



Элементарные шаблоны проектирования

Джейсон Мак-Колм Смит

Предисловие Гради Буча

Элементарные шаблоны проектирования

Elemental Design Patterns

Jason McC. Smith

▼ Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Элементарные шаблоны проектирования

Джейсон Мак-Колм Смит



Москва • Санкт-Петербург • Киев
2013

ББК 32.973.26-018.2.75
М15
УДК 681.3.07

Издательский дом “Вильямс”

Зав. редакцией С.Н. Тригуб

Перевод с английского и редакция докт. физ.-мат. наук Д.А. Клюшина

По общим вопросам обращайтесь
в Издательский дом “Вильямс” по адресам:

info@williamspublishing.com, <http://www.williamspublishing.com>

Смит, Джейсон Мак-Колм.

М15 Элементарные шаблоны проектирования. : Пер. с англ. — М. : ООО “И.Д. Вильямс”, 2013. — 304 с.: ил. — Парал. тит. англ.

ISBN 978-5-8459-1818-5 (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 © 2012 by Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form, by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without written permission from the publisher, except for the inclusion of brief quotations in a review.

Russian language edition is published by Williams Publishing House according to the Agreement with R&I Enterprises International, Copyright © 2013.

Научно-популярное издание
Джейсон Мак-Колм Смит
Элементарные шаблоны проектирования

Литературный редактор Л.Н. Красновон

Верстка Л.В. Чернокозинская

Художественный редактор В.Г. Павлютин

Корректор Л.А. Гордиенко

Подписано в печать 07.12.2012. Формат 70x100/16

Гарнитура Garamond. Печать офсетная

Усл. печ. л. 19,0. Уч.-изд. л. 16,0.

Тираж 1000 экз. Заказ № 3445

Первая Академическая типография “Наука”
199034, Санкт-Петербург, 9-я линия, 12/28

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

ISBN 978-5-8459-1818-5 (рус.)

ISBN 978-0-321-71192-2 (англ.)

© Издательский дом “Вильямс”, 2013

© Pearson Education, Inc., 2012

Оглавление

Предисловие	9
Введение	11
Благодарности	13
Об авторе	15
Глава 1. Введение в шаблоны проектирования	17
Глава 2. Коллекция элементарных шаблонов проектирования	29
Глава 3. Описание шаблонов	61
Глава 4. Работа с элементарными шаблонами проектирования	79
Глава 5. Спецификации элементарных шаблонов проектирования	117
Глава 6. Промежуточные композиции шаблонов	215
Глава 7. Композиции шаблонов Gang of Four	241
Приложение A. ρ-исчисление	257
Библиография	291
Предметный указатель	293

Содержание

Предисловие	9
Введение	11
Благодарности.....	13
Об авторе.....	15
Глава 1. Введение в шаблоны проектирования	17
1.1. Размышления о племенах.....	20
1.2. Искусство или наука?.....	24
1.2.1. Механическая точка зрения на шаблоны.....	24
1.2.2. Точка зрения, зависящая от языка программирования.....	25
1.2.3. От мифа к науке	27
Глава 2. Коллекция элементарных шаблонов проектирования	29
2.1. Основы	30
2.2. Где, почему и как	33
2.2.1. Декомпозиция шаблона <i>Decorator</i>	34
2.2.2. Вниз по кроличьей норе	37
2.2.3. Контекст	46
2.2.4. Пространство проекта	48
2.3. Главные элементарные шаблоны проектирования	57
2.4. Заключение.....	58
Глава 3. Описание шаблонов	61
3.1. Основы	61
3.2. Компонент PINbox	65
3.2.1. Свернутый компонент PINbox.....	65
3.2.2. Стандартный компонент PINbox	67
3.2.3. Раскрытый компонент PINbox	70
3.2.4. Стеки компонентов PINbox и кратность.....	72
3.2.5. Пилинг и конденсация	75
3.3. Заключение.....	78
Глава 4. Работа с элементарными шаблонами проектирования	79
4.1. Композиция шаблонов.....	79
4.1.1. Изотопы.....	84
4.2. Воссоздание шаблона <i>Decorator</i>	88
4.3. Рефакторинг	96

