но проблема с его сложностью не стояла бы так остро, поскольку использоваться должна была только небольшая часть интерфейса. Если бы был разработан компонент, эмулирующий нужное подмножество интерфейса DB2, его можно было бы фактически документировать для разработчиков, просто указав, какое именно подмножество реализуется. Приложение, в которое он интегрировался, уже "общалось" с DB2, так что понадобилось бы совсем немного дополнительной работы. В будущем перепроектировании уровня постоянного хранения объектов разработчики были бы связаны только использованием подмножества DB2 — точно так же, как и до начала своих усовершенствований.

Интерфейс DB2 — это пример ОБЩЕДОСТУПНОГО ЯЗЫКА (PUBLISHED LANGUAGE). В данном случае две модели не находятся в одной предметной области, но все принципы все равно применимы. Поскольку одна из сотрудничающих моделей уже является носителем ОБЩЕДОСТУПНОГО ЯЗЫКА, нет необходимости вводить еще и третий язык.

Пример

Общедоступный язык в химии

Для каталогизации, анализа и манипулирования химическими формулами используется бесчисленное множество программ как в промышленности, так и в науке. Обмен данными всегда был делом нелегким, потому что практически каждая программа имеет свою модель предметной области для представления химических структур. К тому же большинство из них написано на таких языках, как FORTRAN, которые в любом случае не слишком точно отображают модель предметной области. Когда кто-нибудь хочет обменяться или поделиться данными, ему приходится разбираться в подробностях устройства базы данных из другой системы и вырабатывать какую-то схему трансляции.

И тут "на сцене" появляется язык химического кодирования — Chemical Markup Language (CML). Этот диалект языка XML задуман как общий язык обмена данными в этой сфере, а разработан и сопровождается он группой, которая представляет и научные, и производственные круги [20].

Химическая информация характеризуется большой сложностью и разнообразием. Она все время изменяется по мере новых открытий и разработок. Поэтому был разработан язык, с помощью которого можно описать самые основы, такие как химические формулы органических и неорганических молекул, белковых последовательностей, спектры и физические величины.

После того как язык опубликован и стал общедоступным, можно заняться разработкой программ, с которыми раньше не стоило и возиться, раз они все равно годились только для

одной базы данных. Например, было разработано приложение на Java под названием JUMBO Browser, которое генерирует графические представления химических структур, записанных на языке CML. Таким образом, если хранить данные в формате CML, появляется возможность пользоваться подобными средствами визуализации.

На самом деле CML получил двойное преимущество, воспользовавшись как основой XML, своего рода "общедоступным метаязыком". Усвоение CML облегчается знакомством с XML. Его реализация упрощается благодаря наличию различных готовых утилит, например, синтаксических анализаторов, а в дополнение к документации существует множество книг по всем аспектам работы с XML.

Ниже показан небольшой пример в формате CML. В нем мало что поймет неспециалист наподобие меня, но основной принцип все же понятен.

```
<CML.ARR ID="array3" EL.TYPE=FLOAT
      NAME="ATOMIC ORBITAL ELECTRON POPULATIONS"
      SIZE=30 GLO.ENT=CML.THE.AOEPOPS>
1.17947 0.95091 0.97175 1.00000 1.17947 0.95090 0.97174 1.00000
1.17946 0.98215 0.94049 1.00000 1.17946 0.95091 0.97174 1.00000
1.17946 0.95091 0.97174 1.00000 1.17946 0.98215 0.94049 1.00000
0.89789 0.89790 0.89789 0.89789 0.89790 0.89788
</CML.ARR>
```

* * *

Унификация слона

Шесть седовласых мудрецов
Сошлись из разных стран.
К несчастью, каждый был незряч,
Зато умом блистал.
Они исследовать слона
Явились в Индостан.

Один погладил бок слона.
Довольный тем сполна,
Сказал он: "Истина теперь
Как божий день видна:
Предмет, что мы зовем слоном, —
Отвесная стена!"
...
А *третий* хобот в руки взял
И закричал: "Друзья!
Гораздо проще наш вопрос,
Уверен в этом я!
Сей слон — живое существо,
А именно змея!"

Мудрец *четвертый* обхватил
Одну из ног слона
И важно молвил: "Это ствол,
Картина мне ясна!
Слон — дерево, что зацветет,
Когда придет весна!"
...
Тем временем *шестой* из них
Добрался до хвоста.
И рассмеялся от того,
Как истина проста.
"Ваш слон — веревка. Если ж нет
Зашейте мне уста!"

А как известно, мудрецам
Присущ упрямый нрав.
Спор развязав, они дошли
Едва ль не до расправ.
Но правды ни один не знал,
Хотя был в чем-то прав.
...

(Из стихотворения "Слепцы и слон" Дж. Г. Сакса (1816 1887) по мотивам индийской притчи[7].)

[7] Перевод с англ. Б.Б. Авадхута Махараджа. — *Примеч. перев.*

В зависимости от того, какие цели преследуются во взаимодействии со слоном, разные слепцы могут даже продвинуться в решении своих задач, пусть они и не достигли согласия по поводу сущности слона. Если интеграция не требуется, то отсутствие унификации в моделях не имеет никакого значения. Если же какая-то интеграция все-таки нужна, то даже если слепцы по-прежнему не согласны друг с другом, сам факт отсутствия согласия уже может оказаться ценным знанием. По крайней мере, играя в "испорченный телефон", слепцы будут осознавать это.

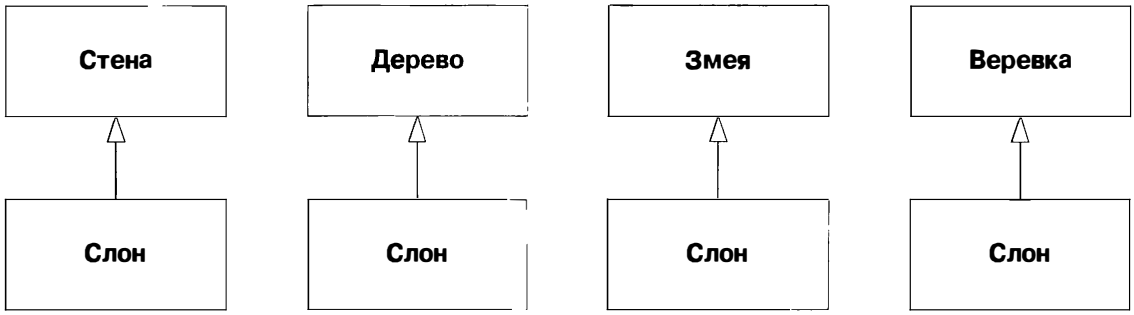
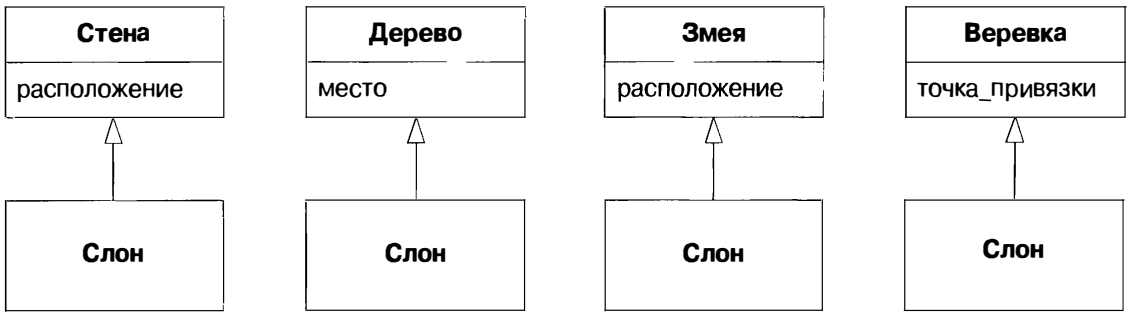


Рис. 14.9. Четыре контекста: никакой интеграции

Диаграммы на рис 14.9 это UML-представления моделей слона, которые построили слепцы. После организации отдельных ОГРАНИЧЕННЫХ КОНТЕКСТОВ (BOUNDED CONTEXTS) ситуация, в которой они находятся, прояснилась настолько, что появился смысл общаться друг с другом о некоторых аспектах слона, которые могут интересовать сразу всех, например о его местонахождении.



Трансляция: {`Стена.расположение↔Дерево.место↔Змея.расположение↔ Веревка.точка_привязки`}

Рис. 14.10. Четыре контекста: минимальная интеграция

Если слепцы захотят обменяться дополнительной информацией о слоне, появится смысл в использовании общего ОГРАНИЧЕННОГО КОНТЕКСТА. Но унификация в корне отличных, несоизмеримых моделей — нетривиальная задача. Ни один из слепцов не хочет отказаться от своей модели и принять чужую. Ведь человек, ухвативший слона за хвост, *знает*, что слон не похож на дерево, и такая модель будет для него бессмысленной и бесполезной. Унификация нескольких моделей почти всегда означает создание новой.

Призвав на помощь воображение и поспорив еще (скорее всего, на повышенных тонах), слепцы смогли бы в конце концов понять и признать, что они описывают и моделируют разные части большего целого. Во многих случаях унификация по типу "часть–целое" может и не потребовать подобных дополнительных усилий. По крайней мере, на первом этапе интеграции требуется только понять, как соотносятся между собой части. Для каких-то целей может оказаться достаточно считать слона стеной, стоящей на стволах деревьев, с веревкой на одном конце и змеей на другом.

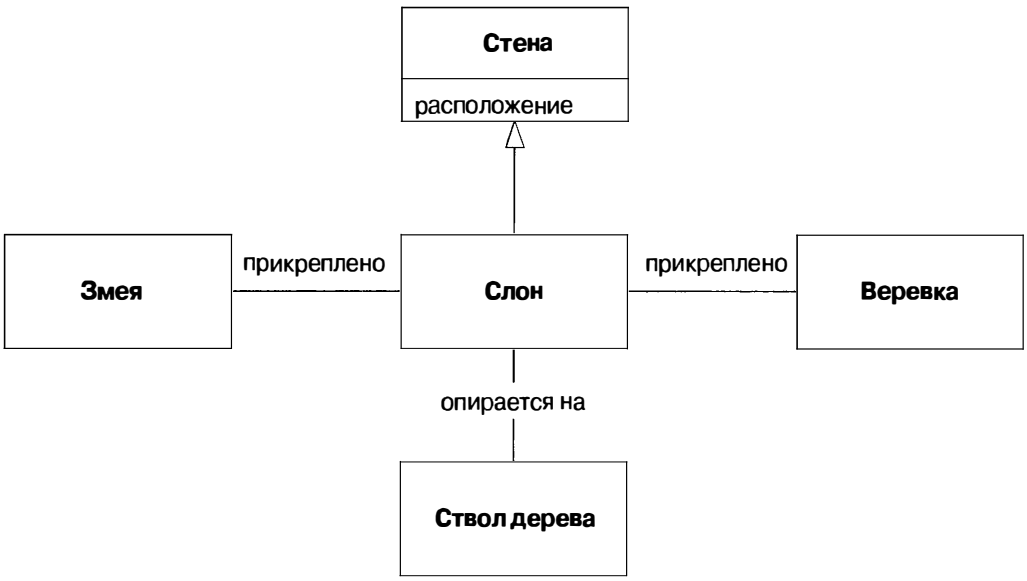


Рис. 14.11. Один контекст: грубая интеграция

Унификация различных моделей слона проще, чем большинство подобных вариантов. К сожалению, это, скорее, исключение, когда две модели просто описывают разные части целого; чаще всего это лишь один из аспектов различия между ними. Все усложняется, когда разные модели воспринимают одну и ту же часть целого по-разному. Если бы двое потрогали хобот, и один описал бы его как змею, а другой — как пожарный шланг, ситуация усложнилась бы. Ни один не принял бы модели другого, поскольку она противоречила бы его собственному опыту. Фактически им нужна новая абстракция, которая бы объединила "живость" змеи с функцией подачи воды, присущей шлангу. Причем в этой абстракции должны быть исключены неподходящие к случаю характеристики моделей: например, наличие ядовитых зубов или способность отделяться от тела, скручиваться в кольцо и укладываться в пожарную машину.

Хотя нам удалось объединить части в целое, итоговая модель получилась грубой и неуклюжей. Она несвязна и никак не обозначает контуры лежащей в ее основе предметной области. Но процесс непрерывного и постепенного ее совершенствования может привести к новым выводам, на основе которых можно будет углубить модель. Стимулировать движение к новой модели может также новое техническое задание. Если слон начнет двигаться, теория "дерева" развалится, и наши слепые моделировщики могут скачкообразно придти к понятию "ног".

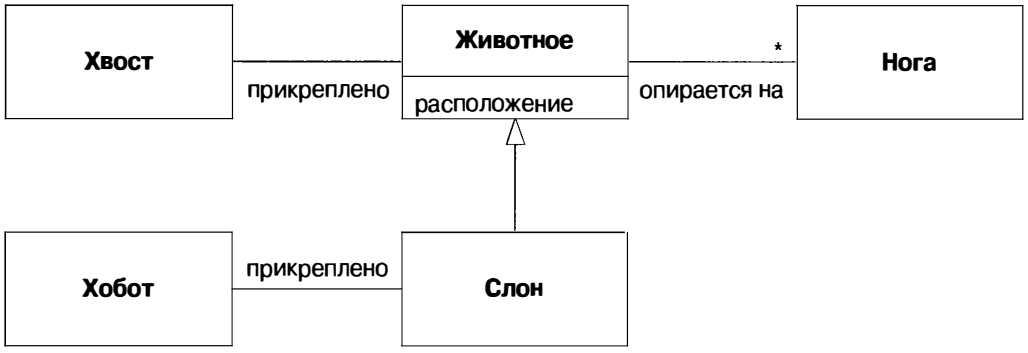


Рис. 14.12. Один контекст: углубленная модель

В этом втором цикле интеграции моделей возникла тенденция отбрасывания случайных или некорректных аспектов отдельных моделей, а также создания новых понятий:

“животного” с частями “хобот”, “нога”, “туловище” и хвост. Каждая из этих частей имеет собственные свойства и четкие взаимосвязи с другими частями. Успешная унификация моделей в значительной мере опирается на принцип минимализма. Хобот слона одновременно и нечто большее, и нечто меньшее, чем змея. Но “меньшее” здесь, вероятно, важнее, чем “большее”. Лучше обойтись без умения разбрызгивать воду, чем приобрести неправильное свойство “ядовитые зубы”.

Если цель состоит только в том, чтобы найти слона, то достаточно ввести трансляцию между выражениями местонахождения в каждой модели. Но если требуется более тесная интеграция, то не обязательно пытаться получить совершенную модель уже в первой версии. Для каких-то целей будет достаточно представить слона как стену, стоящую на стволах деревьев, с веревкой на одном конце и змеей на другом. Позднее, под влиянием новых требований, лучшего понимания и коммуникации, модель можно будет усовершенствовать.

Признавать существование нескольких противоречащих друг другу моделей предметной области — это означает просто принимать реальность такой, какой она есть. Определив контекст, внутри которого применима каждая из моделей, можно сохранить целостность каждой из них и четко увидеть смысл любого из интерфейсов, которые можно построить между двумя моделями. У слепцов нет возможности увидеть всего слона, но их конкретные задачи вполне можно решить, если они признают неполноту своего восприятия.

Выбор стратегии построения контекстов

Всегда важно в нужный момент иметь перед собой КАРТУ КОНТЕКСТОВ (CONTEXT MAP), отражающую текущую реальность. Но как только такая карта построена, можно браться за изменение самой этой реальности. Можно приступать к сознательному выбору границ КОНТЕКСТА и его взаимосвязей. В этом разделе даются некоторые рекомендации, как это делать.

Уровень принятия решений: разработчики или выше

Прежде всего разработчики должны принять решение, где именно определить ОГРАНИЧЕННЫЕ КОНТЕКСТЫ и какие взаимосвязи между ними установить. Эти решения должны приниматься *всей группой разработчиков.* Или, по крайней мере, они должны доводиться до сведения каждого члена группы и быть ему понятными. Такие решения на самом деле часто бывают связаны с договорными отношениями за пределами одной группы. В идеале решение о том, следует расширять или разделять тот или иной КОНТЕКСТ, нужно принимать, взвесив издержки и преимущества как независимой работы группы, так и прямой тесной интеграции, и найдя некий компромисс между ними. На практике же интеграцию систем часто определяют политические отношения между группами разработчиков. Технически полезная унификация оказывается невозможной из-за системы отчетности. Неудобное слияние может диктоваться системой руководства. Не всегда удается получить то, что хочешь, но надо стараться оценить хотя бы какую-то часть издержек, донести эту информацию до других и предпринять меры по исправлению положения. Начните с реальной карты контекстов и проектируйте изменения на ней с сугубо прагматической точки зрения.

Помещение самих себя в контекст

Когда мы работаем над программным проектом, нас интересуют больше всего те части системы, которые затрагивает наша группа (“проектируемая система”), а затем те системы, с которыми она поддерживает связь. Как правило, проектируемая система ока-

зывается встроенной в один или два ОГРАНИЧЕННЫХ КОНТЕКСТА, над которыми работают главные разработчики, и может быть, еще в один или два вспомогательных. Кроме того, есть связи и отношения между этими КОНТЕКСТАМИ и внешними системами. Это обычная картина, примерно представляющая то, что встречается чаще всего.

Но на самом деле и *мы сами* входим в тот КОНТЕКСТ, над которым работаем. Это обязательно нужно отразить в нашей КАРТЕ КОНТЕКСТОВ. В этом нет никакой проблемы, если мы осознаем свою субъективность и знаем, когда мы нарушаем границы применимости такой карты.

Преобразование границ

Что касается определения границ ОГРАНИЧЕННЫХ КОНТЕКСТОВ, то тут существует множество вариантов и неограниченное разнообразие ситуаций. Обычно в таких делах ищется баланс между некоторыми из перечисленных ниже факторов.

Преимущества больших контекстов:

- различные пользовательские операции связаны между собой более плавно и естественно, когда как можно больше разных объектов и операций охвачены одной унифицированной моделью;
- легче понять одну связную модель, чем две разные и уровень отображения между ними;
- трансляция между двумя моделями может оказаться сложной (а то и невозможной);
- общий язык стимулирует четкое взаимодействие в группе разработчиков.

Преимущества малых контекстов:

- снижаются издержки коммуникации между разработчиками;
- НЕПРЕРЫВНУЮ ИНТЕГРАЦИЮ (CONTINUOUS INTEGRATION) проще выполнять в малых группах с небольшими базами кода;
- в больших контекстах могут потребоваться более универсальные и абстрактные модели, требующие редко встречающейся квалификации разработчиков;
- небольшие модели могут обслуживать специальные потребности или соответствовать жаргону специализированных групп пользователей, а также узкоспециальных диалектов ЕДИНОГО ЯЗЫКА (UBIQUITOUS LANGUAGE).

Глубокая и тесная интеграция функций между разными ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ на практике неуместна. Интеграция должна ограничиваться теми частями модели, которые можно строго сформулировать на языке другой модели, и даже такой уровень интеграции может потребовать значительных усилий. Это имеет смысл только тогда, когда интерфейс между двумя системами будет невелик по объему.

Принятие того, что нельзя изменить: контуры внешних систем

Лучше всего начинать с простых решений. Некоторые подсистемы, очевидно, окажутся за пределами всех ОГРАНИЧЕННЫХ КОНТЕКСТОВ проектируемой системы. Это, например, могут быть устаревшие системы, не подлежащие немедленной замене, а также внешние системы, которые предоставляют нужные услуги. Все это можно с самого начала распознать и отделить от своей собственной архитектуры.

Здесь следует быть осторожным в допущениях. Удобно считать, что каждая такая система образует свой собственный ОГРАНИЧЕННЫЙ КОНТЕКСТ, но большинство внешних

систем слабо вписывается в это определение. Прежде всего ОГРАНИЧЕННЫЙ КОНТЕКСТ определяется *намерением* унифицировать модель в определенных границах. Может быть, вы имеете право и возможность сопровождать старую систему, и тогда вы можете декларировать намерение. Или же группа, отвечающая за систему, хорошо скоординирована и следует неформальной процедуре НЕПРЕРЫВНОЙ ИНТЕГРАЦИИ. Но это не следует считать само собой разумеющимся. Проверьте все сами, и если разработка не слишком хорошо скоординирована, то соблюдайте особую осторожность. В таких системах обычное дело — обнаружить семантические противоречия между разными их частями.

Взаимоотношения с внешними системами

Здесь применимы три из описанных шаблонов. Прежде всего нужно рассмотреть ОТДЕЛЬНОЕ СУЩЕСТВОВАНИЕ (SEPARATE WAYS). Да, если бы интеграция не была нужна, эти системы вообще не затрагивались бы. Но нужно иметь полную уверенность. Может быть, будет достаточно предоставить пользователю легкий доступ к обеим системам? Интеграция дорого стоит и отвлекает усилия, поэтому освободите ваш проект от этого бремени, насколько возможно.

Если интеграция действительно существенна, можно попытаться выбрать между двумя крайностями: КОНФОРМИСТОМ (CONFORMIST) или ПРЕДОХРАНИТЕЛЬНЫМ УРОВНЕМ (ANTICORRUPTION LAYER). Вообще, быть КОНФОРМИСТОМ не очень-то приятно. Творческие порывы и выбор новых функциональных возможностей будут сильно ограничены. Проектируя совершенно новую большую систему, вряд ли имеет смысл придерживаться модели устаревшей или внешней системы (иначе почему мы создаем новую?). Однако модель старой системы может оказаться уместной для периферийных расширений большого программного продукта, который в целом сохранит свои ведущие позиции. Примерами могут служить небольшие организационные программы для поддержки принятия решений, часто разрабатываемые в среде Excel, и тому подобные простые утилиты. Если ваше приложение представляет собой расширение для существующей системы и должно иметь с ней большой интерфейс, то трансляция между КОНТЕКСТАМИ легко может превзойти по объему все функции приложения. Но даже здесь есть пространство для качественного проектирования, пусть мы и находимся в ОГРАНИЧЕННОМ КОНТЕКСТЕ другой системы. Если в основе другой системы лежит достаточно отчетливая модель предметной области, вы можете улучшить свою программную реализацию, сделав эту модель более явной, чем она была в старой системе. Для этого просто нужно строго ей следовать. Если вы решите принять архитектуру КОНФОРМИСТА, это придется делать искренне, от всей души. Ограничьтесь созданием расширения-дополнения, не модифицируя существующую модель.

Если же функции проектируемой вами системы выходят за рамки простого расширения-дополнения к существующей системе, если интерфейс с другой системой мал по объему или же если другая система очень плохо спроектирована, то обязательно понадобится свой собственный ОГРАНИЧЕННЫЙ КОНТЕКСТ. А это означает, что придется построить трансляционный уровень — возможно, даже ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ (ANTICORRUPTION LAYER).

Проектируемая система

Под *проектируемой системой* подразумевается то программное обеспечение, которое фактически разрабатывает ваша группа. В этих пределах можно определить несколько ОГРАНИЧЕННЫХ КОНТЕКСТОВ и внутри каждого выполнять НЕПРЕРЫВНУЮ ИНТЕГРАЦИЮ, чтобы поддерживать их унификацию. Но сколько контекстов нужно иметь? В каких отношениях они должны состоять? Ответы на эти вопросы не так триви-

альны, как в случае внешних систем, потому что здесь у нас больше свободы и контроля над ситуацией.

Можно поступить просто: ввести один ОГРАНИЧЕННЫЙ КОНТЕКСТ для всей проектируемой системы. Это хороший вариант, например, для группы менее чем из десяти человек, которая работает над тесно связанным набором функций.

По мере роста численности разработчиков НЕПРЕРЫВНАЯ ИНТЕГРАЦИЯ (CONTINUOUS INTEGRATION) становится все труднее (хотя я видел примеры ее поддержания и в довольно больших коллективах). Может быть, стоит подумать об ОБЩЕМ ЯДРЕ (SHARED KERNEL) и вынести относительно независимые наборы функций в отдельные КОНТЕКСТЫ, чтобы над каждым работало не более десяти человек. Если же все взаимоотношения между двумя такими группами направлены в одну сторону, можно организовать отношения по типу ГРУПП "ЗАКАЗЧИК-ПОСТАВЩИК" (CUSTOMER/SUPPLIER DEVELOPMENT TEAMS).

Бывает, что образ мышления у двух групп столь различен, что в процессе моделирования они все время вступают в конфликт. Может быть, они на самом деле хотят разного от своих моделей. Может быть, у них разная подготовка и общие знания на эту тему. А может быть, так устроено руководство данным проектом. Если причину конфликта устранить никак нельзя или незачем, можно позволить двум моделям "вести" ОТДЕЛЬНОЕ СУЩЕСТВОВАНИЕ (SEPARATE WAYS). Если же интеграция необходима, то можно разработать трансляционный уровень и сопровождать его совместными усилиями двух групп в качестве единственной точки НЕПРЕРЫВНОЙ ИНТЕГРАЦИИ. Это совсем другое дело, чем интеграция с внешними системами, где обычно приходится присоединять всю систему в неизменном виде через ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ (ANTICORRUPTION LAYER) и без особой поддержки с другой стороны.

В целом, на один ОГРАНИЧЕННЫЙ КОНТЕКСТ обычно приходится одна группа разработчиков. Одна группа может сопровождать несколько КОНТЕКСТОВ, но нескольким группам очень трудно (если вообще возможно) вместе работать над одним КОНТЕКСТОМ.

Учет особых случаев отдельными моделями

Различные группы в пределах одной области деятельности часто развивают свою собственную узкоспециализированную терминологию. Локальные жаргоны расходятся и начинают отличаться друг от друга, оставаясь очень точными и приспособленными к решению проблем в своих подобластях. Чтобы изменить такие жаргоны (например, введя единый терминологический стандарт в какой-то отрасли), нужно разрешить противоречия, а для этого придется много поработать. И даже тогда новая терминология может оказаться не настолько удачной, как уже существующая, отлаженная версия.

Можно попробовать учесть специальные случаи в отдельных КОНТЕКСТАХ, позволив моделям "вести" ОТДЕЛЬНОЕ СУЩЕСТВОВАНИЕ, за исключением НЕПРЕРЫВНОЙ ИНТЕГРАЦИИ трансляционных уровней. Вокруг таких моделей и специализированного жаргона, на котором они основаны, будут развиваться различные диалекты ЕДИНОГО ЯЗЫКА. Если у двух диалектов много общего, то, чтобы добиться нужной степени специализации и одновременно минимизировать затраты на трансляцию, можно воспользоваться ОБЩИМ ЯДРОМ (SHARED KERNEL).

Если интеграция не нужна или сравнительно ограничена, это позволяет продолжить использование собственной терминологии, избегая при этом порчи модели. Имеются здесь и свои риски с издержками:

- потеря общего языка ухудшает коммуникацию;
- интеграция требует дополнительных затрат;

- присутствует некоторое дублирование усилий по мере того, как развиваются разные модели одних и тех же процессов и объектов.

Но, возможно, самый большой риск состоит в том, чтобы отказаться от попыток изменений и оправдать любую узкоспециализированную, экзотическую модель. Зададимся вопросом: насколько необходимо перекроить данную конкретную часть системы, для того чтобы она решала особые задачи? Более того: *насколько ценен и важен жаргон этой конкретной группы пользователей*? Необходимо сопоставить преимущества независимой работы группы с рисками, связанными с трансляцией, и стараться рационализировать терминологические различия, в которых нет особого смысла.

Иногда возникает углубленная модель, которая может унифицировать различные языки и удовлетворить обе группы разработчиков. Но в том-то и подвох, что углубленные модели возникают (если возникают) на поздних этапах разработки, в результате значительных усилий и переработки знаний. Спланировать такую модель нельзя, можно только не упустить возможность, когда она возникает, изменить стратегию и выполнить рефакторинг.

Не забывайте, что там, где требования к интеграции велики, стоимость трансляции резко возрастает. Чтобы облегчить трансляцию, не прибегая к полной унификации, можно скоординировать усилия групп до некоторой степени — от точечных модификаций одного объекта со сложной трансляцией до введения ОБЩЕГО ЯДРА (SHARED KERNEL).

Установка системы

Координирование финальной сборки и установки сложной системы — это одна из тех скучных задач, которые почти всегда сложнее, чем они кажутся. Выбор стратегии построения ОГРАНИЧЕННОГО КОНТЕКСТА оказывает на установку свое влияние. Например, когда новые версии приложений устанавливаются ГРУППАМИ "ЗАКАЗЧИК-ПОСТАВЩИК" (CUSTOMER/SUPPLIER TEAMS), им приходится координироваться между собой, чтобы установить совместно протестированные версии. Тут учитывается и код, и перенос данных. Что касается распределенной системы, то в ней полезно разрабатывать трансляционные уровни между КОНТЕКСТАМИ совместно, в рамках одного процесса, чтобы не плодить параллельно существующих разных версий.

Даже установка компонентов одного и того же ОГРАНИЧЕННОГО КОНТЕКСТА может представлять трудности, если перенос данных занимает время или если распределенные системы нельзя обновить одновременно, отчего начинают параллельно сосуществовать две версии кода и данных.

В зависимости от среды и технологии установки приложения возникает множество разных технических соображений. Наиболее критические места вам укажут взаимосвязи между ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ — особенно выделяются в этом плане трансляционные интерфейсы.

Реализуемость плана установки имеет обратную связь с проведением границ между КОНТЕКСТАМИ. Если два КОНТЕКСТА соединены трансляционным уровнем, то один КОНТЕКСТ можно обновить так, чтобы новый трансляционный уровень обеспечивал тот же интерфейс ко второму КОНТЕКСТУ. Наличие ОБЩЕГО ЯДРА (SHARED KERNEL) накладывает гораздо более жесткие требования к координации, и не только при разработке, но и при установке. Упростить жизнь может ОТДЕЛЬНОЕ СУЩЕСТВОВАНИЕ.

Компромиссы

Подытожим высказанные выше рекомендации. Существует целый ряд стратегий унификации или интеграции моделей. В целом, следует искать компромисс между преимуществами интеграции функций в одно целое и необходимостью дополнительной ко-

ординации и коммуникации. На одной чаше весов — независимость действий, на другой — единообразие коммуникации. Чем более полная унификация требуется, тем больше контроля следует иметь над архитектурой всех участвующих в ней подсистем.

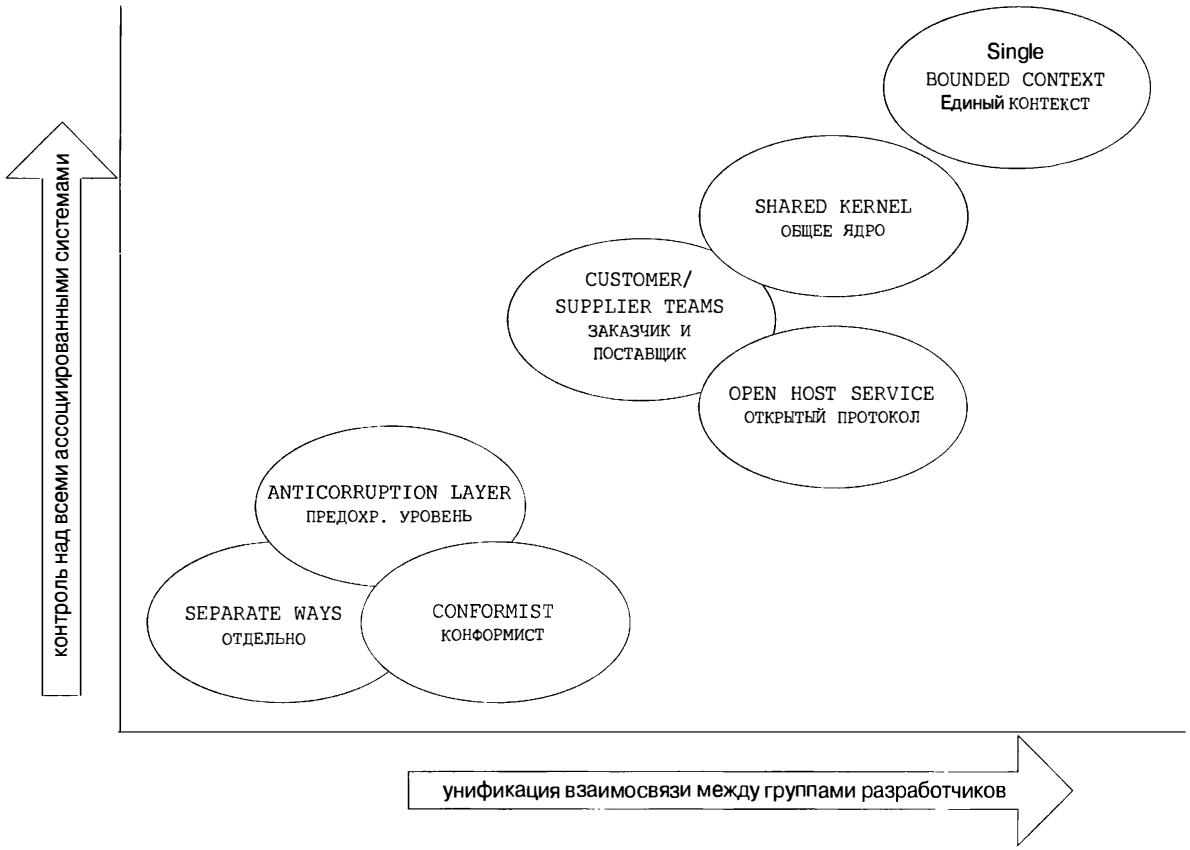


Рис. 14.13. Сравнительная характеристика шаблонов взаимоотношений между КОНТЕКСТАМИ

Если проект уже в работе

Бывает, что мы не начинаем новый проект, а используем уже существующий, находящийся в работе, и пытаемся его усовершенствовать. В этом случае первый шаг — это определить ОГРАНИЧЕННЫЕ КОНТЕКСТЫ (BOUNDED CONTEXTS) *в соответствии с текущей реальностью.* Это критический момент. Чтобы быть эффективным инструментом, КАРТА КОНТЕКСТОВ (CONTEXT MAP) должна отражать *реальную* практику разработчиков, а не *идеальную* организацию, нарисованную по вышеописанным рекомендациям.

После того как определены реальные, действующие в настоящий момент ОГРАНИЧЕННЫЕ КОНТЕКСТЫ и описаны взаимоотношения между ними, следующий шаг — это построить практику разработчиков как можно теснее *вокруг текущей организации.* Улучшайте НЕПРЕРЫВНУЮ ИНТЕГРАЦИЮ (CONTINUOUS INTEGRATION) в пределах КОНТЕКСТОВ. Выделите весь неупорядоченный трансляционный код в ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ (ANTICORRUPTION LAYER). Дайте имена ОГРАНИЧЕННЫМ КОНТЕКСТАМ и обязательно включите их в ЕДИНЫЙ ЯЗЫК проекта.

Вот теперь вы готовы рассматривать изменения границ и взаимосвязей. Эти изменения будут естественным образом основаны на тех же принципах, которые уже описаны здесь для новых проектов. Но их придется разбить на много мелких кусочков, выбирая их прагматически с точки зрения максимальной отдачи при минимуме усилий и разрушений.

В следующем разделе рассказывается, как именно нужно изменять границы КОНТЕКСТА, если уж вы решили это сделать.

Преобразования

Как и другие аспекты моделирования и проектирования, решения насчет ОГРАНИЧЕННЫХ КОНТЕКСТОВ (BOUNDED CONTEXTS) не являются окончательными и бесповоротными. Неизбежно возникают ситуации, в которых приходится изменять первоначальное решение насчет границ и взаимоотношений ОГРАНИЧЕННЫХ КОНТЕКСТОВ. В целом, разбивать КОНТЕКСТЫ на части довольно легко, а вот объединять их или изменять взаимоотношения между ними может оказаться сложнее. Я опишу несколько характерных трудных, но важных изменений. Такие преобразования обычно слишком велики, чтобы проделать их за один рефакторинг или даже за одну итерацию проекта. Поэтому я составил поэтапные планы преобразований, в которых они представлены как серии вполне управляемых модификаций. Это, конечно, всего лишь рекомендации, которые нужно приспособить к конкретным обстоятельствам.

Слияние контекстов: от отдельного существования к общему ядру

Расходы на трансляцию велики. Дублирование — слишком очевидная мера. Есть много соображений в пользу слияния ОГРАНИЧЕННЫХ КОНТЕКСТОВ. Но делать это трудно. Еще не поздно, но терпение понадобится.

Даже если ваша конечная цель — полное слияние в один ОГРАНИЧЕННЫЙ КОНТЕКСТ путем НЕПРЕРЫВНОЙ ИНТЕГРАЦИИ, начинайте с перехода к ОБЩЕМУ ЯДРУ (SHARED KERNEL).

1. Оцените начальную ситуацию. Убедитесь, что два КОНТЕКСТА действительно едины по смыслу, прежде чем начинать их формальную унификацию.
2. Подготовьте процесс. Необходимо выработать порядок работы с общим кодом и установить правила именования модулей. Код ОБЩЕГО ЯДРА должен подвергаться интеграции как минимум раз в неделю. Для него должен существовать набор тестов. Создайте его еще до того, как писать общий код. (Вначале он будет пустым, так что пройти такие тесты — не проблема!)
3. Выберите для начала небольшую подобласть — нечто дублирующееся в обоих КОНТЕКСТАХ, но *не входящее* в СМЫСЛОВОЕ ЯДРО (CORE DOMAIN) предметной области. Первое слияние имеет своей целью только “обкатать” процедуру, так что для него лучше выбрать нечто простое, сравнительно отдельное и некритичное. Изучите уже существующие интеграции и трансляции. Выбор чего-то уже транслируемого хорош тем, что транслируемость проверена. А по итогам работы трансляционный уровень еще уменьшится в размерах.

В этот момент у вас будет две модели, относящиеся к одной и той же подобласти. Существует три основных подхода к слиянию. Можно взять одну модель и выполнить рефакторинг другого КОНТЕКСТА, чтобы привести его к совместимому виду. Это решение можно принимать “оптом”, задавшись целью систематически заменить модель одного КОНТЕКСТА и сохранить при этом связность модели, разрабатывавшейся как одно целое. А можно действовать постепенно, и в результате, возможно, придти к оптимальной комбинации двух вариантов (но придется постараться, чтобы это не была хаотическая куча).

Третий вариант — это найти новую модель, предположительно более глубокую, чем любая из исходных, которая бы смогла взять на себя обязанности обеих сторон.

4. Сформировать группу из двух-трех разработчиков из обеих групп, для выработки общей модели подобласти. Независимо от того, каково происхождение модели, она должна быть проработана во всех подробностях. Это означает, в том числе, определение синонимов и установку соответствия между терминами, которые еще не

транслировались. Объединенная группа набросает и ориентировочный набор тестов дл модели.

5. Разработчики из каждой группы берутся реализовать модель (или адаптировать существующий код, поступающий в совместное пользование), проработать детали и довести до рабочего состояния. Если они сталкиваются с проблемами в модели, то снова собирают группу (см. п. 4) и сами участвуют в необходимой доработке понятий.
6. Разработчики каждой из групп берутся за задачу интегрирования с новым ОБЩИМ ЯДРОМ (SHARED KERNEL).
7. Убрать трансляции, которые больше не нужны.

На этом этапе ОБЩЕЕ ЯДРО (SHARED KERNEL) будет еще очень маленьким; к нему будет прилагаться процедура сопровождения. На последующих итерациях проекта повторяйте пп. 3–7 для расширения ядра. По мере закрепления процедур и приобретения уверенности в себе можно выбирать более сложные подобласти, а то и сразу несколько за один раз или же подобласти из СМЫСЛОВОГО ЯДРА (CORE DOMAIN).

Сделаем одно замечание. Приступая к более специфичным для предметной области частям модели, можно встретить случаи, когда две модели построены в соответствии со специальным жаргоном разных сообществ пользователей. Разумно будет подождать с включением таких моделей в ОБЩЕЕ ЯДРО (SHARED KERNEL), если к этому моменту не произошел скачок к углубленной модели, отчего должен был появиться язык, способный заменить оба специализированных жаргона. Особенность ОБЩЕГО ЯДРА состоит в том, что можно пользоваться преимуществами НЕПРЕРЫВНОЙ ИНТЕГРАЦИИ, притом сохраняя и некоторые преимущества ОТДЕЛЬНОГО СУЩЕСТВОВАНИЯ.

Для слияния в одно ОБЩЕЕ ЯДРО (SHARED KERNEL) существуют определенные правила. Прежде чем двигаться дальше, следует рассмотреть одну альтернативу, решающую некоторые задачи, ради которых и задумывается это преобразование. Если одна из двух моделей однозначно предпочтительнее, подумайте, не стоит ли перейти на нее без интеграции. Вместо организации общих подобластей просто передайте все обязанности, относящиеся к этим подобластям, из одного КОНТЕКСТА в другой. Для этого потребуется рефакторинг приложений к модели из более предпочитаемого КОНТЕКСТА и разные мелкие усовершенствования, требуемые моделью. Итак, и избыточность устранена, и затрат на интеграцию никаких. Потенциально (хотя и не обязательно) предпочитаемый КОНТЕКСТ может вообще вытеснить другой, и получится тот же эффект, что при слиянии. На переходном этапе (который может длиться неопределенно долго) здесь будут иметь место обычные преимущества и недостатки ОТДЕЛЬНОГО СУЩЕСТВОВАНИЯ, которые нужно будет сравнить с преимуществами и недостатками ОБЩЕГО ЯДРА (SHARED KERNEL).

Слияние контекстов: от общего ядра к непрерывной интеграции

По мере того как ваше ОБЩЕЕ ЯДРО (SHARED KERNEL) расширяется, вас может увлечь мысль о преимуществах полной унификации двух КОНТЕКСТОВ. Но дело здесь не только в устранении различий между моделями. Придется изменять структуру рабочих групп и, в конце концов, даже язык, на котором говорят разработчики.

Начните с подготовки сотрудников и групп.

1. Убедитесь, что все процедуры, необходимые для НЕПРЕРЫВНОЙ ИНТЕГРАЦИИ (совместное владение кодом, частая интеграция и т.п.), поручены отдельно *каждой группе*. Приведите в соответствие интеграционные процедуры в обеих группах, чтобы все делали одно и то же одним и тем же образом.

2. Начните циркуляцию разработчиков между группами. Это создаст группу людей, понимающих обе модели, и положит начало объединению двух групп.
3. Объявите отдельную *дистилляцию* каждой модели (см. главу 15).
4. На этом этапе уже должно хватить уверенности в себе для того, чтобы начать переливать СМЫСЛОВОЕ ЯДРО (CORE DOMAIN) предметной области в ОБЩЕЕ ЯДРО (SHARED KERNEL). На это может уйти несколько итераций. Иногда бывают нужны временные трансляционные уровни между новыми общими частями и частями, еще не поступившими в общий доступ. "Взявшись" за СМЫСЛОВОЕ ЯДРО, надо работать быстро. Это очень затратный этап, где встречается много ошибок, и его нужно побыстрее пройти в приоритетном порядке, даже отложив разработку нового. Но не следует брать на себя слишком много.

Для слияния моделей СМЫСЛОВОГО ЯДРА есть несколько способов. Можно придерживаться одной модели, делаю другую совместимой с ней. Или же можно создать новую модель подобласти и приспособить оба КОНТЕКСТА к ее использованию. Берегитесь случая, когда две модели специально спроектированы для удовлетворения разных нужд пользователя. Вам могут понадобиться специализированные возможности обеих исходных моделей. Для этого можно разработать углубленную модель, которая бы превзошла обе исходные модели. Разработка углубленной унифицированной модели — дело очень сложное, но если вы поставили себе задачу полного слияния двух КОНТЕКСТОВ, то больше не можете себе позволить иметь много разных диалектов. Но зато вас ждет награда в виде четкой, ясной интеграции получившейся модели и кода. Убедитесь, что эти преимущества не имеют свою оборотную сторону в виде невозможности дольше соответствовать специальным потребностям пользователей.

5. По мере роста ОБЩЕГО ЯДРА (SHARED KERNEL) увеличивайте частоту процедур интеграции до ежедневной, а там и до НЕПРЕРЫВНОЙ ИНТЕГРАЦИИ.
6. По мере того как ОБЩЕЕ ЯДРО приближается к точке, в которой оно включит в себя оба бывших ОГРАНИЧЕННЫХ КОНТЕКСТА, вы будете оказываться то среди одной большой группы разработчиков, то в двух маленьких с общей базой кода, которую они непрерывно интегрируют. И между этими группами постоянно будут сновать туда-сюда сотрудники.

Вытеснение устаревшей системы

Все хорошее когда-нибудь кончается — и срок службы старого программного обеспечения тоже. Но конец приходит по-разному. Старые системы могут быть так переплетены с предметной областью и другими системами, что извлечение их оттуда может длиться годы. К счастью, совсем не обязательно извлекать его сразу и мгновенно.

Я лишь набросаю здесь общую схему, так как всех вариантов действий слишком много, чтобы их изложить. Я опишу обычный случай: старая система, используемая ежедневно в какой-то отрасли деятельности, недавно дополнена горсткой более современных систем, вступающих с нею в контакт через ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ (ANTI-CORRUPTION LAYER).

Один из первых шагов должен состоять в том, чтобы выработать стратегию тестирования. Для новых функций новой системы нужно написать автоматизированные модульные тесты. Но вытеснение старой системы требует своих, специфических тестов. Некоторые организации в течение какого-то периода времени параллельно запускают старые и новые тесты.

На каждой итерации этого процесса следует выполнить следующее.

1. Определить специфические функции старой системы, которые можно добавить в одну из новых систем за одну итерацию.
2. Определить дополнения, которые понадобятся на ПРЕДОХРАНИТЕЛЬНОМ УРОВНЕ.
3. Реализовать.
4. Установить.

Иногда бывает необходимо потратить несколько итераций на написание эквивалентных функций для модуля старой системы, который будет вскоре выброшен. Но новые функции все равно следует планировать в форме небольших модулей, которые пишутся за одну итерацию, а вот для своей установки требуют несколько.

Установка и развертывание приложений — это слишком многообразная деятельность, чтобы здесь можно было рассмотреть все известные базы кода. Было бы хорошо, если бы все малые, постепенные изменения можно было бы так же оперативно показывать в конечном продукте. Но на практике обычно организовывают выпуск значительно отличающихся друг от друга версий. Пользователей нужно обучать работе с новыми программами. Иногда нужно закончить период параллельной разработки. Следует проработать много логистических проблем.

Как только программное обеспечение установлено и заработало, выполните такие действия.

5. Определите, какие части ПРЕДОХРАНИТЕЛЬНОГО УРОВНЯ не являются необходимыми, и удалите их.
6. Попробуйте "вырезать" уже неиспользуемые модули старой системы, хотя это может оказаться ни к чему. По иронии, чем лучше спроектирована старая система, тем легче будет от нее избавиться. А вот плохо спроектированные программы тяжело демонтировать по маленькому кусочку. Может быть, стоит просто игнорировать неиспользуемые части программы, пока последние остатки не будут вычищены, и можно будет отключить все сразу.

Повторяйте процесс снова и снова. В ходе этого старая система должна все меньше и меньше вовлекаться в работу; со временем станет виден свет в конце туннеля, и ее можно будет отключить совсем. Между тем ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ будет то уменьшаться, то разбухать по мере того, как различные сочетания связей будут увеличивать или уменьшать взаимозависимость между системами. При прочих равных условиях нужно, конечно, перенести первыми те функции, которые уменьшают ПРЕДОХРАНИТЕЛЬНЫЙ УРОВЕНЬ. Но могут выйти на первый план и другие факторы. Тогда придется временно мириться с самыми неожиданными трансляциями.

От открытого протокола к общедоступному языку

Вы интегрируетесь с другими системами через набор открытых протоколов (как OPEN-HOST SERVICES), но по мере того, как новые системы запрашивают доступ к вашей, бремя ее сопровождения становится все тяжелее, и понять все взаимодействия становится все труднее. Необходимо формализовать отношения между системами через ОБЩЕДОСТУПНЫЙ ЯЗЫК (PUBLISHED LANGUAGE).

1. Если в отрасли есть единый стандартный язык обмена данными, оцените его возможности и используйте его, если это допустимо.
2. Если никакого общего стандарта или опубликованного языка нет, начните с четкого выделения СМЫСЛОВОГО ЯДРА (CORE DOMAIN) системы, которая будет служить сервером открытого протокола (см. главу 15).

3. Используйте СМЫСЛОВОЕ ЯДРО как основу для языка коммуникации, подключив стандартную парадигму обмена данными наподобие XML, если это возможно.
4. Сделайте новый язык общедоступным (опубликуйте его) как минимум для всех, кто с вами взаимодействует.
5. Если важна архитектура новой системы, опубликуйте и ее тоже.
6. Постройте трансляционные уровни для каждой подключающейся к вам системы.
7. Переключитесь на новую систему.

На этом этапе дополнительные системы должны получить возможность подключаться с минимальными препятствиями.

Помните, что ОБЩЕДОСТУПНЫЙ ЯЗЫК должен быть стабильным, но при этом вам нужна свобода для изменения модели своего приложения в ходе постоянного рефакторинга. Поэтому *не отождествляйте между собой язык обмена информацией и модель приложения*. Если поддерживать их достаточно близкими, то затраты на трансляцию будут невелики, и приложение можно будет сделать КОНФОРМИСТОМ. Но оставьте за собой право нарастить трансляционный уровень и увести его в сторону, если это окажется более выгодно.

Руководители проектов должны определять ОГРАНИЧЕННЫЕ КОНТЕКСТЫ (BOUNDED CONTEXTS) на основе требований к функциональной интеграции и взаимоотношений между группами разработчиков. Как только ОГРАНИЧЕННЫЕ КОНТЕКСТЫ и КАРТА КОНТЕКСТОВ (CONTEXT MAP) явно определены и строго соблюдаются, можно считать, что логическая непротиворечивость и самосогласованность находится под защитой. По крайней мере, проблемы коммуникации будут лежать на поверхности, и с ними можно будет работать.

Однако иногда контекстами моделей, ограниченными сознательно или естественным путем, злоупотребляют для решения проблем, отличных от логической несогласованности внутри системы. Разработчики могут обнаружить, что модель большого КОНТЕКСТА слишком сложна для понимания или анализа ее в целом. То ли намеренно, то ли случайно это часто приводит к разбиению КОНТЕКСТА на более мелкие части, поддающиеся осмыслению. Такая фрагментация отбирает часть возможностей. Поэтому стоит тщательно проанализировать решение об организации большой модели в широком КОНТЕКСТЕ. И если с организационной или политической точки зрения невозможно удержать КОНТЕКСТ в целости, если он действительно распадается, тогда нужно перерисовать КАРТУ КОНТЕКСТОВ и определить такие границы, которые можно будет соблюдать. Но если большой КОНТЕКСТ соответствует естественным потребностям интеграции и если он кажется обоснованным независимо от сложности самой модели, тогда разбиение такого КОНТЕКСТА уже не будет хорошим решением.

Есть другие способы сделать большие модели управляемыми, которые следует попробовать, прежде чем идти на такие жертвы. В следующих двух главах рассматривается вопрос об “укрощении сложности” в больших моделях путем применения двух стратегических принципов: дистилляции и крупномасштабной структуры.

Глава 15

Дистилляция

$$\nabla \cdot \mathbf{D} = \rho$$

$$\nabla \cdot \mathbf{B} = 0$$

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t}$$

$$\nabla \times \mathbf{H} = \mathbf{J} + \frac{\partial \mathbf{D}}{\partial t}$$

Джеймс Клерк Максвелл, "Трактат об электричестве и магнетизме" (James Clerk Maxwell, A Treatise on Electricity and Magnetism*), 1873 г.*

Эти четыре уравнения, вместе с определениями понятий и лежащим в их основе математическим аппаратом, описывают всю классическую теорию электромагнетизма XIX в.

Как сосредоточиться на главной проблеме и не утонуть в море второстепенных деталей? МНОГОУРОВНЕВАЯ АРХИТЕКТУРА (LAYERED ARCHITECTURE) позволяет отделить понятия предметной области от технической части, винтиков и шестеренок программной системы. Но в больших системах даже изолированная предметная область может быть сложной до неуправляемости.

Дистилляция — это процесс разделения компонентов смеси с целью выделения основного вещества в такой форме, которая делает его более ценным и полезным. Построение модели есть дистилляция знания. С каждым шагом углубляющего рефакторинга мы абстрагируем какой-то ключевой аспект знаний и приоритетов из предметной области. А в этой главе, отступив на шаг назад и придерживаясь соответствующей стратегии, поговорим о дистилляции всей модели предметной области в целом и о том, как набрасывать самые широкие мазки будущей картины.

Как это бывает нередко в случае химической дистилляции, выделяемые ею побочные продукты тоже становятся более ценными в результате этого процесса. Это верно для ЕСТЕСТВЕННЫХ ПОДОБЛАСТЕЙ (GENERIC SUBDOMAINS) и СВЯЗНЫХ МЕХАНИЗМОВ (COHESIVE MECHANISMS). Но все-таки основные усилия вызваны желанием выделить один конкретный, особо ценный компонент — тот, который отличает нашу программу от других и придает ей ценность, т.е. СМЫСЛОВОЕ ЯДРО предметной области (CORE DOMAIN).

Стратегическая дистилляция модели предметной области выполняет сразу несколько функций.

1. Помогает всем разработчикам понять архитектуру системы в целом и взаимосвязь между ее частями.

2. Облегчает процесс коммуникации, выделяя ключевую модель обозримой величины, которая включается в ЕДИНЫЙ ЯЗЫК (UBIQUITOUS LANGUAGE).
3. Задает направление рефакторинга.
4. Фокусирует внимание на тех частях модели, которые имеют наибольшую ценность.
5. Помогает определиться с субподрядами (аутсорсингом), использованием готовых компонентов, распределением задач.

В этой главе излагается систематический подход к проведению стратегической дистилляции СМЫСЛОВОГО ЯДРА предметной области (CORE DOMAIN). Будет рассказано, как распространить правильное восприятие дистилляции на всю группу разработчиков и создать язык, на котором можно обсуждать свою работу.

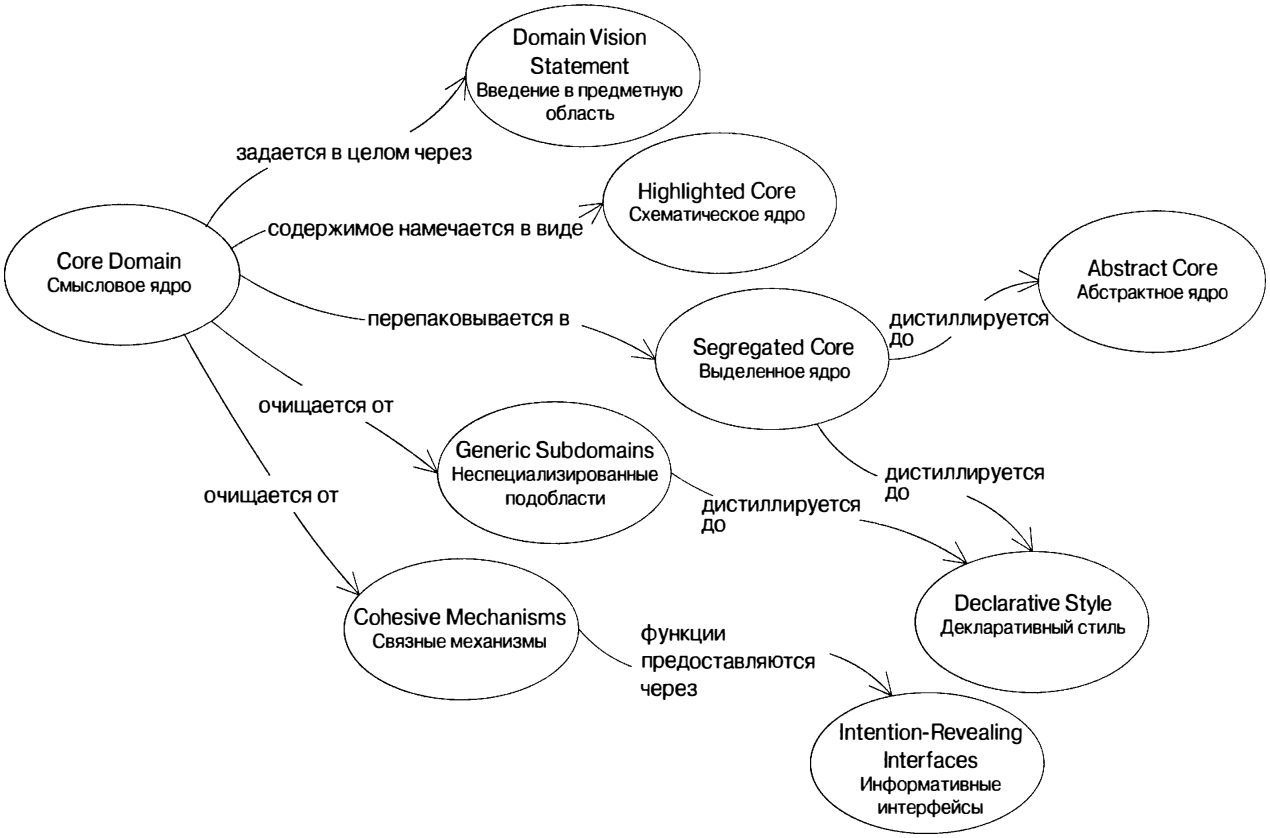


Рис. 15.1. Карта-схема стратегической дистилляции

Как садовник обрезает на деревьях лишнее, чтобы дать возможность расти основным ветвям, так и мы собираемся применить целый набор приемов, чтобы отсечь от модели все несущественное и сосредоточить внимание на самом значимом...

Смысловое ядро

При проектировании большой программной системы нужно учесть так много разных компонентов, причем сложных и совершенно необходимых для успеха дела, что сущность модели предметной области — главного и настоящего делового актива — может затуманиться и пропасть из виду.

Систему, которую трудно понять, трудно и менять. Очень сложно предвидеть, каким будет эффект внесенных изменений. Разработчик, "вышедший за пределы" знакомой ему части приложения, тут же теряется. (В частности, так бывает, когда в группу приходят новые люди, но даже и постоянный ее участник будет испытывать трудности, если не сделать код выразительным и организованным.) Это вынуждает разработчиков специализироваться. Когда они начинают ограничиваться работой только над конкретными модулями, передача знаний снова ухудшается. Такое жесткое разбиение мешает интеграции системы и гибкому распределению работы. Когда программист не знает, что те или иные алгоритмы уже реализованы в другом месте, возникает дублирование, и система снова усложняется.

Таковы некоторые последствия малопонятной архитектуры. Но есть и другой, равно серьезный риск — потерять из виду общую картину предметной области.

В суровой реальности не все части архитектуры совершенствуются одинаково. Следует расставлять приоритеты. Чтобы сделать модель предметной области ценным активом, главное СМЫСЛОВОЕ ЯДРО **модели должно быть отшлифовано до мелочей и полностью задействовано в реализации функций программы. Но редко встречающиеся программисты-профессионалы тяготеют, скорее, к технической инфраструктуре или четко поставленным задачам из предметной области, которые можно понять и без специальных знаний о ней.**

Такие части системы представляют интерес для специалистов-теоретиков программирования. Считается, что на их основе хорошо строить базу переносимых профессиональных знаний и подбирать иллюстративный материал. Специализированное ядро, т.е. та часть модели, которая отличает приложение от всех остальных и делает его ценным активом, обычно собирают в одно целое с другими частями программисты более низкой квалификации. Совместно с администраторами баз данных они создают структуру базы для приложения, а затем программируют одну функцию за другой, не заботясь о том, чтобы воспользоваться концептуальными преимуществами модели.

Плохая архитектура или реализация этой части программы ведет к тому, что ее пользователям всегда субъективно "чего-то не хватает", пусть даже техническая инфраструктура работает прекрасно, а вспомогательные возможности реализованы на высшем уровне. Эта коварная проблема "подкрадывается" тогда, когда в проекте нет четкого видения всей архитектуры в целом и относительной ценности разных ее частей.

Один из наиболее успешных проектов, в которых я работал, вначале тоже страдал от этого синдрома. Цель проекта состояла в разработке очень сложной системы управления синдицированными кредитами. Большинство квалифицированных сотрудников, не напрягаясь, работали над интерфейсом обмена сообщениями и уровнями отображения баз данных, тогда как прикладная модель находилась "в руках" новичков объектных технологий.

Единственным исключением из этой практики был опытный объектно-ориентированный разработчик, работавший над проблемой из предметной области. Он придумал способ добавлять комментарии к любому из долговременных объектов предметной области. Эти комментарии можно было организовывать таким образом, чтобы участники процесса видели письменно зафиксированное обоснование своих прошлых решений. Он же разработал изящный пользовательский интерфейс, который предоставил интуитивно-удобный доступ к гибким возможностям модели комментариев.

Эти возможности были полезны и хорошо спроектированы. Они вошли в окончательный продукт. Но, к сожалению, это были периферийные возможности. Талантливый программист смоделировал интересный и оригинальный способ комментирования решений, качественно реализовал его и вложил в руки пользователя. Тем временем один некомпетентный программист превращал критический для функционирования приложения "кредитный" модуль в "непонятное месиво", которое чуть не сгубило проект.

В процессе планирования следует направлять ресурсы на самые "критические участки" модели и архитектуры. А для этого такие участки в ходе планирования и разработки должен видеть каждый участник.

Части модели, которые можно четко выделить как основные для выполнения главной задачи приложения, образуют ее СМЫСЛОВОЕ ЯДРО (CORE DOMAIN). Именно в СМЫСЛОВОМ ЯДРЕ ваша система производит свой основной "добавочный продукт".

"Упарьте" модель до минимума. Найдите ее СМЫСЛОВОЕ ЯДРО **и сделайте так, чтобы его можно было легко отличить от массы вспомогательных частей модели и кода. Четко, рельефно выделите самые ценные и специализированные понятия.** ЯДРО **должно быть небольшим по размеру.**

Подберите и направьте на разработку СМЫСЛОВОГО ЯДРА **лучшие кадры. При работе над** ЯДРОМ **не жалейте усилий, чтобы построить углубленную модель и гибкую архитектуру, адекватно отражающую видение будущей системы. Вкладывайте ресурсы в другие части системы по тому критерию, насколько они важны для поддержки дистиллированного** СМЫСЛОВОГО ЯДРА**.**

Дистилляция СМЫСЛОВОГО ЯДРА — дело нелегкое, но она приводит к упрощению многих решений. Много усилий уходит на то, чтобы сделать ЯДРО оригинальным, выпукло выделить его, при этом оставляя остальную часть архитектуры настолько общей и стандартной, насколько это удобно практически. Если существует аспект архитектуры, который необходимо хранить в секрете как конкурентное преимущество, то это именно СМЫСЛОВОЕ ЯДРО. Нет необходимости скрывать остальное. И если встает вопрос о выборе (в силу ограниченности времени) между двумя разными процедурами рефакторинга, надо выбирать ту, которая активнее воздействует на СМЫСЛОВОЕ ЯДРО.

* * *

Шаблоны, приведенные в этой главе, помогают лучше увидеть, использовать и модифицировать СМЫСЛОВОЕ ЯДРО.

Выбор ядра

Сейчас в поле нашего зрения находятся те части модели, которые, собственно, и представляют деятельность в предметной области, решение поставленных в ней задач.

Выбор СМЫСЛОВОГО ЯДРА (CORE DOMAIN) зависит от точки зрения на проблему. Например, для многих приложений требуется самая общая модель денег, в которой можно представлять различные валюты, их обменный курс, всевозможные расчеты по обмену валют. С другой стороны, программа для валютных торгов нуждается в более сложной и проработанной модели денег, которая и составит СМЫСЛОВОЕ ЯДРО. Но даже в таком случае часть модели денег может по-прежнему носить самый общий характер. По мере того как с опытом углубляется понимание предметной области, можно продолжать и процесс дистилляции, отделяя общие понятия модели денег и оставляя в СМЫСЛОВОМ ЯДРЕ только узкоспециализированные аспекты.

В программе для обслуживания транспортных перевозок и доставки грузов ЯДРО может включать модель консолидации грузов для их общей доставки, модель передачи юридической ответственности при переходе контейнеров из рук в руки или же модель перемещения контейнера на разных видах транспорта по определенному маршруту в пункт назначения. СМЫСЛОВОЕ ЯДРО в приложении для инвестиционного банка могло бы содержать модель синдикации средств между инвесторами и получателями.

То, что для одного приложения — СМЫСЛОВОЕ ЯДРО, для другого — всего лишь общий вспомогательный компонент. Тем не менее в пределах одного проекта, а обычно и в пределах одной компании можно определить единое СМЫСЛОВОЕ ЯДРО. Как и любая другая часть архитектуры, СМЫСЛОВОЕ ЯДРО должно эволюционировать, развиваться итерационно. Важность того или иного набора взаимоотношений не всегда ясна с первого взгляда. Объекты, которые вначале казались безусловно центральными, в конце концов могут оказаться не более чем вспомогательными.

В последующих разделах, особенно разделе, посвященном НЕСПЕЦИАЛИЗИРОВАННЫМ ПОДОБЛАСТЯМ (GENERIC SUBDOMAINS), будут даны более подробные рекомендации для принятия таких решений.

Как распределить работу

Наиболее технически грамотные программисты-участники проекта, как правило, немного знают о предметной области. Поэтому они приносят пользы меньше, чем могли бы, а закономерная тенденция “ставить” их в связи с этим на вспомогательные компоненты создает порочный круг: из-за недостатка знаний таких программистов не привлекают именно к той работе, которая дала бы им эти знания.

Этот круг очень важно разорвать, собрав такую группу разработчиков, в которой сошлись бы и сильные программисты, мыслящие долговременными категориями и заинтересованные в накоплении знаний, и один или несколько специалистов в предметной области, которые профессионально владеют прикладной проблематикой. Проектирование архитектуры предметной области — это интересная и технически сложная работа, если подходить к ней серьезно. И программистов, которые именно так к ней относятся, найти несложно.

Обычно для конкретной работы по построению СМЫСЛОВОГО ЯДРА нерационально нанимать на короткое время постороннего проектировщика архитектуры. Группе необходимо накопить знания о предметной области, а временный сотрудник — это “брешь” в обороне. С другой стороны, если взять специалиста на роль преподавателя, наставника,

консультанта, то от него может быть очень много пользы: он поможет группе развить навыки архитектурного проектирования в данной предметной области и облегчит им работу со сложными принципами, которыми члены группы могут и не владеть.

По аналогичным причинам есть основания сомневаться, что СМЫСЛОВОЕ ЯДРО можно просто купить. Так, были предприняты большие усилия по созданию специализированных отраслевых сред моделирования. Наиболее заметные результаты этих усилий — разработанная гигантом полупроводниковой индустрии концерном SEMATECH среда CIM для автоматизации производства полупроводников, а также среды San Francisco от IBM для самых разных приложений. Хотя сама идея заманчива, полученные результаты пока неубедительны, если не считать их применения в виде ОБЩЕДОСТУПНЫХ ЯЗЫКОВ для облегчения обмена данными (см. главу 14). В книге [10] дается обзор текущего состояния дел. Но по мере развития данной сферы могут появиться и более "работоспособные" среды моделирования.

При всем этом существует и более веская причина относиться к таким средам с осторожностью. Дело в том, что ценность специализированной программы тем выше, чем больше у нее контроля над СМЫСЛОВЫМ ЯДРОМ. Хорошо спроектированная среда моделирования предоставляет пользователю абстракции высокого уровня, которые можно специализировать для конкретных задач. Благодаря ей пользователь не должен заниматься разработкой самых общих частей приложения и может сосредоточиться на СМЫСЛОВОМ ЯДРЕ. Но если среда накладывает более жесткие ограничения, то возможна одна из трех ситуаций и, соответственно, один из трех вариантов действий.

1. Теряется одно из существенных преимуществ программы. Тогда следует отказаться от использования "стесняющей движения" среды моделирования при проектировании СМЫСЛОВОГО ЯДРА.
2. Область, обслуживаемая средой, не является настолько ключевой, как могло показаться сначала. Тогда следует перенести границы СМЫСЛОВОГО ЯДРА так, чтобы они охватывали действительно значимую и оригинальную часть модели.
3. В СМЫСЛОВОМ ЯДРЕ нет никаких специальных возможностей. Тогда следует подумать о более простом и менее затратном решении, например, приобретении готового приложения и интегрировании его со своим собственным.

Так или иначе, для разработки оригинального программного продукта не обойтись без стабильной группы разработчиков, накапливающих специальные знания и перерабатывающих эти знания в углубленную модель. Ни коротких путей, ни волшебных палочек здесь не существует.

Эскалация дистилляции

Различные приемы дистилляции, которым посвящена оставшаяся часть этой главы, можно применять почти в любом порядке. Тем не менее они отличаются друг от друга по степени радикальности их воздействия на архитектуру модели.

Простой шаблон ВВЕДЕНИЕ В ПРЕДМЕТНУЮ ОБЛАСТЬ (DOMAIN VISION STATEMENT) передает основные понятия и их значение при минимуме вложенных усилий. Шаблон СХЕМАТИЧЕСКОЕ ЯДРО (HIGHLIGHTED CORE) помогает усовершенствовать коммуникацию между разработчиками и процесс принятия решений при этом не требуя практически никаких модификаций архитектуры.

При более радикальном рефакторинге и изменении модульной структуры явно выделяются НЕСПЕЦИАЛИЗИРОВАННЫЕ ПОДОБЛАСТИ (GENERIC SUBDOMAINS), с которыми потом можно работать отдельно. Имея гибкую, информативную и разнообразную архитекту-

ру, можно легко инкапсулировать СВЯЗНЫЕ МЕХАНИЗМЫ (COHESIVE MECHANISMS). Убрав эти отвлекающие факторы, мы тем самым проясняем структуру СМЫСЛОВОГО ЯДРА.

Изменив модульную структуру ВЫДЕЛЕННОГО ЯДРА (SEGREGATED CORE), можно сделать ЯДРО непосредственно видимым даже в коде, что облегчает будущую работу над моделью ЯДРА.

А наиболее амбициозный замысел из всех — это АБСТРАКТНОЕ ЯДРО (ABSTRACT CORE), которое выражает самые фундаментальные понятия и отношения в чистом виде и требует большой работы по реорганизации и рефакторингу модели.

Каждый из перечисленных подходов требует соответственно больших усилий, чем предыдущий. Но чем тоньше сточить лезвие ножа, тем острее оно станет. Последовательная дистилляция модели предметной области порождает ценный актив, который придает всему программному продукту скорость, гибкость и точность исполнения.

Для начала можно попробовать отделить наименее специализированные аспекты модели. НЕСПЕЦИАЛИЗИРОВАННЫЕ ПОДОБЛАСТИ (GENERIC SUBDOMAINS) представляют собой противоположность СМЫСЛОВОМУ ЯДРУ (CORE DOMAIN), и контраст между ними проясняет смысл как того, так и другого понятия...

Неспециализированные подобласти

Некоторые части модели добавляют ей сложности, не содержа и не передавая специального знания. Все лишнее затрудняет понимание и различение СМЫСЛОВОГО ЯДРА **(**CORE DOMAIN**). Модель загромождается общеизвестными или узкоспециализированными принципами, но не основными, а вспомогательными подробностями. Тем не менее, несмотря на их лишенный специализации характер, такие элементы являются существенными для функционирования системы и полноценного выражения модели.**

В вашей модели наверняка есть такая часть, которую нужно воспринимать как само собой разумеющуюся. Она, несомненно, входит в модель предметной области, но абстрагирует такие понятия, которые, скорее всего, встречаются еще во многих областях деятельности. Например, организационная схема компании может понадобиться в той или иной форме в таких разных отраслях, как перевозки, банковское дело или производство. Еще один типичный пример — это учет прихода-расхода и прочая работа с финансами, которая встречается во многих приложениях и может быть представлена одной общей моделью бухгалтерского учета.

Часто как раз на второстепенные вопросы из предметной области уходит слишком много усилий. Лично я своими глазами наблюдал в двух разных проектах, как лучшие программисты несколько недель занимались представлением даты и времени в разных часовых поясах. Такие компоненты, конечно, должны работать, но это не концептуальное ядро системы.

Но даже если настолько общий элемент модели считается критически важным, в модели предметной области, в целом, все же нужно подчеркивать наиболее производительные и специализированные аспекты системы, и структурировать ее нужно так, чтобы именно эта часть получала наиболее мощное развитие. А это трудно сделать, если СМЫСЛОВОЕ ЯДРО смешивается с разными побочными факторами.

Определите, какие из связных подобластей модели не являются основным мотивом к написанию вашего приложения. Рефакторингом выделите модели этих подобластей, имеющие общий характер, и поместите их в отдельные МОДУЛИ**. Не оставляйте в них ни следа специфики вашего приложения.**

Как только такие области выделены, установите для них более низкий приоритет их дальнейшей разработки, чем у СМЫСЛОВОГО ЯДРА**, и постарайтесь не поручать работу**

над ними вашим ведущим программистам (потому, что в процессе такой работы они практически не получат знаний о предметной области). Рассмотрите возможность приобрести готовое решение или взять общеизвестную модель для таких НЕСПЕЦИАЛИЗИРОВАННЫХ ПОДОБЛАСТЕЙ **(**GENERIC SUBDOMAINS**).**

* * *

При разработке таких пакетов у вас может быть несколько дополнительных вариантов действий.

Вариант 1. Готовое решение

Иногда можно купить готовое программное решение или взять для него открытый общедоступный код.

Преимущества

- Меньше кода придется разрабатывать самим.
- Вопросы сопровождения и доработки выносятся за пределы проекта.
- Код, скорее всего, написан более удачно, опробован во многих местах, а поэтому более надежен и завершен, чем код собственного изготовления.

Недостатки

- Прежде чем использовать код, его все равно придется изучить и оценить.
- При нынешнем состоянии контроля качества в нашей индустрии нельзя полностью рассчитывать на правильность и надежность кода.
- Код может быть перегружен лишними элементами с точки зрения ваших потребностей; его интеграция может отнять больше времени, чем написание собственной урезанной версии.
- Элементы извне обычно не так уж "гладко" интегрируются со своими собственными. Может оказаться, что их ОГРАНИЧЕННЫЕ КОНТЕКСТЫ — другие. Но даже если это не так, могут возникнуть трудности с точными ссылками на объекты-СУЩНОСТИ (ENTITIES) из других ваших пакетов.
- Может проявиться зависимость от системной платформы, компилятора и т.д.

Готовые решения по реализации подобластей можно рассмотреть как вариант, но обычно они не стоят затраченных усилий. Я встречал успешные приложения с очень сложными требованиями к делопроизводству, которые пользовались внешними системами делопроизводства через программные интерфейсы (API). Я видел и успешный пример пакета протоколирования ошибок, глубоко интегрированного в приложение. Иногда НЕСПЕЦИАЛИЗИРОВАННЫЕ ПОДОБЛАСТИ (GENERIC SUBDOMAINS) оформляются в виде сред или библиотек, реализующих очень абстрактную модель, которую можно интегрировать с вашим приложением и уже там придать ей специализацию. Чем более общий характер имеет вспомогательный компонент и чем более дистиллирована его собственная модель, тем больше шансов получить от них пользу.

Вариант 2. Общедоступная архитектура или модель

Преимущества

- Более проработанная, чем "доморощенная"; отражает знания многих людей.
- Наличие качественной документации.

Недостаток

- Может оказаться перегруженной лишними элементами или не совсем соответствующей конкретным потребностям.

Том Лерер (Tom Lehrer), автор комических песен 1950–60-х годов, как-то сказал, что секрет успеха в математике — это "плагиат и еще раз плагиат; пусть ничья работа не ускользнет от ваших глаз... только не забудьте назвать это *исследованием*"[1]. Это хороший совет по моделированию предметных областей, особенно при работе с их НЕСПЕЦИАЛИЗИРОВАННЫМИ ПОДОБЛАСТЯМИ.

Лучше всего это получается, если модель широко распространена — как, например, модели из книги [11]. Подробнее см. главу 11.

Если в отрасли уже имеется строго формализованная, установившаяся модель, воспользуйтесь ею. На ум сразу приходят бухгалтерия и физика. Это не просто строгие и четко определенные области знания, они еще и повсеместно распространены, общеизвестны, что снижает затраты на текущее и перспективное обучение сотрудников. (См. в главе 10 об использовании общепринятого формализма.)

Не считайте себя обязанными реализовать все аспекты общедоступной модели, если можете выделить упрощенное замкнутое подмножество, которое отвечает вашим потребностям. Но в случае наличия хорошо обкатанной и хорошо документированной а еще лучше формализованной готовой модели нет смысла изобретать велосипед.

Вариант 3. Вынос реализации на аутсорсинг

Преимущества

- Освобождает ключевых сотрудников для работы над СМЫСЛОВЫМ ЯДРОМ, где требуется и накапливается больше всего знаний по предметной области.
- Позволяет вести дополнительную разработку, не увеличивая число собственных сотрудников, но и "не рассеивая" знания о СМЫСЛОВОМ ЯДРЕ.
- Вынуждает к интерфейсно-ориентированному проектированию и помогает сохранить обобщенный вид подобласти, поскольку наружу передается только техническое задание.

Недостатки

- Все-таки требует затрат времени со стороны основных разработчиков, потому что интерфейс, стандарты разработки кода и другие важные аспекты необходимо согласовывать.
- Влечет за собой дополнительные трудозатраты при передаче кода заказчику, поскольку требуется понять полученный код. (Но эти затраты все же меньше, чем для специализированных подобластей, потому что модель общего характера, скорее всего, не потребует для своего понимания каких-то особых знаний.)
- Качество кода может оказаться переменчивым, хуже или лучше в зависимости от соотношения между группами разработчиков.

При выносе работ на аутсорсинг важную роль могут сыграть автоматизированные тесты. От субподрядчиков необходимо требовать модульных тестов для кода, который они предоставляют. Самый надежный подход, который гарантирует должный уровень качества, четкое техническое задание и безупречную реинтеграцию это спецификация

[1] Plagiarize! Plagiarize. Let no one's work evade your eyes. ...Only be sure always to call it please, *research*.

или даже написание автоматизированных приемочных тестов для компонентов, вынесенных на аутсорсинг. Кроме того, "аутсорсинговая реализация" может прекрасно сочетаться с "общедоступной архитектурой или моделью".

Вариант 4. Собственная реализация

Преимущества

- Простота интеграции.
- Получаете именно то, что нужно, и ничего лишнего.
- Можно привлечь временных исполнителей.

Недостатки

- Затраты на сопровождение и обучение.
- Легко недооценить время и стоимость разработки таких пакетов.

Конечно, и этот подход тоже хорошо сочетается с "общедоступной архитектурой или моделью".

Именно в НЕСПЕЦИАЛИЗИРОВАННЫХ ОБЛАСТЯХ уместно попробовать применить внешние архитектурные решения, поскольку для этого не требуется глубокое знание специализированного СМЫСЛОВОГО ЯДРА, и в процессе работы такое знание не приобретается. Соображения конфиденциальности не играют тут особой роли, поскольку в таких модулях практически отсутствует конфиденциальная информация, технологии или методики. Неспециализированная подобласть позволяет не нагружать лишними знаниями тех, кого не интересует глубокое знание предметной области.

Надо полагать, что со временем наше определение того, что входит в СМЫСЛОВОЕ ЯДРО модели, будет сужаться. Все больше и больше неспециализированных моделей будут доступны в виде готовых сред, библиотек или, по крайней мере, общедоступных моделей либо аналитических шаблонов. Пока что большую часть таких моделей приходится разрабатывать самостоятельно. Однако отделять их от СМЫСЛОВОГО ЯДРА — это полезный задел на будущее.

Пример

Повесть о двух часовых поясах[2]

Дважды в разных проектах мне приходилось наблюдать, как лучшие программисты целыми неделями занимались проблемой хранения и преобразования времени из одного часового пояса в другой. К таким работам я отношусь с подозрением, но иногда они необходимы. В этих же двух проектах можно увидеть пример практически идеального контраста.

В первом предпринималась попытка спроектировать программу для организации грузоперевозок. Для составления графиков движения по международным маршрутам критически важно правильно вычислять время, а поскольку все такие графики отслеживаются по местному времени, скоординировать перевозки без преобразования времени невозможно.

Убедительно обосновав потребность в этой функции, разработчики приступили к разработке СМЫСЛОВОГО ЯДРА (CORE DOMAIN) и первых итераций самого приложения, используя имеющиеся в наличии классы для работы со временем и фиктивные данные. По мере развития приложения стало ясно, что имеющиеся классы не отвечают своей задаче, и

[2] В заголовке раздела (*A Tale of Two Time Zones*) обыгрывается название романа Ч.Диккенса "Повесть о двух городах" (*A Tale of Two Cities*). — *Примеч. перев.*

проблема очень сложна из-за вариаций по странам и неопределенности линии перемены дат (*International Date Line*). Требования к функциям прояснились, и разработчики поискали готовое решение, но не нашли. Оставался единственный вариант — написать его самим.

Для решения задачи требовалось провести исследования и выполнить очень точное проектирование, так что руководители группы выделили для этого одного из лучших программистов. Но выполнение этой работы не требовало специальных знаний в области грузоперевозок и не развивало их. Поэтому руководство решило поручить ее программисту, работавшему по временному контракту.

Программист не начал работу с нуля. Он изучил несколько существующих реализаций часовых поясов, большинство из которых не удовлетворяло выдвигавшимся требованиям, и решил адаптировать открытое общедоступное решение из системы BSD Unix, где имелась обширная база данных и реализация на языке C. Он восстановил программную логику и написал процедуру импортирования для базы данных.

Проблема оказалась еще сложнее, чем можно было ожидать (например, в особых случаях нужно было импортировать базу данных), но код был написан и интегрирован с ЯДРОМ, и финальный программный продукт сдан в эксплуатацию.

В другом проекте все пошло совершенно по-другому. Страховая компания разрабатывала новую систему обработки страховых претензий и планировала фиксировать время различных событий (время дорожно-транспортного происшествия, бури с градом и т.д.) Эти данные должны были записываться по местному времени, так что требовалась функция преобразования времени часовых поясов.

К моменту, когда я появился в проекте, у них уже был назначен на эту работу молодой, но очень толковый программист. Правда, еще не было готово точное техническое задание на приложение и не предпринималась попытка написать даже первую его итерацию. Но программист с готовностью взялся за *априорное* построение модели часовых поясов.

Так как было неизвестно, что конкретно потребуется, руководство решило, что модель должна быть достаточно гибкой и адаптироваться к чему угодно. Задача была трудная, и программист, которому поручили работу, нуждался в помощи, поэтому над ним был назначен старший программист. Они написали сложный код, но так как его не использовало никакое конкретное приложение, оставалось неясным, правильно он работает или нет.

В силу разных причин проект “заглох”, и код для работы с часовыми поясами так и не нашел
бы достаточно просто сохранять в базе данных местное время с меткой часового пояса и даже без преобразования, поскольку это были базовые справочные данные, а не основа для каких-то вычислений. Даже если бы преобразование оказалось необходимым, все данные собирались только по Северной Америке, где переходы между часовыми поясами сравнительно несложны.

Главное, чем пришлось заплатить за излишнее внимание к часовым поясам, — это пренебрежение СМЫСЛОВЫМ ЯДРОМ модели. Если бы столько же энергии вложили именно туда, мог бы получиться работающий прототип приложения и первый набросок модели предметной области. Кроме того, программисты-участники проекта, связанные с ним долговременными договорами, должны были бы работать в основной, страховой его части, накапливая для группы важные знания.

Обе группы роднило то, что одну “вещь” они сделали правильно: четко отделили НЕСПЕЦИАЛИЗИРОВАННУЮ модель часовых поясов от СМЫСЛОВОГО ЯДРА. Если бы модель часовых поясов содержала специфику страхового дела или грузоперевозок, то главная, специализированная, модель оказалась бы слишком тесно связана со вспомогательной, неспециализированной, и СМЫСЛОВОЕ ЯДРО стало бы труднее для понимания (поскольку содержало бы ненужные подробности о часовых поясах). А весь МОДУЛЬ для

работы с часовыми поясами стало бы тяжелее сопровождать, поскольку ответственные за сопровождение лица должны были бы понимать СМЫСЛОВОЕ ЯДРО и его взаимосвязь с часовыми поясами.

Стратегия проекта по грузопоставкам	Стратегия проекта по страховому обслуживанию
Преимущества	***Преимущество***
• НЕСПЕЦИАЛИЗИРОВАННАЯ модель отделена от ЯДРА. • ЯДРО практически готово, и отвлечение ресурсов от смысловой модели ему не повредит. • Было точно известно, что именно нужно в проекте. • Критически важная вспомогательная функция для составления графиков международных перевозок.	• НЕСПЕЦИАЛИЗИРОВАННАЯ модель отделена от ЯДРА.
Недостаток	***Недостатки***
• Отвлечение высококвалифицированного программиста от работы над смысловым ядром.	• Модель ЯДРА недоработана, и отвлечение на другие проблемы усугубляет эту недоработку. • Из-за неопределенных требований принималась самая общая модель, тогда как могло быть достаточно простых преобразований по Северной Америке. • Работа была поручена постоянным членам группы, которые могли бы вместо этого стать вместилищем знаний по предметной области.

Мы, "технари", предпочитаем заниматься четко определенными проблемами, наподобие преобразований времени между часовыми поясами, и запросто найдем оправдание потраченному на такие проблемы времени. Но если точно придерживаться правильных приоритетов, то предпочтительнее все-таки оказывается СМЫСЛОВОЕ ЯДРО.

"Неспециализированный" не значит "хорошо переносимый"

Следует отметить, что хотя рассматриваемые подобласти имеют неспециализированный характер, это еще не означает, что код для них хорошо переносим. Готовые переносимые решения в каких-то ситуациях уместны, а в каких-то — нет. Но если говорить только о коде, который разрабатывается самостоятельно (в пределах основной группы или на аутсорсинге), не следует особо заботиться о его переносимости для повторного использования. Это противоречило бы основной мотивации к дистилляции: вкладывать как можно больше усилий в СМЫСЛОВОЕ ЯДРО и лишь по необходимости — во вспомогательные НЕСПЕЦИАЛИЗИРОВАННЫЕ ПОДОБЛАСТИ.

Повторное использование практикуется, но не всегда в отношении кода. Часто более высокий его уровень это перенос и повторное использование модели (например, общедоступной архитектуры). Если вам приходится создавать собственную модель, она может оказаться полезной в будущем аналогичном проекте. Основная концепция такой модели может оказаться применимой во многих ситуациях, а вот саму модель не обязательно прорабатывать во всех подробностях. Бывает достаточно смоделировать и реализовать только ту часть, которая необходима для практической деятельности.

Нет необходимости всегда строить переносимые архитектуры, но следует строго придерживаться общей концепции. Внесение в модель специализированных элементов чревато двумя проблемами. Во-первых, это затормозит будущее развитие. Пусть сейчас вам нужна только небольшая часть модели подобласти, но со временем потребности мо-

гут вырасти. Внося в архитектуру что-либо, не входящее в основную концепцию, вы сильно затрудняете корректное расширение системы без полной перестройки ее старой части и перепроектирования других модулей, которые на нее опираются.

Вторая, и более важная, причина состоит в том, что понятия/концепции, специфичные для данной области деятельности, содержатся или в СМЫСЛОВОМ ЯДРЕ, или в своих собственных специализированных подобластях. Такие специализированные модели гораздо ценнее общих.

Управление рисками в проекте

В *Agile*-методиках программирования принято управлять рисками, в первую очередь направляя усилия на самые рискованные задачи. В экстремальном программировании специально подчеркивается, как важно сразу запустить систему полного рабочего цикла. Эта начальная система часто и оказывается технической архитектурой проекта. В связи с ней возникает искушение сначала построить периферийную систему для работы с какой-то из НЕСПЕЦИАЛИЗИРОВАННЫХ
вать. Будьте осторожны — это может лишить управление рисками всякого смысла.

Проекты "сталкиваются" с рисками с обеих сторон: в некоторых более рискованной является техническая сторона, а в других — моделирование предметной области. Готовая система полного цикла снижает риск только в том смысле, что она представляет "зачаточную" версию самых проблемных мест в реальной системе. Легко недооценить риск, связанный с моделированием предметной области. Он может принять форму непредвиденных осложнений в модели, недостаточно эффективного доступа к специалистам по предметной области, пробелов в ключевых знаниях программистов.

Поэтому во всех случаях, кроме тех, когда разработчики имеют большой опыт в хорошо знакомой предметной области, система первого приближения должна быть основана на какой-то части смыслового ядра, пусть даже простейшей.

Тот же принцип относится и к любой методике разработки, где на первый план выдвигаются задачи с самым высоким уровнем риска. СМЫСЛОВОЕ ЯДРО относится к таким, поскольку его сложность часто непредсказуема, а без него проект вообще не будет реализован.

Большинство из шаблонов в этой главе демонстрирует работу над моделью и кодом с целью дистилляции СМЫСЛОВОГО ЯДРА (CORE DOMAIN). Однако два следующих шаблона, ВВЕДЕНИЕ В ПРЕДМЕТНУЮ ОБЛАСТЬ (DOMAIN VISION STATEMENT) и СХЕМАТИЧЕСКОЕ ЯДРО (HIGHLIGHTED CORE), показывают, как с помощью вспомогательной документации можно при совсем небольших затратах улучшить информативность и содержательность ЯДРА, а также сконцентрировать усилия разработчиков...

Введение в предметную область

В начале проекта модели, как правило, еще нет, но потребность в придании ей нужного "направления" уже существует. На более поздних же этапах разработки бывает нужно разъяснить кому-нибудь ценность системы, не прибегая к глубокому ее изучению. Кроме того, критически важные аспекты модели предметной области могут "простираться" на несколько ОГРАНИЧЕННЫХ КОНТЕКСТОВ**, но по определению эти различные модели нельзя структурировать так, чтобы выделить в них общий фокус.**

Многие группы разработчиков представляют своему руководству "концептуальное введение" (*vision statement*). Наилучшие документы такого рода рассказывают о том, что особенно ценного приложение даст организации-заказчику. В некоторых упоминается,

что создаваемая модель предметной области будет представлять собой стратегический актив. Обычно "концептуальное введение" перестает быть актуальным, как только проект получает финансирование. Оно никогда не используется в реальном процессе разработки, а технические сотрудники его даже не читают.

ВВЕДЕНИЕ В ПРЕДМЕТНУЮ ОБЛАСТЬ (DOMAIN VISION STATEMENT) построено по образцу таких документов, но фокусируется на общих характеристиках модели предметной области и ее ценности для предприятия. Оно может непосредственно использоваться как руководством, так и техническими сотрудниками на всех этапах разработки с целью правильного распределения ресурсов, принятия проектно-модельных решений, повышения квалификации разработчиков. Если модель предметной области призвана "служить многим господам", этот документ может показать, как именно проект учитывает и балансирует разные интересы.