4.4. Общая картина	105
4.5. Зачем читать приложение	110
4.6. Расширенные возможности	111
4.6.1. Специализированная документация и обучение	111
4.6.2. Метрики	112
4.6.3. Процедурный анализ	114
4.7. Выводы	115
Глава 5. Спецификации элементарных шаблонов проектирования	117
Create Object	119
Retrieve	127
Inheritance Relation	131
Abstract Interface	139
Delegation Behavioral	144
Redirection	149
Conglomeration	155
Recursion	160
Extend Method	174
Delegated Conglomeration	179
Redirected Recursion	184
Redirected Recursion	190
Trusted Redirection	196
Deputized Delegation	202
Deputized Redirection	208
Глава 6. Промежуточные композиции шаблонов	215
Fulfill Method	217
Retrieve New	221
Retrieve Shared	225
Objectifier	229
Object Recursion	235
Глава 7. Композиции шаблонов Gang of Four	241
7.1. Порождающие шаблоны	242
7.1.1. Шаблон <i>Abstract Factory</i>	242
7.1.2. Шаблон <i>Factory Method</i>	245
7.2. Структурные шаблоны	247
7.2.1. Шаблон <i>Decorator</i>	247
7.2.2. Шаблон <i>Proxy</i>	248
7.3. Поведенческие шаблоны	251
7.3.1. Шаблон <i>Chain of Responsibility</i>	251
7.3.2. Шаблон <i>Template Method</i>	252
7.4. Заключение	255

Приложение А. ρ-исчисление.	257
A.1. Операторы зависимости	258
A.2. Транзитивность и изотопы	260
A.3. Сходство	262
A.4. Формализм элементарных шаблонов проектирования	263
A.5. Правила композиции и редукции	265
A.6. Обозначения и роли в экземпляре шаблона	268
A.7. Определения элементарных шаблонов проектирования	269
A.7.1. Шаблон <i>Create Object</i>	269
A.7.2. Шаблон <i>Retrieve</i>	270
A.7.3. Шаблон <i>Inheritance</i>	271
A.7.4. Шаблон <i>Abstract Interface</i>	272
A.7.5. Шаблон <i>Delegation</i>	272
A.7.6. Шаблон <i>Redirection</i>	273
A.7.7. Шаблон <i>Conglomeration</i>	273
A.7.8. Шаблон <i>Recursion</i>	274
A.7.9. Шаблон <i>Revert Method</i>	274
A.7.10. Шаблон <i>Extend Method</i>	275
A.7.11. Шаблон <i>Delegated Conglomeration</i>	275
A.7.12. Шаблон <i>Redirected Recursion</i>	276
A.7.13. Шаблон <i>Trusted Delegation</i>	276
A.7.14. Шаблон <i>Trusted Redirection</i>	277
A.7.15. Шаблон <i>Deputized Delegation</i>	278
A.7.16. Шаблон <i>Deputized Redirection</i>	279
A.8. Определения промежуточных шаблонов	280
A.8.1. Шаблон <i>Fulfill Method</i>	280
A.8.2. Шаблон <i>Retrieve New</i>	281
A.8.3. Шаблон <i>Retrieve Shared</i>	282
A.8.4. Шаблон <i>Objectifier</i>	283
A.8.5. Шаблон <i>Object Recursion</i>	284
A.9. Шаблоны <i>Gang of Four</i>	285
A.9.1. Шаблон <i>Abstract Factory</i>	285
A.9.2. Шаблон <i>Factory Method</i>	286
A.9.3. Шаблон <i>Decorator</i>	287
A.9.4. Шаблон <i>Proxy</i>	288
A.9.5. Шаблон <i>Chain of Responsibility</i>	289
A.9.6. Шаблон <i>Template Method</i>	290
Библиография	291
Предметный указатель	293

Предисловие

В кинофильме “Космическая одиссея 2001 года” есть прекрасная сцена. Проведя несколько месяцев в одиночестве на покинутом корабле *Discovery* (после лоботомии вышедшего из строя компьютера ЭАЛ), доктор Дэвид Боуман приблизился к монолиту, ведущему его в новый мир. Его последнее сообщение, переданное на Землю, заканчивалось фразой “Он полон звезд!”