Кратко опишите (примерно на страницу) СМЫСЛОВОЕ ЯДРО **(**CORE DOMAIN**) предметной области и ее полезность — своего рода "деловое предложение". Игнорируйте те аспекты, которые не отличают данную модель предметной области от других. Покажите, как модель служит разным интересам и создает баланс между ними. Напишите этот документ как можно раньше и вносите изменения по мере развития знаний о предмете.**

ВВЕДЕНИЕ В ПРЕДМЕТНУЮ ОБЛАСТЬ можно использовать как ориентир, который задает группе разработчиков общее направление в непрерывном процессе дистилляции как модели, так и самого кода. Этим ВВЕДЕНИЕМ можно делиться с нетехническим персоналом группы, руководством и даже клиентами (конечно, в той мере, в какой там не содержится конфиденциальной информации).

Включается во ВВЕДЕНИЕ В ПРЕДМЕТНУЮ ОБЛАСТЬ	**Не включается во ВВЕДЕНИЕ В ПРЕДМЕТНУЮ ОБЛАСТЬ (хотя и важно само по себе)**
Система заказа авиабилетов	***Система заказа авиабилетов***
Модель может представлять приоритетность пассажиров и схемы заказа билетов в авиакомпаниях, балансируя их по гибкой методике. Модель пассажира должна отражать "взаимоотношения", которые авиакомпания хочет построить с постоянными клиентами. Поэтому она должна представлять историю пассажира в полезной концентрированной форме, его участие в специальных программах, принадлежность к стратегическим корпоративным клиентам и т.д.	Интерфейс должен быть "подогнан" под опытных пользователей, но доступен и для начинающих.
Представление разных ролей для разных пользователей (пассажир, агент, менеджер) позволяет расширять модель отношений и передавать необходимую информацию в систему безопасности.	Предлагается доступ через Веб, путем пересылки данных через другие сетевые системы, а также, возможно, через другие интерфейсы, так что архитектура интерфейса будет строиться на основе XML с уровнями трансляции веб-страниц и других систем данных.
Модель должна поддерживать эффективный поиск по маршрутам и местам, а также интеграцию с другими распространенными системами заказа авиабилетов.	Цветная анимированная версия эмблемы этой системы должна кешироваться в клиентской системе, чтобы при следующем входе она появлялась быстрее.
	Когда заказчик резервирует билеты, нужно выполнить визуальное подтверждение заказа в течение 5 секунд.
	Система безопасности проверяет личность пользователя и ограничивает доступ к тем или иным возможностям на основе привилегий, присвоенных пользовательским ролям.

Включается во ВВЕДЕНИЕ В ПРЕДМЕТНУЮ ОБЛАСТЬ	Не включается во ВВЕДЕНИЕ В ПРЕДМЕТНУЮ ОБЛАСТЬ (хотя и важно само по себе)
Автоматизация завода полупроводников	***Автоматизация завода полупроводников***
Модель предметной области будет представлять состояние материалов и оборудования в процессе производства полупроводниковых пластин таким образом, чтобы обеспечить отслеживание поставок и автоматизацию распределения продукции. Модель не будет включать кадровые ресурсы, требующиеся для процесса, но должна позволять выборочную автоматизацию процесса через загрузку наборов команд. Представление состояния завода должно быть понятным менеджерам (людям), чтобы снабжать их знаниями и улучшать качество принятия решений.	Приложение должно работать через Веб посредством сервлета, но допускать и другие интерфейсы. Где это возможно, следует использовать стандартные для индустрии технологии, чтобы как можно меньше разрабатывать и сопровождать самостоятельно, а также максимизировать доступ к внешней базе знаний. Предпочтение следует отдать решениям с открытым кодом (таким как веб-сервер Apache). Веб-сервер будет работать на выделенном специализированном сервере. Приложение будет работать на одном специализированном сервере.

* * *

ВВЕДЕНИЕ В ПРЕДМЕТНУЮ ОБЛАСТЬ (DOMAIN VISION STATEMENT) дает разработчикам общее направление. Но обычно требуется еще навести некий "мост" между достаточно общим ВВЕДЕНИЕМ и полной детализацией кода или модели...

Схематическое ядро

ВВЕДЕНИЕ В ПРЕДМЕТНУЮ ОБЛАСТЬ (DOMAIN VISION STATEMENT) определяет СМЫСЛОВОЕ ЯДРО (CORE DOMAIN) в самых общих выражениях, а определения конкретных элементов ядра при этом отдаются на прихоть индивидуальных интерпретаций. Но если в группе не установлен исключительно высокий уровень коммуникабельности, то само по себе ВВЕДЕНИЕ В ПРЕДМЕТНУЮ ОБЛАСТЬ не будет иметь существенного влияния на работу группы.

* * *

Пусть члены группы в целом имеют представление о том, из чего состоит СМЫСЛОВОЕ ЯДРО **(**CORE DOMAIN**), все равно разные люди выберут для него разные элементы, и даже один и тот же человек может изменить свое мнение за несколько дней. Умственные усилия, требующиеся для постоянного фильтрования ключевых элементов модели, "поглощают" внимание, отвлекая от проектирования, и к тому же требуют обширных знаний модели.** СМЫСЛОВОЕ ЯДРО **следует сделать более простым для понимания —** СХЕМАТИЧЕСКИМ ЯДРОМ **(**HIGHLIGHTED CORE**).**

Идеальный способ определения СМЫСЛОВОГО ЯДРА **— это существенные структурные изменения в коде, но это не всегда удобно делать за короткие сроки. Фактически настолько серьезные изменения вообще нельзя предпринять, если в группе отсутствует нужное видение модели.**

Такие изменения в организации модели, как разграничение НЕСПЕЦИАЛИЗИРОВАННЫХ ПОДОБЛАСТЕЙ (GENERIC SUBDOMAINS) и некоторые другие, описанные в этой главе позже,

дают возможность почерпнуть нужную информацию из структуры МОДУЛЕЙ. Но не следует с ходу замахиваться на то, чтобы сделать этот способ выражения СМЫСЛОВОГО ЯДРА единственным.

Желательно найти какие-то более легкие вспомогательные приемы в дополнение к этим радикальным способам. В проекте могут быть какие-то ограничения, препятствующие физическому выделению СМЫСЛОВОГО ЯДРА. Или же вы начинаете с существующего кода, в котором ЯДРО не слишком хорошо выделено, но при этом его очень нужно увидеть и поделиться с другими участниками для эффективного рефакторинга в направлении лучшей дистилляции. Но и на зрелых этапах проекта несколько тщательно отобранных графиков или документов дают "зацепки" для ума и "опорные точки" для группы.

Такие вопросы возникают в равной степени и в таких проектах, где используются подробные UML-модели, и в таких (например, XP-проектах), где внешней документации практически нет, а основное изложение модели содержится в коде. Группа экстремальной разработки предпочитает минимализм, ограничиваясь только временным использованием таких вспомогательных средств (например, нарисованной от руки диаграммой на стене), но тем не менее и они находят себе удачное применение.

Выделение особо привилегированной части модели, а также воплощающей ее программной реализации — это отражение модели, а не обязательно какая-то ее часть. Тут подойдет любой прием или способ, лишь бы он помог всем понять СМЫСЛОВОЕ ЯДРО. Для этого класса решений характерны два специальных приема.

Дистилляционный документ

Для описания и разъяснения СМЫСЛОВОГО ЯДРА (CORE DOMAIN) я часто пишу отдельный документ. Он может быть очень простым, — например, списком наиболее существенных концептуальных объектов. Это может быть набор графиков, концентрирующихся на этих объектах и показывающих наиболее важные взаимосвязи между ними. Документ может давать представление о самых фундаментальных взаимодействиях в модели на примерах или на абстрактном уровне. В нем могут использоваться UML-диаграммы классов или последовательности операций, нестандартные диаграммы, характерные для предметной области, тщательно сформулированные текстовые описания, а также сочетания всего перечисленного. *Дистилляционный документ — это не полный проектный документ.* Это минимально необходимое вступление, кратко очерчивающее ЯДРО и располагающее к более подробному исследованию тех или иных его частей. Читателю дается общая картина связи между частями и отсылки к соответствующим фрагментам кода за дальнейшими подробностями.

Итак, это одна из форм СХЕМАТИЧЕСКОГО ЯДРА (HIGHLIGHTED CORE).

Напишите очень краткий документ (от трех до семи страниц крупным шрифтом), в котором описывается СМЫСЛОВОЕ ЯДРО (CORE DOMAIN) и основные взаимодействия между элементами этого ЯДРА.

Здесь возникают все обычные риски, связанные с наличием отдельных документов.

1. Документ может быть лишен сопровождения.
2. Документ могут не читать.
3. Из-за умножения источников информации документ может потерять свое первоначальное предназначение: упрощение сложного.

Лучший способ снизить этот риск — полный минимализм. Не вдаваясь в рутинные детали, сосредотачиваясь только на основных абстракциях и взаимодействиях между ними, можно сделать документ более устойчивым во времени, поскольку этот уровень модели обычно "стареет медленнее".

Напишите такой документ, чтобы его поняли и "нетехнические" сотрудники группы. Пользуйтесь им как совместным представлением знаний, общих и необходимых для всех, и руководством, с которого все члены группы могут начинать знакомство с моделью и кодом.

Разметка ядра

В первый день моей работы над проектом для большой страховой компании мне выдали экземпляр "модели предметной области" — двухсотстраничный документ, приобретенный за большие деньги у промышленного консорциума. Несколько дней я пробирался через завалы диаграмм классов, которые описывали все на свете — от подробного устройства страховых полисов до исключительно абстрактных моделей отношений между людьми. Качество факторинга этих моделей было различным (в некоторых даже описывались деловые регламенты, по крайней мере в сопроводительном тексте). Но с чего начать? Двести страниц...

В культуре данного проекта очень поощрялось построение абстрактных архитектурных сред, и мой предшественник сосредоточился на весьма абстрактной модели взаимосвязей людей между собой, с вещами, видами деятельности и соглашениями. Это был очень неплохой анализ таких взаимосвязей, и их эксперименты с этой моделью имели качество университетского исследовательского проекта. Но к приложению для обслуживания страховой деятельности все это никак нас не приближало.

Моим первым побуждением было рубить сплеча: вернуться к небольшому СМЫСЛОВОМУ ЯДРУ (CORE DOMAIN), выполнить рефакторинг и по ходу дела ввести остальные сложности. Но руководство переполошилось из-за такого отношения. Документ был очень авторитетным. В его разработке участвовали специалисты всей отрасли, и в любом случае они уже заплатили консорциуму значительно больше, чем платили мне, поэтому мои рекомендации по радикальным изменениям не могли восприниматься ими слишком серьезно. Но я знал, что мы должны получить совместную картину нашего ЯДРА и сосредоточить на нем усилия группы.

Вместо рефакторинга я прошелся по документу с помощью бизнес-аналитика, который знал много о страховом деле в целом и о требованиях к разрабатываемому приложению в частности. Так я определил несколько разделов, содержавших самые существенные, оригинальные концепции и понятия, с которыми нам и нужно было работать. Я сделал карту модели, которая четко показывала СМЫСЛОВОЕ ЯДРО и его взаимосвязь со вспомогательными функциями.

Разработка нового прототипа приложения началась с этой отправной точки. Было быстро создано упрощенное приложение, которое демонстрировало некоторые из необходимых функций.

Килограмм макулатуры был превращен в бизнес-актив с помощью нескольких закладок и желтого маркера.

Этот прием не ограничивается одними лишь объектными диаграммами на бумаге. В группе, где пользуются UML-диаграммами, для выделения элементов ядра можно применять "стереотипы". Если единственным хранилищем модели группа сделала программный код, то можно использовать комментарии к нему, — например, структурированные в формате Java Doc — или какую-нибудь утилиту в среде разработки. Конкретный способ не имеет значения, лишь бы благодаря ему разработчики четко знали, что входит, а что не входит в СМЫСЛОВОЕ ЯДРО.

Итак, это другая форма СХЕМАТИЧЕСКОГО ЯДРА (HIGHLIGHTED CORE).

Отметьте все элементы СМЫСЛОВОГО ЯДРА **(**CORE DOMAIN**) в основном информационном хранилище модели, не пытаясь особо разъяснять их роли. Сделайте так, чтобы разработчики легко видели, что входит в** ЯДРО**, а что в него не входит.**

Теперь СМЫСЛОВОЕ ЯДРО четко видимо всем, кто работает с моделью, при достаточно небольших усилиях и затратах на сопровождение — по крайней мере, настолько, насколько хорошо модель факторизована для различения сравнительной значимости ее частей.

Дистилляционный документ как методическое средство

Теоретически в проекте, организованном на принципах экстремального программирования, любая пара (два совместно работающих программиста) может изменять любой код в системе. На практике некоторые изменения влекут за собой важные последствия, и поэтому требуют более серьезного сопровождения и координирования. При работе на инфраструктурном уровне влияние изменений может быть очевидным, но на типичном уровне предметной области это не так.

Имея понятие о СМЫСЛОВОМ ЯДРЕ (CORE DOMAIN) предметной области, такое влияние можно прояснить. Изменения в СМЫСЛОВОМ ЯДРЕ должны иметь большое влияние на модель. Изменения в широко используемых неспециализированных элементах могут потребовать больших модификаций кода, но они все равно не приводят к таким концептуальным сдвигам, как изменения в ЯДРЕ.

Используйте дистилляционный документ как руководство к действию. Когда разработчики понимают, что в дистилляционный документ тоже нужно вносить изменения, чтобы синхронизировать его с кодом или моделью, тогда они обращаются за консультацией. Они либо фундаментально изменяют элементы СМЫСЛОВОГО ЯДРА, либо меняют границы ЯДРА, а может быть, и что-нибудь еще. Необходимо наладить распространение информации об изменении ЯДРА между всеми членами группы по всем используемым в ней каналам связи, в том числе путем раздачи новых версий дистилляционного документа.

Если дистилляционный документ выделяет существенные элементы СМЫСЛОВОГО ЯДРА**, то он служит практическим индикатором значимости изменений в модели. Если изменения в модели или коде влияют на дистилляционный документ, это требует согласования с другими членами группы. При внесении изменения необходимо немедленно сообщать о нем всем участникам и распространять среди них новую версию документа. Изменения за пределами** ЯДРА **или в деталях, которые не включены в дистилляционный документ, можно вносить без консультаций и извещений; члены группы ознакомятся с ними в рабочем порядке. Так разработчики получают полную автономию, декларируемую в экстремальном программировании.**

* * *

Хотя ВВЕДЕНИЕ В ПРЕДМЕТНУЮ ОБЛАСТЬ (DOMAIN VISION STATEMENT) и СХЕМАТИЧЕСКОЕ ЯДРО (HIGHLIGHTED CORE) информируют и направляют группу, они фактически не приводят к модификации модели или кода. А вот выделение НЕСПЕЦИАЛИЗИРОВАННЫХ ПОДОБЛАСТЕЙ (GENERIC SUBDOMAIN) физически устраняет некоторые отвлекающие элементы. В следующих шаблонах мы будем иметь дело со структурными изменениями модели и архитектуры, чтобы сделать СМЫСЛОВОЕ ЯДРО (CORE DOMAIN) более четко видимым и управляемым...

Связные механизмы

Инкапсуляция механизмов работы — это стандартный принцип объектно-ориентированного программирования. Упаковка сложных алгоритмов в методы с информативными именами отделяет “как делать” от “что делать”. Такие приемы помогают лучше понимать архитектуру программ и пользоваться ею. И все же здесь есть свои естественные ограничения.

Иногда вычисления настолько усложняются, что начинают "затуманивать" архитектуру. Концептуальное "что" становится трудно увидеть за обилием механических "как". Большое количество методов с алгоритмами для решения задачи заслоняет собой те методы, которые эту задачу ставят.

Изобилие функциональных процедур — это симптом проблем с моделью. Углубляющий рефакторинг может дать модель и архитектуру, элементы которых лучше приспособлены к решению основной задачи. Первое, что нужно искать и чего добиваться, — это модель, которая упрощает вычислительные механизмы. Но время от времени приходит понимание, что те или иные части механизма сами по себе (внутренне) концептуально связны. Такая концептуальная расчетная часть, вероятно, не включает в себя все громоздкие вычисления, которые нам нужны. Мы не говорим здесь о некоем универсальном "калькуляторе". Но извлечение связной части механизма позволяет лучше понять то, что осталось.

Выделите концептуально СВЯЗНЫЙ МЕХАНИЗМ **(**COHESIVE MECHANISM**) в отдельную небольшую программную среду или библиотеку. Особое внимание окажите формализациям или хорошо документированным категориям алгоритмов. Функциональные возможности среды покажите с помощью** ИНФОРМАТИВНОГО ИНТЕРФЕЙСА **(**INTENTION-REVEALING INTERFACE**). После этого другие элементы предметной области можно будет сосредоточить на выражении основной задачи ("что делать"), а тонкости ее решения ("как делать") передать новой среде.**

Выделенные механизмы ставятся во вспомогательное положение и оставляют после себя небольшое, выразительное СМЫСЛОВОЕ ЯДРО (CORE DOMAIN), которое затем пользуется механизмами через интерфейс в декларативном стиле.

Если распознать в модели известный стандартный алгоритм или формальный аппарат, это позволит перенести некоторую часть сложной архитектуры в уже изученный набор понятий и концепций. Имея общий ориентир, можно реализовать нужное решение с уверенностью и минимумом проб и ошибок. Можно рассчитывать на то, что этой информацией обладают другие разработчики или, по крайней мере, что они могут легко найти ее. Здесь есть сходство с общедоступной моделью НЕСПЕЦИАЛИЗИРОВАННОЙ ПОДОБЛАСТИ (GENERIC SUBDOMAIN), но задокументированный алгоритм или вычислительный метод попадается чаще, потому что это хорошо изученный пласт науки программирования. Тем не менее достаточно часто приходится и создавать что-то новое. В таком случае сосредоточьтесь на узком частном расчетном случае и не примешивайте сюда смысловую модель предметной области. Необходимо разделение обязанностей. Модель СМЫСЛОВОГО ЯДРА или НЕСПЕЦИАЛИЗИРОВАННОЙ ПОДОБЛАСТИ — это формулировка факта, правила или задачи. А СВЯЗНЫЙ МЕХАНИЗМ — это реализация правила или выполнение расчета по требованиям модели.

Пример

Механизм в организационной диаграмме

Мне довелось поучаствовать в подобном процессе, работая в проекте, где понадобилась довольно сложная модель организационной диаграммы. Модель представляла тот факт, что один человек работает на другого (в определенном подразделении организации). Она предлагала интерфейс, посредством которого можно было задавать нужные вопросы и получать ответы. Большинство этих вопросов имело примерно такой вид: "кто в данной цепочке субординации имеет полномочия утверждать это?" или "кто в этом отделе имеет право решать такую-то проблему?". Разработчики поняли, что главную сложность в этой работе представляет трассировка ветвей организационного дерева в поисках

нужных людей или взаимосвязей. Именно такие задачи решаются хорошо формализованным аппаратом *графов*, т.е. наборов узлов, соединенных отрезками (*ребрами*), вместе с правилами и алгоритмами обхода графов.

Субподрядчик реализовал программную среду для трассировки графов в виде СВЯЗНОГО МЕХАНИЗМА (COHESIVE MECHANISM). В этой среде использовалась стандартная терминология графов и алгоритмы, знакомые большинству программистов-прикладников и подробно описанные в учебниках. Реализованный граф ни в коей мере не имел самого общего характера. Это было подмножество концептуального набора понятий, охватывающее только функции, необходимые для нашей организационной модели. Но имея ИНФОРМАТИВНЫЙ ИНТЕРФЕЙС (INTENTION-REVEALING INTERFACE), не стоит слишком беспокоиться о средствах, которыми "добывается" правильный ответ.

Теперь в организационной модели можно было просто декларировать, используя стандартную терминологию графов, что каждый человек — это узел, а каждая взаимосвязь между людьми — это ребро графа, соединяющее узлы. После этого механизм среды для работы с графами предположительно был способен найти взаимосвязь между любыми двумя людьми.

Если бы механизм был встроен в модель предметной области, это обернулось бы двумя неприятностями. Во-первых, модель была бы привязана к конкретному методу решения задачи, что ограничивало бы варианты будущего развития. Во-вторых, что более существенно, модель организации стала бы гораздо сложнее и запутаннее. Разделение механизма и модели позволило описывать организации в декларативном стиле, что гораздо нагляднее. А сложный код для манипулирования графами был вынесен в чисто механистическую вспомогательную среду и основывался на проверенных алгоритмах, которые можно было сопровождать и тестировать отдельно.

Еще один пример СВЯЗНОГО МЕХАНИЗМА — это среда для конструирования объектов-СПЕЦИФИКАЦИЙ с поддержкой базовых операций сравнения и комбинирования. Если реализовать такую среду, то СМЫСЛОВОЕ ЯДРО (CORE DOMAIN) и НЕСПЕЦИАЛИЗИРОВАННЫЕ ПОДОБЛАСТИ (GENERIC SUBDOMAINS) получат возможность объявлять нужные СПЕЦИФИКАЦИИ на четком, легко понятном языке, как описано в соответствующем шаблоне (см. главу 10). Сложные операции по проведению сравнений и комбинирования можно полностью вынести во вспомогательную среду.

* * *

Сравнение связных механизмов и неспециализированных подобластей

Как НЕСПЕЦИАЛИЗИРОВАННЫЕ ПОДОБЛАСТИ (GENERIC SUBDOMAINS), так и СВЯЗНЫЕ МЕХАНИЗМЫ (COHESIVE MECHANISMS) вводятся тогда, когда есть необходимость разгрузить СМЫСЛОВОЕ ЯДРО (CORE DOMAIN). Разница состоит в принимаемых ими на себя обязанностях. НЕСПЕЦИАЛИЗИРОВАННАЯ ПОДОБЛАСТЬ основана на смысловой модели, которая представляет определенные аспекты взгляда разработчиков на общую проблему. В этом она принципиально не отличается от СМЫСЛОВОГО ЯДРА, только является менее центральной, важной и узкоспециализированной. СВЯЗНЫЙ МЕХАНИЗМ же никоим образом не представляет предметную область; он решает некую частную вычислительную задачу, поставленную смысловыми моделями.

Итак, модель говорит "надо" — СВЯЗНЫЙ МЕХАНИЗМ отвечает "есть!".

На практике, за исключением тех случаев, когда перед вами четко формализованный и общеизвестный метод расчета, это различие не столь уж очевидно — особенно на первый взгляд. В ходе последовательного рефакторинга можно либо дистиллировать более четкий механизм, либо прийти к НЕСПЕЦИАЛИЗИРОВАННОЙ ПОДОБЛАСТИ с набором ранее не опознанных понятий, которые этот механизм упрощают.

Когда механизм входит в смысловое ядро

МЕХАНИЗМЫ почти всегда необходимо удалять из СМЫСЛОВОГО ЯДРА модели (CORE DOMAIN). Единственное исключение — когда МЕХАНИЗМ сам по себе является неотъемлемой и ключевой частью программы, создающей ее конкурентную ценность. Так иногда бывает с узкоспециализированными алгоритмами. Например, если бы одну из характерных черт программы по управлению грузопоставками составлял особенно эффективный алгоритм для составления графиков перемещений грузов, то такой МЕХАНИЗМ можно было бы считать частью концептуального ЯДРА. Я в свое время работал в проекте для инвестиционного банка, в котором их собственные специализированные алгоритмы оценки риска совершенно точно принадлежали к СМЫСЛОВОМУ ЯДРУ. (На самом деле конфиденциальность этих алгоритмов охранялась столь тщательно, что даже большинство разработчиков ЯДРА не имели возможности с ними ознакомиться.) Алгоритмы, вероятно, представляли собой конкретную реализацию набора правил, действительно прогнозирующих риск. Если провести более глубокий анализ, можно было бы построить углубленную модель с явной формулировкой этих правил и инкапсуляцией вычислительного механизма.

Но это было бы уже дополнительное усовершенствование архитектуры, следующий шаг в проекте. Решение о том, делать ли этот шаг, нужно было принимать по итогам анализа выгоды и издержек: насколько трудно было бы разработать новую архитектуру? Насколько трудно понимать и дорабатывать текущую? Насколько упростилась бы работа при новой, усовершенствованной архитектуре для конкретного состава сотрудников? И, конечно же, имеет ли кто-нибудь представление, какой вид может принять новая модель?

Пример

Круг замкнулся: организационная диаграмма поглотила собственный механизм

Если быть точным, через год после того, как мы завершили организационную модель из предыдущего примера, другие разработчики перепроектировали ее и ликвидировали выделение работы с графами в отдельную среду. Им показалось, что наличие дополнительных объектов и усложнение структуры в виде выделения МЕХАНИЗМА в отдельный пакет необоснованны. Вместо этого они добавили операции над узлами графов в родительский класс организационных объектов-СУЩНОСТЕЙ (ENTITIES). При этом разработчики все-таки оставили декларативный *public*-интерфейс организационной модели. Сохранилась даже инкапсуляция МЕХАНИЗМА, но уже в пределах СУЩНОСТЕЙ организационной модели.

Часто кажется, что, совершив полный круг, мы приходим к тому, что уже было. Но это не совсем так. В результате обычно возникает углубленная модель, в которой более четко различаются факты, цели и МЕХАНИЗМЫ. Рациональный рефакторинг сохраняет то ценное, что накоплено на промежуточных этапах работы, но при этом убирает ненужные осложнения.

Дистилляция к декларативному стилю

Декларативная архитектура и "декларативный стиль" программирования составляли предмет главы 10, но они заслуживают особого упоминания и здесь, в главе о стратегической дистилляции. Смысл дистилляции состоит в том, чтобы сознательно пробиваться к сути, не давая отвлечь себя несущественными деталями. Важные части СМЫСЛОВОГО ЯДРА (CORE DOMAIN) могут следовать декларативному стилю тогда, когда вспомогательные элементы архитектуры обеспечивают поддержку экономичного языка для выражения понятий и правил ЯДРА, в то же время инкапсулируя методы вычислений или реализации правил.

СВЯЗНЫЕ МЕХАНИЗМЫ (COHESIVE MECHANISMS) наиболее полезны тогда, когда предоставляют доступ через ИНФОРМАТИВНЫЕ
с концептуально связными КОНТРОЛЬНЫМИ УТВЕРЖДЕНИЯМИ (ASSERTIONS) и ФУНКЦИЯМИ БЕЗ ПОБОЧНЫХ ЭФФЕКТОВ (SIDE-EFFECT-FREE FUNCTIONS). МЕХАНИЗМЫ и гибкая архитектура (*supple design*) позволяют СМЫСЛОВОМУ ЯДРУ оперировать осмысленными декларациями вместо вызова малопонятных функций. Но самый исключительный результат получается тогда, когда часть СМЫСЛОВОГО ЯДРА совершает качественный скачок до уровня углубленной модели и начинает функционировать как язык, на котором можно выразить рабочие сценарии приложения в гибкой и краткой форме.

Углубленная модель часто возникает вместе с сопутствующей ей гибкой архитектурой. Когда гибкая архитектура достигает зрелости, она превращается в наглядный набор элементов, которые можно сочетать по однозначным правилам для решения сложных задач или передачи сложной информации — так же, как слова сочетаются в предложения. На этом этапе развития клиентский код приобретает декларативный стиль и высокую степень дистилляции.

Выделение НЕСПЕЦИАЛИЗИРОВАННЫХ ПОДОБЛАСТЕЙ (GENERIC SUBDOMAINS) путем рефакторинга делает код менее запутанным, а СВЯЗНЫЕ МЕХАНИЗМЫ (COHESIVE MECHANISMS) позволяют инкапсулировать сложные операции. В результате возникает более сфокусированная модель, с меньшим количеством посторонних факторов, которые не добавляют к процессу выполнения пользовательских операций ничего ценного. Маловероятно, что в модели предметной области найдется удачное место для всего того, что не относится к СМЫСЛОВОМУ ЯДРУ. В шаблоне ВЫДЕЛЕННОГО ЯДРА (SEGREGATED CORE) предпринимается непосредственная попытка структурно отделить СМЫСЛОВОЕ ЯДРО от всего остального...

Выделенное ядро

Элементы модели могут частично работать на СМЫСЛОВОЕ ЯДРО **(**CORE DOMAIN**), а частично играть вспомогательные роли. Элементы ядра бывают тесно связаны с неспециализированными элементами. Концептуальная связность** ЯДРА **может быть не слишком сильной или плохо различимой. Смысл** ЯДРА **теряется, если оно слишком загромождено или расплывчато. Разработчики недостаточно четко видят наиболее важные взаимосвязи, отчего страдает качество архитектуры.**

Выделяя путем рефакторинга НЕСПЕЦИАЛИЗИРОВАННЫЕ ПОДОБЛАСТИ (GENERIC SUBDOMAINS), мы "вычищаем" из предметной области часть загромождающих ее подробностей, и этим делаем СМЫСЛОВОЕ ЯДРО более четко различимым. Но найти и разграничить все эти подобласти — дело нелегкое, иногда кажущееся бесполезным. А между тем главное в модели — СМЫСЛОВОЕ ЯДРО — остается замусоренным.

Выполните рефакторинг модели так, чтобы отделить понятия СМЫСЛОВОГО ЯДРА **(**CORE DOMAIN**) от вспомогательных элементов (включая неудачно определенные) и усилить связность** ЯДРА**, одновременно снижая его зависимость от остального кода. Факторизуйте все неспециализированные или вспомогательные элементы в другие объекты и поместите их в другие пакеты, даже если в ходе такого рефакторинга модели будут отделены сильно зависимые элементы.**

Здесь применяются, в общем, те же принципы, что и в НЕСПЕЦИАЛИЗИРОВАННЫХ ПОДОБЛАСТЯХ (GENERIC SUBDOMAINS), но с другой стороны. Связные подобласти, являющиеся центральными для нашего приложения, обнаруживаются и выделяются в свои собственные связные пакеты. Что делать с бесформенной массой, оставшейся позади — тоже важно, но не настолько. Ее можно оставить более-менее в том же виде или разделить по пакетам, выделив основные классы. Со временем все больше и больше этой остаточной массы будет выделено рефакторингом в НЕСПЕЦИАЛИЗИРОВАННЫЕ ПОДОБЛАСТИ, но в ближней перспективе подойдет любое несложное решение, лишь бы фокус внимания удерживался на ВЫДЕЛЕННОМ ЯДРЕ (SEGREGATED CORE).

* * *

Обычно рефакторинг, порождающий ВЫДЕЛЕННОЕ ЯДРО, состоит из следующих этапов.

1. Определяется подобласть СМЫСЛОВОГО ЯДРА (иногда ее можно взять из дистилляционного документа).
2. Имеющие к ней отношение классы перемещаются в новый МОДУЛЬ под именем, соответствующим основному понятию, которое их объединяет.
3. Выполняется рефакторинг кода, при котором "отрезаются" данные и функциональные возможности, не выражающие это понятие непосредственно. Удаленные аспекты выносятся в классы (возможно, новые) других пакетов. Можно постараться объединить их с концептуально родственными задачами, но не тратить слишком много времени на поиск идеального решения. Основное внимание следует уделить очистке подобласти ЯДРА, а также сделать ссылки из него на другие пакеты явными и наглядными.
4. Выполняется рефакторинг нового МОДУЛЯ ВЫДЕЛЕННОГО ЯДРА с целью упрощения и повышения информативности взаимосвязей и взаимодействий в нем, а также для минимизации и прояснения его взаимосвязей с другими модулями. (Это становится постоянной целью рефакторинга.)
5. Вышеописанное повторяется с другой подобластью ЯДРА, и так до тех пор, пока ВЫДЕЛЕННОЕ ЯДРО не будет построено.

Цена создания выделенного ядра

Выделение ЯДРА иногда делает отношения с тесно взаимосвязанными классами, не входящими в ЯДРО, менее очевидными или даже более сложными. Но эта цена невелика по сравнению с такими преимуществами, как прояснение СМЫСЛОВОГО ЯДРА (CORE DOMAIN) и облегчение работы с ним.

Создание ВЫДЕЛЕННОГО ЯДРА (SEGREGATED CORE) позволяет повысить связность СМЫСЛОВОГО ЯДРА. Есть много рациональных способов разбиения модели на части. Иногда при создании ВЫДЕЛЕННОГО ЯДРА может нарушиться целостность хорошо организованного МОДУЛЯ — его связность фактически приносится в жертву связности самого СМЫСЛОВОГО ЯДРА. Но в итоге получается чистая прибыль, так как наибольшая до-

бавленная стоимость корпоративного приложения содержится в профессиональных, специализированных аспектах модели.

За выделение ядра приходится платить и такую цену, как большие трудозатраты. Следует признать, что решение о создании выделенного ядра нагружает программистов работой по внесению изменений во все части системы.

Момент для создания ВЫДЕЛЕННОГО ЯДРА наступает тогда, когда у вас есть большой ОГРАНИЧЕННЫЙ КОНТЕКСТ (BOUNDED CONTEXT), критически важный для системы, но при этом существенная часть модели запутана большим количеством вспомогательных функций.

Эволюция коллективных решений

Как и в случае многих других стратегических проектных решений, решение о переходе к ВЫДЕЛЕННОМУ ЯДРУ (SEGREGATED CORE) должно приниматься всей группой разработки совместно. Для этого требуется, чтобы в группе была налажена процедура коллективного принятия решений, а также присутствовала достаточная дисциплина и скоординированность для воплощения этих решений. Основная трудность состоит в том, чтобы заставить всех следовать одному и тому же определению ЯДРА, не закрепляя "намертво" само определение. СМЫСЛОВОЕ ЯДРО (CORE DOMAIN) эволюционирует так же, как и любой другой аспект архитектуры, поэтому опыт работы с ВЫДЕЛЕННЫМ ЯДРОМ приводит к новым выводам о том, что в нем составляет суть, а что является вспомогательным элементом. Эти выводы должны влиять через обратную связь на определение СМЫСЛОВОГО ЯДРА и МОДУЛЕЙ ВЫДЕЛЕННОГО ЯДРА.

Все это означает, что новые выводы и знания следует регулярно распространять в группе, но отдельные программисты (или пары программистов) не должны действовать на их основе односторонне. Какова бы ни была процедура принятия коллективных решений (достижение консенсуса или волевое решение руководителя группы), она должна обладать достаточной гибкостью для регулярной коррекции курса действий. Коммуникация в группе должна быть достаточно налаженной, чтобы все в группе имели одинаковое видение ЯДРА.

Пример

Выделение ядра в модели грузопоставок

Начнем с модели, показанной на рис. 15.2, как основы для координирования грузопоставок.

Следует заметить, что модель сильно упрощена по сравнению с той, которая потребовалась бы для реального приложения. Но реальная модель была бы слишком громоздкой для примера. Поэтому, пусть даже пример и недостаточно сложен, чтобы побуждать к построению ВЫДЕЛЕННОГО ЯДРА (SEGREGATED CORE), давайте подключим воображение и будем относиться к этой модели как к слишком сложной для интерпретации и восприятия в виде единого целого.

Так в чем же суть модели грузопоставок? Часто бывает полезно начать анализ с конца. Это может привести нас к вопросам ценообразования и счетов-фактур. Но в данном случае полезнее взглянуть на ВВЕДЕНИЕ В ПРЕДМЕТНУЮ ОБЛАСТЬ (DOMAIN VISION STATEMENT). Вот отрывок из него:

> *...Улучшить наглядность операций и предоставить средства для более быстрого и надежного выполнения требований клиентов...*

Это приложение проектируется не для отдела сбыта. Оно будет использоваться основными операторами компании. Поэтому отнесем все вопросы денежных расчетов к вспомогательным (хотя и несомненно важным). Некоторые из этих элементов уже вынесены до нас в отдельный пакет **Billing (Выставление счетов)**. Мы можем согласиться с этим решением, которое подтверждает вспомогательный характер денежных вопросов.

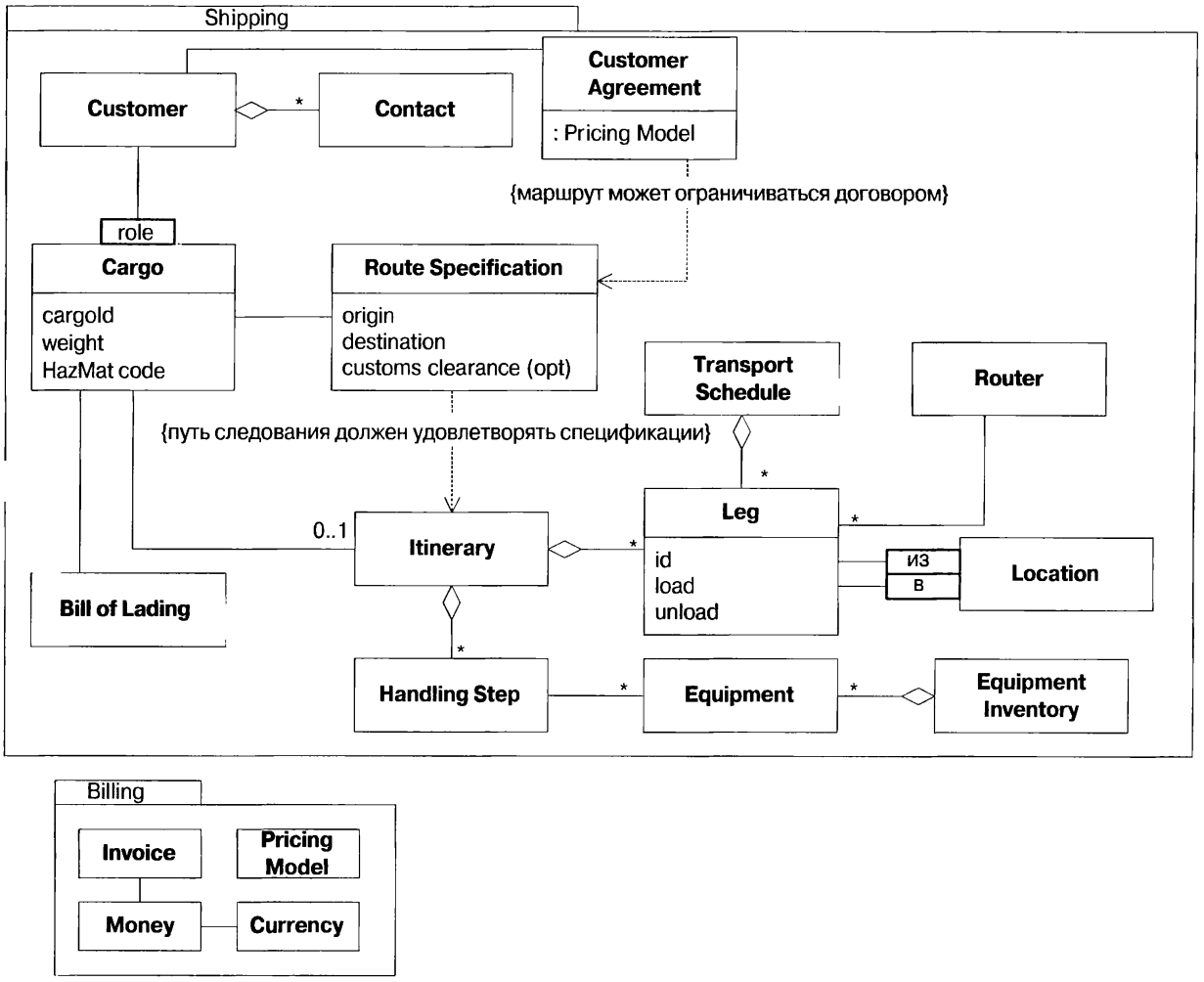


Рис. 15.2.

В фокусе у нас должны находиться вопросы манипуляций с грузом: доставка его клиенту согласно выдвинутым требованиям. Извлечение классов, непосредственно занятых в этой деятельности, позволяет вынести ВЫДЕЛЕННОЕ ЯДРО в новый пакет под названием **Delivery (Доставка)**, как показано на рис. 15.3.

Фактически классы просто "переехали" в новый пакет. Но все-таки и в самой модели произошло несколько изменений.

Во-первых, возможные **Манипуляции (Handling Step)** теперь ограничены **Договором с клиентом (Customer Agreement)**. Это типичное возникновение новых знаний о предмете при выделении ЯДРА разработчиками. По мере концентрации внимания на качестве и эффективности доставки становится ясно, что связи-ограничения, накладываемые **Договором с клиентом**, играют фундаментальную роль и должны быть прописаны в модели *явно*.

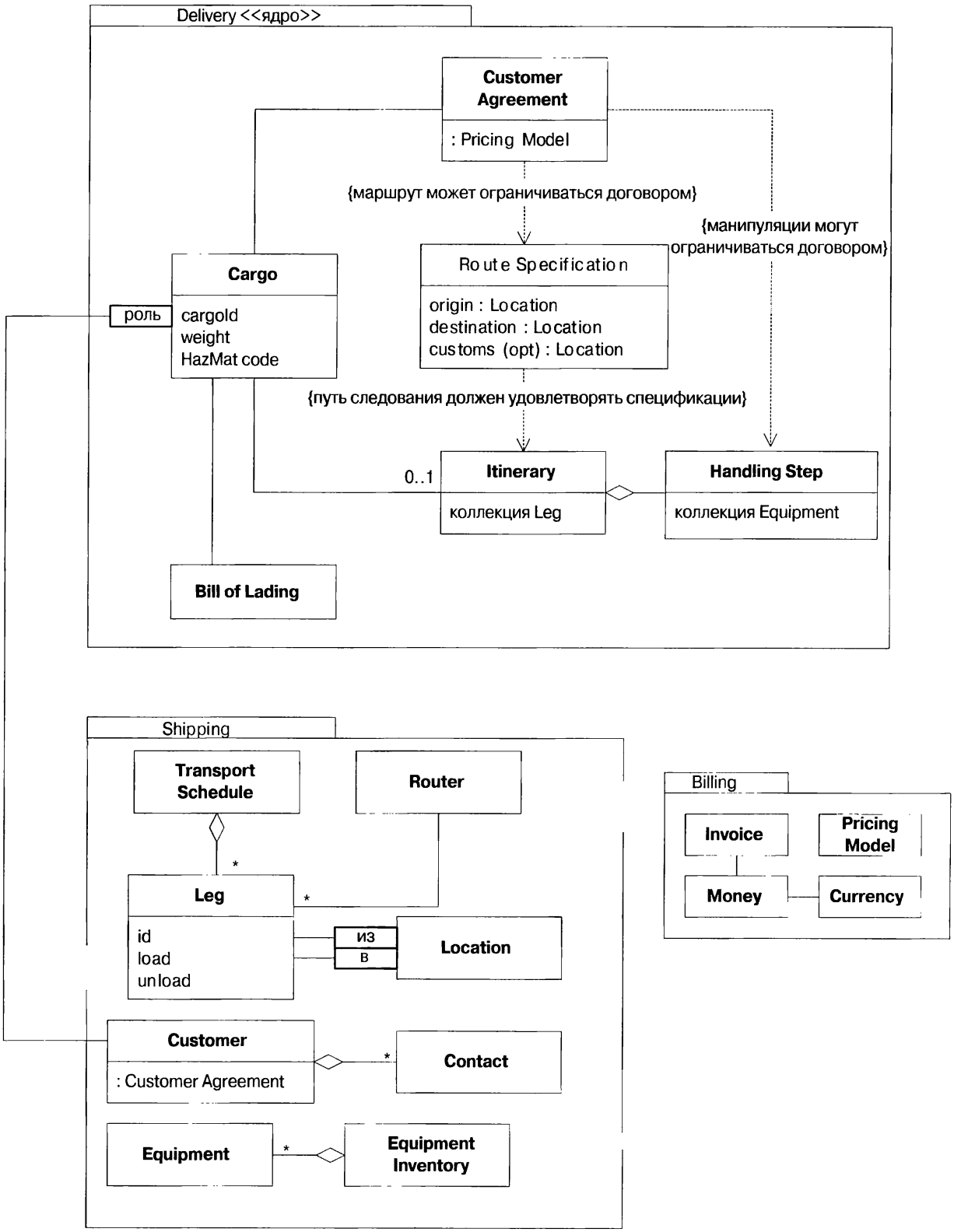


Рис. 15.3. Ключевая цель проекта (его смысловое ядро) — надежная доставка груза в соответствии с требованиями пользователя

Другое изменение носит более прагматический характер. В модели после рефакторинга **Договор с клиентом** (**Customer Agreement**) ассоциирован с **Грузом** (**Cargo**) напрямую, а не опосредованно через **Клиента** (**Customer**). Это ассоциирование выполняется при заказе груза, как и ассоциирование самого **Клиента**. В процессе собственно доставки

груза сам **Клиент** не так важен для правильного выполнения операций, как договор с ним. В другой модели требовалось сперва найти нужного **Клиента** согласно роли, которую тот играл при доставке, а потом запросить у него нужный **Договор**. Эта взаимосвязь загромождала любой сценарий, который можно было выстроить по модели. Новая же ассоциация упрощает основные рабочие сценарии и придает им прямой, непосредственный характер. Теперь можно вообще вынести **Клиента (Customer)** за пределы модели.

Кстати, как обстоят дела с этим вынесением? Основное внимание уделяется выполнению требований **Клиента**, поэтому на первый взгляд он должен принадлежать к СМЫСЛОВОМУ ЯДРУ. Но теперь, когда **Договор с клиентом (Customer Agreement)** стал доступен напрямую, во взаимодействиях между объектами в ходе доставки груза собственно объект **Клиент (Customer)** как раз не участвует. К тому же базовая модель **Клиента** имеет довольно общий характер.

Есть веские доводы в пользу сохранения в ЯДРЕ объекта **Участок пути (Leg)**. ЯДРО должно носить минималистский характер, а **Участок** достаточно тесно связан с **Графиком движения транспорта (Transport Schedule)**, **Службой маршрутизации (Routing Service)** и **Местоположением (Location)**. Однако ни один из этих объектов в ЯДРЕ не нужен. Но если **Участок пути** стал бы фигурировать во многих сценариях, которые можно построить по модели, то я бы переместил его в пакет **Delivery (Доставка)** и смирился бы с неудобствами его отделения от перечисленных классов.

В этом примере все определения классов остались такими же, какими были раньше. Но при дистилляции часто требуется рефакторинг и самих классов, чтобы разделить в них неспециализированные и специфические для предметной области обязанности, а потом выполнить выделение ЯДРА.

Как только у нас появилось ВЫДЕЛЕННОЕ ЯДРО (SEGREGATED CORE), рефакторинг завершен. Но пакет **Shipping (Поставка)** представляет собой остаточную "бесформенную массу", образовавшуюся после выделения ЯДРА. К нему можно применить новый рефакторинг для более содержательной перепаковки модулей, как показано на рис. 15.4.

Чтобы добиться этого результата, может понадобиться несколько рефакторингов; не обязательно делать все сразу. Здесь у нас в конце концов получился один пакет с ВЫДЕЛЕННЫМ ЯДРОМ (SEGREGATED CORE), одна НЕСПЕЦИАЛИЗИРОВАННАЯ ПОДОБЛАСТЬ (GENERIC SUBDOMAIN) и два пакета, специализированных по предметной области, но во вспомогательных ролях. Углубленный анализ мог бы со временем привести к созданию неспециализированной подобласти для класса **Клиент (Customer)** или же этот класс стал бы более специализированным в области грузопоставок.

Работа по выделению полезных и информативных модулей относится к области моделирования (см. главу 5). В ходе стратегической дистилляции имеет место сотрудничество между разработчиками и специалистами в предметной области, так как это часть процесса переработки знаний (*knowledge crunching*).

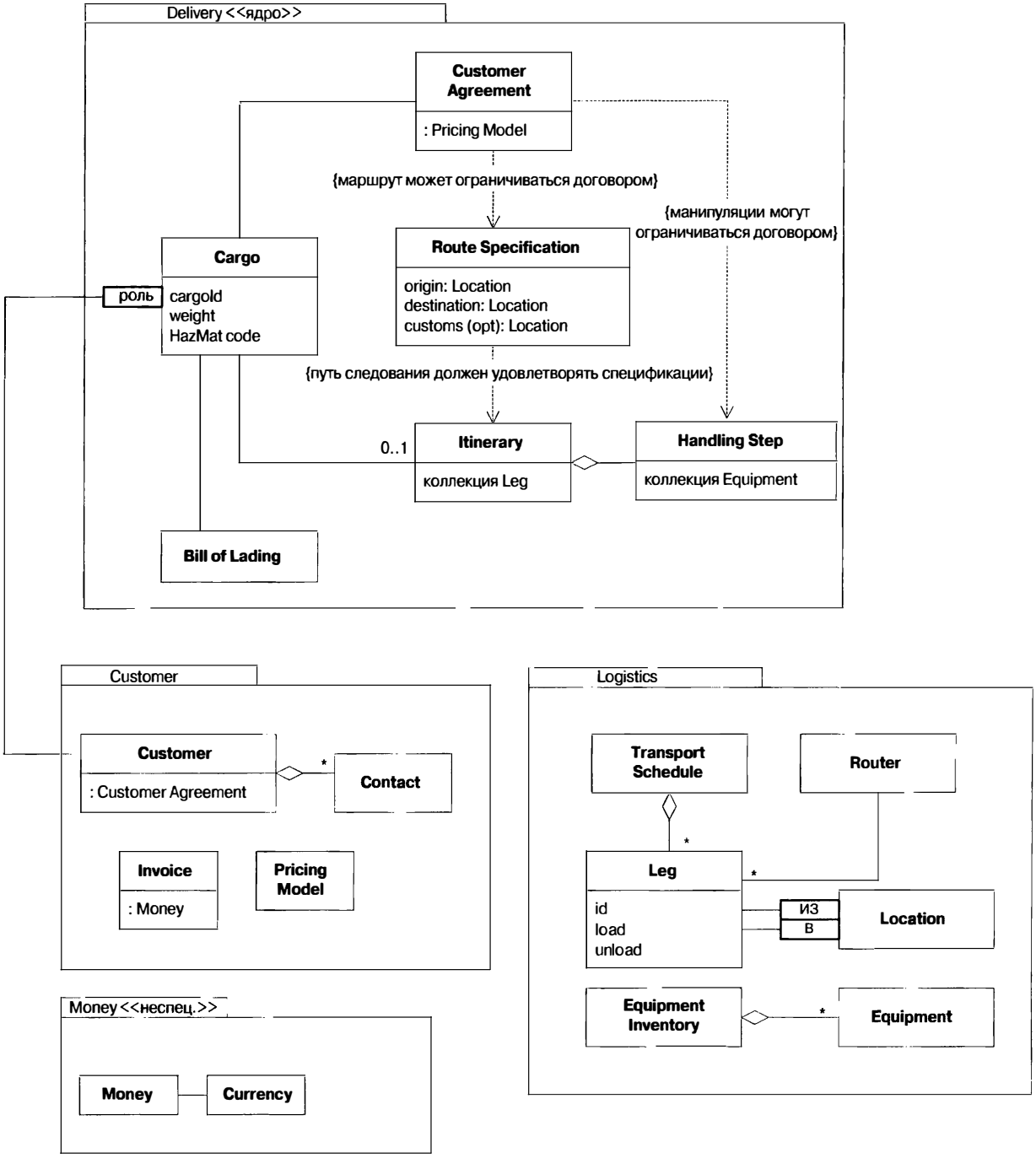
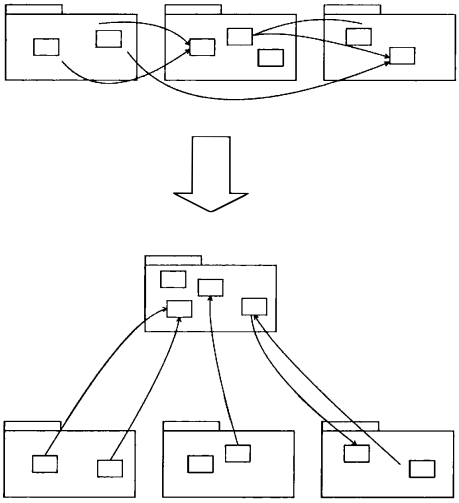


Рис. 15.4. Перепаковка МОДУЛЕЙ для подобластей, не входящих в ЯДРО, после построения ВЫДЕЛЕННОГО ЯДРА

Абстрактное ядро



Даже СМЫСЛОВОЕ ЯДРО (CORE DOMAIN) модели иногда может иметь такие размеры, что показать общий план оказывается не так-то просто.

* * *

Мы обычно работаем с большими моделями, разбивая их на более узкие подобласти, обозримые по своим размерам, и помещаем их в отдельные МОДУЛИ. Такой исключающий стиль модульной организации часто помогает сделать сложную модель более управляемой и поддающейся обработке. Но иногда создание отдельных МОДУЛЕЙ запутывает или усложняет взаимосвязи между подобластями.

Если между подобластями, находящимися в отдельных МОДУЛЯХ**, происходит интенсивное взаимодействие, то либо приходится создавать много ссылок между ними, что частично лишает смысла само модульное разбиение, либо организовывать косвенные способы взаимодействия, что ухудшает наглядность модели.**

Подумайте над возможностью установить горизонтальные, а не вертикальные границы. Полиморфизм дает нам право игнорировать детали различий между экземплярами абстрактного типа. Если большинство взаимодействий между МОДУЛЯМИ можно выразить в виде полиморфных интерфейсов, то, может быть, имеет смысл выделить эти типы рефакторингом в отдельный МОДУЛЬ ЯДРА.

Здесь нас не интересуют чисто технические подробности. Ценным этот прием становится только тогда, когда полиморфизм интерфейсов соответствует фундаментальным понятиям предметной области. В этом случае отделение абстракций позволяет снизить взаимную зависимость МОДУЛЕЙ и одновременно дистиллировать небольшое по размеру, более связное СМЫСЛОВОЕ ЯДРО (CORE DOMAIN).

Определите наиболее фундаментальные понятия в модели и выделите их рефакторингом в отдельные классы, абстрактные классы или интерфейсы. Спроектируйте эту абстрактную модель так, чтобы она выражала большую часть взаимодействий между существенными компонентами. Поместите эту абстрактную модель в собственный МОДУЛЬ**, оставив более специализированные классы конкретных программных реализаций в их собственных** МОДУЛЯХ**, определяемых подобластью.**

Большинство специализированных классов теперь будет ссылаться на МОДУЛЬ АБСТРАКТНОГО ЯДРА, но не на другие специализированные МОДУЛИ. АБСТРАКТНОЕ ЯДРО (ABSTRACT CORE) дает самое сжатое представление основных понятий и взаимодействий между ними.

Процесс выделения-факторизации АБСТРАКТНОГО ЯДРА не носит механического характера. Например, если автоматически перенести в отдельный МОДУЛЬ все классы, на

которые часто ссылаются другие МОДУЛИ, в результате, скорее всего, получится "каша". Моделирование АБСТРАКТНОГО ЯДРА требует глубокого понимания ключевых понятий предмета и ролей, которые они играют в основных взаимодействиях внутри системы. Другими словами, это пример углубляющего рефакторинга, при котором обычно происходит серьезная перестройка архитектуры.

АБСТРАКТНОЕ ЯДРО должно в итоге стать похожим на дистилляционный документ (если бы оба применялись в одном проекте, и дистилляционный документ развивался бы вместе с приложением по мере накопления знаний и углубления понимания). Разумеется, АБСТРАКТНОЕ ЯДРО выражается в виде кода, а потому имеет более строгий и завершенный вид.

* * *

Дистилляция в углубленных моделях

Дистилляция — это не только примитивное отделение тех или иных частей предметной области от СМЫСЛОВОГО ЯДРА (CORE DOMAIN). В ходе дистилляции эти подобласти совершенствуются, особенно само СМЫСЛОВОЕ ЯДРО, посредством непрерывного углубляющего рефакторинга. Происходит движение в сторону углубленной модели и гибкой архитектуры. Целью является такая архитектура, которая бы сделала модель очевидной, и модель, выражающая предметную область в самой простой форме. Углубленная модель сама дистиллирует наиболее существенные аспекты предметной области в простые элементы, сочетание которых дает возможность решать важные практические задачи, стоящие перед приложением.

Хотя качественный скачок к углубленной модели приносит пользу везде, где он случается, все же именно в СМЫСЛОВОМ ЯДРЕ **(**CORE DOMAIN**) он может коренным образом изменить характер всего проекта.**

Выбор целей рефакторинга

Когда вам попадается плохо организованная большая система, с чего нужно начинать ее реорганизацию? В сообществе сторонников экстремального программирования обычно дают один из следующих ответов на этот вопрос.

1. Начинайте откуда угодно, все равно нужен полный рефакторинг всей системы.
2. Начинайте с самого наболевшего. Рефакторинг нужен там, где именно сейчас требуется решить конкретную задачу.

Я не могу согласиться ни с одной из этих точек зрения. Первый подход практически нереализуем, за редкими исключениями, когда в проекте работают сплошь большие профессионалы, причем в достаточном количестве. Во втором случае есть тенденция обходить острые углы, лечить симптомы, игнорируя причины, уклоняться от работы с самыми сложными и запутанными местами. В итоге код будет все хуже и хуже поддаваться рефакторингу.

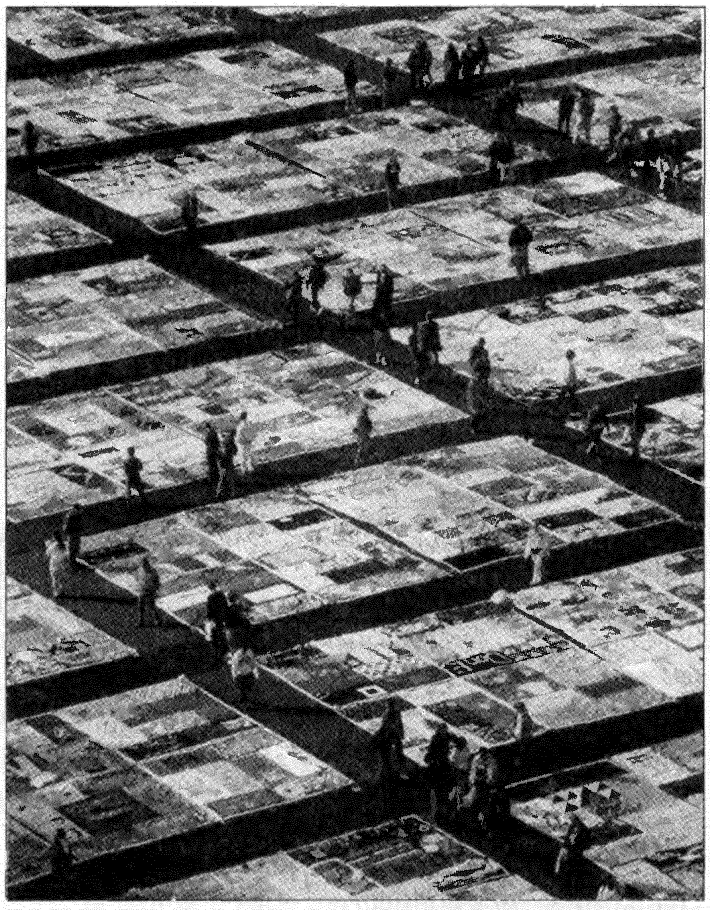
Итак, если не браться за все сразу, и не лечить только там, где болит, что же тогда делать?

1. В рефакторинге "по наболевшему" следует проверить, есть ли проблема в СМЫСЛОВОМ ЯДРЕ (CORE DOMAIN) или в его взаимосвязях со вспомогательными элементами. Если проблема есть, решите ее.
2. Если вы можете позволить себе роскошь решать, с чего начать, начните с реорганизации СМЫСЛОВОГО ЯДРА, более четкого выделения этого ЯДРА, очистки вспомогательных подобластей до состояния НЕСПЕЦИАЛИЗИРОВАННЫХ (GENERIC SUBDOMAINS).

Только так можно получить максимальную отдачу от приложенных усилий.

Глава 16

Крупномасштабная структура



Тысячи людей, работая независимо друг от друга, создали мозаичное панно[1] памяти жертв СПИДа

Как-то раз небольшую проектную фирму из Кремниевой Долины (Silicon Valley) подрядили разработать симулятор для системы спутниковой связи. Работа шла нормально, разрабатывалась архитектура, основанная на модели (MODEL-DRIVEN DESIGN), которая выражала и моделировала широкий спектр состояний и отказов сети.

Но ведущие разработчики проекта были недовольны — задача была сложная. Ощущая потребность прояснить запутанные взаимосвязи в модели, разработчики разбили

[1] Дословно "лоскутное одеяло" — *quilt*. Дальше в главе речь пойдет о его наборе из лоскутов ткани. *Примеч. перев.*

архитектуру на внутренне связные МОДУЛИ обозримого размера. И в итоге у них стало очень много МОДУЛЕЙ. В каком пакете программисту нужно искать тот или иной аспект функциональных возможностей? Куда поместить новый класс? Что конкретно имеется в виду в некоторых из этих "мелко нарезанных" пакетов? Как они склеиваются в единое целое? А приложению было еще далеко до завершения.

Разработчики хорошо взаимодействовали друг с другом, нормально ставили и решали повседневные задачи, но руководители проекта испытывали беспокойство от того, что проект находился на грани понятности и вразумительности. Они хотели каким-то образом организовать архитектуру так, чтобы она стала понятной и управляемой при переходе на следующий уровень сложности.

Был проведен "мозговой штурм", выдвинуто много вариантов, предложены альтернативные схемы распределения по пакетам. Можно было бы сделать обзор системы в каком-нибудь документе или же использовать представления диаграммы классов в программе моделирования для перехода в нужный МОДУЛЬ. Но руководству проекта было недостаточно этих трюков.

Разработчики могли просто и наглядно рассказать о процессе работы симулятора, о прохождении данных через инфраструктуру, о поддержании их целостности и маршрутизации с помощью многоуровневой системы телекоммуникационных технологий. Все детали процесса были заложены в модели, но вот общего, широкого взгляда на нее не хватало.

Недоставало некоей общей идеи, концепции из предметной области. И это были не два-три отсутствующих в модели класса. Здесь вся модель в целом была лишена структуры.

Разработчики посидели над решением проблемы неделю-другую, и идея начала выкристаллизовываться. Нужно было наложить определенную структуру на имеющуюся систему. Весь симулятор должен был представляться в виде последовательности уровней, описывающих разные аспекты телекоммуникационной системы. Нижний уровень представлял бы физическую инфраструктуру, т.е. способность передавать цифровые данные с одного узла на другой. Затем был бы уровень маршрутизации пакетов, в котором решались бы задачи перенаправления потоков данных. Другие концептуальные уровни проблемы представлялись бы другими архитектурными уровнями приложения. *Вся совокупность слоев отражала бы рабочий процесс в системе.*

И разработчики приступили к рефакторингу кода в соответствии с новой структурой. Необходимо было переопределить МОДУЛИ так, чтобы разграничить уровни. В некоторых случаях выполнялся рефакторинг обязанностей объектов, чтобы четко вынести каждый объект на свой уровень. Соответственно, в ходе этого сами концептуальные определения уровней претерпели изменения по итогам их практического применения. Уровни, МОДУЛИ и объекты развивались совместно, пока в конце концов вся архитектура не приобрела многоуровневый характер, как и было задумано.

Эти уровни не были ни МОДУЛЯМИ, ни какими-либо другими единицами кода. Они представляли собою совокупность правил более высокого уровня, определяющих границы и взаимоотношения определенного МОДУЛЯ или объекта в целостной архитектуре и даже при взаимодействии с другими системами.

Введение такой упорядоченности вернуло архитектуру в состояние, удобное для понимания. Теперь разработчики примерно представляли себе, где искать ту или иную функцию. Программисты, работавшие отдельно, могли принимать проектные решения, в целом совместимые друг с другом. Уровень сложности был поднят на новую высоту.

Даже при наличии модульного разбиения большая модель может оказаться слишком сложной. МОДУЛИ делят архитектуру приложения на обозримые фрагменты, но их может оказаться слишком много. Кроме того, модульность не обязательно привносит в ар-

хитектуру единообразие. Объект за объектом, пакет за пакетом нагромождаются хотя и необходимые, но не подчиненные единой идее проектные решения.

Строгое разграничение, создаваемое ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ (BOUNDED CONTEXTS), предотвращает путаницу, но само по себе еще не облегчает видение системы в целом.

Дистилляция полезна тем, что фокусирует внимание на СМЫСЛОВОМ ЯДРЕ (CORE DOMAIN) и отводит другим подобластям вспомогательные роли. Но все-таки остается еще необходимость понять вспомогательные элементы и их взаимосвязи как со СМЫСЛОВЫМ ЯДРОМ, так и друг с другом. В идеале СМЫСЛОВОЕ ЯДРО должно быть настолько ясным и четким, чтобы не требовать дополнительных пояснений, но не всегда нам удается выйти на этот уровень.

В проекте любого размера программистам приходится работать над разными частями системы более-менее независимо. Без координации и четких правил возникает путаница из разных стилей и разных решений одних и тех же задач, из-за чего становится трудно понять связь между частями целого и вовсе невозможно — представить себе общую картину. Изучение одной части архитектуры не "выводит" на другие ее части, и в конце концов в проекте собираются специалисты по разным МОДУЛЯМ, которые не способны помочь друг другу за пределами своей узкой компетенции. НЕПРЕРЫВНАЯ ИНТЕГРАЦИЯ (CONTINUOUS INTEGRATION) нарушается, ОГРАНИЧЕННЫЙ КОНТЕКСТ (BOUNDED CONTEXT) фрагментируется.

В большой системе, лишенной общего руководящего принципа, который позволяет воспринимать элементы с точки зрения их ролей в структурах, распространяющихся на масштаб всей архитектуры приложения, разработчики не видят леса за деревьями. Необходимо понимать роль отдельной части в целом, не углубляясь в излишние подробности этого целого.

"Крупномасштабная структура" — это своего рода язык, на котором можно обсуждать и описывать систему грубыми штрихами. Архитектурная схема для всей системы задается набором высокоуровневых концепций или правил (или того и другого вместе). Такой принцип организации направляет проектирование архитектуры и одновременно способствует лучшему пониманию. Облегчается координирование независимых разработчиков, потому что у них теперь есть общая картина, показывающая, как роли отдельных частей способствуют формированию целого.

Разработайте схему правил или ролей/взаимоотношений, распространяющуюся на всю систему и позволяющую понять место каждой части в едином целом — пусть даже без подробного знания обязанностей всех этих частей.

Структурная схема может ограничиваться и одним КОНТЕКСТОМ, но, как правило, распространяется на несколько, обеспечивая концептуальную совместную организацию групп разработчиков и подсистем, участвующих в проекте. Хорошая структура помогает разобраться в модели и дополняет дистилляцию.

Большинство крупномасштабных структур нельзя представить диаграммами UML, да это и не нужно. Такие структуры определяют форму модели и архитектуры, объясняют ее, но сами в ней не фигурируют. Это дополнительный уровень информации о программной архитектуре. В примерах к этой главе вы увидите много неформальных UML-диаграмм, на которые наложена информация о соответствующей крупномасштабной структуре.

Когда группа разработчиков невелика, а их модель не слишком сложна, для ее структурной организации бывает вполне достаточно разбиения на МОДУЛИ с удобными именами, некоторой дистилляции и неформального координирования работ в группе.

Введение крупномасштабной структуры может спасти проект, но неподходящая структура способна сильно помешать работе. В этой главе рассматриваются архитектурные шаблоны, позволяющие успешно структурировать приложение и модель на этом уровне знания.

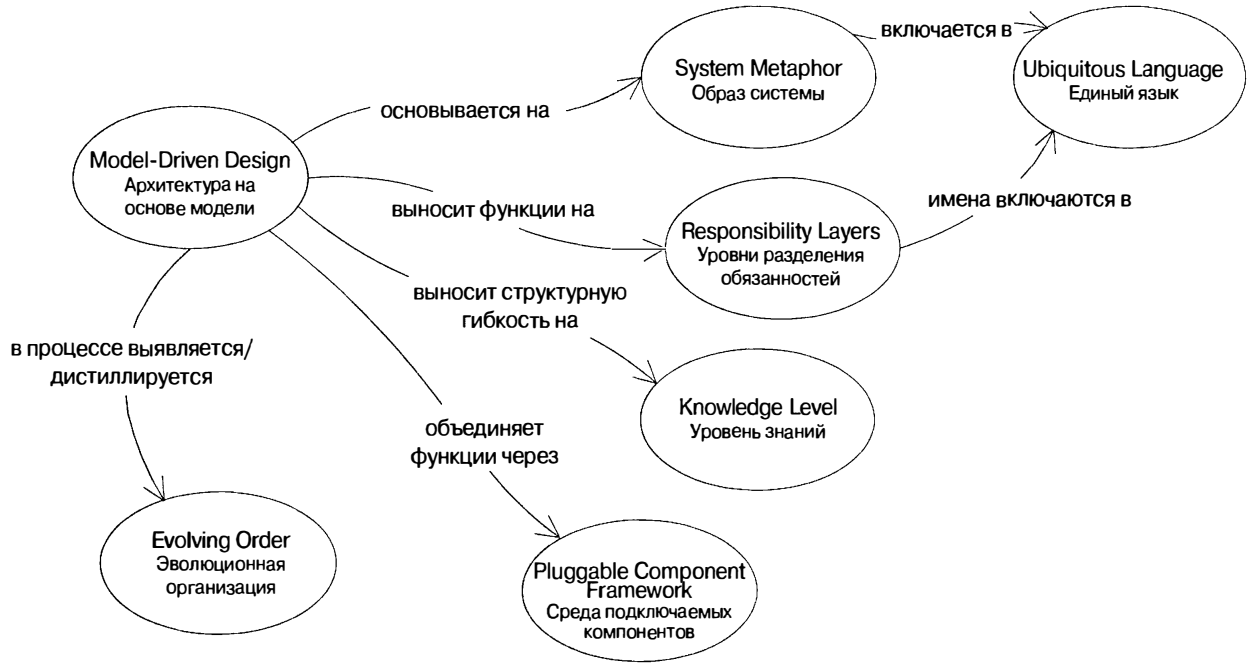


Рис. 16.1. Некоторые архитектурные шаблоны крупномасштабной структуры

Эволюционная организация

Многие разработчики ощутили на себе все прелести отсутствия структурированной архитектуры. Чтобы избежать анархии, в проектах часто создаются такие архитектуры, которые накладывают те или иные ограничения на процесс разработки. Некоторые технические архитектуры действительно решают частные проблемы, — например, управление сетевой коммуникацией или обеспечение сохранности данных. Но когда они проникают в ту сферу, где главенствуют модели прикладных проблем и предметной области, они могут сами создавать там беспорядок. Часто такие архитектуры мешают разработчикам спроектировать модели, которые лучше всего бы подходили специфике задачи. Наиболее навязчивые даже могут лишить разработчиков какой-то части знакомых технических возможностей из их языка программирования. Неважно, технические это архитектуры или предметно-ориентированные, — если в них приходится заранее принимать и фиксировать слишком много проектных решений, то в будущем они могут стать тормозом развития при изменении требований заказчика и углублении понимания среди самих разработчиков.

Некоторые технические архитектуры (например, J2EE) за много лет приобрели большой авторитет, тогда как крупномасштабные структуры на уровне предметной области остаются слабо исследованными. Потребность в них сильно меняется от одного приложения к другому.

Заблаговременное введение крупномасштабной структуры часто имеет свою цену. В ходе разработки почти наверняка найдется более подходящая для приложения структура. Может даже оказаться, что заранее предписанная структура мешает выбрать такое направление работы, которое бы прояснило и упростило программу. Какие-то возможности, конечно, удастся использовать, но потенциал теряется. Применение разных технических хитростей или попытки договориться с архитекторами только замедляют работу. Но руководство уверено, что архитектура уже готова. Она была призвана упростить разработку, поэтому почему бы вам не заняться написанием кода приложения, вместо того чтобы решать архитектурные проблемы? Руководство и архитекторы могут даже

выражать готовность выслушать предложения, но если каждое изменение в архитектуре сопровождается эпическими битвами, это несколько утомляет.

Хаотическое, несистематическое проектирование порождает системы, которые никто не воспринимает как единое целое и не способен сопровождать и дорабатывать. Но и наличие заблаговременно выбранной архитектуры может стать слишком стесняющим для проекта, отбирая у разработчиков и архитекторов отдельных частей программы слишком много полномочий. Вскоре разработчики начинают подгонять приложение под структуру либо игнорировать ее вообще, возвращаясь ко всем проблемам нескоординированной разработки.

Проблема состоит, скорее, не в самом наличии обязательных к исполнению правил, а в источнике и чрезмерной жесткости этих правил. Если правила, определяющие архитектуру, соответствуют обстоятельствам, то они не мешают, а, наоборот, помогают вести разработку в нужном направлении, а также создают единообразие.

Пусть концептуальная крупномасштабная структура эволюционирует вместе с приложением, имея возможность превратиться в нечто совершенно отличающееся от первоначального. Не накладывайте слишком жестких и подробных ограничений на архитектуру и модель; такие решения следует принимать на основе детализированных знаний.

Для отдельных частей программы могут существовать естественные или особо удачные способы организации и выражения, неприменимые к целому. Соответственно, при введении жестких общих правил качество этих частей падает. Выбор в пользу крупномасштабной структуры облегчает манипулирование моделью в целом, но структуризация отдельных частей может оказаться далеко не идеальной. Поэтому следует искать компромисс между унификацией структуры и свободой естественного выражения отдельных компонентов. Это противоречие можно смягчить, выбрав структуру, основанную на реальном опыте и знании предметной области, а также избегая слишком жестких структур. Если между предметной областью и техническим заданием на программу, с одной стороны, и выбранной структурой, с другой стороны, наблюдается хорошее соответствие, то моделировать и проектировать архитектуру становится намного легче, поскольку с ходу решается много вопросов выбора.

Наличие структуры позволяет быстрее выйти на проектные решения, которые в принципе можно было бы найти, работая на уровне отдельных объектов, но с большими затратами времени и вразнобой. Конечно, от непрерывного рефакторинга по-прежнему никуда не деться, но он становится более управляемым и помогает разным людям прийти к согласованным решениям.

Применимость крупномасштабной структуры, как правило, шире одного ОГРАНИЧЕННОГО КОНТЕКСТА (BOUNDED CONTEXT). В реальном проекте итерационное совершенствование структуры лишает ее тесной привязки к конкретной модели и развивает в ней черты, соответствующие КОНЦЕПТУАЛЬНЫМ КОНТУРАМ (CONCEPTUAL CONTOURS) предметной области. Это не значит, что структура *совсем никак* не зависит от модели. Но она не навязывает всему проекту идеи, имеющие смысл только в определенной локальной ситуации. Структура должна предоставлять полную свободу группам разработчиков в различных КОНТЕКСТАХ, чтобы те могли варьировать модель в соответствии со своими локальными потребностями.

Кроме того, крупномасштабные структуры все-таки должны накладывать полезные, практически ценные ограничения на процесс разработки. Например, архитекторам может быть отказано в контроле над моделями некоторых частей системы, особенно если это внешние или старые подсистемы. В таких случаях структуру можно изменить для подстройки под те или иные внешние элементы. Можно задать конкретные способы взаимосвязи с внешним миром. А можно ослабить жесткость структуры настолько, чтобы она позволяла легко обходить неудобные места.

В отличие от КАРТЫ КОНТЕКСТОВ (CONTEXT MAP), крупномасштабная структура — вещь необязательная. Ее следует вводить, когда это выгодно по балансу затрат и результата и когда имеется действительно подходящая структура. В ней нет нужды в системах, которые достаточно просты для понимания после разбиения их на МОДУЛИ. **Крупномасштабная структура должна вводиться в тех случаях, когда можно найти такую структуру, которая бы сделала систему намного понятнее, не накладывая неестественных ограничений на развитие модели. Плохо подходящая структура хуже, чем вообще никакой, поэтому не надо гнаться за всеобщностью охвата, а найти минимальное решение для возникших проблем. Лучше меньше, да лучше.**

Крупномасштабная структура может оказаться удобной за несколькими исключениями. Эти исключения следует как-то пометить, чтобы разработчики четко знали: структуре нужно следовать всегда, не считая оговоренных случаев. Если же исключения становятся слишком многочисленными, структуру нужно менять или вообще отбрасывать.

* * *

Как уже говорилось, построить структуру, дающую достаточно свободы разработчикам и при этом предотвращающую хаос, — немалый подвиг. По техническим архитектурам для программных систем наработано очень много материала, тогда как по структуризации уровня предметной области публикаций практически нет. Некоторые подходы работают против объектно-ориентированной парадигмы, — например, такие, в которых предметная область разбивается по задачам приложения либо по прецедентам (сценариям использования). Вся эта область пока плохо разработана. В разных проектах я наблюдал несколько общих образцов крупномасштабных структур, четыре из которых будут рассмотрены в этой главе. Какой-то из них может подойти и вам — по крайней мере, навести на мысли относительно структуризации вашего собственного проекта.

Метафорический образ системы

В индустрии разработки программного обеспечения образное, метафорическое мышление вообще широко распространено, особенно при работе с моделями. Но методика экстремального программирования под названием "[метафорический] образ" (*metaphor*) стала известна как конкретный способ использования образных представлений для упорядочения разработки больших систем.

* * *

Как брандмауэр[2] спасает здание от пожара, охватившего соседние постройки, так и программный "брандмауэр" или "межсетевой экран" (*firewall*) защищает локальную сеть от опасностей, приходящих из внешних глобальных сетей. Этот образ повлиял на архитектуру компьютерных сетей и породил целую категорию программных продуктов. Потребителю теперь доступно множество конкурирующих систем-брандмауэров, разработанных независимо друг от друга и в значительной степени взаимозаменяемых. Новички в сетевых технологиях легко ухватывают основную идею, и единое понимание этой концепции как среди разработчиков, так и среди пользователей не в последней степени возникло благодаря образному представлению.

И все же аналогия неполна; ее удобство оборачивается палкой о двух концах. Использование образа брандмауэра привело к разработке программных экранов, иногда недос-

[2] Это название глухой противопожарной стены здания — нередкий пример того, как при переводе на русский язык сугубо английское слово, составленное из английских же корней, приходится заменять заимствованным немецким (*brandmauer*, "пожарная стенка"). — *Примеч. перев.*

таточно избирательных, подавляющих полезные потоки данных и в то же время не обеспечивающих никакой защиты от угроз, возникающих внутри защищаемой области. Сильно уязвимы, например, беспроводные локальные сети. Образ брандмауэра нагляден и удобен, но всякий образ имеет свою оборотную сторону[3].

Программные архитектуры часто бывают слишком абстрактными и трудными для понимания. Как разработчикам, так и пользователям нужно иметь реальный способ понять систему и совместно воспринять ее как единое целое.

Метафорические образы так глубоко пронизывают наше мышление, что находят свое отражение в любой архитектуре. Системы состоят из "уровней", расположенных "один над другим". В "центре" системы находится ее "ядро". Но иногда возникает такой образ, который становится ключом к пониманию всей архитектуры и дает разработчикам единую точку зрения на систему.

Когда случается подобное, облик системы фактически уже определяется ее метафорическим образом. Разработчики теперь принимают проектные решения, согласуясь с образом системы. Самые разные части сложной системы получают единообразное истолкование на основе все того же образа. У разработчиков и специалистов по предметной области появляется точка отсчета в дискуссиях, которая может стать более реальной, чем модель.

ОБРАЗ СИСТЕМЫ (SYSTEM METAPHOR) — это нестрогая, легко понятная крупномасштабная структура, находящаяся в соответствии с объектной парадигмой. ОБРАЗ СИСТЕМЫ в конечном счете является всего лишь аналогией предметной области, и разные модели могут по-разному находиться в приблизительном соответствии с ним. Это позволяет применять ОБРАЗ в нескольких ОГРАНИЧЕННЫХ КОНТЕКСТАХ, способствуя координированию работ над ними.

ОБРАЗ СИСТЕМЫ приобрел большую популярность, поскольку это одна из ключевых методик в экстремальном программировании [5]. К сожалению, действительно полезные ОБРАЗЫ удалось найти в очень немногих проектах. Предпринимались многочисленные попытки протолкнуть этот подход в такие предметные области, где от него становилось только хуже. Чем убедительнее ОБРАЗ, тем больше риск, что архитектура воспримет такие аспекты этой аналогии, которые нежелательны при решении конкретной задачи, или что "соблазнительная" аналогия попросту окажется неуместной.

Учитывая все это, следует подытожить, что ОБРАЗ СИСТЕМЫ (SYSTEM METAPHOR) — это распространенная разновидность крупномасштабной структуры, полезная для некоторых проектов, которая хорошо иллюстрирует само понятие структуры.

Если возникает образ (метафора) системы, захватывающий воображение разработчиков и, по всей видимости, направляющий их мышление в нужном направлении, примите его в качестве КРУПНОМАСШТАБНОЙ СТРУКТУРЫ**. Постройте архитектуру системы вокруг этого образа и внедрите его в** ЕДИНЫЙ ЯЗЫК **(**UBIQUITOUS LANGUAGE**).** ОБРАЗ СИСТЕМЫ **(**SYSTEM METAPHOR**) должен облегчать обсуждение системы и направлять ее разработку. Это улучшает согласованность между отдельными частями системы, и даже между различными** ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ **(**BOUNDED CONTEXTS**). Но поскольку никакое образное представление не может быть точным, непрерывно испытывайте** ОБРАЗ **на соответствие задаче и будьте готовы отказаться от него, если он станет мешать в работе.**

* * *

[3] Я наконец понял, что такое ОБРАЗ СИСТЕМЫ (SYSTEM METAPHOR), именно тогда, когда Уорд Каннингем (Ward Cunningham) привел пример с брандмауэром на одном из семинаров.

“Наивный образ”: почему он нам не нужен

Так как в большинстве проектов полезный образ построить не получается, в сообществе экстремального программирования стали поговаривать о так называемом “наивном образе” (*naive metaphor*), под которым имеется в виду сама модель предметной области.

Этот термин некорректен уже в том отношении, что зрелая модель предметной области очень далека от наивности. На самом деле образ типа “обработка платежной ведомости напоминает сборочный конвейер” намного более наивен, чем модель, родившаяся в результате многих итераций переработки знаний, совместной со специалистами, и проверенная в тесной связи с программной реализацией в рабочем приложении.

От термина “наивный образ” (*naive metaphor*) следует отказаться.

ОБРАЗЫ СИСТЕМЫ (SYSTEM METAPHORS) не обязательно полезны во всех проектах. В целом, наличие крупномасштабной структуры не так уж существенно. В 12 практиках экстремального программирования роль образа системы может успешно играть ЕДИНЫЙ ЯЗЫК (UBIQUITOUS LANGUAGE). В этот ЯЗЫК следует включить ОБРАЗ СИСТЕМЫ и другие крупномасштабные структуры, если они хорошо подходят для решения задачи.

Уровни разделения обязанностей

В ходе изложения мы то и дело наделяли разные объекты их собственным узким кругом обязанностей. Проектирование архитектуры по разделению обязанностей применимо также и в крупном масштабе.

* * *

Если для каждого отдельного объекта искусственно конструируется свой круг обязанностей, то в такой системе нет правил, нет единообразия и нет возможности управлять на уровне больших подобластей предметной области. Чтобы придать большой модели связность, распределению обязанностей полезно придать какую-то структуру.

Когда вы приобретаете глубокое понимание предметной области, то становятся видны крупномасштабные структуры внутри нее. В некоторых областях имеется естественная стратификация (расслоение). Определенные понятия и операции существуют на фоне других элементов, которые изменяются независимо от них, с другой скоростью и по другим причинам. Как можно воспользоваться этой естественной структурой, сделать ее более наглядной и полезной? Стратификация предполагает деление на уровни, и это один из самых успешных архитектурных образцов-шаблонов, см. [7] и других авторов.

Уровни — это части системы, устроенные таким образом, что элементы каждой из частей знают о существовании и могут пользоваться услугами “нижних” уровней, но не знают о существовании и независимы от уровней, лежащих “выше”. При изображении взаимосвязей между МОДУЛЯМИ на схемах МОДУЛЬ, от которого зависят другие МОДУЛИ, часто располагают ниже их. Таким образом происходит естественная сортировка уровней, так что никакие объекты на нижних уровнях не зависят концептуально от верхних объектов.

Но такое хаотическое расслоение, пусть оно даже помогает прослеживать взаимосвязи и иногда привносит в схему некий интуитивный смысл, не позволяет глубоко понять модель и руководить принятием решений. Нам необходимо нечто более систематическое.

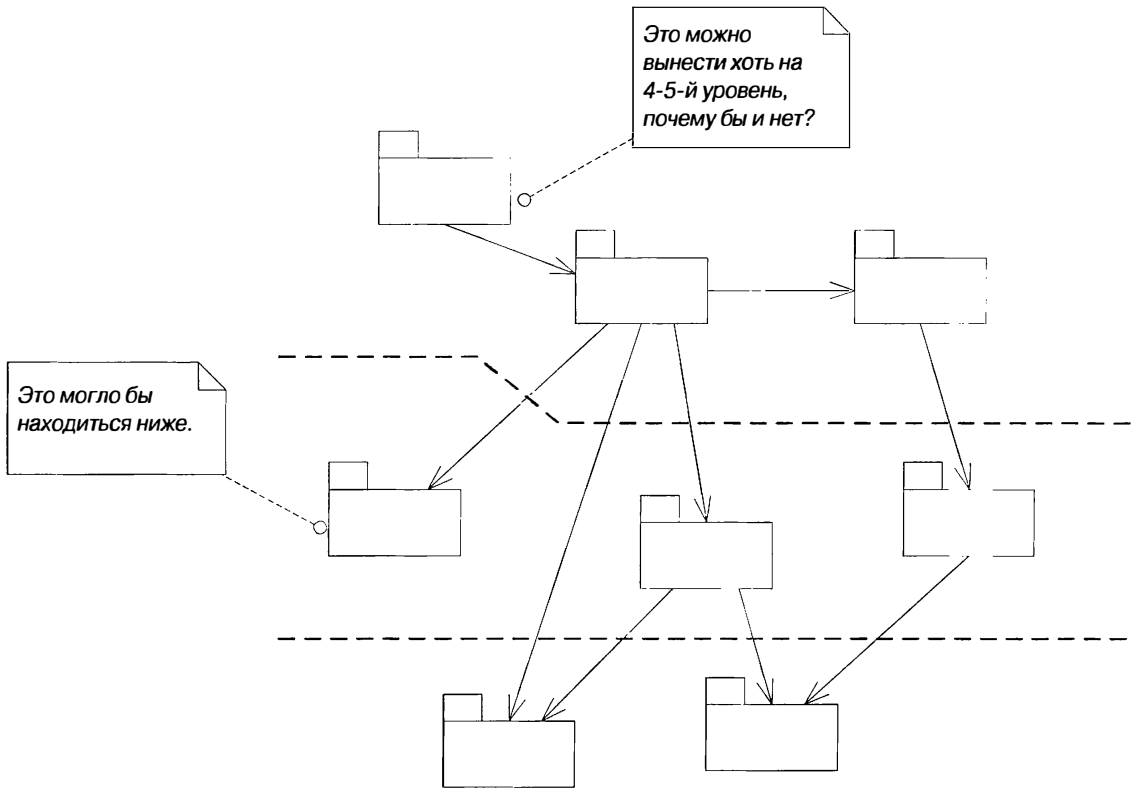


Рис. 16.2. Хаотическое расслоение модели: какой смысл во всех этих пакетах?

В моделях, где присутствует естественная стратификация, концептуальные уровни можно построить вокруг основных групп обязанностей, тем самым объединяя два мощных принципа: многоуровневую организацию и проектирование по обязанностям/ответственности (*responsibility-driven design*)[4].

Эти обязанности должны быть существенно шире, чем те, которые обычно назначаются отдельным объектам, как это будет вскоре показано. По мере проектирования новых МОДУЛЕЙ и АГРЕГАТОВ выполняется их факторизация, чтобы удержать их обязанности в пределах, задаваемых общей схемой. Уже само по себе распределение обязанностей по группам с их наименованием может улучшить наглядность разбитой на модули системы, поскольку так легче интерпретировать обязанности отдельных МОДУЛЕЙ. Но сочетание высокоуровневого распределения обязанностей с многоуровневой структурой дает нам сразу принцип организации системы.

> Многоуровневая структура, наилучшим образом подходящая для УРОВНЕЙ РАЗДЕЛЕНИЯ ОБЯЗАННОСТЕЙ (RESPONSIBILITY LAYERS), называется НЕСТРОГОЙ МНОГОУРОВНЕВОЙ СИСТЕМОЙ (RELAXED LAYERED SYSTEM), см. [7], с. 45. В ней компонентам того или иного уровня разрешается обращаться к любым нижним уровням, а не только к тому, который расположен сразу под ними.

Изучите концептуальные взаимосвязи в вашей модели, а также различия в скорости и источниках изменений, происходящих в разных частях предметной области. Если в предметной области обнаруживается естественное расслоение, представьте слои

---

[4] В то время как *проектирование по ответственности* уже стало устоявшимся термином, смысл выражения все же ближе к *проектированию по обязанностям*, т.е. на основе разделения обязанностей между программными объектами. Причина неточности — многозначность слова *responsibility*. — *Примеч. перев.*

в виде обобщенных абстрактных группы обязанностей. Они должны нести информацию об высокоуровневом назначении вашей системы и ее архитектуре. Выполните рефакторинг модели так, чтобы обязанности каждого объекта предметной области, АГРЕГАТА **и** МОДУЛЯ **как можно точнее совпадали с обязанностями одного слоя-уровня.**

Это достаточно абстрактное определение, но оно станет понятнее после нескольких примеров. В симуляторе спутниковой связи, пример которого открывал эту главу, обязанности также разделялись на уровни. Успешное применение УРОВНЕЙ РАЗДЕЛЕНИЯ ОБЯЗАННОСТЕЙ (RESPONSIBILITY LAYERS) встречалось мне в самых разнообразных приложениях: от управления производством до обслуживания финансовой деятельности.

* * *

В следующем примере, рассматривая УРОВНИ РАЗДЕЛЕНИЯ ОБЯЗАННОСТЕЙ, мы попытаемся прочувствовать процесс выявления крупномасштабной структуры *любого* вида, его направляющую и ограничивающую роль в моделировании и проектировании архитектуры.

Пример

Подробности: многоуровневое устройство системы грузопоставок

Давайте посмотрим, что будет, если применить УРОВНИ РАЗДЕЛЕНИЯ ОБЯЗАННОСТЕЙ (RESPONSIBILITY LAYERS) к приложению, обслуживающему грузопоставки и неоднократно рассмотренному в примерах предыдущих глав.

К моменту нашего возвращения к этому приложению группа его разработчиков сильно продвинулась в проектировании его архитектуры по модели и дистилляции СМЫСЛОВОГО ЯДРА (CORE DOMAIN). Но по мере того, как вырисовывалась архитектура, начали появляться проблемы с объединением всех составных частей в одно целое. Сейчас разработчики ищут такую крупномасштабную структуру, которая бы подчеркнула главную тему системы и создала для всех один общий контекст.

Вот вариант представления одной из ключевых частей модели.