Системы, интенсивно использующие программное обеспечение, — это новые миры, которые мы создаем своими умственными усилиями. Мир, который видел Боуман, состоял из атомов и потому был полон звезд, а наши миры созданы из битов... и полны шаблонов проектирования.

Вольно или невольно все хорошо структурированные системы, основанные на интенсивном использовании программного обеспечения, полны шаблонов проектирования. Идентификация шаблонов в системе позволяет повысить уровень абстракции в размышлениях о ней, а внедрение шаблонов в систему позволяет улучшить ее упорядоченность, элегантность и простоту. Мой опыт подсказывает, что появление шаблонов проектирования — одно из самых важных событий в программировании за прошлые два десятилетия.

Я имел удовольствие сотрудничать с Джейсоном, когда он разворачивал свою работу над системой SPQR, и позвольте мне вас уверить, что он внес очень большой вклад в теоретическое освоение и практическое использование шаблонов проектирования. Книга *Элементарные шаблоны проектирования* поможет вам думать о шаблонах по-новому и применять их для улучшения миров программного обеспечения, которые вы создаете и развиваете. Если вы плохо знакомы с шаблонами, примите к сведению, что благодаря этой прекрасной книге вы начнете их изучать. Если же вы имеете опыт работы с шаблонами, то, я полагаю, вы узнаете нечто новое. Я, определенно, узнал много нового.

Гради Буч (Grady Booch)
Почетный сотрудник IBM (IBM Fellow)
Февраль, 2012

Введение

Эта книга представляет собой введение в новую тему — элементарные шаблоны проектирования, являющиеся основой для исследования и применения шаблонов в программной инженерии. Эти шаблоны образуют ткань теории разработки программного обеспечения, но в то же время они очень практичны и прагматичны. Эта книга предназначена как для начинающих программистов, так и для опытных разработчиков. Она поможет студентам освоиться в индустрии программного обеспечения и даст исследователям новую информацию для размышлений.

Короче говоря, эта книга предназначена для *использования*.

Изучив ее, вы овладеете новой коллекцией инструментов, глубже поймете основные концепции программирования, которые будете использовать ежедневно, а также узнаете, как они взаимодействуют между собой, чтобы делать удивительные вещи. Элементарные шаблоны проектирования, или сокращенно EDP (Elemental Design Patterns), — это коллекция фундаментальных идей программирования, которые разработчики обычно используют рефлексорно, почти не задумываясь. В этой книге содержится их явное описание и им присваиваются “официальные” имена, чтобы создать основу для дискуссий, а также для их совместного использования и сравнения преимуществ. Если вы студент, то изучение элементарных шаблонов заменит вам постоянно растущую гору литературы, в которой шаблоны проектирования представлены в виде наборов устрашающих блоков, не допускающих никаких отклонений. Вы получите шанс освоить их систематически, шаг за шагом. Если вы давно занимаетесь разработкой программного обеспечения и знакомы с шаблонами, то сможете по-новому взглянуть на старые подходы и увидеть новые возможности.

Предполагается, что вы немного знакомы с шаблонами как с областью знаний, но не использовали и не изучали их подробно. Для того чтобы начать их освоение, достаточно просто знать, что они существуют, и иметь о них хотя бы приблизительное представление. Книга не требует от читателей подготовки в области теории программирования, умения разрабатывать языки и даже хорошего владения одним из языков программирования. Достаточно иметь желание научиться критически думать о проектировании программного обеспечения. Мы коснемся этих тем только в качестве отправной точки для тех, кто хочет изучать их более глубоко, следуя за приведенными в конце книги библиографическими ссылками.