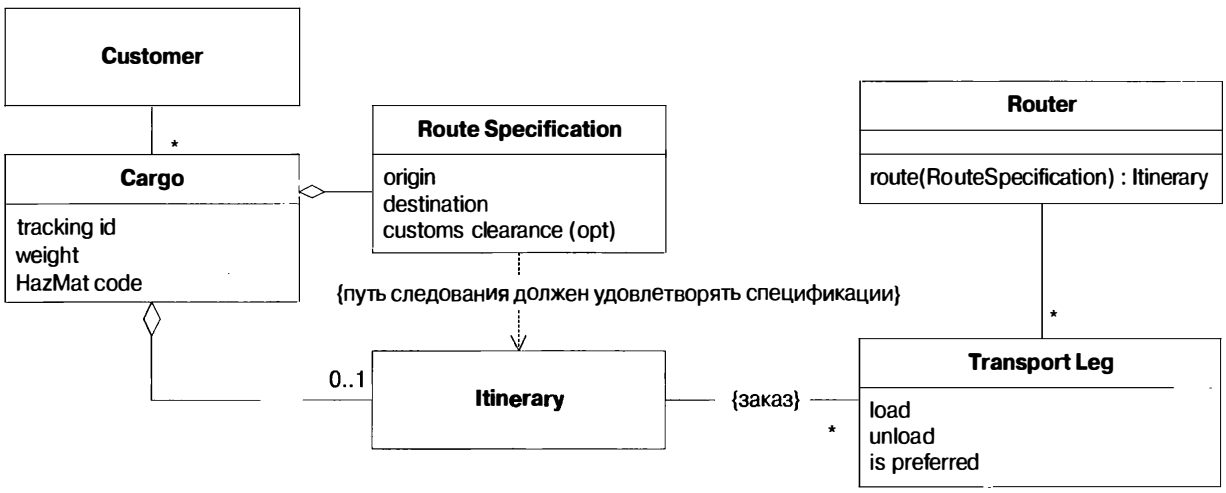


Рис. 16.3. Элементарная модель из предметной области для маршрутизации грузов

Разработчики "варились" в тематике грузопоставок месяцами, и в конце концов заметили естественное расслоение в ее понятиях. Вполне разумно рассматривать графики движения транспорта (расписания рейсов судов и поездов), не ссылаясь на грузы, находящиеся на борту этих транспортных средств. В то же время трудно говорить об отсле-

живании груза, не ссылаясь на транспорт, который его перевозит. Концептуальные взаимосвязи тут вполне понятны. Группа легко выделила два уровня: уровень **Операций** (**Operations**) и нижний по отношению к нему уровень **Ресурсов** (**Capability**).

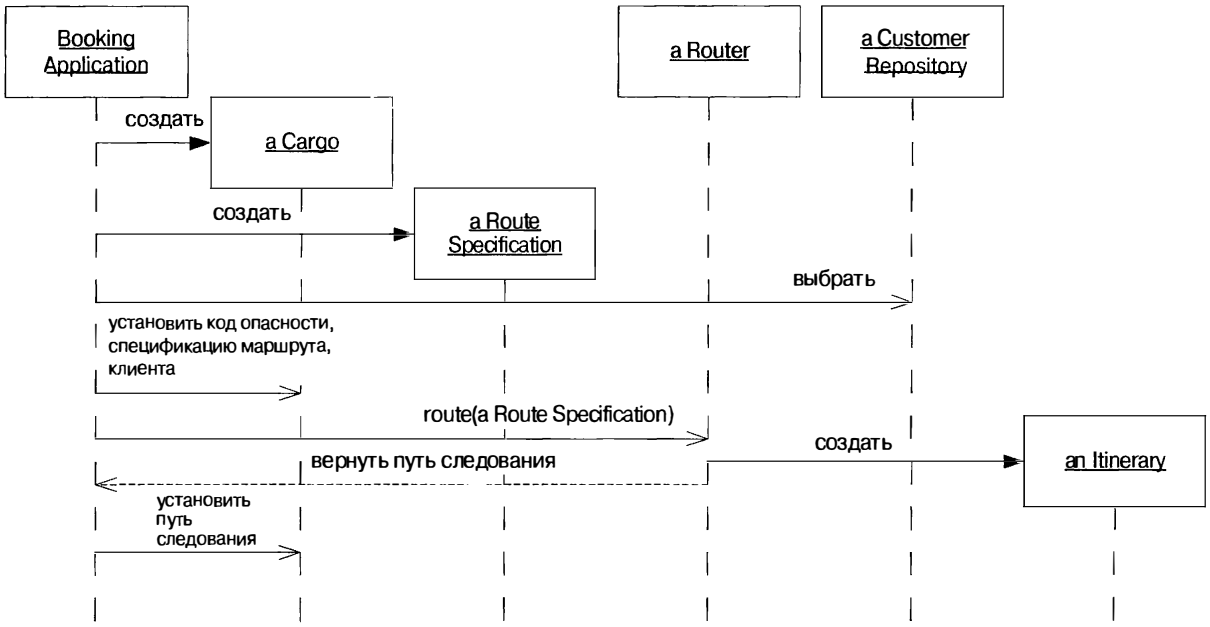


Рис. 16.4. Маршрутизация груза с помощью модели в процессе его заказа

Обязанности уровня Операций

На уровне **Операций** (**Operations**) собрана вся деятельность компании: прошлая, нынешняя и планируемая. Наиболее очевидный объект этого уровня — объект **Груз** (**Cargo**), на котором концентрируется большая часть повседневной работы. **Спецификация маршрута** (**Route Specification**) является органичной составной частью **Груза**, поскольку она задает требования к доставке. Операционный план доставки содержится в объекте **Путь следования** (**Itinerary**). Оба эти объекта входят в АГРЕГАТ **Груз**, и срок их существования привязан к временным рамкам доставки.

Обязанности уровня Ресурсов

Этот уровень отражает ресурсы, которые компания привлекает для выполнения операций доставки. Классическим примером является объект **Участок** (**Transit Leg**). Для морских судов составляется график рейсов и предусматривается определенная грузоподъемность, которая может использоваться полностью или частично.

Если бы нас интересовало управление грузовым флотом, объект **Участок пути** находился бы на уровне **Операций**. Но пользователей данной системы эта проблема не волнует. (Если бы компания занималась обеими видами деятельности и хотела их скоординировать, то разработчикам следовало бы подумать о другой схеме деления на уровни: например, “Транспортные операции” и “Операции с грузами”.)

Сложнее решить, куда же отнести **Клиента** (**Customer**). В некоторых областях деятельности клиенты преходящи: они представляют интерес, пока посылка доставляется, а потом о них забывают до следующего раза. При таком подходе — как у службы почтовой доставки посылок отдельным получателям — объекты-клиенты имели бы чисто операционный характер. Но наша гипотетическая компания по доставке грузов склонна раз-

вивать долговременные отношения с клиентами, и большинство ее заказов — повторные. *При условии такого отношения к потребителям данной услуги* объект **Клиент (Customer)** относится к уровню потенциальных ресурсов. Как видите, *это не техническое решение*. Это попытка зафиксировать и передать знание о предметной области.

Поскольку ассоциацию между **Грузом (Cargo)** и **Клиентом (Customer)** можно проследить только в одном направлении, ХРАНИЛИЩЕ объектов **Груз** должно отвечать на запрос о всех **Грузах** конкретного **Клиента**. В любом случае есть веские причины спроектировать эту функцию именно так, но с применением крупномасштабной структуры это стало уже нормативным требованием.

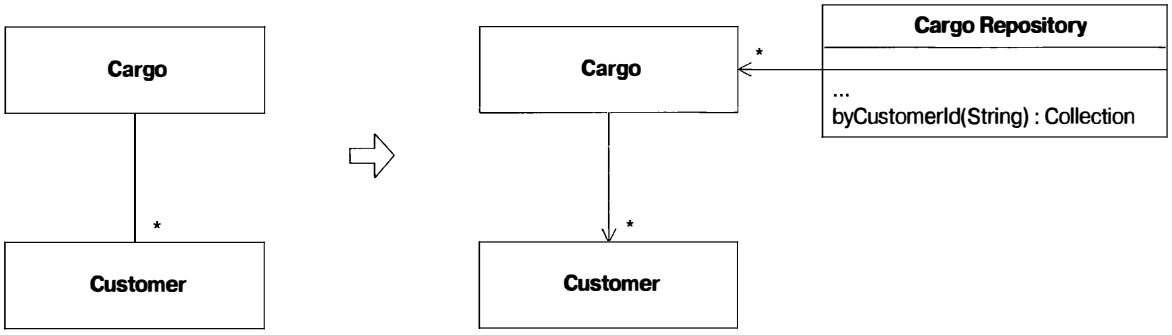


Рис. 16.5. Запрос заменяет двунаправленную ассоциацию, нарушающую деление на уровни

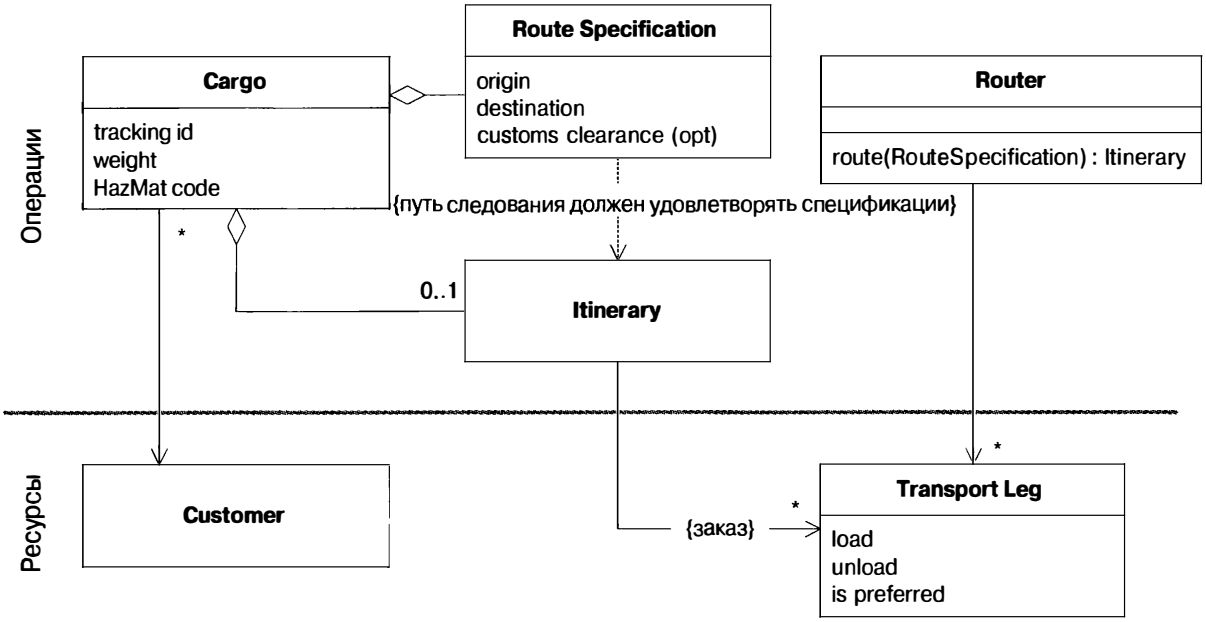


Рис. 16.6. Двухуровневая модель первого приближения

По мере того как проясняются различия между операциями и ресурсами, эволюционирует и организация приложения. После нескольких недель экспериментирования разработчики "нацеливаются" на другое различие. Дело в том, что большей частью оба первоначальных уровня сосредоточены на ситуациях или планах *как они есть*. Между тем **Маршрутизатор (Router)** и многие другие элементы, исключенные из этого примера, не относятся к текущей операционной ситуации или плану действий. Эта функция помогает принимать решения об изменении планов. И вот разработчики определяют новый уровень, отвечающий за **Поддержку решений (Decision Support)**.

Обязанности уровня Поддержки решений

Этот уровень приложения дает пользователю в руки инструмент планирования и принятия решений, потенциально пригодный даже для автоматизации некоторых из них (например, автоматического генерирования нового маршрута для **Грузов** в случае, если график движения транспорта изменяется).

Маршрутизатор (Router) — это СЛУЖБА, которая помогает приемщику заказа выбрать наилучший маршрут пересылки **Груза (Cargo)**. Поэтому **Маршрутизатору**, вне всяких сомнений, место на уровне **Поддержки решений**.

Ссылки внутри этой модели четко соответствуют трехуровневой структуре, если не считать одного элемента: атрибута "предпочтительный" (`is preferred`) в объекте **Transport Leg (Этап перевозки)**. Этот атрибут присутствует, потому что компания предпочитает пользоваться своим транспортом, когда это возможно, или транспортом некоторых определенных компаний, с которыми у них заключены выгодные договоры. Атрибут "предпочтительный" используется для того, чтобы склонить решение **Маршрутизатора** в пользу этих видов транспорта. Он не имеет ничего общего с уровнем **Ресурсов (Capability)**; это стратегия принятия решений. Теперь, чтобы применить новые УРОВНИ РАЗДЕЛЕНИЯ ОБЯЗАННОСТЕЙ, необходим рефакторинг модели.

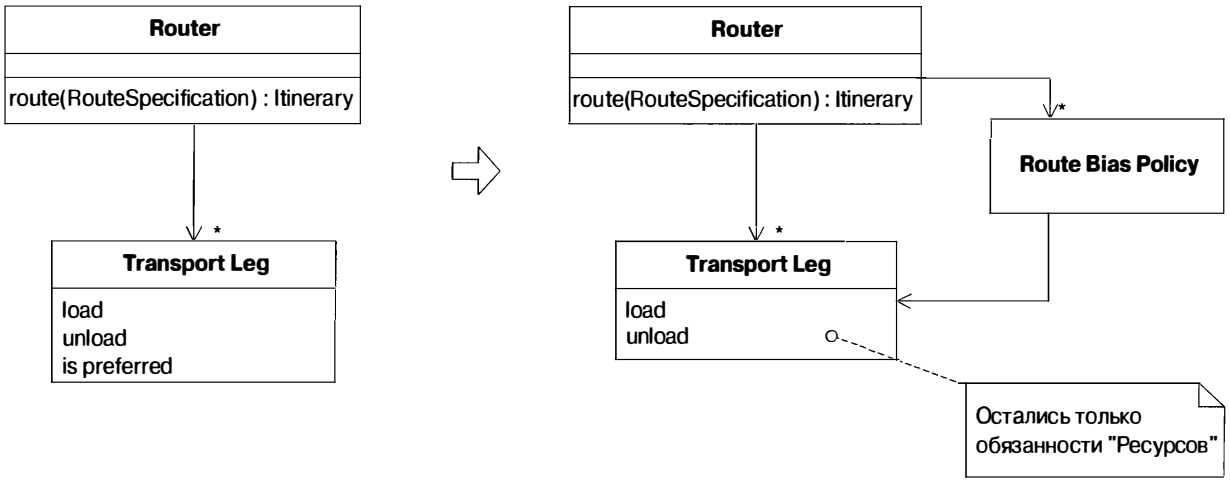


Рис. 16.7. Рефакторинг модели для обновления многоуровневой структуры

Такая структуризация четче выделяет **Route Bias Policy (Регламент предпочтения маршрута)**, при этом объект **Этап перевозки (Transport Leg)** стал в большей степени соответствовать фундаментальному понятию транспортных мощностей или ресурсов. Крупномасштабная структура, основанная на глубоком понимании предметной области, часто подталкивает модель в таком направлении, которое проясняет смысл ее составляющих.

Новая модель идеально вписывается в крупномасштабную многоуровневую структуру.

Разработчик, привыкший к выбранным уровням, сумеет легче выделить роли отдельных частей и взаимосвязи между ними. Ценность крупномасштабной структуры возрастает с увеличением сложности.

Следует отметить, что хотя этот пример проиллюстрирован модифицированной UML-диаграммой, она здесь просто *изображает* расслоение на уровни. В UML нет соответствующих обозначений, поэтому в схему добавлена дополнительная информация для удобства читателя. Если в вашем проекте основным архитектурным документом является код, то было бы полезно иметь какое-то средство для просмотра или хотя бы перечисления классов по уровням.

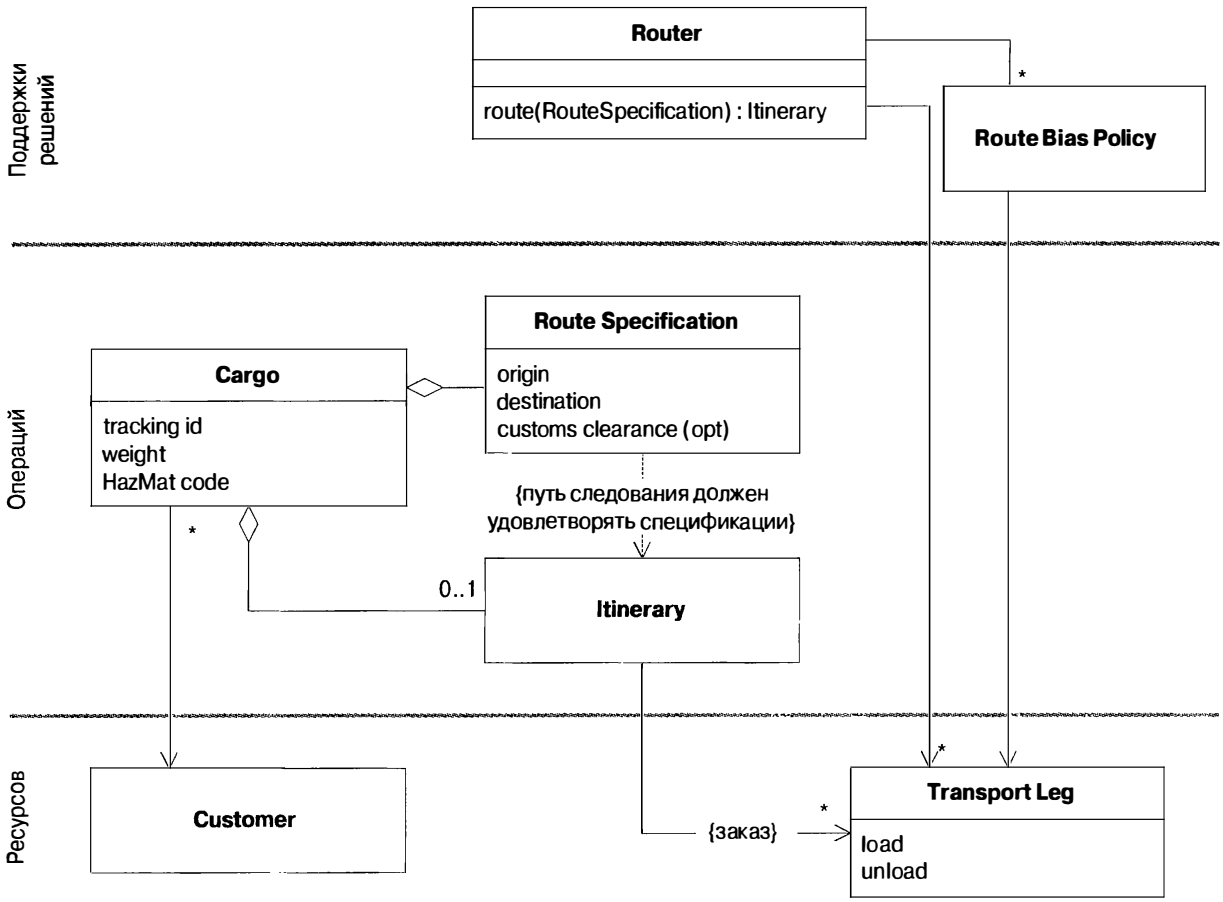


Рис. 16.8. Реструктуризированная и рефакторизированная модель

Как структура влияет на архитектурное проектирование

Раз уж в проекте принята определенная крупномасштабная структура, она должна учитываться при последующем принятии модельных и проектных решений. Для иллюстрации представим себе, что нам нужно добавить новую функцию в нашу уже многоуровневую архитектуру. Специалисты по предметной области только что сообщили, что для определенных категорий опасных материалов существуют ограничения возможных маршрутов. Те или иные материалы не разрешается провозить некоторыми средствами транспорта или через некоторые порты. Необходимо, чтобы **Маршрутизатор (Router)** подчинялся этим нормативным правилам.

К этой проблеме есть несколько подходов. В отсутствие крупномасштабной структуры привлекательным кажется такой вариант: передать обязанность по реализации этих правил выбора маршрута тому объекту, который владеет **Спецификацией маршрута (Route Specification)** и содержит код HazMat (*Hazardous Material*, "опасный материал"). Это объект **Груз (Cargo)**.

Беда в том, что эта архитектура не соответствует крупномасштабной структуре. Объект **HazMat Policy Service (Служба регламентов обращения с опасными материалами)** не составляет проблемы — он хорошо вписывается в обязанности уровня **Поддержки решений**. Проблема заключается в зависимости **Груза** ("операционного" объекта) от **Службы регламентов обращения с опасными материалами** (объекта "поддержки решений"). Если в проекте принято к исполнению деление на эти уровни, такая модель недопустима. Она сбивает с толку разработчиков, которые ожидают, что структура будет соблюдаться.

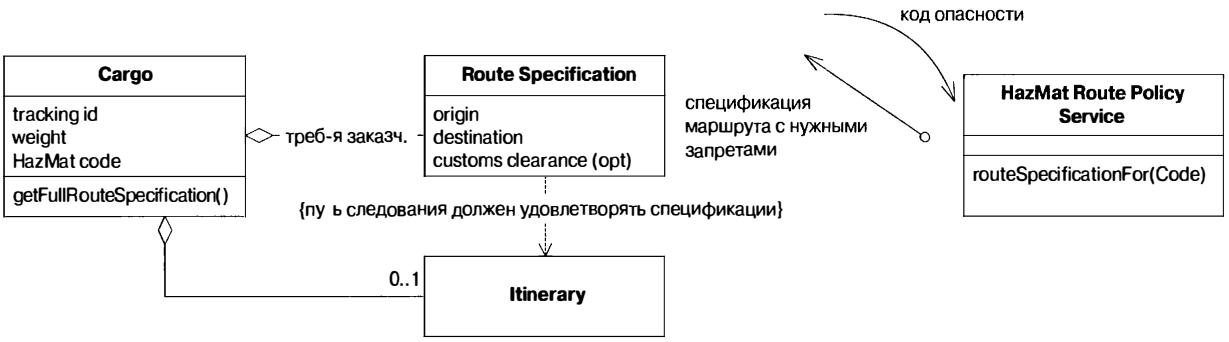


Рис. 16.9. Одна из возможных архитектур для маршрутизации опасного груза

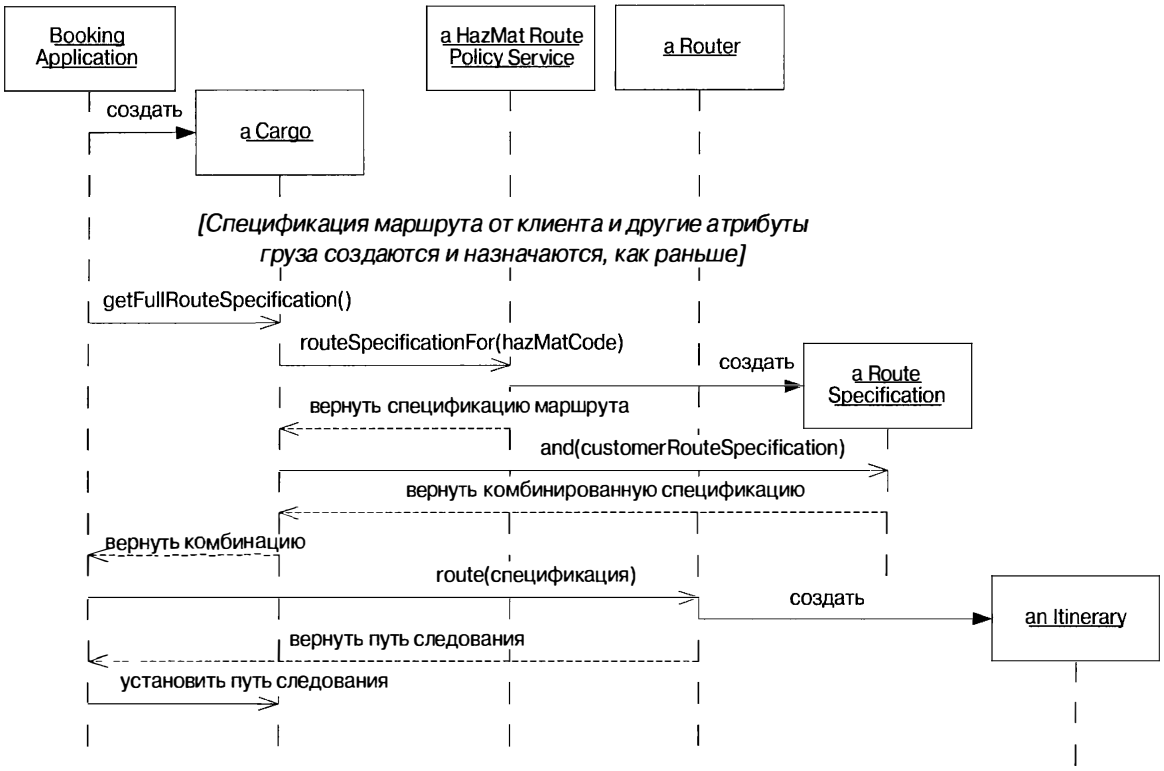


Рис. 16.10.

Всегда есть много вариантов построения архитектуры, и нам нужно выбрать всего один: тот, который следует правилам крупномасштабной структуры. Объект **`HazMat Policy Service`** (**`Служба регламентов обращения с опасными материалами`**) в изменениях не нуждается, но необходимо передать куда-то право использования регламентов. Давайте попробуем возложить на **`Маршрутизатор`** (**`Router`**) обязанность собирать соответствующие регламенты, прежде чем подбирать маршрут. Для этого нужно изменить интерфейс **`Маршрутизатора`** так, чтобы он включал объекты, от которых могут зависеть регламенты. Далее показан вариант архитектуры для этой цели.

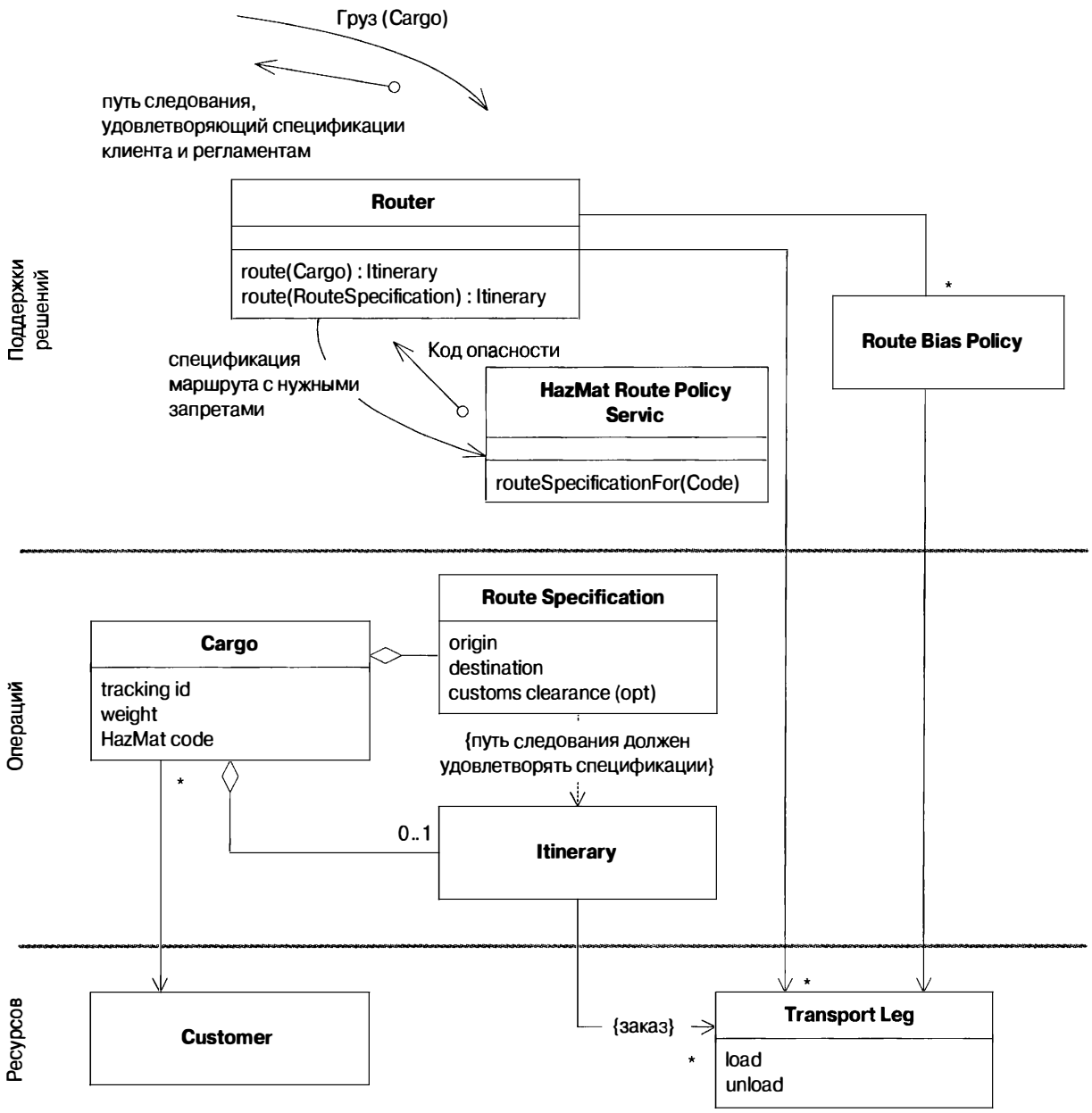


Рис. 16.11. Архитектура, не противоречащая многоуровневой структуре

Типичные взаимодействия показаны далее на рис. 16.12.

Надо сказать, что эта архитектура *не обязательно* лучшая, чем та, другая. У обеих есть свои достоинства и недостатки. Но если все участники проекта принимают единое решение, то архитектура, в целом, будет намного проще для восприятия, а это стоит того, чтобы кое-чем поступиться в "архитектурных частностях".

Если же структура навязывает слишком много неуклюжих архитектурных решений, то в соответствии с принципом ЭВОЛЮЦИОННОЙ ОРГАНИЗАЦИИ (EVOLVING ORDER) ее следует пересмотреть и модифицировать, а может быть, и исключить.

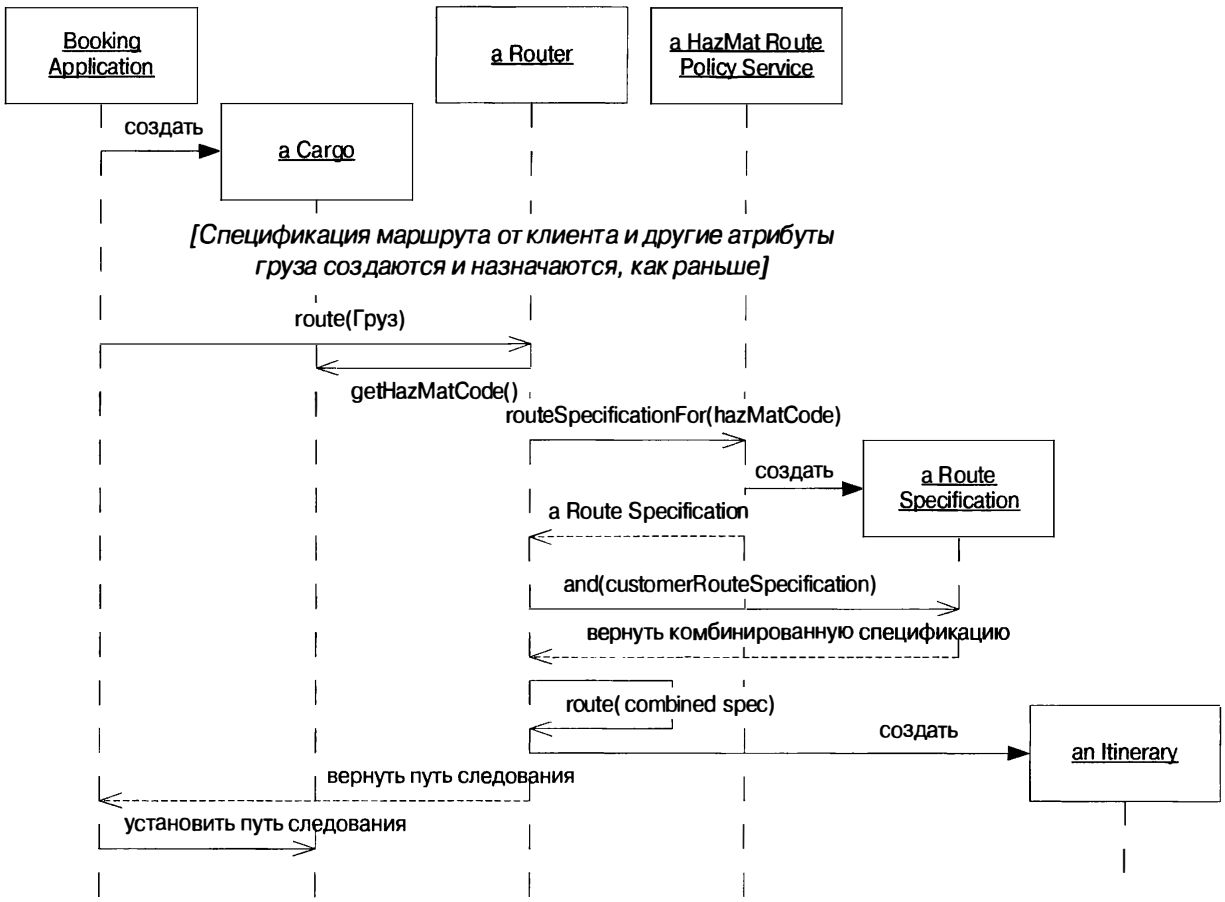


Рис. 16.12.

Выбор подходящих уровней

Чтобы найти удачные УРОВНИ РАЗДЕЛЕНИЯ ОБЯЗАННОСТЕЙ (RESPONSIBILITY LAYERS), как и любую другую крупномасштабную структуру, надо понимать предметную область и много экспериментировать. Если в проекте допускается ЭВОЛЮЦИОННАЯ ОРГАНИЗАЦИЯ (EVOLVING ORDER), то не так уж важно, с чего начинать, хотя неудачно выбранная исходная точка добавляет лишней работы. Структура вполне может эволюционировать в нечто неузнаваемое. Поэтому предлагаемые здесь рекомендации следует применять тогда, когда вы обдумываете значительную трансформацию структуры, по размаху напоминающую создание ее заново.

В ходе жонглирования уровнями, их слияния, разбиения и переопределения следует выявлять и стараться сохранить некоторые полезные свойства структуры.

- *Информативность.* Уровни должны отражать ключевые приоритеты и реалии предметной области. Выбор крупномасштабной структуры — это в большей степени высокоуровневое модельное решение, а не технически-инфраструктурное. Разбиение на уровни должно подчеркивать приоритеты профессиональной сферы приложения.
- *Концептуальная взаимосвязанность.* Понятия верхних уровней должны иметь смысл на фоне поддержки от нижних, тогда как понятия нижних уровней должны иметь самостоятельный смысл и значение.
- *Наличие концептуальных контуров.* Если объекты разных уровней меняются с разной скоростью или по разным причинам, уровни должны отражать наличие сдвигов между ними.

Проектируя уровни для новой модели, не всегда обязательно начинать с чистого листа. Некоторые уровни могут фигурировать в целых семействах взаимосвязанных предметных областей.

Так, на предприятиях, основанных на использовании крупных материальных активов (например, заводов или грузовых судов), программы по управлению материально-техническим снабжением часто организуются в виде "Потенциального" уровня (это другое название для уровня "Ресурсов", который рассматривался в примере) и "Операционного" уровня.

- *Потенциал* (Potential). Что мы можем сделать? Неважно, что именно планируется сделать. Что *вообще* можно сделать? Ядром потенциального уровня служат ресурсы организации, включая ее работников, и организация этих ресурсов. Потенциал также создается договорами с поставщиками. Этот уровень можно видеть практически в любом бизнесе, но наиболее важную роль он играет в таких отраслях (например, транспорте или производстве), где сама деятельность становится возможной только благодаря наличию крупных материальных активов. Потенциал включает также и временные активы, но если деятельность основана большей частью на таких активах, то лучше выбрать для модели уровни, подчеркивающие именно это, как будет показано дальше. В примере этот уровень назывался **Ресурсы** (**Capability**).
- *Операции* (Operations). Что на самом деле происходит? Что нам удалось извлечь из имеющегося потенциала? Как и потенциальный, этот уровень должен отражать реальное, а не желаемое, положение дел. На этом уровне мы пытаемся увидеть наши собственные усилия и виды деятельности – что именно мы продаем, а не благодаря чему мы можем это продавать. Как правило, операционные объекты ссылаются на объекты потенциальные и даже состоят из них, но потенциальные объекты не должны ссылаться на операционный уровень.

Во многих существующих системах (может быть, даже в большинстве из них) из предметных областей такого рода эти два уровня полностью решают проблему, хотя возможны и качественно другие, более наглядные структуры. В них отслеживается текущая ситуация и активные операционные планы, генерируются отчеты и документация. Но отслеживать не всегда бывает достаточно. Если программный проект призван помочь пользователю, направить его, автоматизировать принятие решений, то в нем существует еще один набор обязанностей, который можно организовать в виде дополнительного уровня, лежащего выше операционного.

- *Поддержка решений* (Decision Support). Какие действия следует предпринять, какой установить регламент? На этом уровне выполняется анализ и принимаются решения. Анализ основывается на информации с более низких уровней — потенциального и операционного. Программные объекты на уровне поддержки решений могут пользоваться накопленной ранее информацией для поиска новых возможностей в текущих и будущих операциях.

Системы поддержки решений концептуально зависят от других уровней (операционного и потенциального), потому что решения принимаются не в вакууме. Во многих проектах уровень поддержки решений реализуется с применением технологии хранилищ данных (*data warehouses*). Тогда этот уровень становится отчетливо отдельным ОГРАНИЧЕННЫМ КОНТЕКСТОМ (BOUNDED CONTEXT) и находится с "операционными" приложениями в отношениях ЗАКАЗЧИК-ПОСТАВЩИК (CUSTOMER/SUPPLIER). В других проектах он глубже интегрирован с другими уровнями, как показано в предыдущем развернутом примере. Од-

но из естественных преимуществ многоуровневой структуры заключается в том, что нижние уровни могут существовать без верхних. Это помогает при постепенном введении новых функций или наращивании высокоуровневых расширений на основу из старых операционных систем.

Еще один случай — приложения, реализующие сложные деловые регламенты (*business rules*) или юридические требования. Они тоже могут образовывать один из УРОВНЕЙ РАЗДЕЛЕНИЯ ОТВЕТСТВЕННОСТИ (RESPONSIBILITY LAYER).

- *Регламент* (Policy). Каковы установленные правила и цели? Правила и цели в основном носят пассивный характер, но накладывают ограничения на работу других уровней. Проектирование таких взаимодействий — дело тонкое. Правила могут передаваться в виде аргумента в метод более низкого уровня. Иногда здесь применим шаблон STRATEGY (СТРАТЕГИЯ). Этот уровень хорошо работает в связке с уровнем поддержки решений, который предоставляет средства для реализации поставленных регламентным уровнем целей и учитывает ограничения, наложенные регламентным уровнем.

Регламентные уровни можно писать на тех же языках, что и другие, но иногда они реализуются с применением серверов регламентов (*rules engines*). Это не обязательно требует их вынесения в отдельный ОГРАНИЧЕННЫЙ КОНТЕКСТ. На самом деле координацию таких разных технологий реализации можно сильно упростить, педантично и последовательно применяя в обеих одну и ту же модель. Если правила записываются на основе не той модели, на которой основаны подчиняемые им объекты, то либо сильно растет сложность приложения, либо объекты приходится упрощать, чтобы они не вышли из-под контроля.

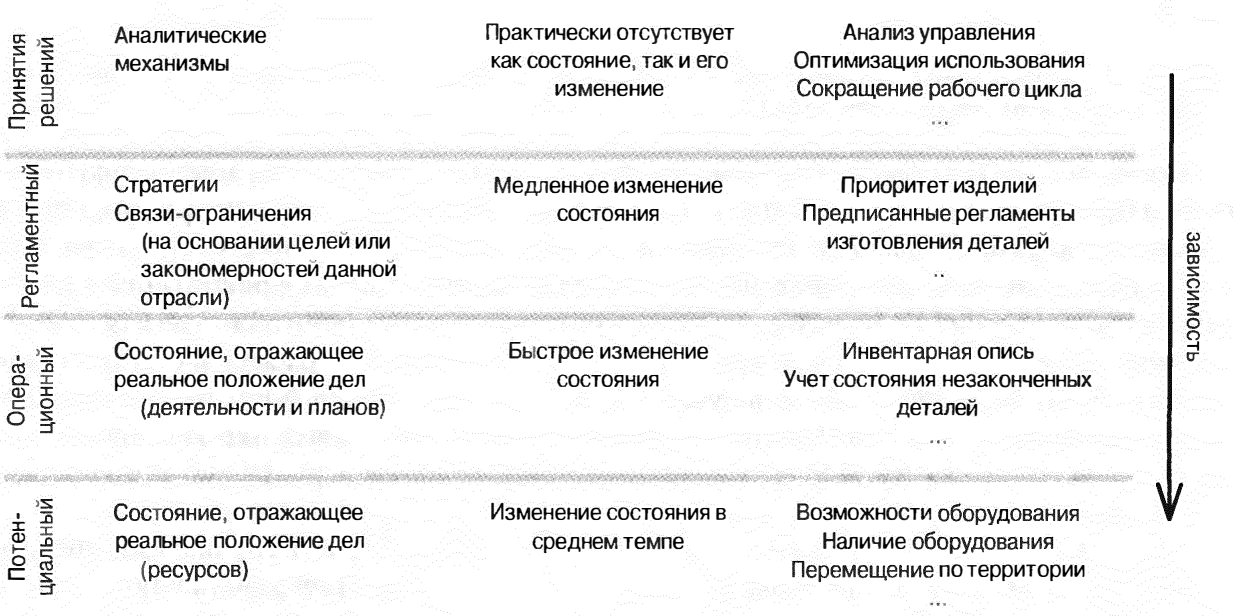


Рис. 16.13. Концептуальные взаимосвязи и точки относительного сдвига уровней в системе автоматизации производства

Многие предприятия не опираются в своей деятельности на заводские мощности и оборудование. Например, в сфере финансовых консалтинговых услуг или в страховании потенциал во многом определяется именно текущими операциями. Способность страховой компании принять на себя новый риск путем подписания нового страхового соглашения зависит от диверсификации ее текущей деятельности. Здесь потенциальный

уровень, скорее всего, вошел бы в операционный, и возникло бы совсем другое деление на уровни.

В подобных ситуациях часто на передний план выходит такой важный фактор, как обязательства перед клиентами.

- *Обязательства* (Commitment). Какие мы дали обещания и кому? Этот уровень по характеру похож на регламентный в том, что он задает цели для будущих операций, но и на операционный тоже, поскольку обязательства возникают и изменяются в ходе текущей деятельности.

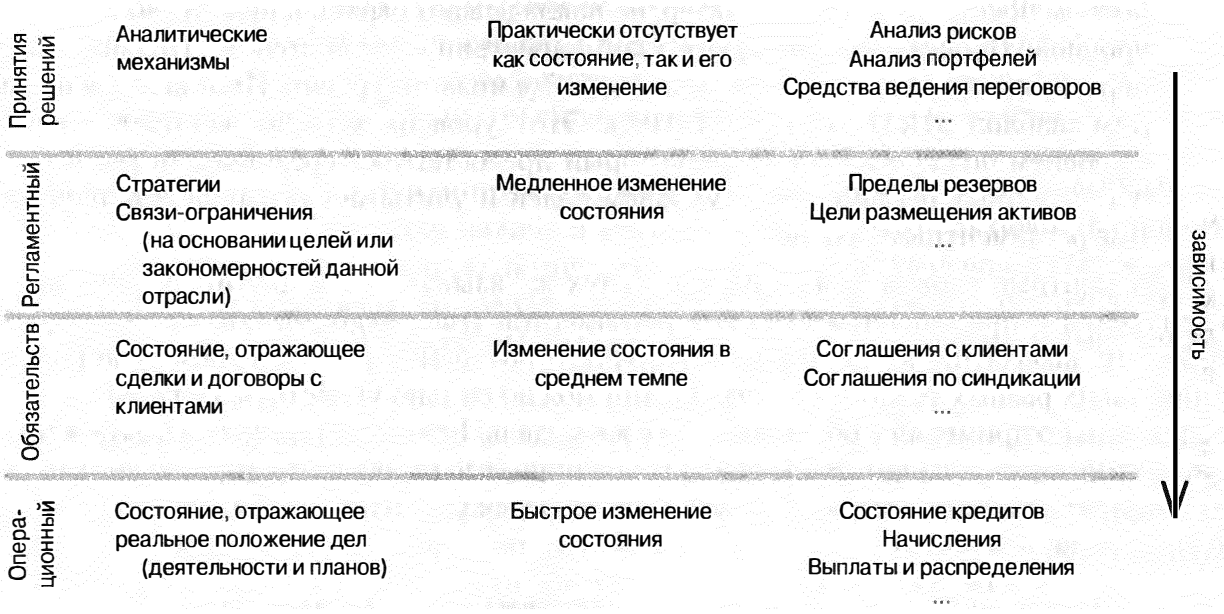
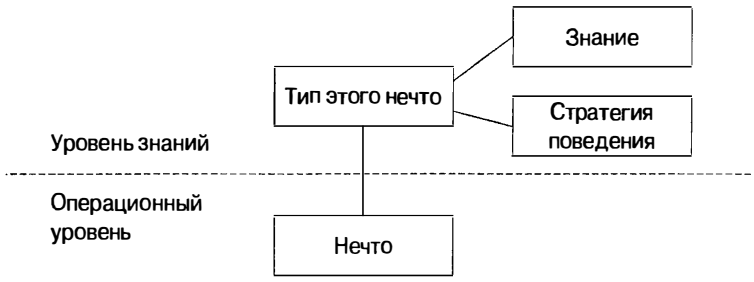


Рис. 16.14. Концептуальные взаимосвязи и точки относительного сдвига уровней в системе обслуживания инвестиционного банка

Потенциальный уровень и уровень обязательств не являются взаимоисключающими. Если в предметной области заметную роль играют оба уровня — как, например, в транспортной компании с большим разнообразием услуг по доставке — то и в приложении будет присутствовать и тот, и другой. Могут понадобиться и другие уровни, специфичные для таких областей. Варьируйте, экспериментируйте, но стремитесь сделать систему уровней как можно проще. Если уровней больше четырех-пяти, работать становится неудобно: слишком трудно передать сущность операций, устройство предметной области, и все проблемы сложности, которые призвано было решить введение многоуровневой структуры, возвращаются в другом виде. Крупномасштабную структуру нужно постоянно дистиллировать.