Для описания небольших примеров используется унифицированный язык моделирования (Unified Modeling Language — UML). Если вы не знаете язык UML, я предлагаю вам прочитать книгу [20] или [33]. Вам нужны базовые знания в области программирования, процедурного или объектно-ориентированного. Знание последнего является

полезным, но не абсолютно необходимым — в этой книге достаточно доступно объясняются основы объектно-ориентированного программирования. Разработчики, имеющие опыт проектирования объектно-ориентированных систем, с удивлением обнаружат новые точки зрения на концепции, которые они освоили давным-давно, и еще больше оценят преимущества объектно-ориентированного проектирования в целом.

Многие программисты представляют себе “сообщество специалистов по шаблонам проектирования” как эзотерическую группу экспертов и не причисляют себя к ним. Формируя новую точку зрения на внутреннюю структуру шаблонов проектирования, эта книга должна убедить *каждого* программиста, что он также является членом сообщества специалистов по шаблонам проектирования, независимо от того, знает он об этом или нет. Каждый отдельно взятый программист использует шаблоны проектирования каждый раз, когда пишет строку программы, даже если он об этом не думает. Однако он, вероятно, не осознает своих возможностей. Шаблоны проектирования представляют собой общее концептуальное пространство, в котором мы реализуем свои электронные идеи, формирующие наш мир. Настало время создать карту ландшафта, в котором мы работаем и играем.

Следуя примеру основополагающей книги GoF¹ [21], я разделил эту книгу на две логические части. В первой части объясняется, зачем написана эта книга и что такое элементарные шаблоны проектирования, откуда они появились и почему так важны. Здесь же объясняются *причины* появления элементарных шаблонов проектирования. Затем предлагается введение в систему диаграмм Pattern Instance Notation, предназначенную для работы с шаблонами на разных уровнях детализации и в разных средах. Наконец рассматриваются способы использования элементарных шаблонов проектирования для создания традиционных шаблонов проектирования, описанных в многочисленных книгах.

Во второй части этой книги приведены описание коллекции шаблонов проектирования, начиная с элементарных, и пример их сочетания для создания шаблонов Intermediate. В заключение представлены некоторые из шаблонов GoF в виде композиции элементарных шаблонов. Коллекция элементарных шаблонов проектирования, описанная здесь, — это всего лишь часть коллекции EDP Catalog, содержащей первую версию формально определенных и описанных базовых шаблонов. Сообщество разработчиков программного обеспечения продолжает уточнять определение существующих и предлагать новые элементарные шаблоны проектирования по мере развития основной концепции. Я надеюсь, что вы присоединитесь к этим попыткам. Добро пожаловать, мы рады вам!

¹ GoF (Gang of Four, “Банда четырех”) — южное название авторов классической книги: Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. *Приемы объектно-ориентированного проектирования. Паттерны проектирования*. — СПб: “Питер”, 2007. — Примеч. ред.

Благодарности

З а выход в свет моей книги я хочу поблагодарить многих людей. Они перечислены здесь в не совсем хронологическом порядке.

Я благодарен Дэвиду Стоттсу (David Stotts), научному руководителю моей докторской диссертации из Университета Северной Каролины (University of North Carolina) из Чапел-Хилла (Chapel Hill), который многие годы наблюдал за рождением и развитием системы SPQR (System for Pattern Query and Recognition — система запросов и распознавания шаблонов) и коллекции EDP. Я также благодарю научный совет, члены которого сочли мое исследование интересным и позволили мне достичь успеха, хотя и думали, что оно почти неосуществимо: Яна Принса (Jan Prins), Дэвида Плейстеда (David Plaisted), Ала Сегарса (Al Segars) и Сида Чаттерджи (Sid Chatterjee). Все они оказывали мне неоценимую помощь в критические моменты.