Перечисленные пять уровней применимы в целом ряде систем управления предприятиями. Но не во всех предметных областях они способны моделировать ключевые обязанности объектов. В некоторых случаях принудительное приведение архитектуры к этому образцу может сделать только хуже. Но и тогда еще может существовать естественный набор УРОВНЕЙ РАЗДЕЛЕНИЯ ОБЯЗАННОСТЕЙ, соответствующий задаче. Если же предметная область совсем далека от тех, которые мы рассмотрели, то, возможно, придется выбрать совершенно другие уровни. В общем, следуйте своей интуиции, выберите стартовую точку и дайте архитектуре эволюционировать.

Уровень знаний



[УРОВЕНЬ ЗНАНИЙ — это] группа объектов, которая описывает, как должна вести себя другая группа объектов. [Martin Fowler, "Accountability", www.martinfowler.com]

УРОВЕНЬ ЗНАНИЙ (KNOWLEDGE LEVEL) решает следующую задачу. Предположим, нам нужно, чтобы некоторая часть модели была "податливой" в руках пользователя, но при этом подчинялась более-менее широкому набору правил. Такие свойства соответствуют техническому заданию на программу с конфигурируемым поведением, в которой роли и взаимоотношения между СУЩНОСТЯМИ (ENTITIES) должны изменяться при установке или даже в ходе работы.

Этот шаблон возникает при рассмотрении подотчетности и ответственности внутри организаций, а затем применяется к правилам проводки в бухгалтерском учете [11]. Хотя шаблон фигурирует в нескольких главах, там нет отдельной главы, специально посвященной ему, потому что он отличается от большинства шаблонов в книге. Вместо моделирования предметной области, а именно это компетенция других аналитических шаблонов, УРОВЕНЬ ЗНАНИЙ структурирует модель.

Чтобы поставить задачу конкретнее, рассмотрим модели "подотчетности". Организации состоят из людей и из меньших организаций; в них определяются исполняемые роли и взаимоотношения. Правила (регламенты), задающие эти роли и отношения, очень отличаются в разных организациях. В одной фирме "отделение" может возглавляться "директором", подотчетным "вице-президенту". В другой фирме "модуль" возглавляется "менеджером", который подотчетен "старшему менеджеру". Есть еще и "матричные" организации, в которых каждый сотрудник по разным видам работ подотчетен разным лицам.

В типичном приложении обычно принимается ряд предположений. Если они неадекватны задаче, пользователи станут использовать поля ввода данных не для того, для чего требуется. Операции приложения будут "бить" мимо, поскольку их семантику изменили пользователи. Тогда пользователи придумают обходные пути, чтобы не оказаться отрезанными от высокоуровневых функций приложения. Им придется научиться сложным переходам от того, что они делают в своей работе, к тому, как работает само приложение. Такое приложение — плохой помощник.

Когда систему нужно будет изменять или заменять, разработчики обнаружат (рано или поздно), что смысл ее функций совсем не таков, как они думали. Эти функции могут восприниматься совсем по-разному в разных сообществах пользователей или в разных ситуациях. Изменить что-либо, не нарушая сложившихся неформальных практик пользования такой программой, будет очень сложно. Перенос данных в более продуманную систему потребует глубокого понимания сути и грамотного программирования всех этих уловок и "приемчиков".

Пример

Ведомость зарплаты и пенсионных отчислений, часть I

Отдел кадров небольшой компании работает с простой программой для расчета ведомостей заработной платы и пенсионных отчислений.

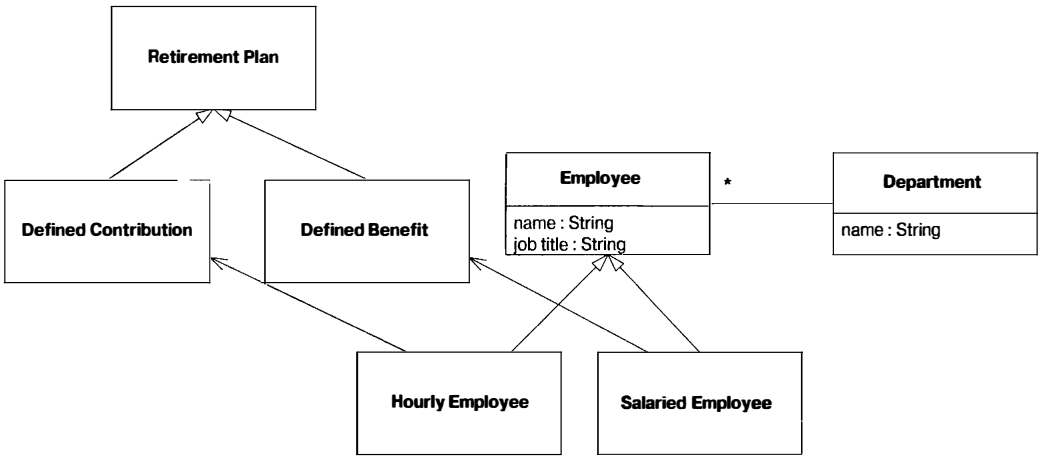


Рис. 16.15. Старая модель, перегруженная ограничениями с точки зрения новых требований

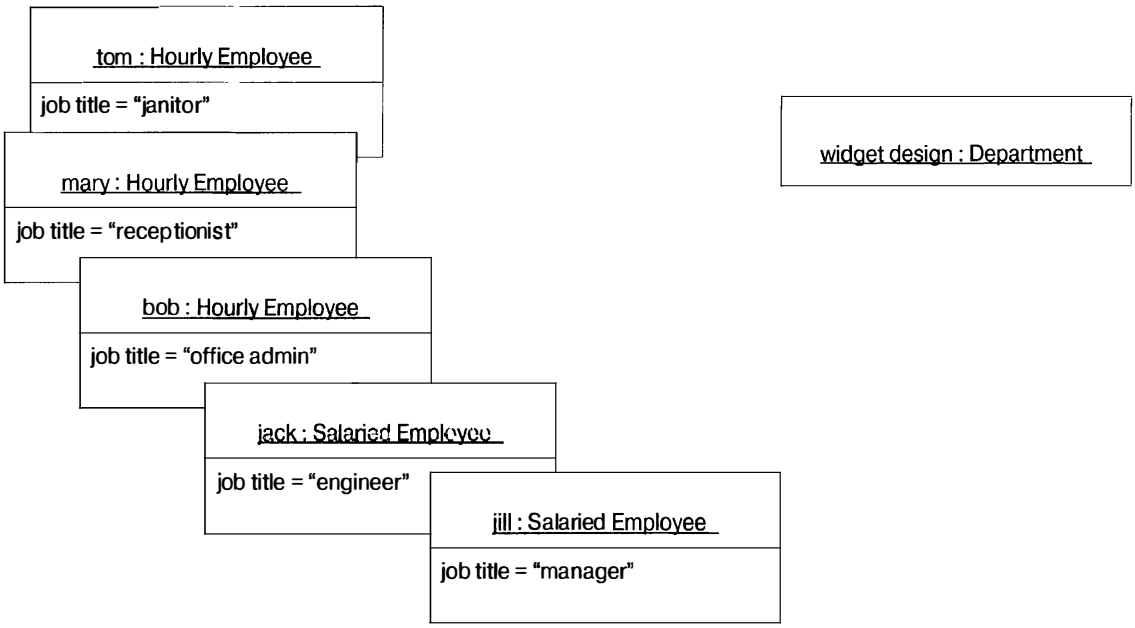


Рис. 16.16. Представление некоторых сотрудников с помощью старой модели

И вот руководство решает, что офисные администраторы должны перейти на пенсионный план с фиксированным размером пособия (*defined benefit*). Но проблема в том, что офисным администраторам платят почасово, а данная модель не допускает смешанных вариантов. Поэтому нужно менять саму модель.

Следующее предложение очень просто: убрать все связи-ограничения.

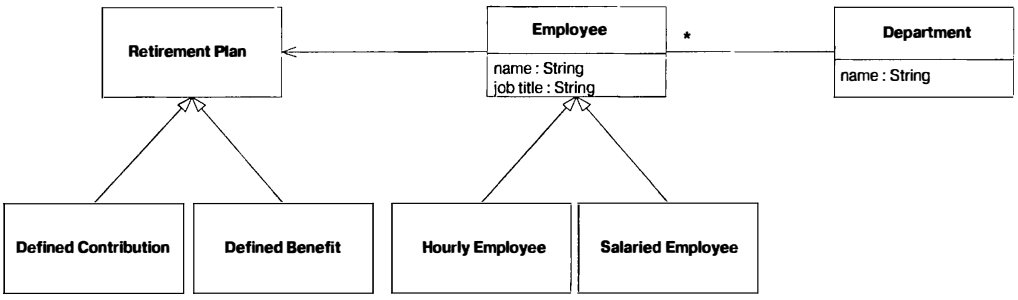


Рис. 16.17. Предложенная модель: теперь ограничений не хватает

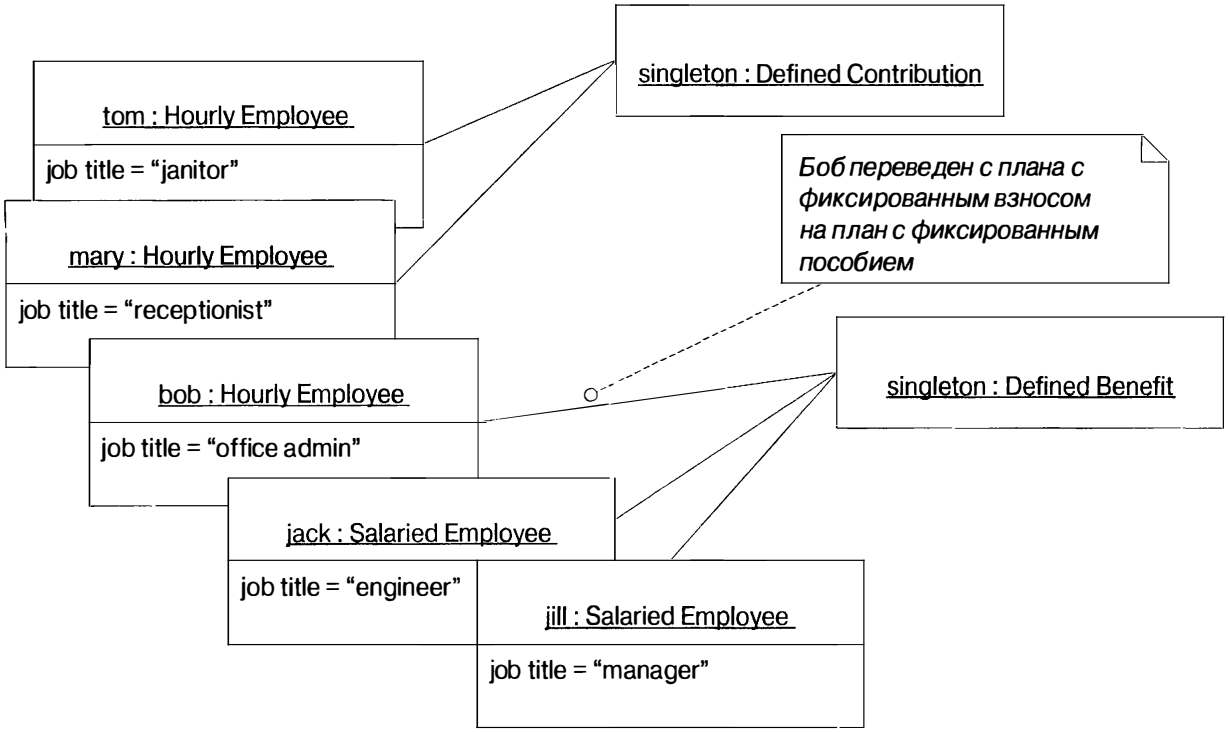


Рис. 16.18. Сотрудники могут выбрать неверный пенсионный план

Такая модель позволяет любому сотруднику выбрать любой вариант пенсионного плана. Каждому офисному администратору теперь доступна смена плана. Эту модель руководство отвергает, потому что она не отражает политику фирмы. Некоторые администраторы перейдут на другой план, а некоторые — нет. План может сменить даже уборщица. Руководство хочет иметь модель, которая реализует следующий регламент.

> *Офисные администраторы — это сотрудники с почасовой оплатой и фиксированным размером пенсии (план "defined benefit").*

Это правило предполагает, что поле "должность" (`job title`) должно представлять важное понятие предметной области. Программисты могут выполнить рефакторинг и вынести это понятие в явные под названием **`Employee Type`**, т.е. **`Тип сотрудника`**.

Требования можно выразить на ЕДИНОМ ЯЗЫКЕ следующим образом.

> ***Типу сотрудника*** *(**`Employee Type`**) ставится в соответствие один из* ***Пенсионных планов*** *(**`Retirement Plan`**) или одна из ведомостей оплаты труда.*
>
> *На* ***Сотрудников*** *(**`Employees`**) наложена связь-ограничение в виде* ***Типа сотрудника*** *(**`Employee Type`**).*

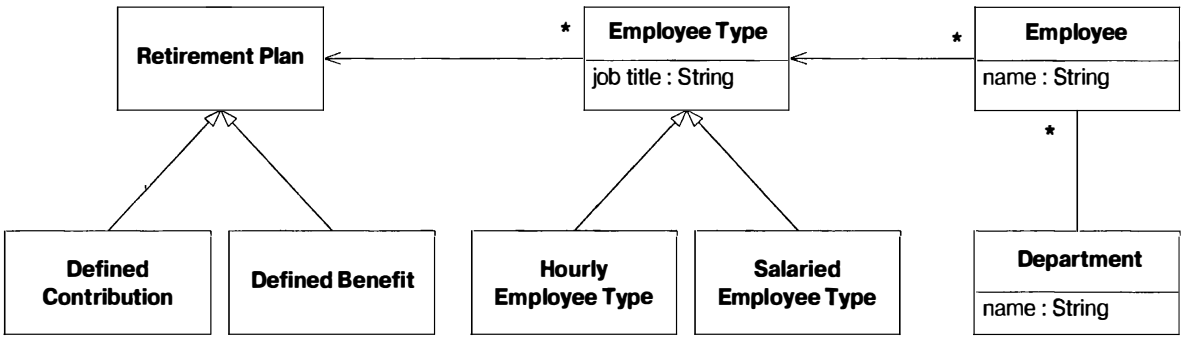


Рис. 16.19. Объект ***Type*** *позволяет удовлетворить требования*

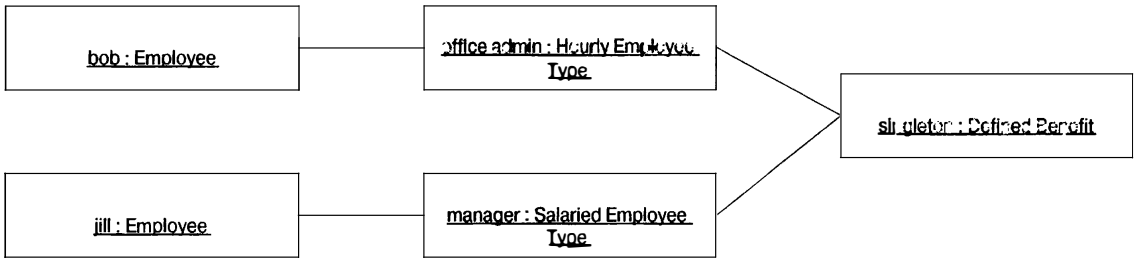


Рис. 16.20. Каждому ***Типу сотрудника*** *назначается пенсионный план*

Доступ к объекту **Тип сотрудника (Employee Type)** для его редактирования будет предоставлен только "суперпользователю", который будет вносить изменения только при изменении политики компании. Обыкновенный пользователь в отделе кадров сможет изменять объекты **Сотрудник (Employee)** или ассоциировать их с другими **Типами (Employee Type)**.

Эта модель удовлетворяет заданным требованиям. Разработчики предполагают, что здесь еще есть одно-два неявных понятия, но на данный момент это не более чем мелкие придирки. Никаких значительных идей для реализации нет, так что они сходятся на этом.

Статическая модель порождает проблемы. Но не менее серьезные проблемы могут возникнуть и во вполне гибкой системе, которая способна выразить любое возможное отношение между объектами. Такой системой было бы неудобно пользоваться, и через нее нельзя было бы реализовать правила, установленные организацией.

Писать полностью настраиваемые приложения, пригодные для любой организации, не слишком практично. Даже если каждая организация готова была бы платить за "подгонку" приложения под ее нужды, все равно организационная структура имеет тенденцию изменяться.

Итак, подобная программа должна предлагать *пользователю* возможность ее конфигурирования для отражения текущей организационной структуры. Но беда в том, что добавление таких возможностей в объекты модели делает их громоздкими. Чем больше гибкости, тем сложнее система.

В приложении, где роли и взаимоотношения между СУЩНОСТЯМИ **(**ENTITIES**) меняются в разных ситуациях, сложность может нарастать лавинообразно. Ни самые общие, ни гибко настраиваемые модели могут не отвечать потребностям пользователя. В объектах появляются ссылки на другие типы, отражающие все многообразие случаев, или же атрибуты, которые используются по-разному в разных ситуациях. Могут размножиться классы, содержащие одни и те же данные и операции — только для того, чтобы реализовать разные правила сборки.**

В нашу модель встраивается другая модель, описывающая ее. Самоопределение модели выносится в отдельный УРОВЕНЬ ЗНАНИЙ (KNOWLEDGE LEVEL), отчего все связи-ограничения становятся явными.

УРОВЕНЬ ЗНАНИЙ представляет собою применение к уровню предметной области шаблона REFLECTION (ОТРАЖЕНИЕ), который используется во многих программных архитектурах и технических инфраструктурах. Он хорошо описан в книге [7]. Чтобы учесть меняющиеся потребности, ОТРАЖЕНИЕ придает программе элементы "самосознания" и открывает как некоторые аспекты ее структуры, так и функции для адаптации и изменения. Это достигается разбиением приложения на "базовый уровень" (*base level*), в котором содержатся собственно его операционные обязанности, и "мета-уровень" (*meta level*), представляющий знания о строении и поведении программы.

Показательно, что в названии этого шаблона слово "уровень" отсутствует. Хотя сходство с многоуровневой структурой тут есть, для ОТРАЖЕНИЯ характерны двунаправленные взаимосвязи.

В языке Java имеется встроенная минимальная реализация ОТРАЖЕНИЯ в виде протоколов для опроса классов об их методах. Такие механизмы дают программе возможность задавать вопросы о своей собственной структуре. В CORBA есть несколько более расширенные, но аналогичные протоколы ОТРАЖЕНИЙ. В некоторых технологиях для поддержания непрерывности существования данных (*persistence technologies*) возможности самоописания расширены — с целью поддержки полуавтоматического отображения данных между таблицами баз данных и объектами. Есть и другие технические примеры. Этот шаблон может также применяться на уровне предметной области.

Сравнение терминологии шаблонов УРОВЕНЬ ЗНАНИЙ (KNOWLEDGE LEVEL) и ОТРАЖЕНИЕ (REFLECTION)

Терминология М.Фаулера	Терминология POSA[5]
Уровень знаний (*knowledge level*)	Мета-уровень (*meta level*)
Операционный уровень (*operations level*)	Базовый уровень (*base level*)

Сделаем разъяснение. Средства языка программирования, имеющие то же название *reflection* (*отражение*), не должны использоваться при реализации УРОВНЯ ЗНАНИЙ в модели предметной области. Эти мета-объекты описывают структуру и функции самих языковых конструкций. А вот УРОВЕНЬ ЗНАНИЙ должен строиться из обыкновенных объектов.

УРОВЕНЬ ЗНАНИЙ имеет две полезные особенности. Во-первых, он сосредоточен на предметной области приложения, в отличие от хорошо известных примеров использования ОТРАЖЕНИЙ. Во-вторых, он не претендует на максимальную общность. Как СПЕЦИФИКАЦИЯ (SPECIFICATION) может быть полезнее, чем общий предикат, так и узкоспециализированный набор ограничений, наложенный на совокупность объектов и взаимосвязей между ними, может оказаться ценнее, чем обобщенная архитектурная среда. УРОВЕНЬ ЗНАНИЙ устроен проще и может передавать конкретные намерения проектировщика.

Создайте отдельный набор объектов, которые могут использоваться для описания структуры и функций базовой модели, а также накладывания на нее связей-ограничений. Отделите эти части друг от друга как два "уровня": один — предельно

[5] Это сокращение от названия книги *Pattern-Oriented Software Architecture* [Buschmann et al., 1996].

конкретный, а другой — отражающий правила и знания, изменяемые пользователем или "суперпользователем".

Как и любая мощная идея, шаблоны ОТРАЖЕНИЕ (REFLECTION) и УРОВЕНЬ ЗНАНИЙ (KNOWLEDGE LEVEL) могут оказаться слишком соблазнительным. А использовать такой шаблон нужно с осторожностью. Он может снизить сложность кода, освободив объекты от необходимости быть "мастерами на все руки", но вводимая им же косвенность взаимосвязей частично возвращает в программу эту сложность. Если УРОВЕНЬ ЗНАНИЙ усложняется, поведение системы становится трудным для понимания как разработчиками, так и пользователями. Пользователям (или "суперпользователю"), конфигурирующим программу, в конце концов понадобятся навыки программистов — да еще и мета-программистов. Если же они сделают ошибки, программа будет функционировать неправильно.

В дополнение к этому не полностью исчезает проблема переноса данных. Когда изменяется структура УРОВНЯ ЗНАНИЙ, надо что-то делать с объектами операционного уровня. Бывает, что старые и новые объекты могут сосуществовать, но в любом случае тут потребуется тщательный анализ.

Все эти проблемы ложатся тяжким бременем на плечи проектировщика УРОВНЯ ЗНАНИЙ. Проектируемая архитектура должна быть достаточно устойчивой, чтобы реализовать не только все сценарии, сформулированные в ходе разработки, но и любые, под которые пользователь может сконфигурировать приложение в будущем. Если применять УРОВЕНЬ ЗНАНИЙ с разумной умеренностью, до той точки, в которой возможность конфигурирования оказывается критической и без этого уровня грозит исказить архитектуру, то можно решать проблемы, которые другим способом одолеть очень трудно.

Пример

Ведомость зарплаты и пенсионных отчислений, часть II: уровень знаний

Итак, наши разработчики снова в строю. Один из них, отдохнувший и выспавшийся, начинает "подбираться" к одному из "неуклюжих" мест в программе. Почему некоторые объекты защищены, тогда как другие могут свободно подвергаться изменениям? Совокупность защищенных объектов напомнила ему о шаблоне УРОВЕНЬ ЗНАНИЙ (KNOWLEDGE LEVEL), и он решил опробовать его как способ восприятия модели. И обнаружил, что существующую модель уже можно воспринимать таким образом.

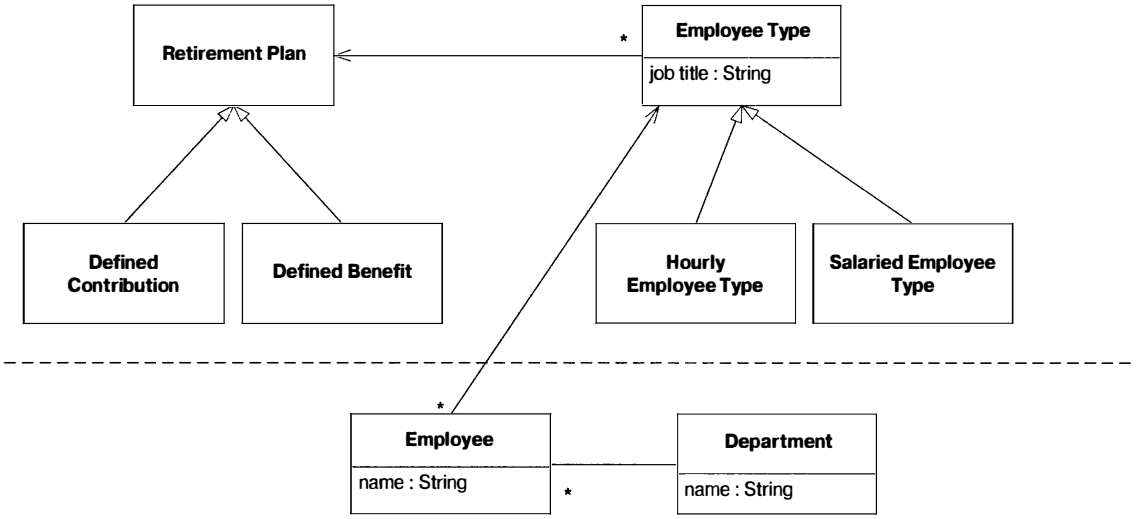


Рис. 16.21. Выделение УРОВНЯ ЗНАНИЙ, присутствовавшего в текущей модели неявно

Ограничения на модификацию оказались на уровне знаний, тогда как обычные, повседневно используемые средства редактирования — на операционном уровне. Это удача. Все объекты над чертой описывают типы или долговременные стратегии. Тип **Employee Type (Тип сотрудника)** сразу же накладывает на тип **Employee (Сотрудник)** четко определенные функции.

Разработчик как раз делился своей находкой с коллегами, когда одного из остальных программистов посетило еще одно озарение. Четкая организация модели посредством УРОВНЯ ЗНАНИЙ позволила ему понять, что именно беспокоило его в последнее время. В одном и том же объекте объединились два различных понятия. Он услышал это в выражении, сказанном на день раньше, но тогда не придал этому значения.

> ***Типу сотрудника (Employee Type)*** *ставится в соответствие один из* ***Пенсионных планов (Retirement Plan)*** *или одна из ведомостей оплаты труда.*

Но это же *не* утверждение на ЕДИНОМ ЯЗЫКЕ модели. В модели нет никакой "ведомости". Фраза сказана на языке *желательном*, но не фактическом. Понятие ведомости выплаты зарплаты неявно присутствовало в модели, склеиваясь с **Типом сотрудника (Employee Type)**. До выделения УРОВНЯ ЗНАНИЙ это не было настолько очевидно, и сами элементы из этой ключевой фразы фигурировали все на одном и том же уровне... за исключением одного.

На основе этих выводов снова был выполнен рефакторинг модели, поддерживающей данное утверждение.

Необходимость в контроле пользователей над правилами ассоциирования объектов привела разработчиков к модели, содержавшей неявный УРОВЕНЬ ЗНАНИЙ.

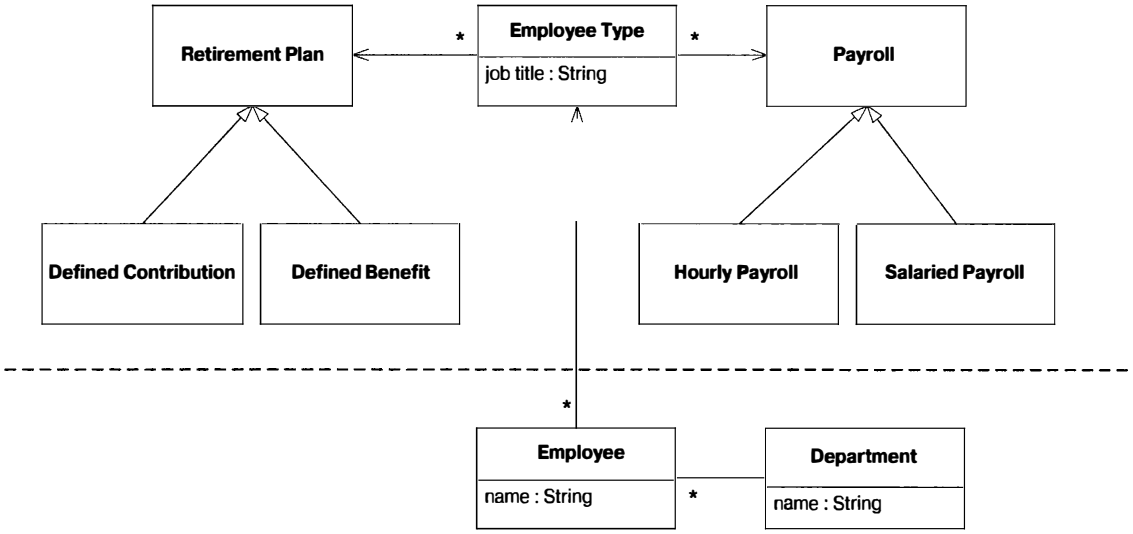


Рис. 16.22. Теперь ***Payroll (Ведомость)*** *фигурирует явно, отдельно от* ***Employee Type (Типа сотрудника)***

На присутствие УРОВНЯ ЗНАНИЙ намекали специфические ограничения доступа и взаимосвязи типа "предмет-предмет". Его формальное введение в модель прояснило ситуацию, и были сделаны новые выводы, в результате которых, благодаря выделению **Ведомости (Payroll)**, обособились два важных понятия предметной области.

УРОВЕНЬ ЗНАНИЙ (KNOWLEDGE LEVEL), как и другие крупномасштабные структуры, строго говоря, не является обязательным для применения. Объекты будут работать и без него, и даже в таком случае можно будет сделать и использовать выводы, приведшие к

отделению ВЕДОМОСТИ (PAYROLL) от **Типа сотрудника (Employee Type)**. Может придти время, когда эта структура уже не будет оправдывать себя, и от нее можно будет отказаться. Но пока она, по-видимому, передает важное знание о системе и помогает разработчикам "бороться" с моделью.

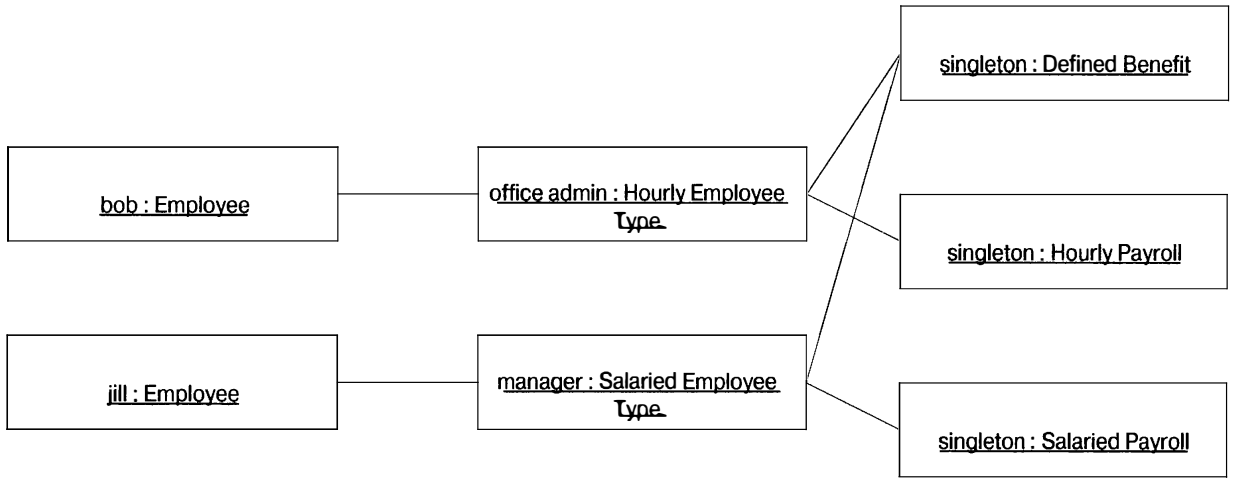


Рис. 16.23. Каждый ***Тип сотрудника (Employee Type)*** *теперь имеет* ***Пенсионный план (Retirement Plan)*** *и* ***Ведомость (Payroll)***

* * *

На первый взгляд УРОВЕНЬ ЗНАНИЙ (KNOWLEDGE LEVEL) похож на частный случай одного из УРОВНЕЙ РАЗДЕЛЕНИЯ ОБЯЗАННОСТЕЙ (RESPONSIBILITY LAYERS), особенно "регламентного" (*policy*). Но это не так. Во-первых, между УРОВНЕМ ЗНАНИЙ и остальными уровнями устанавливаются двусторонние связи, тогда как среди УРОВНЕЙ РАЗДЕЛЕНИЯ ОБЯЗАННОСТЕЙ нижние уровни независимы от верхних[6].

На самом деле УРОВЕНЬ ЗНАНИЙ может сосуществовать с большинством остальных крупномасштабных структур, добавляя к ним дополнительный аспект организации.

Среда подключаемых компонентов

В очень "зрелой" модели, углубленной и хорошо дистиллированной, возникает много разных возможностей. СРЕДА ПОДКЛЮЧАЕМЫХ КОМПОНЕНТОВ (PLUGGABLE COMPONENT FRAMEWORK) обычно становится актуальной уже после того, как в одной и той же предметной области реализуется несколько приложений.

* * *

Когда реализуется совместная работа целого ряда разных приложений, основанных на одних и тех же абстракциях, но спроектированных отдельно, их интеграцию ограничивает необходимость трансляции между несколькими ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ **(**BOUNDED CONTEXTS**). ОБЩЕЕ ЯДРО (**SHARED KERNEL**) не годится для групп, связаных недостаточно тесным сотрудничеством. Дублирование и фрагментация работ поднимает стоимость разработки и установки приложений, а их совместная работа затрудняется.**

---

[6] Автор употребляет в отношении УРОВНЯ ЗНАНИЙ слово *level* (собственно "уровень"), тогда как в отношении уровней разделения обязанностей — слово *layer* (фактически "слой"). Это различие не так уж велико и важно, но оно присутствует. *Примеч. перев.*

В некоторых успешных проектах архитектура разбивается на компоненты, каждый из которых отвечает за определенную категорию рабочих функций. Обычно все эти компоненты подключаются к центральному распределительному модулю, или "концентратору" (*hub*), который поддерживает все нужные протоколы и умеет общаться через все предоставляемые интерфейсы. Существуют и другие формы подключения компонентов. Структуру этих интерфейсов и распределительного модуля необходимо скоординировать между собой, а внешние части программного комплекса можно проектировать относительно независимо.

Этот шаблон поддерживается в нескольких широко используемых технических средах разработки, но не это главное. Техническая среда нужна только тогда, когда она решает какую-то существенную техническую проблему — например, распределение или совместное использование компонента разными приложениями. А шаблон предлагает, собственно, концептуальную организацию распределения обязанностей, и применить его можно в пределах одной программы на Java.

Дистиллируйте АБСТРАКТНОЕ ЯДРО **(**ABSTRACT CORE**) интерфейсов и взаимосвязей и постройте среду или библиотеку, в которой различные реализации этих интерфейсов были бы легко заменяемы. Аналогично, предоставьте любому из приложений возможность использования этих компонентов, но с условием обращения к ним строго через интерфейсы** АБСТРАКТНОГО ЯДРА**.**

Высокоуровневые абстракции определяются и распределяются по всей системе, а специализация достигается в модулях. Центральным распределительным модулем системы является АБСТРАКТНОЕ ЯДРО (ABSTRACT CORE) внутри ОБЩЕГО ЯДРА (SHARED KERNEL). Но за интерфейсами инкапсулированных компонентов может стоять множество разных ОГРАНИЧЕННЫХ КОНТЕКСТОВ (BOUNDED CONTEXTS). Такая структура построена специально для тех случаев, когда разные компоненты поступают из разных источников или же инкапсулируют ранее написанные программы с целью их интеграции.

Вообще-то, компоненты вовсе не обязаны иметь в своей основе разные модели. Несколько компонентов вполне может быть разработано в одном КОНТЕКСТЕ, если разработчики практикуют НЕПРЕРЫВНУЮ ИНТЕГРАЦИЮ (CONTINUOUS INTEGRATION) или могут определить другое ОБЩЕЕ ЯДРО (SHARED KERNEL) для набора "близкородственных", тесно связанных компонентов. Все эти подходы легко могут сосуществовать в крупномасштабной структуре ПОДКЛЮЧАЕМЫХ КОМПОНЕНТОВ (PLUGGABLE COMPONENTS). Еще один вариант, применимый в некоторых случаях, — это использование общедоступного языка (PUBLISHED LANGUAGE) для интерфейса, к которому подключаются компоненты.

У СРЕДЫ ПОДКЛЮЧАЕМЫХ КОМПОНЕНТОВ есть и свои недостатки. Один из них — чрезвычайная сложность этого шаблона в применении. Он требует высокой точности при проектировании интерфейсов и достаточно глубокой модели, содержащей все необходимые функции в АБСТРАКТНОМ ЯДРЕ (ABSTRACT CORE). Еще один существенный недостаток — ограниченная изменчивость приложения, недостаток вариантов развития. Если потребуется применить совершенно другой подход в СМЫСЛОВОМ ЯДРЕ (CORE DOMAIN), этому помешает имеющаяся крупномасштабная структура. Разработчики могут специализировать модель, но не смогут изменить СМЫСЛОВОЕ ЯДРО, не изменив одновременно и протокол для всех отличающихся компонентов. В результате процесс непрерывного совершенствования ЯДРА и углубляющий рефакторинг практически остановятся.

В книге [10] дается хороший обзор "дерзких" попыток применения СРЕД ПОДКЛЮЧАЕМЫХ КОМПОНЕНТОВ (PLUGGABLE COMPONENT FRAMEWORKS) в нескольких областях. В числе других рассматривается SEMATECH CIM. Такие среды пользовались переменным успехом. Самое большое препятствие к их применению, по-видимому, состоит в том, что для проектирования полезной среды нужна очень глубокая степень понимания проблемы. СРЕДА ПОДКЛЮЧАЕМЫХ КОМПОНЕНТОВ не должна быть ни первой,

ни даже второй крупномасштабной структурой, применяемой к проекту. Наиболее успешные примеры возникали тогда, когда вначале были полностью и в достаточном количестве разработаны специализированные приложения.

Пример

Среда SEMATECH CIM

На заводе по производству компьютерных микросхем (чипов) группы (*партии*) кремниевых пластин движутся от одного агрегата к другому, проходя через сотни этапов обработки, пока процесс нанесения и протравливания микроскопических электрических цепей не будет завершен. Заводу требуется программное обеспечение, которое бы умело отслеживать каждую отдельную партию, точно записывать всю выполненную над ней обработку, а затем направлять либо работников завода, либо автоматические устройства для переноса партии к следующему агрегату и применения следующего технологического процесса. Такие программы называются *системами управления производством*, *СУП* (*manufacturing execution system*, MES).

В ходе всего этого используются сотни разных агрегатов от десятков поставщиков, и на каждый этап работ есть четко прописанные инструкции и спецификации. Создать СУП, которая бы смогла справиться с этим нагромождением, — дело сложное и запредельно дорогое. Для решения этой проблемы промышленный консорциум SEMATECH разработал среду CIM Framework.

Система CIM — очень большая и сложная, в ней много разных аспектов, но нам здесь важны два из них. Во-первых, среда определяет абстрактные интерфейсы для базовых понятий, связанных с управлением производством полупроводниковых устройств, другими словами, СМЫСЛОВОЕ ЯДРО (CORE DOMAIN) предметной области в форме АБСТРАКТНОГО ЯДРА (ABSTRACT CORE). В этих определениях интерфейсов присутствует и функциональность, и семантика.

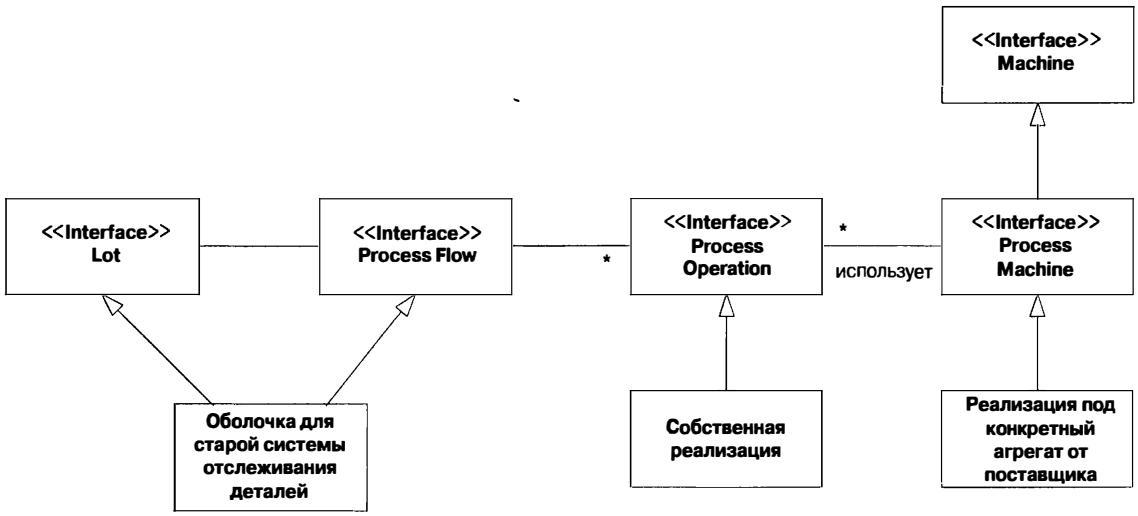


Рис. 16.24. Сильно упрощенное подмножество интерфейсов CIM с примерами реализаций

Если поставщик оборудования вводит в строй новый агрегат, ему необходимо разработать специализированную реализацию интерфейса **`Process Machine`** (**`Технологический агрегат`**). Если соблюсти все требования интерфейса, их управляющий агрегатом компонент сможет подключаться к любому приложению, основанному на среде CIM Framework.

Определив эти интерфейсы, SEMATECH определил и правила, по которым они могут взаимодействовать в приложении. Любое приложение на базе CIM Framework должно реализовать протокол, включающий объекты, которые реализуют какое-то подмножество таких интерфейсов. Если протокол реализован, и приложение строго придерживается абстрактных интерфейсов, то оно может рассчитывать на обещанный этими интерфейсами набор функций независимо от конкретной реализации. Сочетание интерфейсов и протокола их использования образует жесткую, существенно ограничительную крупномасштабную структуру.

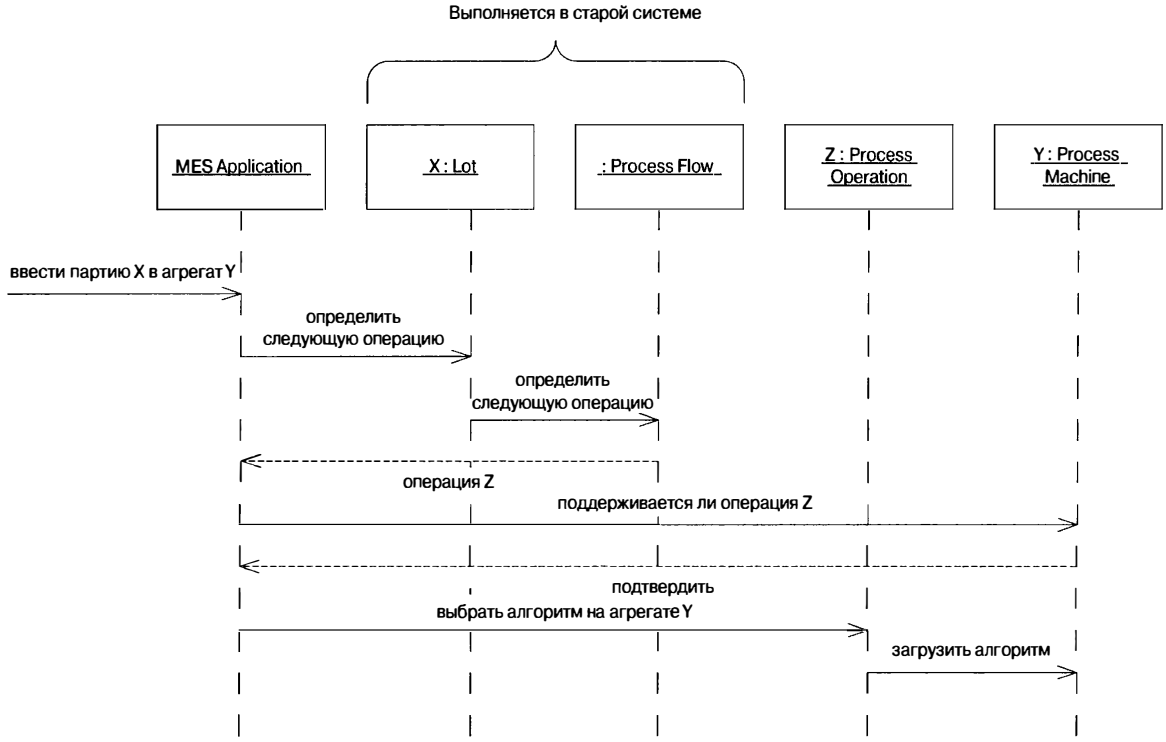


Рис. 16.25. Пользователь закладывает партию пластин в следующий агрегат и регистрирует их перемещение в компьютере

В среде имеются очень специфические требования к инфраструктуре. Она тесно привязана к CORBA для обеспечения непрерывности существования данных, обработки транзакций и событий, выполнения других технических задач. Но что в ней самое интересное — это определение СРЕДЫ ПОДКЛЮЧАЕМЫХ КОМПОНЕНТОВ, которая каждому позволяет разрабатывать свое программное обеспечение и беспрепятственно интегрировать его в огромные существующие системы. Никто не знает всех деталей такой системы, но все понимают ее общие свойства.

* * *

Как могут тысячи людей, работая независимо, сделать панно из более чем 40 000 панелей?

Крупномасштабная структура для мозаичного панно памяти жертв СПИДа возникла благодаря нескольким простым общим правилам, а о подробностях позаботились отдельные участники. Обратите внимание, что в правилах поставлена общая цель (почтить память людей, умерших от СПИДа), определяются свойства компонентов, позволяющие интегрировать их в общее целое, и задается возможность манипулировать большими фрагментами панно (например, складывать его).

Создание лоскута для панно

[*взято с веб-сайта проекта мозаичного панно памяти жертв СПИДа*, `www.aidsquilt.org`]

Оформление лоскута

Включите в него имя человека, которого вы поминаете. Можно включить дополнительную информацию: даты рождения и смерти, место рождения... Посвятите один лоскут одному человеку...

Выбор материала

Помните, что панно будут много раз складывать и разворачивать, так что имеет значение его долговечность. Свойства клея со временем ухудшаются, поэтому если к лоскуту нужно что-то прикрепить, это лучше пришить. Лучше всего подходит нерастягивающаяся ткань средней плотности — парусина или поплин.

Композиция может быть вертикальной или горизонтальной, но окончательный лоскут с подрубленной кромкой должен иметь размеры 3 фута на 6 (90x180 см) — не более и не менее! Когда будете резать ткань, оставьте на каждой стороне лишних 5–7 см для кромки. Если вы не сможете подрубить кромку сами, оставьте это нам. Рекомендуется сделать подкладку, не обязательно толстую — это позволяет поддерживать чистоту лоскутов при раскладывании их на земле, а также сохранять форму ткани.

Создание лоскута

Для создания лоскута воспользуйтесь одним или несколькими нижеперечисленными способами.

- **Аппликация**. На ткань основы пришиваются кусочки другой ткани, буквы, небольшие памятные изображения. Клей использовать не нужно — он не удержится.
- **Краска**. Нанесите рисунок текстильной краской, стойким красителем для ткани или несмываемыми чернилами. Не используйте объемные краски (*puffy paint*) — они слишком клейкие.
- **Трафарет**. Нанесите рисунок на ткань карандашом по трафарету, поднимите его и нанесите окончательный рисунок кисточкой или несмываемыми чернилами.
- **Коллаж из предметов**. Подберите не слишком объемные и тяжелые предметы из такого материала, чтобы не порвать ткань (например, стекло и блестки не подходят).
- **Фотопечать**. Лучше всего наносить фотографии или буквы, перенеся их на специальные переводные картинки, которые наносятся проглаживанием горячим утюгом. Нанесите их на 100% хлопковую ткань и пришейте ее к основе. Фотографии можно нанести также на прозрачный виниловый пластик и пришить его к основному лоскуту (не в центре, чтобы их не складывали).

Насколько жесткой должна быть структура

Шаблоны крупномасштабных структур, рассмотренные в этой главе, по своей жесткости охватывают весьма широкий диапазон: от весьма вольного ОБРАЗА СИСТЕМЫ (SYSTEM METAPHOR) до строго определенной СРЕДЫ ПОДКЛЮЧАЕМЫХ КОМПОНЕНТОВ (PLUGGABLE COMPONENT FRAMEWORK). Конечно, возможны и другие структуры. Даже в пределах общего структурного шаблона всегда есть выбор, насколько жестко определить его правила.

Например, УРОВНИ РАЗДЕЛЕНИЯ ОБЯЗАННОСТЕЙ (RESPONSIBILITY LAYERS) диктуют способ выделения понятий модели и взаимосвязи между ними, но можно добавить и свои правила, которые будут задавать образцы взаимодействия между уровнями.

Рассмотрим завод, на котором автоматизированная программная система направляет каждую деталь к нужному агрегату, где она обрабатывается согласно какому-то технологическому процессу. Определение нужного процесса поступает с уровня регламентов (*policy*), а выполняется он на операционном уровне. Но на нижнем, производственном, уровне неизбежны ошибки. Реальная ситуация не будет соответствовать правилам, заданным в программе. Так вот, операционный уровень должен отражать реальность *как она есть*. Это означает, что если деталь случайно поместили не в тот агрегат, информация об этом должна быть воспринята безо всяких оговорок и условий. Эту исключительную ситуацию надо каким-то образом перенаправить на более высокий уровень, где включатся механизмы принятия решений, и проблема будет решена по другим регламентам — например, деталь будет направлена в ремонт или выброшена за негодностью. Но операционный уровень ничего не знает об уровнях выше него. Обмен информацией следует организовать так, чтобы он не создавал двусторонних взаимосвязей между нижними и верхними уровнями.

Обычно оповещение подобного рода организуется через механизмы событий. Объекты операционного уровня генерируют события при каждом изменении своего состояния. А объекты уровня регламентов "прослушивают" интересующие их события с нижних уровней. Если событие нарушает какое-то регламентное правило, то либо выполняется конкретное ответное действие (фигурирующее в определении правила), либо генерируется событие для оповещения еще более высокого уровня.

В примере с банком стоимость активов изменяется (операционный уровень), изменяя и стоимость сегментов портфеля. Когда она превышает установленные для портфеля пределы ассигнований (регламентный уровень), посылается извещение трейдеру, который может купить или продать активы для восстановления баланса.

Все это можно решать в рабочем порядке, но можно и установить некий согласованный порядок действий, которому бы следовали все в ходе взаимодействия между объектами того или иного уровня. Более жесткая структура увеличивает единообразие и облегчает интерпретацию архитектуры. Если структура подходит к задаче, ее регламентные правила, скорее всего, будут подталкивать разработчиков к хорошей архитектуре, и отдельные части будут удачно сходиться вместе.

С другой стороны, структурные ограничения могут ударить по гибкости, которая требуется разработчикам. Специфические схемы коммуникации могут оказаться неэффективными для реализации между разными ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ (BOUNDED CONTEXTS) в неоднородной системе, особенно если используются разные технологии реализации.

Итак, следует бороться с искушением строить жесткие архитектурные среды и строго регламентировать реализацию крупномасштабной структуры. Наиболее важная заслуга крупномасштабной структуры — достижение концептуальной связности и углубление понимания предметной области. *Каждое структурное правило должно облегчать разработку.*

Структурирующий рефакторинг

В эпоху, когда наша индустрия старается стряхнуть с себя оковы жесткого заблаговременного проектирования программ, многие сочтут крупномасштабную структуру "откатом" к старым недобрым временам "каскадной" архитектуры с односторонним потоком данных[7]. Но на самом деле найти действительно полезную структуру можно только через углубленное понимание предметной области и стоящей задачи. А практический способ достижения такого понимания — это итерационный процесс разработки.

---

[7] Автор употребляет метафору *waterfall*, т.е. "водопад", который никогда не течет вверх, а только вниз. *Примеч. перев.*

Группа разработчиков, стремящаяся к ЭВОЛЮЦИОННОЙ ОРГАНИЗАЦИИ модели (EVOLVING ORDER), должна бесстрашно пересматривать свою крупномасштабную структуру в течение всего времени реализации проекта. Разработчики не обязаны тащить на себе структуру, порожденную в самом начале, когда никто еще не понимал предметную область или техническое задание как следует.

К сожалению, эволюция означает, что окончательная структура не будет известна на старте. Пока структура не вырисуется в своем финальном виде, будет много рефакторинга. Это дорого и трудно, но это *необходимо*. Существует несколько общих подходов, позволяющих снизить стоимость при максимизации выгоды.

Минимализм

Один из ключей к удешевлению структуры — это ее простота и легкость. Не пытайтесь охватить все на свете. Решайте самые насущные и серьезные задачи, а остальное пусть складывается в рабочем порядке.

На ранних этапах работы бывает полезно выбрать нежесткую структуру наподобие ОБРАЗА СИСТЕМЫ (SYSTEM METAPHOR) или нескольких УРОВНЕЙ РАЗДЕЛЕНИЯ ОБЯЗАННОСТЕЙ (RESPONSIBILITY LAYERS). Даже самая минималистическая и нежесткая структура дает общие подсказки, помогающие избежать хаоса.

Коммуникативность и самодисциплина

Следовать структуре как при написании нового кода, так и при рефакторинге должна вся группа. Для этого вся группа должна ее понимать. Терминология и описания взаимосвязей внутри структуры должны войти в ЕДИНЫЙ ЯЗЫК (UBIQUITOUS LANGUAGE).

Крупномасштабная структура может породить словарь общих понятий, облегчающий проекту связь с системой, а разным разработчикам — единообразие в принятии решений. Но большинство таких структур имеют нежесткий, концептуальный и рекомендательный характер, поэтому в группе разработчиков должна царить самодисциплина.

Если не все участники в достаточной мере привержены структуре, она имеет тенденцию к разложению. Взаимосвязь структуры с теми или иными конкретными частями модели или реализации обычно не слишком очевидна в коде программы, и функциональные тесты от структуры не зависят. К тому же структура имеет тенденцию быть абстрактной, так что единообразие разработки бывает нелегко соблюсти в большом коллективе (или нескольких коллективах).

Коммуникативный уровень дискуссий, практикуемый в большинстве групп разработчиков, недостаточен для поддержания в системе согласованной крупномасштабной структуры. Критически важно инкорпорировать ее в ЕДИНЫЙ ЯЗЫК проекта и приучить всех сотрудников постоянно пользоваться этим языком.

Реструктуризация дает гибкую архитектуру

Любое изменение в структуре может вызвать необходимость масштабного рефакторинга. Структура эволюционирует по мере возрастания сложности системы и углубления ее понимания. Каждый раз, когда структура меняется на другую, *необходимо менять всю систему, чтобы приспособить ее к новому порядку*. Очевидно, это немалый труд.

Но все не так плохо, как кажется. Мне случалось убедиться, что архитектуру, в которой есть крупномасштабная структура, трансформировать во что-то новое гораздо легче, чем такую, где ее нет. Это верно даже для тех случаев, когда сам тип структуры изменяется на другой — например, от ОБРАЗА (METAPHOR) мы переходим к УРОВНЯМ (LAYERS).

Я не могу полностью объяснить этот феномен. Частично ответ состоит в том, что реструктуризировать нечто легче тогда, когда понимаешь, как оно устроено в текущий момент, а наличие структуры облегчает такое понимание. Частично дело в дисциплине, которой требует поддержание структуры — ее наличие отражается на всех аспектах системы. Но есть и нечто большее, думается мне. Дело в том, что изменять систему, в которой ранее сменилось уже *две* крупномасштабных структуры, — *еще легче*.

Новая кожаная куртка — слишком жесткая и неудобная, но после первого дня ее ношения локти, разогнувшись и согнувшись много раз, становятся подвижнее. Еще за несколько дней "разнашиваются плечи", и куртку становится легче надевать. Через несколько месяцев кожа становится гибкой и податливой, а куртка — удобной и легкой. Похоже, именно так обстоит дело и с моделями, которые несколько раз подвергались разумным преобразованиям. В них закладывается постоянно пополняющаяся база знаний, *определяются и "прокачиваются" основные направления изменений*, тогда как стабильные аспекты упрощаются. В структуре модели вырисовываются наиболее общие КОНЦЕПТУАЛЬНЫЕ КОНТУРЫ предметной области, лежащей в ее основе.

Дистилляция

Еще один важнейший этап при создании модели — это непрерывная дистилляция. Она облегчает изменение структуры сразу несколькими способами. Прежде всего, в ходе нее механизмы, НЕСПЕЦИАЛИЗИРОВАННЫЕ ПОДОБЛАСТИ (GENERIC SUBDOMAINS) и другие вспомогательные структуры выносятся вовне из СМЫСЛОВОГО ЯДРА (CORE DOMAIN), после чего там попросту остается меньше материала для реструктуризации.

Если это возможно, вспомогательные элементы следует определять так, чтобы они встраивались в крупномасштабную структуру как можно проще. Например, в системе УРОВНЕЙ РАЗДЕЛЕНИЯ ОБЯЗАННОСТЕЙ (RESPONSIBILITY LAYERS) можно так определить НЕСПЕЦИАЛИЗИРОВАННУЮ ПОДОБЛАСТЬ (GENERIC SUBDOMAIN), что она поместится в один уровень. Если реализуются ПОДКЛЮЧАЕМЫЕ КОМПОНЕНТЫ (PLUGGABLE COMPONENTS), то такая подобласть может принадлежать целиком к одному компоненту или же быть ОБЩИМ ЯДРОМ (SHARED KERNEL) для набора родственных компонентов. Возможно, эти вспомогательные элементы придется подвергнуть рефакторингу, чтобы найти им подходящее место в структуре. Но они развиваются независимо от СМЫСЛОВОГО ЯДРА (CORE DOMAIN) и имеют тенденцию к более узкой специализации, что облегчает задачу. В конечном счете, они еще и не настолько критичны, чтобы их усовершенствование так уж много значило.

Принципы дистилляции и углубляющего рефакторинга применимы и к крупномасштабной структуре. Например, изначально уровни структуры могут быть выбраны на основе поверхностного знания предметной области; затем они постепенно заменяются более глубокими абстракциями, которые выражают фундаментальные задачи, решаемые системой. Отточенная четкость представления помогает разработчикам глубже понимать архитектуру системы, в чем и состоит цель структуризации. Но это не только цель, а еще в какой-то мере и средство, поскольку благодаря ей манипуляции с системой в глобальном масштабе становятся легче и безопаснее.

ГЛАВА 17

Объединение стратегических подходов

В предыдущих трех главах было представлено множество принципов и приемов стратегического предметно-ориентированного проектирования (DDD). Создавая архитектуру большой и сложной системы, часто приходится “пускать” в ход сразу несколько из них. Как крупномасштабная структура может сосуществовать с КАРТОЙ КОНТЕКСТОВ (CONTEXT MAP)? Как сочетаются друг с другом отдельные “кирпичики”? С чего следует начинать? Какой должен быть первый шаг? А третий? Как вообще взяться за разработку собственной стратегии?

Сочетание крупномасштабных структур и ограниченных контекстов

Сочетание крупномасштабных структур и ограниченных контекстов

Рис. 17.1.

Три основополагающих принципа стратегического проектирования (контекстность, дистилляция, крупномасштабная структура) не заменяют, а дополняют друг друга и находятся в разных формах взаимодействия. Например, крупномасштабная структура может существовать в пределах одного ОГРАНИЧЕННОГО КОНТЕКСТА (BOUNDED CONTEXT), а может распространяться сразу на несколько и являться организующим началом для КАРТЫ КОНТЕКСТОВ (CONTEXT MAP).

Приведенные ранее примеры УРОВНЕЙ РАЗДЕЛЕНИЯ ОБЯЗАННОСТЕЙ (RESPONSIBILITY LAYERS) были ограничены одним КОНТЕКСТОМ. Так легче всего объяснить основную идею, и именно таково основное применение этого шаблона. В этом простом сценарии смысловые значения имен уровней сосредоточены в одном и том же КОНТЕКСТЕ, как и имена элементов модели или интерфейсов подсистем, существующих внутри этого КОНТЕКСТА.

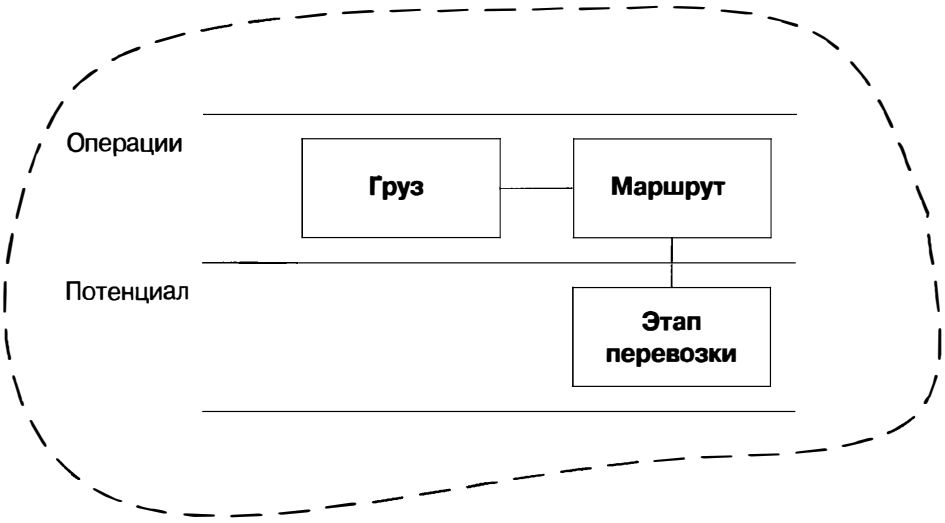


Рис. 17.2. Структуризация модели в пределах одного ОГРАНИЧЕННОГО КОНТЕКСТА

Подобная локальная структура может быть полезна в очень сложной, но унифицированной и единой модели; она поднимает уровень сложности до тех пределов, в каких систему еще можно держать в рамках одного КОНТЕКСТА.

Но во многих проектах возникает более сложная и важная задача: понять, как стыкуются между собой разные по смыслу части. Пусть даже они разделены на несколько разных КОНТЕКСТОВ, но какую роль каждая из частей играет в целостной системе и как эти части соотносятся одна с другой? В этом случае для организации КАРТЫ КОНТЕКСТОВ может использоваться крупномасштабная структура. В таком случае терминология структуры будет применима во всем проекте (или по крайней мере в четко очерченной его подобласти).

Предположим, вы хотите ввести УРОВНИ РАЗДЕЛЕНИЯ ОБЯЗАННОСТЕЙ (RESPONSIBILITY LAYERS), но у вас имеется старая система, организация которой несовместима с желаемой крупномасштабной структурой. Нужно ли отказываться от мысли о введении УРОВНЕЙ? Нет, но нужно правильно определить место, которое в этой структуре будет занимать старое приложение. Для этого бывает полезно как-то охарактеризовать это приложение, дать ему определение. Предоставляемые им услуги могут на самом деле ограничиваться одним-двумя УРОВНЯМИ. Если вы можете сказать, что старая система целиком помещается в такой-то УРОВЕНЬ (или УРОВНИ) РАЗДЕЛЕНИЯ ОБЯЗАННОСТЕЙ, это означает, что вы сумели кратко определить ключевую роль и сферу применения этой системы.

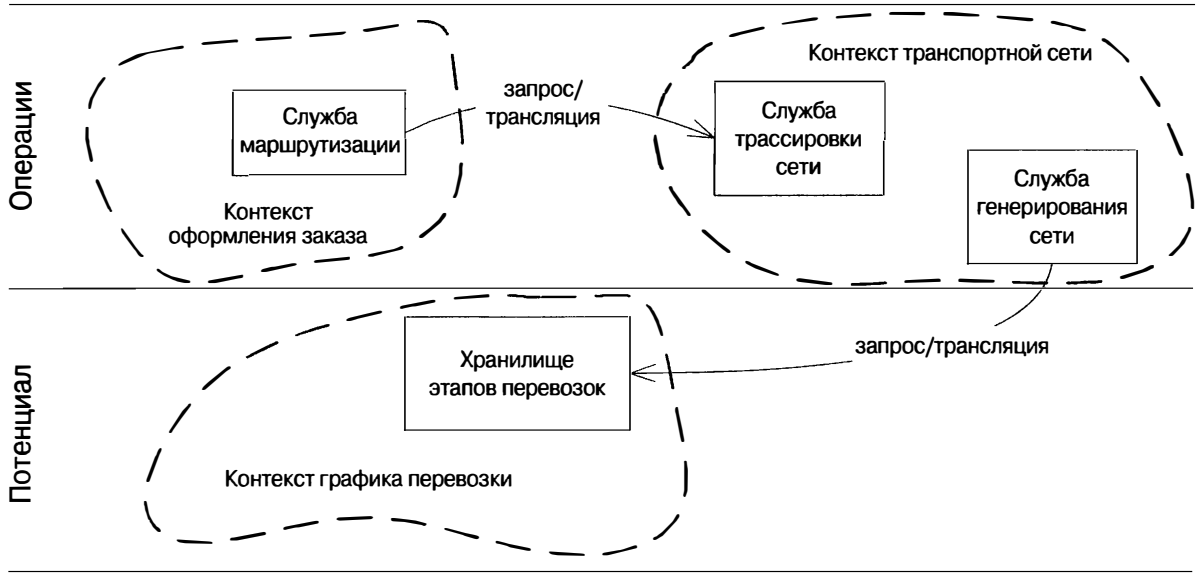


Рис. 17.3. Структура, введенная во взаимоотношениях между компонентами различных ОГРАНИЧЕННЫХ КОНТЕКСТОВ

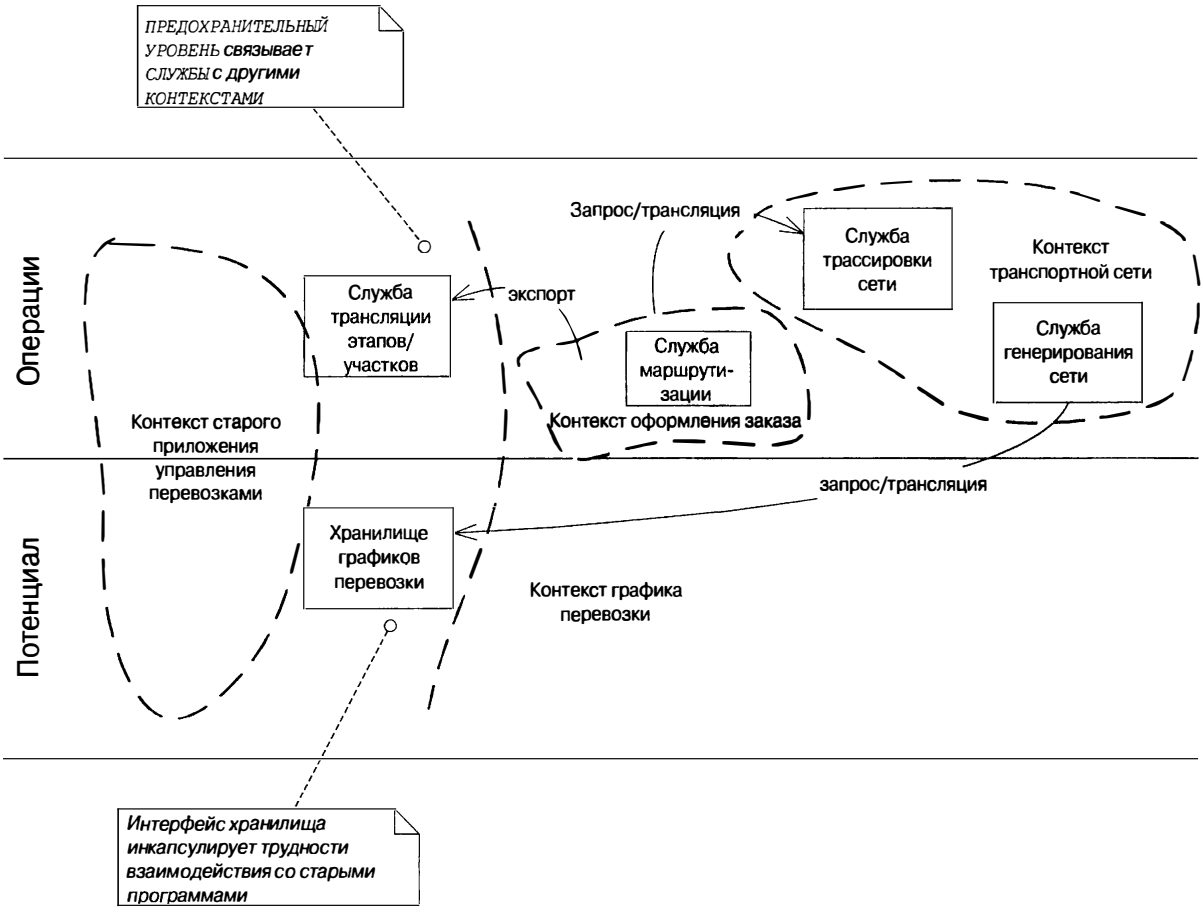


Рис. 17.4. Структура, в которой некоторым компонентам позволяется распространяться на несколько уровней

Если обращение к функциям старой системы реализовано через ФАСАДНЫЕ МЕТОДЫ (FACADE), то есть все шансы, что каждую из фасадных СЛУЖБ (SERVICES), которая ста-

вится в соответствие этим функциям, можно спроектировать так, чтобы она попадала целиком в один уровень.

Внутреннее устройство приложения по координированию поставки (Shipping Coordination), которое в данном примере является устаревшим, представлено в виде единой массы, где деталей устройства не видно. Но в проекте, где присутствует четкая крупномасштабная структура, простирающаяся через несколько ОГРАНИЧЕННЫХ КОНТЕКСТОВ, разработчики могут принять решение упорядочить свою модель в пределах своего КОНТЕКСТА по тем же самым, хорошо знакомым им УРОВНЯМ.

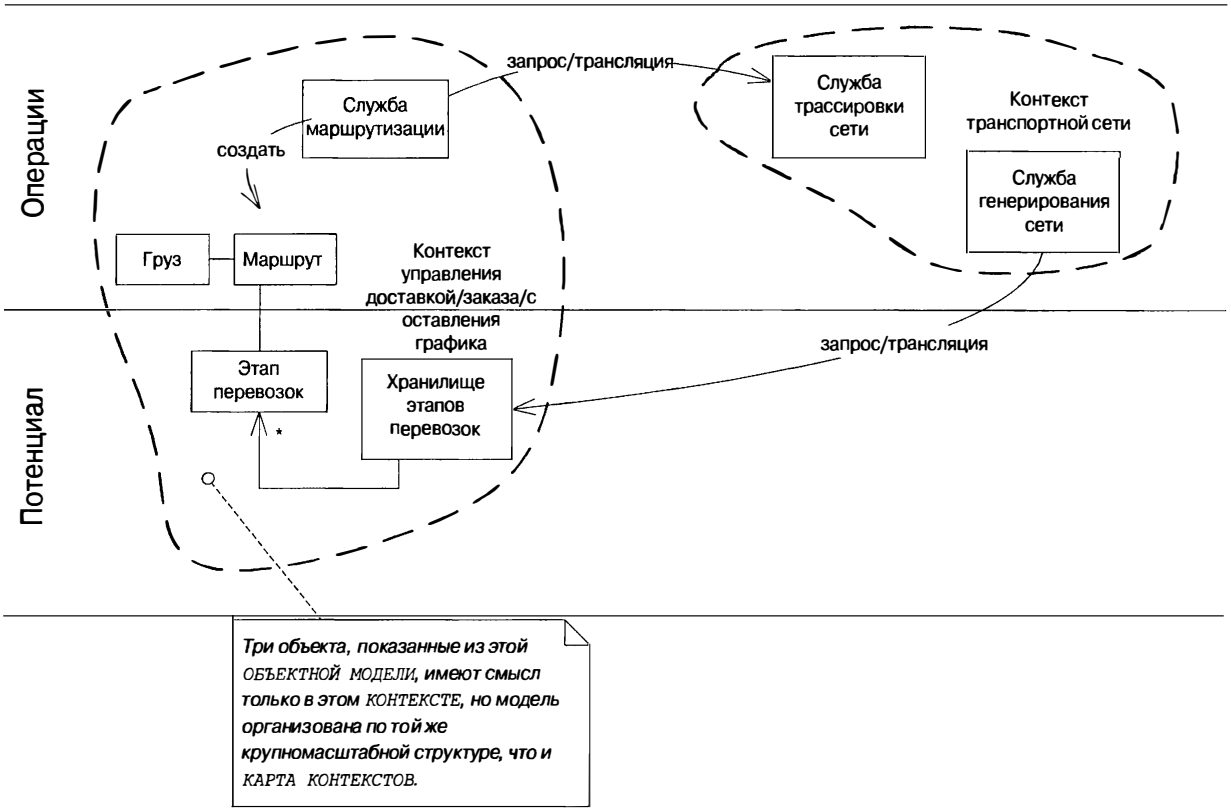


Рис. 17.5. Та же структура, примененная и внутри контекста, и в рамках всей КАРТЫ КОНТЕКСТОВ

Конечно, каждый ОГРАНИЧЕННЫЙ КОНТЕКСТ находится в своем собственном пространстве имен, и поэтому в пределах одного КОНТЕКСТА можно ввести одну структуру, а в пределах соседнего — другую, а всю КАРТУ организовать в соответствии с третьей. Но если слишком увлечься подобными вещами, можно обесценить саму идею крупномасштабной структуры как набора понятий, унифицирующего проект.

Сочетание крупномасштабной структуры и дистилляции

Понятия крупномасштабной структуры и дистилляции также взаимно дополняют друг друга. Крупномасштабная структура может помочь в объяснении взаимоотношений в пределах СМЫСЛОВОГО ЯДРА (CORE DOMAIN), а также между НЕСПЕЦИАЛИЗИРОВАННЫМИ ПОДОБЛАСТЯМИ (GENERIC SUBDOMAINS).

В то же время и сама крупномасштабная структура может являться важной частью СМЫСЛОВОГО ЯДРА. Например, если выделить уровни потенциала, операций, регламентов и поддержки принятия решений, это само по себе дистиллирует такое знание, которое яв-

ляется фундаментальным для решения той прикладной проблемы, для которой и пишется программное обеспечение. Это знание особенно полезно, если проект поделен между несколькими ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ (BOUNDED CONTEXTS), так что объекты СМЫСЛОВОГО ЯДРА модели не имеют прямого значения для большей части проекта.

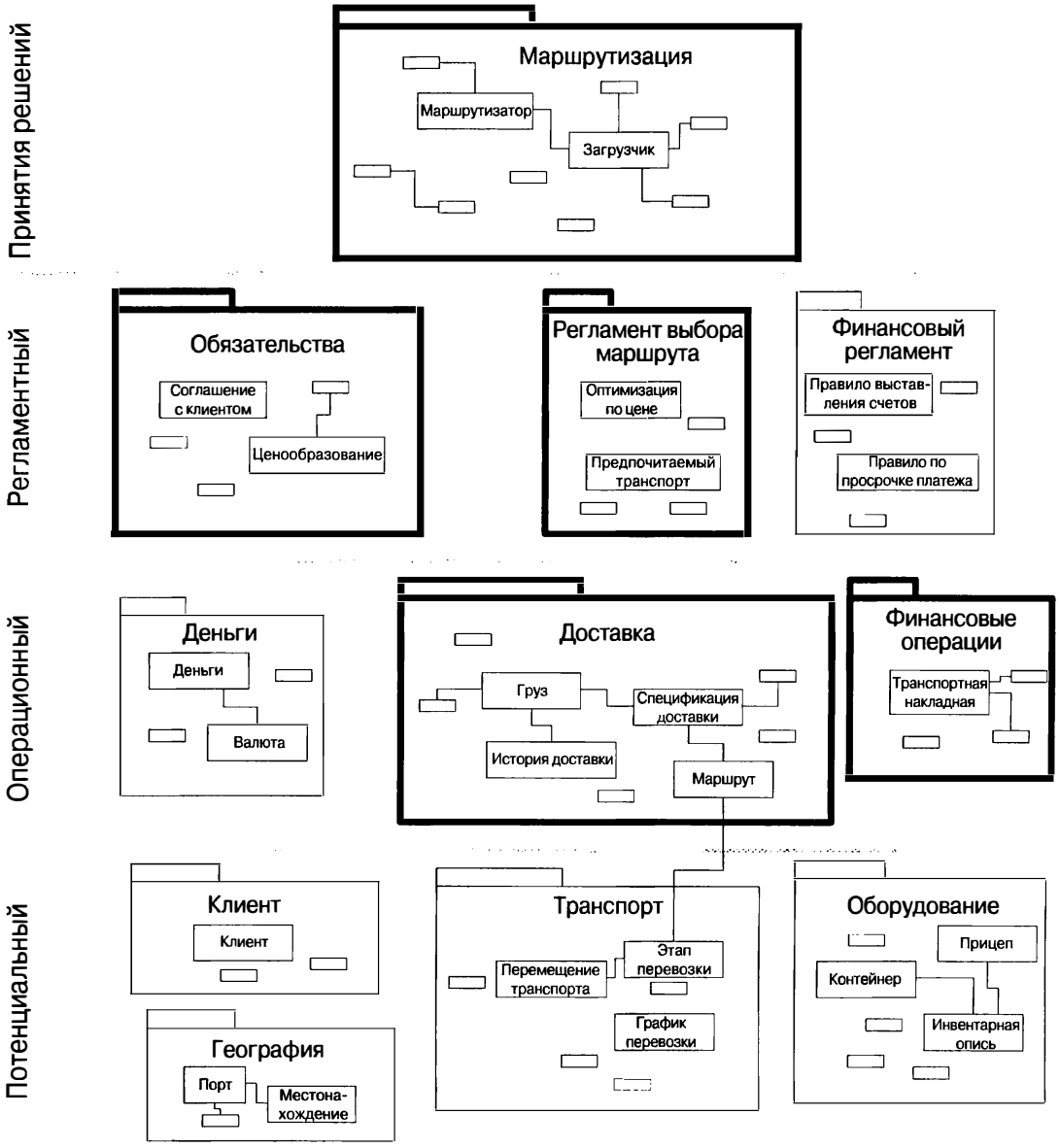


Рис. 17.6. Модули СМЫСЛОВОГО ЯДРА (CORE DOMAIN) и НЕСПЕЦИАЛИЗИРОВАННЫЕ ПОДОБЛАСТИ (GENERIC SUBDOMAINS) выявляются четче в многоуровневой структуре

Первоначальная оценка

Выполняя стратегическое проектирование в проекте, начните с четкой оценки текущей ситуации.

1. Начертите КАРТУ КОНТЕКСТОВ. Можно ли это сделать естественным способом, или есть препятствие в виде неоднозначности?
2. Обратите внимание на использование языка в проекте. Существует ли в нем ЕДИНЫЙ ЯЗЫК (UBIQUITOUS LANGUAGE)? Достаточно ли он богат, чтобы помочь в процессе разработки?

3. Определите самое важное. Обозначено ли в проекте СМЫСЛОВОЕ ЯДРО (CORE DOMAIN)? Написано ли введение в предметную область (DOMAIN VISION STATEMENT)? А вы можете его написать?
4. Помогает или мешает технология реализации проекта в ПРОЕКТИРОВАНИИ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN)?
5. Обладают ли разработчики достаточными техническими навыками?
6. Хорошо ли разработчики знают саму предметную область? Вызывает ли она у них *интерес*?

Идеальных ответов, конечно, найти не удастся. В настоящий момент вы знаете о проекте гораздо меньше, чем будете знать в будущем. Но все-таки эти вопросы дают хорошую "отправную точку". К моменту, когда вы получите на них пусть и первоначальные, но конкретные ответы, у вас уже выработается представление о том, что нужно сделать в первую очередь. С течением времени эти ответы можно усовершенствовать — особенно КАРТУ КОНТЕКСТОВ, ВВЕДЕНИЕ В ПРЕДМЕТНУЮ ОБЛАСТЬ и другие материалы проекта — для отражения изменившейся ситуации и новых знаний.

Кому планировать стратегию

В традиционной методике архитектура создается еще до начала разработки приложения и спускается сверху группой, у которой больше полномочий, чем у непосредственных разработчиков. Но так работать не обязательно — и получается, честно говоря, не слишком эффективно.

Стратегическое проектирование по самому своему определению должно применяться к проекту в целом. Существует много способов организации проектов, и мне не хотелось бы сводить все к слишком конкретным рецептам. Тем не менее, чтобы процесс принятия проектных решений давал результаты, необходимо знать кое-какие основы.

Вначале давайте бросим беглый взгляд на два стиля работы, которые, по моему опыту, дают практически полезные результаты (и проигнорируем старый подход "мудрость свыше").

Самозарождение структуры в ходе разработки

Группа с должной самодисциплиной, состоящая из хорошо контактирующих друг с другом разработчиков, может работать без "централизованной власти" и придти к некоему общему набору принципов путем ЭВОЛЮЦИОННОЙ ОРГАНИЗАЦИИ (EVOLVING ORDER). Порядок в этом случае органично развивается, а не навязывается.

Это типичная модель работы в экстремальном программировании. Теоретически структура может возникнуть совершенно спонтанно в результате интеллектуального прорыва любой из пар программистов. Но чаще бывает, что унифицированную крупномасштабную структуру помогает поддерживать один человек или группа людей, имеющая на это некие контрольные полномочия. Особенно хорошо этот подход работает, если подобный неформальный лидер — еще и практикующий разработчик, который умеет правильно судить и общаться, а не только выдавать идеи. В знакомых мне группах экстремального программирования лидерство по стратегическому проектированию часто возникало спонтанно и переходило, например, к коучу[1]. Но кем бы ни был этот естественный лидер, он все-таки остается и членом группы разработчиков. Отсюда следует, что

[1] *Coach* — это тренер, преподаватель, инструктор и т.п. Но так уж устоялось в экстремальном программировании, и не только в нем, что теперь мы говорим "коуч". — *Примеч. перев.*

во всей группе разработчиков должно быть как минимум несколько работников такого уровня для принятия проектных решений, которые повлияют на весь проект в целом.

Когда крупномасштабная структура распространяется на несколько групп разработчиков, тесно связанные группы могут перейти к неформальным видам взаимодействия. В такой ситуации каждая из групп способна делать открытия, ведущие к воплощению идеи крупномасштабной структуры, но конкретные варианты обсуждаются неформальным комитетом, составленным из представителей разных групп. После оценки влияния архитектурного решения участники могут решить принять, изменить или отвергнуть его. Группы пытаются двигаться вперед совместно, в такой вот неформальной связке. Это возможно, когда взаимодействующих групп сравнительно немного, когда они мотивированы на взаимное координирование усилий, когда их навыки проектирования сравнимы, а потребности в структурировании приложения настолько сходны, чтобы их удовлетворила одна крупномасштабная структура.

Смежная группа по разработке архитектуры

Если разработка стратегии ведется совместно несколькими группами, возникает желание иметь какую-то централизацию в принятии решений. Дискредитировавшая себя модель восседающего в вышине архитектора — не единственный возможный вариант. Группа проектирования архитектуры может действовать на равных правах с другими разработчиками, помогая им координировать и приводить в согласие их крупномасштабные структуры и границы ОГРАНИЧЕННЫХ КОНТЕКСТОВ (BOUNDED CONTEXTS), а также решать другие технические проблемы, представляющие общий интерес. Чтобы приносить пользу в таком контексте, проектировщики архитектуры должны настроиться на практическую разработку приложения.

В организационной диаграмме такая группа может походить на традиционную группу разработки архитектуры, но на деле она отличается во всех аспектах своей работы. Члены такой группы реально работают совместно с непосредственными разработчиками, прорабатывают вместе с ними шаблоны и формы, экспериментируют с целью дистилляции — в общем, не боятся марать руки.

Я встречал подобное пару раз. В подобных проектах главный архитектор обычно склонен делать все то, что перечислено далее в списке.

Шесть принципов принятия решений при стратегическом проектировании

Решения должны доводиться до всех членов группы

Очевидно, если не все участники проекта знают стратегию и следуют ей, то она оказывается вообще ни при чем. Чтобы этого не случилось, разработчики склонны организовываться вокруг централизованных групп проектирования архитектуры с официальными “властными” полномочиями, — чтобы гарантировать следование одним и тем же нормативным правилам. По иронии судьбы, архитекторов, запершихся в высоких кабинетах, чаще всего игнорируют, а их указания обходят. Собственно, у разработчиков и нет другого выхода, когда в результате отсутствия обратной связи с архитекторами попытки применить их требования к реальным приложениям кончаются совершенно нереальными схемами.

В проекте с высоким уровнем коммуникации стратегическое архитектурное проектирование в исполнении группы разработки имеет более реальный шанс получить всеоб-

щее распространение. Такая стратегия будет иметь связь с жизнью, а соответствующие полномочия помогут в принятии коллективных решений.

Какова бы ни была система, следует меньше беспокоиться об уровне полномочий, исходящем от руководства, чем о реальной взаимосвязи между разработчиками и стратегией.

В процессе принятия решений следует учитывать обратную связь

Чтобы выработать организующий принцип, крупномасштабную структуру или достаточно тонкую дистилляцию, необходимо глубоко понимать цели и задачи проекта, а также понятия его предметной области. Единственные люди, которые обладают должной глубиной познаний — это члены группы разработки. Это объясняет, почему архитектуры приложений, построенные специальными архитектурными группами, так редко бывают полезными, несмотря на несомненный талант архитекторов.

В отличие от разработки технической инфраструктуры и архитектуры, стратегическое проектирование не подразумевает написания больших объемов кода, хотя и влияет на весь процесс разработки. А вот активное участие всех групп разработчиков оно как раз подразумевает. Опытный архитектор обладает умением слушать идеи, исходящие от различных людей, и способствовать выработке общего решения.

Одна из технических архитектурных групп, с которой я работал, постоянно посылала своих членов на работу в другие группы разработки, где пользовались их средой и библиотеками. Благодаря такой ротации кадров в архитектурную группу поступал реальный опыт решения проблем, стоявших перед разработчиками, а к разработчикам — знания о тонкостях применения среды. Именно такая тесная обратная связь и требуется в процессе стратегического проектирования.

План должен допускать эволюцию, развитие

Эффективная разработка программного обеспечения — это очень динамичный процесс. Если на самом верхнем уровне принятия решений все уже “высечено в камне”, разработчикам остается меньше вариантов выбора в тех случаях, когда они вынуждены реагировать на изменения. ЭВОЛЮЦИОННАЯ ОРГАНИЗАЦИЯ (EVOLVING ORDER) позволяет избежать этой ловушки, поскольку требует непрерывных изменений в крупномасштабной структуре по итогам углубления знаний о предмете.

Если слишком много проектных решений жестко принимается заранее, разработчики оказываются “стреноженными” и не могут гибко реагировать на те проблемы, которые перед ними ставятся. Так что принцип, приводящий приложение к единообразию — это ценно и хорошо, но он должен меняться и развиваться по мере реализации проекта, а также не отнимать слишком много возможностей у разработчиков, которым и так приходится нелегко.

Если в проекте присутствует тесная обратная связь, то при возникновении препятствий в ходе разработки возникают также и неожиданные возможности, и даже инновации.

Разработчики архитектуры не должны переманивать к себе лучшие кадры

Проектирование на этом уровне требует такой интеллектуальной подготовки, какая есть у немногих. Руководители имеют привычку переводить наиболее технически способных разработчиков в группы проектирования архитектуры и инфраструктуры, потому что хотят найти их квалификации наилучшее применение среди опытных проектировщиков. Со своей стороны разработчиков привлекает возможность более активно влиять на проект или заниматься “более интересными” задачами. Да и быть членом элитной группы уже престижно само по себе.

Таким образом, для реальной текущей разработки приложения часто остаются лишь наименее технически подготовленные программисты. Но для разработки тоже требуются

хорошие навыки проектирования, иначе затея обречена на провал. Даже если стратегическая группа спроектирует хорошую архитектуру, у разработчиков может не оказаться должной подготовки для ее реализации.

В противоположность уже сказанному, в такие группы почти никогда не включаются разработчики, которые не так искушены технически, но имеют самые обширные знания предметной области. Стратегическое проектирование — это не чисто техническая задача; размежевание с программистами, имеющими глубокие знания по предмету, ограничивает возможности архитекторов. И специалисты из самой предметной области, кстати, тоже нужны.

Важно иметь сильных проектировщиков во всех группах разработчиков. А в каждой группе, которая занимается стратегическим проектированием, необходимо иметь достаточно знаний по предметной области. Иногда бывает просто необходимо нанять дополнительно лучших проектировщиков-архитекторов. Бывает, что архитектурную группу достаточно привлечь на частичную занятость. В общем, вариантов может быть много, но любая эффективная группа стратегического проектирования должна иметь в партнерах не менее эффективную группу технической разработки.

В стратегическом проектировании нужны минимализм и скромность

Дистилляция и минимализм важны для любой хорошей архитектуры, но в стратегическом проектировании минимализм еще важнее. Даже мелкая нестыковка может привести к проблемам. Специально выделенным архитектурным группам следует соблюдать осторожность, поскольку они хуже чувствуют препятствия, возникающие благодаря их деятельности перед разработчиками. В то же время энтузиазм архитекторов, порождаемый их широкими полномочиями, часто уводит их куда-то в сторону. Я это не только видел много раз, но и сам бывал грешен. Одна хорошая идея цепляется за другую, и в конце концов получаем перегруженную архитектуру, работающую против нас же.

Вместо этого необходимо дисциплинировать себя и произвести на свет организующие принципы и ключевые модели, очищенные от всего, что не помогает улучшить четкость и ясность архитектуры. По правде говоря, практически любой элемент чему-то да мешает, поэтому любая мелочь должна иметь веское обоснование своего существования. Требуется скромность, чтобы признать, что даже самая лучшая ваша идея для кого-то может являться препятствием.

Объекты — для специализации, разработчики — для обобщения

Сущность хорошего объектно-ориентированного проектирования состоит в том, чтобы дать каждому объекту четкие и узкоспециализированные обязанности, снизив взаимозависимость с другими объектами до абсолютного минимума. Иногда мы пытаемся и взаимодействие в группах выстроить так же аккуратно, как в самом приложении. Но в хорошем проекте всегда находится много людей, которым интересны дела других. Разработчики экспериментируют с архитектурными средами. Архитекторы пишут код приложений. Все общаются со всеми. Фактически, царит хаос. Пусть специализация будет присуща объектам, а разработчики должны заниматься обобщением.