Вспоминая годы работы в компании IBM Watson Research в Нью-Йорке, снова благодарю Сида Чаттерджи, убедившего меня связаться с Голубым Гигантом, Клея Уильямса (Clay Williams), предоставившего мне свободу для дальнейшего развития моих безумных идей (я до сих пор скучаю по нашим разговорам за чашкой кофе), Петера Сантанама (Peter Santhanam), отстаивавшего эти идеи и научившего меня больше ценить унаследованные системы, Брента Хайльперна (Brent Hailpern), человеколюбивого знатока черных анекдотов из корпоративной жизни, давшего мне много ценных уроков по менеджменту, Эдит Шонберг (Edith Schonberg), относившуюся к моим затеям более терпимо, чем любой другой менеджер, и многих других людей, слушавших мои маниакальные рассказы о моей работе при каждой встрече. Я очень скучаю по всем вам, мои друзья.

Особой благодарности заслуживает еще один сотрудник IBM — Гради Буч (Grady Booch), принявший меня под свое крыло и подвергавший меня безумным перегрузкам, которые я ни на что не променял. Гради, я безмерно благодарен за Ваше руководство, воспитание и защиту и надеюсь на будущее сотрудничество и продолжение дружбы.

Я хотел бы поблагодарить за понимание и поддержку каждого сотрудника компании Software Revolution, Inc., из Киркланда, шт. Вашингтон (Kirkland, Washington), в которой сейчас работаю старшим научным сотрудником. Мне действительно приятно работать со всеми вами.

Спасибо моим многочисленным рецензентам за их советы и комментарии, которые были поучительными и полезными. Вы сделали эту книгу намного лучше, и я выражаю свою глубочайшую признательность всем вам: Ли Аккерману (Lee Ackerman), Ларсу Бишопу (Lars Bishop), Роберту Богетти (Robert Bogetti), Роберту Коучу (Robert Couch), Бернарду Фарреллу (Bernard Farrell), Мэри Лу Хайнс Фриттс (Mary Lou Hines Fritts), Гейл Мерфи (Gail Murphy), Джейффи Овербею (Jeffrey Overbey), Этану Робертсу (Ethan

Roberts), Карлоте Сэйдж (Carlota Sage), Дэви Свейсу (Davie Sweis), Пети Тэрр (Peri Tarr) и Ребекке Вирфс-Брок (Rebecca Wirs-Brock). Элизабет Райан (Elizabeth Ryan), Раина Хробак (Raina Chrobak), Крис Зан (Chris Zahn) и Крис Гузиковски (Chris Guzikowski) из издательства Addison-Wesley продемонстрировали образцовое понимание и поддержку в процессе подготовки книги. Большое спасибо вам и остальным сотрудникам издательства, в частности Кэрол Лалльер (Carol Lallier), помощь которой в шлифовке текста просто неоценима. Я также благодарю своих друзей и семью, проявивших невероятное терпение, пока я проводил бесконечные часы в работе над этой книгой, несмотря на то что после возвращения в Сиэттл они надеялись меня видеть чаще.

В заключение я хотел бы упомянуть мою жену Леа (Leah). Она очень сильно поддерживала меня все это время. Она посвятила мне много времени, терпения и любви. Я переполнен любовью и благодарностью к ней. Спасибо. Словами этого просто не передать.

Спасибо всем. Каждый из упомянутых людей внес свой вклад в уточнение идей и улучшение текста. Если сравнить эту книгу с ребенком, то у нее много повивальных бабок.

Джейсон Мак-Колм Смит (Jason McC. Smith)

Сиэттл

4 сентября 2011 года

Об авторе

Джейсон Мак-Колм Смит получил докторскую степень по компьютерным наукам в 2005 году в Университете Северной Каролины в Чапел-Хилле. (Элементарные шаблоны проектирования появились как часть проекта по созданию системы запросов и распознавания шаблонов.) Доктор Смит получил два патента США на разработки, выполненные в университете UNC-CH: один — на технологии, связанные с системой SPQR, а второй — на систему распределенной совместной работы над документами FaceTop.