В этой книге я подчеркнул различие между *стратегическим проектированием* и другими его видами, чтобы прояснить особенности решаемых в процессе задач. Теперь же необходимо отметить, что двумя разными видами проектирования вовсе не обязаны заниматься две разных категории людей. Построить гибкую архитектуру на основе углубленной модели — это сложная проектная деятельность, но подробности в ней так важны, что это должен делать кто-то, непосредственно работающий с кодом. Стратегическое проектирование вырастает из проектирования конкретных программных структур, и все

же оно требует целостного, масштабного видения всей проектной деятельности — может быть, даже деятельности нескольких групп разработчиков. Программисты обожают находить способы так "нарезать" задачи, чтобы архитекторы не обязаны были понимать предметную область, а специалисты в этой области не должны были бы разбираться в технологии реализации. Конечно, есть предел у способности отдельного человека освоить некую деятельность, но сверхспециализация лишает предметно-ориентированное проектирование самой его сути и силы.

То же верно и для технических сред проектирования

Технические архитектурные среды (*frameworks*) могут сильно ускорить разработку приложения, в том числе и уровня его предметной области, предоставив для этой цели готовый уровень инфраструктуры. Благодаря этому приложение освобождается от необходимости реализовать основные технические службы. Кроме того, предметная область изолируется от других задач. Но есть и риск, что техническая архитектура *окажет влияние на выразительную смысловую реализацию модели предметной области и внесет изменчивость*. Это может произойти даже тогда, когда проектировщики архитектурной среды вовсе не имеют намерения вмешиваться в уровни предметной области или прикладных операций.

Те же требования, которые устанавливают планку для стратегического проектирования, могут помочь и в разработке технической архитектуры. Эволюционность, минимализм, тесное сотрудничество с непосредственными разработчиками могут породить постоянно совершенствующийся набор технических служб и правил, реально помогающих в разработке и ничем не мешающих. Архитектуры, которые не следуют этому принципу, либо снижают творческую активность разработчиков, либо просто игнорируются и обходятся, отчего разработка приложения практически лишается всякой архитектурной составляющей.

Существует один подход, следование которому гарантированно убивает техническую архитектурную среду разработки.

Не пишите архитектурные среды для "чайников"

Если делить разработчиков на группы, подразумевая при этом, что некоторые из них неспособны к архитектурному проектированию, то впереди, скорее всего, вас ждет неудача, поскольку этим недооценивается сложность технической разработки приложения. Если вы считаете, что у этих людей не хватает ума проектировать, нечего поручать им разработку программ. А если ума у них хватает, тогда нянчиться с ними — это только ставить преграды между ними и требующимися им для работы инструментами.

Такой подход приводит к проблемам и во взаимоотношениях между группами. Мне приходилось оказываться в числе таких снобов, после чего извиняться перед программистами в каждой беседе за группы, в которые я входил. (Изменить что-либо в такой группе мне, боюсь, не удалось.)

Следует отметить, что *инкапсуляция несущественных технических деталей резко отличается* от того вида заблаговременной структуризации, который я терпеть не могу. Техническая среда разработки может вложить в руки разработчика мощные абстракции и программные средства, освобождая его от тяжелой рутины. Трудно описать эту разницу в каком-то обобщенном виде, но все же ее можно обозначить следующим способом. Попросите создателей среды ответить, чего они ожидают от пользователя их среды/компонента/инструментального средства. Если они высоко оценивают уровень квалификации пользователя среды, то, скорее всего, находятся на правильном пути.

Долой генеральный план

Группа архитекторов (тех, которые проектируют реальные здания), возглавляемая Кристофером Александером (Christopher Alexander), отстаивала принцип постепенного, пошагового роста в архитектуре и градостроительстве. Они очень хорошо объяснили, почему генеральные планы проваливаются.

> *Не имея некоего процесса планирования, совершенно невозможно когда-либо придать Орегонскому университету (University of Oregon) такой глубокий и гармоничный архитектурный порядок, как тот, что лежит в основе строений Кембриджского университета (University of Cambridge).*
>
> *Подобные проблемы всегда решались разработкой генерального плана. В таком плане пытаются дать достаточно точные указания, чтобы обеспечить единство среды в целом, и все же оставить место для адаптации отдельных зданий и участков пространства к местным нуждам.*
>
> *...и все разнообразные части этого будущего университета образуют единое, связное целое, поскольку все они просто вставляются в ячейки-гнезда единой архитектуры.*
>
> *...на практике генеральный план терпит поражение — потому что он создает тоталитарный порядок, а не естественный. Генеральному плану присуща излишняя "жесткость", он не может адаптироваться к естественным и непредсказуемым изменениям, которые неизбежны в жизни сообщества. Как только подобные изменения случаются... генеральный план устаревает, и ему больше не следуют. Но даже в той мере, в какой генплану все-таки следуют... там недостаточно говорится о связях между зданиями, населенности, сбалансированности функций и т.п., чтобы он мог помочь каждому местному строительству или объекту архитектуры хорошо вписаться в окружающую среду как в единое целое.*
>
> *...Попытка идти этим курсом напоминает, скорее, "заполнение" цветов в детской книжке-раскраске... В лучшем случае порядок, возникающий в результате такой процедуры, банален.*
>
> *...Итак, в качестве источника органичного, естественного порядка генеральный план одновременно и слишком точен, и недостаточно точен. Все в целом задано слишком жестко, тогда как детали определены недостаточно точно.*
>
> *...наличие генерального плана отталкивает пользователей объекта, [поскольку по определению] члены жилищного сообщества практически не могут влиять на свои будущие жилищные условия ведь большинство самых важных проектных решений уже принято.*
>
> *Из книги "The Oregon Experiment", [1]*

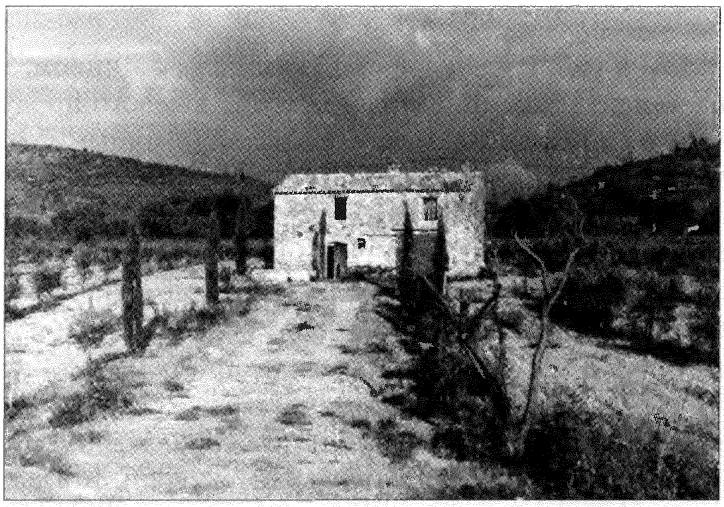
Вместо генерального плана Александер и его коллеги отстаивали такой подход, как *набор принципов*, действующий в отношении всех членов сообщества и применяемый при любом мелком, локальном акте застройки, чтобы в результате возник "органичный порядок", хорошо приспособленный к реальным обстоятельствам.

Заключение

Работать над интересным проектом, экспериментировать с новыми идеями и методами — такой опыт всегда приносит большое удовлетворение. И все же я считаю его бесплодным, если получающееся в результате программное обеспечение не находит практического и эффективного применения. Реальная проверка успеха программы — это проверка временем, т.е. ее служба на протяжении некоторого периода времени. И мне довелось наблюдать за "судьбой" некоторых моих прошлых проектов в течение многих лет.

Здесь будут рассмотрены пять таких проектов. В каждом из них предпринималась серьезная попытка предметно-ориентированного проектирования (*domain-driven design*), пусть даже несистематическая и, конечно, не под этим названием. Во всех этих проектах были разработаны программы; только в некоторых удалось продержаться до конца и построить архитектуру по модели.

Программа автоматизированного проектирования печатных плат из главы 1 имела оглушительный успех у бета-пользователей в этой области. К сожалению, инновационная компания, которая запустила этот проект, потерпела полное поражение в своей рыночной деятельности и была в конце концов ликвидирована. А сама программа в настоящее время используется горсткой инженеров-схемотехников, у которых сохранились старые копии от бета-тестирования. Как и всякое оставленное без сопровождения программное обеспечение, она будет работать до тех пор, пока в одну из программ, с которыми она интегрирована, не внесут какое-нибудь роковое для нее изменение.

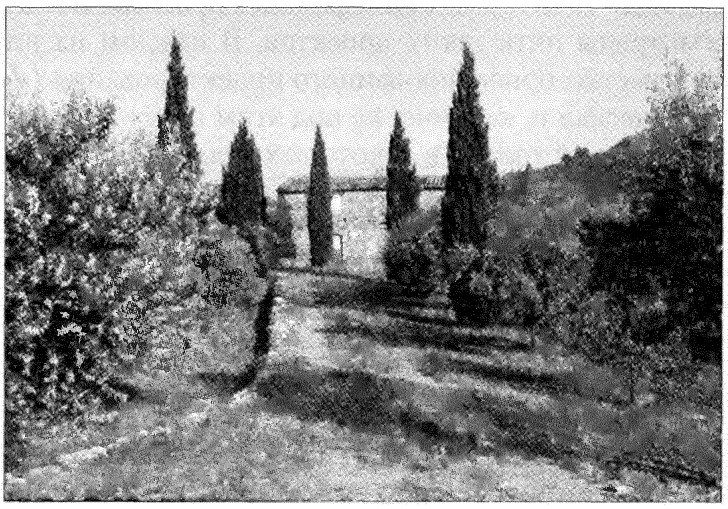


Недавно посаженные оливковые деревья

Программа по управлению кредитованием из главы 9 существовала и развивалась примерно по тому же пути около трех лет после качественного скачка, о котором я писал. Затем в какой-то момент этот программный проект отделился в виде независимой компании. В суматохе реорганизации из проекта "выжили" человека, который руководил им с самого начала, и многие ключевые разработчики ушли вместе с ним. Новая группа раз-

работчиков придерживалась несколько другого подхода к проектированию, не столь тесно привязанного к идеям объектного моделирования. Но они все же сохранили четко выраженный уровень предметной области с реализацией сложных операций и продолжали поощрять приобретение разработчиками знаний о предметной области. Через семь лет после отделения проекта программа продолжает "обогащаться" новыми функциями. Это "лидирующее приложение" в своей области, обслуживающее все большее количество организаций-клиентов и дающее основной поток прибыли компании.

Пока предметно-ориентированный подход не приобрел более широкой популярности, интересные программы во многих проектах разрабатываются за довольно короткие промежутки времени с ударной производительностью труда. Впоследствии такой проект превращается в нечто более-менее обыкновенное, где нет возможности в полной мере использовать силу углубленных и дистиллированных моделей, не говоря уже о совершенствовании в этом направлении. Но, хотя я бы желал от этих проектов большего, на самом деле они оказались очень успешными и приносят практическую пользу в течение многих лет.



Семь лет спустя

Над одним проектом я работал в паре с другим разработчиком; мы писали утилиту, которая была нужна клиенту для производства его ключевого продукта. Функции этой утилиты были довольно сложными и хитро скомбинированными. Мне нравилось работать над проектом; мы разработали гибкую архитектуру с АБСТРАКТНЫМ ЯДРОМ (ABSTRACT CORE). Когда программу передали заказчику, на этом всякое участие разработчиков в ее судьбе закончилось. По причине такого резкого разрыва я полагал, что специфические свойства архитектуры, позволявшие комбинировать элементы программы, могут оказаться малопонятными, и их заменят более типичной вариантной логикой. Но вначале этого не случилось. Когда мы сдавали программу клиенту, в пакет входил полный набор тестов и документ по дистилляции. Новые разработчики воспользовались им, чтобы сориентироваться в своих поисках, и по мере освоения продукта им все больше нравились возможности, предоставляемые архитектурой. Когда я услышал их замечания год спустя, я понял, что в новой группе сохранился и продолжил развиваться наш прежний единый язык (UBIQUITOUS LANGUAGE).

Затем, еще через год, я услышал уже другую историю. Разработчики столкнулись с новыми требованиями, возможности удовлетворить которые в рамках унаследованной архитектуры они не нашли. Им пришлось изменить архитектуру практически до неузнаваемости. Выяснив некоторые подробности, я понял, что "изящное" решение возникших задач было бы невозможно из-за некоторых аспектов нашей модели. Именно в такие мо-

менты становится возможным качественный скачок к более углубленной модели — особенно тогда, когда, как в данном случае, разработчики накопили глубокие знания и понимание предметной области — в результате они изменили и модель, и архитектуру.

Мне эту историю рассказывали осторожно, дипломатично, ожидая, как мне кажется, разочарования с моей стороны из-за того, что столь большая часть моей работы оказалась "отброшена". Но я не слишком привязан к своим архитектурным произведениям. Успешная архитектура не обязана сохраняться вечно. Постройте систему, поставьте людей в зависимость от нее, сделайте ее непрозрачной, и она будет существовать вечно, как неприкосновенное наследие прошлого. Углубленная же модель допускает ясное видение, дающее новое знание, а гибкая архитектура помогает в изменениях. Модель, которую они в итоге построили, оказалась глубже, лучше приспособлена к реальным запросам пользователей. Их архитектура решала реальные проблемы. Программному обеспечению свойственно изменяться, и эта программа продолжала развиваться.

Примеры, посвященные управлению поставками/перевозками и разбросанные по всей книге, более или менее основаны на проекте для большой международной компании, занимающейся контейнерными перевозками. С самого начала руководство проекта выбрало за основу предметно-ориентированный подход, но так и не развило культуру программирования, которая послужила бы ему "питательной средой". За разработку модулей взялось сразу несколько групп программистов весьма разной квалификации в области проектирования и объектного моделирования, координирование же их осуществлялось только неформальным общением между руководителями групп, а также архитектурной группой, ориентированной на заказчика. И нам удалось разработать достаточно глубокую модель СМЫСЛОВОГО ЯДРА (CORE DOMAIN) предметной области, а также работоспособный единый язык (UBIQUITOUS LANGUAGE).

Но культура, принятая в организации, яростно сопротивлялась итерационному процессу разработки, и мы слишком задержались с выпуском работающей внутренней версии. Поэтому проблемы стали видны только на позднем этапе, когда исправлять их уже было рискованно и дорого. В какой-то момент мы обнаружили в модели специфические аспекты, тормозившие быстродействие в базе данных. В ПРОЕКТИРОВАНИИ ПО МОДЕЛИ (MODEL-DRIVEN DESIGN) совершенно естественно двигаться от проблем в реализации к изменениям в модели, но к тому времени у нас было чувство, что мы уже слишком далеко зашли, чтобы вносить изменения в фундаментальную модель. Вместо этого изменения вносились в код, чтобы повысить его эффективность, и связь кода с моделью все ослаблялась. Первая версия приложения также выявила наличие ограничений на масштабируемость в технической инфраструктуре, которые повергли руководство в панику. Для решения инфраструктурных проблем были привлечены специалисты, и проект несколько "выровнялся". Но цикл реализации и моделирования предметной области так и не стал замкнутым.

Некоторые группы разработали прекрасные приложения со сложными функциями и выразительными моделями. Другие произвели на свет нечто жесткое и застывшее, в котором модель была сведена к структурам данных, хотя даже в этом случае какие-то следы ЕДИНОГО ЯЗЫКА сохранялись. Возможно, КАРТА КОНТЕКСТОВ (CONTEXT MAP) могла бы оказать решающую помощь, поскольку конечные продукты разных групп находились в довольно хаотическом отношении друг с другом. И все же модель ЯДРА, заключенная в ЕДИНОМ ЯЗЫКЕ, помогла разработчикам в конечном счете "склеить" систему воедино.

Хотя и реализованный в неполном объеме, проект все-таки заменил собой несколько устаревших систем. Одно целое удерживалось вместе благодаря наличию общего набора понятий, хотя большую часть архитектуры нельзя было назвать достаточно гибкой. К настоящему времени, многие годы спустя, это приложение частично и само устарело, хотя

до сих пор обслуживает потребности глобального бизнеса. Хотя влияние более успешных групп разработчиков неизбежно распространяется со временем, как раз времени-то может и не хватить, даже в самой процветающей компании. В культуру проекта ПРОЕКТИРОВАНИЕ ПО МОДЕЛИ так и не вошло окончательно. Новые разработки теперь ведутся на других платформах, и наша прежняя работа влияет на них только косвенным образом — тогда, когда новые разработчики должны приспосабливаться к старым компонентам системы.

В некоторых кругах относятся подозрительно к амбициозным целям наподобие тех, которые изначально поставила себе компания по перевозкам. Есть мнение, что лучше писать небольшие программы, с которыми можно легко управиться, лучше придерживаться наименьшего общего знаменателя в архитектуре и делать все просто. Что ж, этот консервативный подход имеет право на существование и удобен при реализации срочных проектов с четко очерченной областью применения. Но интегрированные, основанные на сложных моделях, системы обещают то, на что эти быстрые разработки не способны. Существует и третий путь. Предметно-ориентированное проектирование (DDD) позволяет расширять большие системы с богатыми функциональными возможностями поэтапно, постепенно развивая углубленную модель и гибкую архитектуру.

Я завершу свой разговор упоминанием о компании Evant, которая разрабатывает программы для управления материальными запасами. Я играл в их проекте второстепенную, вспомогательную роль, всего лишь делая свой скромный вклад в уже и без того высокую культуру проектирования. Об этом проекте писали как о “рекламном лице” экстремального программирования; при этом обычно не отмечалось, что в проекте была очень сильная предметно-ориентированная составляющая. Дистиллировались все более глубокие модели, они находили свое выражение во все более гибких архитектурах. Проект хорошо развивался до краха “доткомов” в 2001 году. После этого из-за постоянной нехватки капиталовложений начались сокращения, разработка программ замерла, и казалось, что уже близок конец. Но летом 2002 года на Evant вышла одна из десяти крупнейших в мире компаний розничной торговли. Этому потенциальному клиенту нравился сам программный продукт, но в нем требовалось доработать архитектуру, чтобы иметь возможность масштабирования на огромную операцию по планированию товарно-материальных запасов. Это был последний шанс для Evant.

Группа разработчиков, хоть и сократившаяся до четырех человек, все еще имела кое-что в запасе. У них была квалификация, знание предметной области, а у одного из членов группы — еще и опыт по части масштабирования. У них была очень эффективная культура программирования. А еще у них была база кода с гибкой архитектурой, которая облегчала внесение изменений. Тем летом эти четверо разработчиков предприняли героические усилия, получив в результате возможность манипулировать миллиардами единиц планирования и сотнями пользователей. Благодаря силе этих возможностей фирма Evant заполучила в клиенты настоящего “монстра”, а чуть позже была куплена другой фирмой, желавшей приспособить их программы и проверенные возможности к новым требованиям.

Пи всем этом культура предметно-ориентированного программирования (равно как и культура экстремального программирования) пережила эпоху перемен и возродилась вновь. В настоящее время и модель, и архитектура продолжают эволюционировать. Кстати, разработчики Evant не ассимилировались в компании-покупателе, а, наоборот, стимулировали ее группы разработчиков следовать своему примеру. Так что их история продолжается.

Ни в одном проекте никогда не будут применены сразу все методы, описанные в этой книге. Но даже при этом любой проект, в основе которого лежит DDD, можно узнать сразу по нескольким признакам. Его определяющая характеристика это акцент на изу-

чение и понимание предметной области с последующим внедрением полученных знаний в приложение. Все остальное проистекает отсюда. Все члены рабочей группы сознательно пользуются языком проекта и культивируют его усовершенствование. Их трудно удовлетворить качеством модели предметной области, потому что они все время узнают из этой области что-то новое. Они считают непрерывное усовершенствование модели вполне нормальным, а плохо "подогнанную" модель — опасной. Они всерьез относятся к навыкам проектирования, потому что не так-то легко разработать программное обеспечение прикладного профессионального уровня, которое бы четко отражало предметную область. Они преодолевают препятствия и идут дальше.

Взгляд в будущее

Климат, экосистемы, биология в недавнем прошлом считались хаотическими, неточными областями знания в противоположность таким наукам, как физика или химия. Однако в последнее время люди осознали, что мнимая "хаотичность" на самом деле предоставляет широчайшие перспективы для обнаружения и понимания порядка в этих очень сложных явлениях. "Сложность" — это понятие, находящееся на переднем крае многих наук. Хотя наиболее интересными и трудными для решения инженеры-программисты всегда считали чисто технологические, системные задачи, предметно-ориентированное проектирование открывает новую область для исследований, которая в сложности ничуть им не уступит. Прикладное программное обеспечение не должно быть нагромождением кое-как связанных вместе компонентов. Превращение сложной области знания в стройную и обозримую программную архитектуру — это главная цель для технически грамотных программистов.

Еще очень далеки мы от тех времен, когда любой неспециалист сможет легко разрабатывать работоспособные программы. Армия программистов с примитивными навыками, конечно, сумеет написать какое-то программное обеспечение, но вряд ли такое, которое спасет компанию в ответственный момент. Разработчикам (утилит, пакетов, библиотек, сред и т.д.) неплохо бы переориентировать свои усилия на новую цель: как помочь талантливым программистам расширить свои возможности и повысить производительность труда. Необходимы четкие методы исследований моделей предметных областей и выражения их в работающих прикладных программах. С нетерпением жду возможности поэкспериментировать с новыми средствами и технологиями, созданными для этой цели.

Но, хотя новые средства разработки и будут оценены по достоинству, нельзя зацикливаться на них и терять из виду простую истину: разработка хороших программ — это дело, требующее обучения и мышления. Для моделирования нужны воображение и самодисциплина. Средства, которые помогают нам думать и не отвлекаться, — это хорошо. А вот попытки автоматизировать то, что должно быть продуктом мышления — наивны и контрпродуктивны.

Имея в руках нынешние средства и технологии разработки, уже можно строить системы значительно более полезные и работоспособные, чем это делается в большинстве нынешних проектов. Можно писать программы, которыми приятно пользоваться и которые приятно совершенствовать, такие, которые способствуют развитию новых возможностей и пополняют интеллектуальный капитал их владельцев.

Использование шаблонов в этой книге

Моя первая "хорошая машина", доставшаяся мне вскоре после окончания колледжа, — восьмилетний "Пежо", который иногда называли французским "Мерседесом". Машина была хорошо сделана, ее было приятно водить, и в свое время она показывала высокую надежность. Но к моменту попадания в мои руки она уже достигла возраста, когда начинаются серьезные проблемы и требуется постоянное техобслуживание и ремонт.

Компания Peugeot имеет долгую историю, она прошла собственный путь эволюции в течение многих десятилетий. В ней пользуются своей собственной механической терминологией, а ее проектные решения весьма необычны; даже распределение функций между частями механизма часто бывает нестандартным. В результате создаются машины, работать с которыми могут только специалисты компании, что для владельца — молодого выпускника колледжа с весьма скромным доходом — может представлять проблему.

Как-то я повез машину к местному механику, чтобы разобраться с утечкой жидкости. Он заглянул под днище и сообщил, что "вытекает масло из небольшой коробки, находящейся примерно на одной трети длины машины, считая сзади; эта коробка, кажется, имеет отношение к распределению торможения между передним и задним мостом". После этого механик предложил поехать на фирменную техстанцию, находившуюся в восьмидесяти километрах. А вот с "Фордом" или "Хондой" справился бы всякий, поэтому эти машины намного удобнее и дешевле содержать, пусть даже в механическом отношении они устроены не менее сложно.

Я любил свою машину, но никогда больше не стал покупать необычных автомобилей. Настал день, когда обнаружилась особенно дорогостоящая проблема, и этого с меня было довольно. Я отвез свою "Пежо" в местную благотворительную организацию, которая принимала автомобили в качестве пожертвований. Затем я купил старую подержанную Honda Civic примерно за ту же сумму, в которую обошелся бы ремонт прежней машины.

Стандартных элементов архитектуры недостаточно для подробной проработки предметных областей, так что всякая модель предметной области и соответствующая программная реализация всегда трудны для понимания. Более того, каждой группе разработчиков приходится заново изобретать велосипед (а также коробку передач и "дворники" лобового стекла). В мире объектно-ориентированного программирования все представляет собой объект, ссылку или сообщение — и это, конечно, полезная абстракция. Но для разумного сужения диапазона проектных решений, а также для экономического анализа модели предметной области этого недостаточно.

Остановиться на утверждении "все есть объект" — это все равно что строителю или архитектору сказать о доме: "все есть комната". В доме есть комната с высоковольтными розетками и раковиной-мойкой, в которой можно готовить еду. Есть маленькая комната на втором этаже, где можно спать. Чтобы полностью описать обычный жилой дом, по-

требуется много страниц. Люди, которые строят дома и живут в них, понимают, что комнаты следуют определенным образцам (шаблонам) со специально придуманными именами, — такими как “кухня”. Используя этот язык, можно обсуждать экономические аспекты проектирования и строительства домов.

Далее, не всякая комбинация функций может оказаться практически полезной. Почему бы не устроить комнату, где можно и мыться, и спать? Вот удобно было бы. Но долгий опыт, перешедший в обычай, тем не менее предписывает отделять “спальни” от “ванных”. Оборудование ванных комнат совместно используются бо́льшим количеством людей, чем обстановка спален. К тому же в ванной требуется максимальное уединение — даже от людей, которые пользуются той же спальней. Для ванных комнат имеются специальные и дорогостоящие инфраструктурные требования. Ванны и унитазы часто ставятся в одном помещении — санузле — именно потому, что для них требуется одна и та же инфраструктура (подвод воды и канализация), и пользуются ими в одинаковом уединении.

Еще одно помещение, к которому выдвигаются особые инфраструктурные требования — это помещение, где готовят пищу, и называется оно “кухня”. В противоположность санузлу, на кухне никакого особого уединения не требуется. Из-за дороговизны оборудования она обычно одна в доме, даже довольно большом. Это свойство отвечает и нашим устоявшимся обычаям совместного приготовления и приема пищи.

Если я говорю, что мне нужен дом с тремя спальнями, двумя санузлами и кухней открытой планировки, я вкладываю в это короткое предложение огромное количество информации, а также избегаю множества дурацких ошибок, — например, установки унитаза рядом с холодильником.

В проектировании чего угодно — домов, автомобилей, гребных лодок или программ — мы следуем образцам-шаблонам, работоспособность которых уже проверена в прошлом, и импровизируем только в заданных пределах. Иногда, конечно, приходится изобретать нечто совершенно новое. Но, основывая стандартные элементы на шаблонных образцах, мы не тратим силы на задачи, решения которых известны, чтобы можно было сосредоточиться на наших необычных потребностях. Кроме того, работа с известными шаблонами помогает избегать проектов настолько необычных и причудливых, что их даже трудно описать.

Проектирование архитектуры предметных областей в программировании пока не столь традиционная область деятельности, как другие виды проектирования, и его разнообразие подчас не позволяет применять в работе настолько конкретные шаблоны, как образцы автомобильных запчастей или комнат дома. Тем не менее и в нем есть потребность пойти дальше утверждения “все есть объект” как минимум до того этапа, когда отличают болты от гаек.

Способ обмена проектными знаниями и их стандартизации был предложен в 1970-е годы группой архитекторов под руководством Кристофера Александера (Cristopher Alexander), см. [2]. Их “язык описания шаблонов” (*pattern language*) свел воедино ряд проверенных и работающих проектных решений распространенных задач (и гораздо более тонко, чем мой пример с кухней, от которого некоторые читатели Александера, скорее всего, поморщились бы). Цель создания этого языка состояла в том, чтобы строители и пользователи общались на нем между собой, руководствуясь шаблонами-образцами для создания красивых зданий, которые бы хорошо служили и нравились людям.

Что бы там сами архитекторы ни думали об этой идее, язык описания шаблонов оказал большое влияние на проектирование и разработку программных продуктов. В 1990-х годах архитектурные шаблоны программирования уже применялись с переменным успехом в разных областях, особенно в детальном рабочем проектировании [14] и при создании технических архитектур [7]. В более поздние времена шаблоны использовались для документирования базовых приемов объектно-ориентированного программирования

[17] и корпоративных архитектур [13], [3]. Язык описания шаблонов в настоящее время является главным методом организации идей в области проектирования программного обеспечения.

Имена шаблонов призваны стать терминами языка группы разработчиков, и в этой книге они используются именно в таком качестве. Если имя шаблона фигурирует в изложении, оно ВЫДЕЛЯЕТСЯ МАЛЫМИ ПРОПИСНЫМИ БУКВАМИ.

Ниже показано, как описания шаблонов отформатированы в этой книге. В конкретных случаях могут быть небольшие вариации, поскольку я отдавал предпочтение ясности и логике изложения, а не жесткой структуре.

Имя шаблона

[Иллюстрация понятия. Иногда визуальная метафора или поясняющий текст.]

[Контекст. Краткое описание связи понятия с другими шаблонами. В некоторых случаях — краткий обзор шаблона.

Впрочем, значительная часть контекста в этой книге излагается во вступлениях к главам и других частях повествования, а не внутри описаний шаблонов.

* * *]

[Обсуждение проблемы.]

Краткое резюме проблемы.

Как задействованные в проблеме факторы приводят к ее решению.

Краткое резюме решения.

Следствия. Соображения по реализации. Примеры.

* * *

Итоговый контекст: краткое объяснение, как от текущего шаблона возможен переход к последующим.

[Обсуждение трудностей реализации. В исходном формате Александера это обсуждение было бы помещено в раздел, где описывается решение проблемы, и часто в этой книге я следую этому принципу. Но для некоторых шаблонов требуется более обстоятельное обсуждение их реализации. Чтобы сократить центральную часть обсуждения до минимума, длинные подробные обсуждения выносятся за пределы описания шаблона.

Туда же часто выносятся длинные примеры, особенно те, в которых используется сразу несколько шаблонов.]

Глоссарий

Здесь приводятся краткие определения некоторых терминов, названий архитектурных шаблонов и других понятий, употребляемых в книге[1].

Агрегат (aggregate). Совокупность ассоциированных друг с другом объектов, воспринимаемых с точки зрения изменения данных как единое целое. Внешние ссылки возможны только на один объект АГРЕГАТА, именуемый корневым (*root*). В границах АГРЕГАТА действует определенный набор правил согласования и единообразия.

Аналитический шаблон (analysis pattern). Группа понятий, в совокупности представляющих некоторое общепринятое построение в прикладном моделировании. Может относиться к одной предметной области или распространяться на несколько [11].

Контрольное утверждение (assertion). Утверждение о правильном состоянии программы в определенной точке, независимо от способа его достижения. Обычно КОНТРОЛЬНОЕ УТВЕРЖДЕНИЕ задает результат операции или инвариант элемента архитектуры.

Ограниченный контекст (bounded context). Область применения конкретной модели с определенными границами. ОГРАНИЧЕННЫЕ КОНТЕКСТЫ дают членам группы разработчиков четкое и общее представление о том, где следует соблюдать согласованность, а где можно работать независимо.

Клиент (client). Элемент программы, который обращается к проектируемому элементу (вызывает его) для использования его функциональных возможностей.

Связность (cohesion). Логическая согласованность и взаимозависимость.

Команда (command), или **модификатор (modifier)**. Операция, вносящая в систему какое-то изменение, например, устанавливающая значение переменной. Операция, создающая намеренный побочный эффект.

Концептуальный контур (conceptual contour). Внутренняя согласованность самой предметной области, которая, будучи отражена в модели, помогает более естественно модифицировать архитектуру.

Контекст (context). Окружение, в котором фигурирует слово или утверждение и которое определяет их смысл. См. **Ограниченный контекст (bounded context)**.

Карта контекстов (context map). Представление ОГРАНИЧЕННЫХ КОНТЕКСТОВ (BOUNDED CONTEXTS), задействованных в проекте, вместе с фактическими связями между ними и их моделями.

Смысловое ядро (core domain). Четко выделяемая часть модели, наиболее существенная для достижения целей пользователя программы, которая придает программе ценность и отличает ее от других.

---

[1] Из соображений удобства поиска по оригинальному термину в глоссарии сохранен алфавитный порядок английского текста. Следует учитывать это при поиске по ссылкам. — *Примеч. перев.*

Declarative design (декларативное проектирование). Форма программирования, в которой точное описание свойств объектов реально служит для управления программой — фактически исполняемая спецификация.

Углубленная модель (deep model). Четкое и выразительное представление основных задач и интересов специалистов в предметной области, а также наиболее существенных знаний о ней. В углубленной модели отбрасываются поверхностные аспекты проблемы и наивные ее интерпретации.

Архитектурный шаблон, или **образец (design pattern)**. Описание взаимодействующих друг с другом объектов и классов, приспособленных к решению общей архитектурной задачи в конкретном контексте [14], с. 3.

Дистилляция (distillation). Процесс разделения компонентов смеси для выделения главного компонента в такой форме, которая делает его важным и ценным. В области проектирования программного обеспечения — абстрагирование ключевых аспектов модели или проведение границ в большой системе с целью выделения ее СМЫСЛОВОГО ЯДРА (CORE DOMAIN).

Предметная область (domain). Отрасль или сфера знаний, влияния, деятельности.

Специалист в предметной области (domain expert). Участник проекта по разработке программного обеспечения, специализирующийся не в программировании, а в той области, в которой программа должна применяться. Специалист — это не просто рядовой пользователь программы, а носитель глубоких знаний о предмете.

Уровень предметной области (domain layer). Та часть архитектуры и программной реализации, которая отвечает за алгоритмизацию предметной области в МНОГОУРОВНЕВОЙ АРХИТЕКТУРЕ (LAYERED ARCHITECTURE). Именно на уровне предметной области реализуется программное выражение модели предметной области.

Сущность (entity). Объект, суть которого определяется не его атрибутами, а непрерывностью существования и идентификации.

Фабрика (factory). Механизм обслуживания клиента, инкапсулирующий сложные алгоритмы создания и абстрагирующий тип создаваемого объекта.

Функция (function). Операция, вычисляющая и возвращающая результат без наблюдаемых побочных эффектов.

Неизменяемый (immutable). Обладающий свойством никогда не изменять своего наблюдаемого состояния после создания.

Неявное понятие (implicit concept). Понятие, необходимое для понимания модели или архитектуры, но не упоминаемое в них.

Информативный интерфейс (intention-revealing interface). Архитектура, в которой имена классов, методов и других элементов четко описывают, для чего их создавал исходный разработчик, а также для чего ими может пользоваться разработчик-клиент.

Инвариант (invariant). КОНТРОЛЬНОЕ УТВЕРЖДЕНИЕ (ASSERTION) относительно некоторого элемента архитектуры, которое должно быть истинным всегда, кроме специфических переходных состояний, таких как процесс выполнения метода или незавершенной транзакции базы данных.

Итерация (iteration). Процесс, в котором программа постепенно совершенствуется маленькими шагами. *Также* один из таких шагов.

Крупномасштабная структура (large-scale structure). Набор высокоуровневых понятий, правил, или того и другого, задающий образец для проектирования всей системы. Язык, позволяющий обсуждать и воспринимать систему крупными блоками.

Многоуровневая архитектура (layered architecture). Способ изоляции друг от друга выполняемых в программной системе задач — в числе прочего, способ выделения уровня предметной области.

Цикл существования (life cycle). Последовательность состояний объекта, которые он может принимать от момента создания до момента удаления — обычно вместе со связями-ограничениями, которые гарантируют его целостность при переходе от одного состояния к другому. Может включать миграцию СУЩНОСТИ (ENTITY) между системами и разными ОГРАНИЧЕННЫМИ КОНТЕКСТАМИ (BOUNDED CONTEXTS).

Модель (model). Система абстракций, которая описывает избранные аспекты предметной области и может использоваться для решения задач, относящихся к этой области.

Проектирование по модели (model-driven design). Проектирование архитектуры, при котором соблюдается максимально точное соответствие между некоторым подмножеством элементов программы и элементами модели. *Также*, процесс совместной разработки модели и ее программной реализации с сохранением такого соответствия между ними.

Парадигма моделирования (modeling paradigm). Оригинальный стиль построения понятий в предметной области в сочетании со средствами создания программных аналогов этих понятий (например, объектно-ориентированное программирование и логическое программирование).

Хранилище (repository). Механизм для инкапсулирования средств хранения, извлечения и поиска объектов в виде некоторой совокупности.

Обязанность (responsibility). Возложенное на кого-либо или что-либо обязательство выполнять определенную задачу или владеть информацией [24].

Служба (service). Операция, предлагаемая в виде интерфейса и находящаяся в модели автономно, без инкапсулируемого состояния.

Побочный эффект (side effect). Любое наблюдаемое изменение состояния в результате некоторой операции — неважно, намеренное или нет, пусть даже от предумышленной модификации данных.

Функция без побочных эффектов (side-effect-free function). См. **Функция (function)**.

Изолированный класс (standalone class). Класс, который можно воспринимать и тестировать без всякой связи с другими классами, если не считать системных примитивов и базовых библиотек.

Без состояния (stateless). Свойство элемента архитектуры, позволяющее клиентскому коду пользоваться его операциями, не обращая внимания на историю изменений. Элемент без состояния может использовать информацию, доступную глобально, и даже изменять ее (т.е. иметь побочные эффекты), но не содержит никакого закрытого состояния, которое бы влияло на его поведение.

Стратегическое проектирование (strategic design). Принятие модельных и архитектурных решений, касающихся крупных частей системы. Такие решения оказывают влияние на весь проект и должны приниматься на уровне всей группы разработчиков в целом.

Гибкая архитектура (supple design). Архитектура программы, которая вкладывает всю силу, заключенную в углубленной модели (*deep model*) в руки разработчика клиентского кода, помогая ему четко и гибко выражать идеи и выдавать устойчивые, ожидаемые результаты. Не менее важно, что *та же самая* модель применяется и для того, чтобы облегчить ее реализатору модификацию или перестройку архитектуры для адаптации новых знаний и выводов.

Единый язык (ubiquitous language). Язык, структурированный в соответствии с моделью предметной области и используемый всеми разработчиками для координации работ группы над программным проектом.

Унификация (unification). Внутренняя согласованность модели, при которой все термины имеют единообразные значения, а в правилах нет противоречий.

Объект-значение (value object). Объект, описывающий некоторую характеристику или атрибут, но не передающий понятий и не имеющий индивидуальности.

Целостный объект-значение (whole value). Объект, моделирующий отдельное, законченное понятие.

Список литературы

[1] Alexander C., Silverstein M., Angel S., Ishikawa S., Abrams D. *The Oregon Experiment*, Oxford University Press, 1975.

[2] Alexander C., Ishikawa S., Silverstein M. *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, 1977.

[3] Alur D., Crupi J., Malks D. *Core J2EE Patterns*, Sun Microsystems Press, 2001.

[4] Beck K. *Smalltalk Best Practice Patterns*, Prentice Hall PTR, 1997.

[5] Beck K. *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2000.

[6] Beck K. *Test-Driven Development: By Example*, Addison-Wesley, 2003.

[7] Buschmann F., Meunier R., Rohnert H., Sommerlad P., Stal M. *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, 1996.

[8] Cockburn A. *Surviving Object-Oriented Projects: A Manager's Guide*, Addison-Wesley, 1998.

[9] Evans E., Fowler M. "Specifications." Труды конференции PLoP-97, 1997.

[10] Fayad M., Johnson R. *Domain-Specific Application Frameworks*, Wiley, 2000.

[11] Fowler M. *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997.

[12] Fowler M. *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[13] Fowler M. *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.

[14] Gamma E., Helm R., Johnson R., Vlissides J. *Design Patterns*, Addison-Wesley, 1995.

[15] Kerievsky J. *Continuous Learning. Extreme Programming Perspectives*, Michele Marchesi et al. Addison-Wesley, 2003.

[16] Kerievsky J. 2003. `http://www.industriallogic.com/xp/refactoring`.

[17] Larman C. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design*, Prentice Hall PTR, 1998.

[18] Merriam-Webster. *Merriam-Webster's Collegiate Dictionary*. Tenth edition. Merriam-Webster, 1993.

[19] Meyer B. *Object-oriented Software Construction*. Prentice Hall PTR, 1988.

[20] Murray-Rust P., Rzepa H., Leach C. *Abstract 40*. Стенд на 210th ACS Meeting, Chicago, 21 августа 1995 г. `http://www.ch.ic.ac.uk/cml/`

[21] Pinker S. *The Language Instinct: How the Mind Creates Language*, HarperCollins, 1994.

[22] Succi G. J., Wells D., Marchesi M., Williams L. *Extreme Programming Perspectives*, Pearson Education, 2002.

[23] Warmer J., Kleppe A. *The Object Constraint Language: Precise Modeling with UML*, Addison-Wesley, 1999.

[24] Wirfs-Brock R., Wilkerson B., Wiener L. *Designing Object-Oriented Software*, Prentice Hall PTR, 1990.

[25] Wirfs-Brock R., McKean A. *Object Design: Roles, Responsibilities, and Collaborations*, Addison-Wesley, 2003.

Фотографии

Все фотографии, приведенные в этой книге, были опубликованы с разрешения владельцев.

Richard A. Paselk, Humboldt State University
"Астролябия" (глава 3, с. 62).

© Безвозмездно/Corbis
"Отпечаток пальца" (глава 5, с. 95), "Автосервис" (глава 5, с. 107), "Автозавод" (глава 6, с. 133), "Библиотекарь" (глава 6, с. 141).

Martine Jousset
"Виноград" (глава 6, с. 124), "Оливковые деревья" (молодые и старые) (заключение, с. 423–424).

Biophoto Associates/Photo Researchers, Inc.
Электронная микрофотография бактерий Oscillatoria (глава 14, с. 298).

Ross J. Venables
"Гребцы" (группа и одиночка) (глава 14, с. 302 и 325).

Photodisc Green/Getty Images
"Бегуны" (глава 14, с. 313), "Ребенок" (глава 14, с. 319).

U.S. National Oceanic and Atmospheric Administration
Великая Китайская стена (глава 14, с. 320).

© 2003 NAMES Project Foundation, Atlanta, Georgia. Фотограф Paul Margolies. `www.aidsquilt.org`
Мозаика памяти жертв СПИДа (глава 16, с. 375)

Предметный указатель

О

П

Р

С

Т

У

Ф

Х

Ц

Ч

Ш

Э

Разработка программного обеспечения

ПРЕДМЕТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ

СТРУКТУРИЗАЦИЯ СЛОЖНЫХ ПРОГРАММНЫХ СИСТЕМ

"Эта книга должна стоять на полке у каждого мыслящего программиста."

Кент Бек (Kent Beck)

"Эрику удалось ухватить суть того, что опытные проектировщики программных объектов всегда знали, но проваливали все попытки донести это знание до своих коллег в смежных областях. Мы охотно делимся отдельными секретами... но никогда не заботились об организации и систематизации принципов построения логической структуры предметной области. Вот почему эта книга так важна."

Кайл Браун (Kyle Brown), автор книги "Enterprise Java Programming with IBM WebSphere"

Мировое сообщество программистов признает, что моделирование предметных областей — ключевой раздел проектирования программного обеспечения. В моделях предметных областей разработчики выражают сложные функции своих программ, реализуя их затем в таком виде, который отвечает реальным потребностям пользователей. Но несмотря на очевидную важность предмета, существует очень мало пособий по эффективному внедрению моделирования предметных областей в практику разработки программ.

Книга Эрика Эванса восполняет этот пробел. Она посвящена не отдельным технологиям, а систематическому предметно-ориентированному подходу. В ней представлен широкий набор приемов и методик, основанных на практическом опыте, и фундаментальных принципов, помогающих в реализации программных проектов из сложных предметных областей. Органично переплетая практику проектирования и реализации программ, эта книга содержит множество фактических примеров, иллюстрирующих применение общих стратегических принципов в реальных программных проектах.

Из книги читатель узнает, как с помощью модели предметной области придать разработке сложной системы нужную направленность и динамику. Выделены основные приемы и образцы-шаблоны, образующие общий язык группы разработчиков. Особо подчеркивается необходимость рефакторинга не только кода, но и модели в его основе, что в сочетании с итерационной agile-методикой приводит к углублению знаний о предметной области и повышению качества взаимодействия между специалистами и программистами. Подход книги строится именно на этом фундаменте, предлагая модели и архитектуры для систем и организаций любой сложности.

В частности, в книге рассматриваются следующие темы:

- Единый язык общения для всей группы разработчиков
- Глубокая связь между моделью и программной реализацией
- Выделение ключевых черт модели
- Управление циклом существования объектов
- Написание легко интегрируемого кода предметной области
- Как сделать сложный код очевидным и предсказуемым
- Формулировка введения в предметную область
- Дистилляция ядра предметной области
- Поиск неявных понятий, скрытых в модели
- Применение аналитических шаблонов
- Архитектурные шаблоны в моделях
- Поддержание целостности больших систем
- Сосуществование нескольких моделей в одном проекте
- Организация систем в соответствии с крупномасштабными структурами
- Качественные скачки в моделях

Имея под рукой эту книгу, разработчики объектно-ориентированных программ, системные аналитики и архитекторы будут всегда располагать набором рекомендаций по организации своего труда, созданию сложных и полезных моделей предметных областей, превращению их в высококачественные программные продукты.

Эрик Эванс является основателем Domain Language — консалтинговой группы, которая помогает различным фирмам строить и развивать программные системы, тесно связанные с их профессиональной деятельностью. Автор работал в качестве архитектора и программиста над большими объектно-ориентированными системами начиная с 1980-х годов. Он также занимается повышением квалификации групп разработчиков в области экстремального программирования.

www.awprofessional.com

www.DomainDrivenDesign.org

Изображение на обложке — Василий Кандинский, "Композиция 8", 1923 г., из коллекции Solomon R. Guggenheim Museum

ВИЛЬЯМС

www.williamspublishing.com

Addison-Wesley
Pearson Education