До этого доктор Смит много лет работал инженером по физическому моделированию и консультантом, начав с получения двух дипломов бакалавра по физике и математике в Университете штата Вашингтон. Среди заслуживающих внимания проектов следует упомянуть моделирование распространения звука в океане, электронных процессов, полетов гражданской и военной авиации, а также графические обучающие системы реального времени.

Четыре года работы в компании IBM Watson Research позволили доктору Смиту применить опыт работы над системой SPQR и каталогом EDP, а также композиционный подход к их использованию в программном обеспечении, как унаследованном, так и современном.

В настоящее время доктор Смит является старшим научным сотрудником в компании Software Revolution, Inc., в Киркланде, шт. Вашингтон, где продолжает уточнять каталог EDP и искать способы повышения эффективности работы компании за счет автоматизированной модернизации и трансформации унаследованных систем.

Введение

шаблоны

проектирования

Шаблоны проектирования — это одно из самых выдающихся достижений программной инженерии по любым меркам. Тем не менее история шаблонов проектирования довольно странная, и со временем их исходная польза и элегантность были забыты, недооценены или просто неправильно поняты. Эта книга может заполнить возможные пробелы знаний у читателей, имеющих опыт работы с шаблонами проектирования, и позволит студентам постепенно ими овладеть. Литература по шаблонам проектирования представляет собой коллекцию довольно больших блоков информации разной степени доступности. Данная книга позволит практикам, знакомым с методологией шаблонов проектирования, объединить эти блоки в более крупную логическую систему, а студентам, только начинающим изучение шаблонов проектирования, предоставит возможность изучить основные принципы и освоить тему шаг за шагом. Элементарные шаблоны проектирования действительно элементарны и составляют основу для изучения шаблонов проектирования как научной дисциплины.

Коллективный разум сообщества разработчиков программного обеспечения является одной из самых больших ценностей, и мы по-прежнему можем научиться друг у друга. Книга и исследование, на котором она основана, представляют собой попытку восполнить то, что мы потеряли в ходе эволюции шаблонов проектирования. Наша цель — вспомнить исходное предназначение шаблонов проектирования, создать более широкую платформу для дискуссий о шаблонах и углубить понимание программного

обеспечения, которое мы создаем и используем. Наше сообщество разрабатывало шаблоны проектирования “в ширину”, забыв о глубине. Иначе говоря, мы знаем многое о многом, но не умеем объединить все это в нечто целостное. Это напоминает мне переход от алхимии к химии. Пока не появилась периодическая таблица элементов, коллективный разум многих талантливых исследователей был силен, но слабо согласован. Вероятно, самым крупным следствием открытия Дмитрием Менделеевым периодической таблицы был не столько способ, позволивший химикам распознать систему среди строительных блоков вещества, а возможность использования этих элементов для предсказания свойств еще неоткрытых элементов. Первыми примерами стали галлий и германий, химические и физические свойства которых Менделеев точно описал еще до их открытия. Периодическая таблица перевела химию из категории описательных дисциплин в категорию прогнозных наук.

Появление шаблонов проектирования в среде разработчиков программного обеспечения связано с появлением в 1995 году основополагающей книги *Design Patterns: Elements of Reusable Object-Oriented Software*. “Банда четырех” (Gang of Four — GoF), состоявшая из Эриха Гаммы (Erich Gamma), Ричарда Хелма (Richard Helm), Ральфа Джонсона (Ralph Johnson) и Джона Влиссидеса (John Vlissides), собрала разнообразные изобретения, циркулировавшие в научных и академических кругах с момента выхода в свет докторской диссертации Гаммы в 1991 году. В этой работе в значительной степени использованы идеи Кристофера Александера (Christopher Alexander), опубликованные им в 1960-х годах. Александр был строителем и архитектором, и его работа была посвящена поиску шаблонных решений, которые можно было бы применять в конкретном контексте. Его отправной точкой была идея о том, что в архитектуре существует два типа шаблонов, которые он назвал *неосознанными* (*unselfconscious*) и *осознанными* (*selfconscious*).

Неосознанное проектирование характерно для так называемых примитивных культур: конструкции домов копируются точно и постоянно, а обучение сводится к безукирзенному и адекватному воспроизведению конкретного образца. Конструкция изменяется редко, а приверженность конкретной форме считается самоцелью, в основном потому, что такая конструкция представляет собой результат тысячелетнего опыта. Основной принцип этого вида проектирования — от добра добра не ищут. Поскольку проблема строительства жилья является универсальной, разные условия жизни, например дожди, пустыня, лед, болото или лес, привели к появлению разных стилей и конструкций, но в конкретном контексте приемлемым считается только одно-единственное решение, которое, впрочем, часто оказывается невероятно эффективным в специфичной природной среде. Тем не менее проект применяется без учета индивидуальных особенностей или проявления свободы выбора.

Осознанное проектирование является более современным изобретением; проектировщик свободен принимать осознанное решение, которое практически всегда зависит от его стиля, эстетических взглядов и материалов. Эту архитектурную свободу можно увидеть во многих произведениях современной архитектуры в любом городе или поселке. Даже на вашей улице вы можете увидеть множество стилей и характерных выкрутасов, являющихся следствием сознательного решения архитектора. Современные проектировщики имеют широкий выбор стилей, и, вообще говоря, единственной проблемой является согласование

их эстетических предпочтений и сметы. Конечно, дома соответствуют основным критериям жилья, но это далеко не единственный фактор, влияющий на их архитектуру. Если архитектор имеет свободу выбора, ему труднее выбрать эффективное решение среди практически бесконечного количества неприемлемых и просто плохих вариантов. Для того чтобы как-то ограничить возможность выбора плохих решений при строительстве жилья, были приняты строительные нормы, но даже они не облегчили задачу. Простое чтение строительных норм и следование им не гарантируют эффективности архитектуры. Строительные нормы носят обобщенный характер, а хорошая архитектура учитывает особенности окружающей среды на любом уровне детализации: от глобального климата и региональной погоды до свойств почвы и растительности на участке.

Результаты осознанного проектирования можно увидеть в любом городе. Один дом может быть построен в георгианском стиле, другой — в псевдовикторианском, третий может состоять из стекла и стали, а четвертый представлять собой двухуровневое ранчо или вообще смешение всех стилей архитектуры и типов конструкций и материалов. Тем не менее мы должны спросить себя, является ли данная конструкция оптимальной или хотя бы достаточно эффективной с учетом конкретных природных условий в данной местности. Например, город Остин в штате Техас — не самое лучшее место для строительства открытых стеклянных сооружений, поскольку летом здесь очень солнечно и на охлаждение зданий придется затратить дополнительные средства. В то же время Нью-Йорк, возможно, не подходит для зданий с плоской крышей, потому что зимой здесь выпадает много снега, создающего дополнительную нагрузку на перекрытия. Окружающая среда и факторы, обуславливающие конкретную ситуацию, в которой необходимо принять проектное решение, часто игнорируются, а принимаемые решения часто лишь в минимальной степени соответствуют критериям эффективности или даже создают дополнительные проблемы.

Должно быть очевидным, как все сказанное относится и к разработке программного обеспечения: мы можем сделать практически все, что придет в голову, и даже больше, чем в архитектуре реальных зданий. Программирование обладает удивительной мощью, но у него есть и ахиллесова пятка. Мы можем сделать почти все что угодно, и обычно нам это удается, но, к сожалению, набор действительно полезных вещей среди всего этого разнообразия достаточно мал, и наши проекты часто не выполняются вовремя, выходят за рамки бюджета, с треском проваливаются или тихо умирают. Мы редко заканчиваем проекты с чувством выполненного долга, намного чаще нам кажется, что мы избежали опасности. Это повторяется снова и снова. Почему? Ведь мы десятилетиями накапливали коллективный опыт и тратили миллионы человеко-часов работы. Почему же, решая очередную задачу, мы каждый раз буксуем, сталкиваясь с новой проблемой? Некоторые проектировщики и разработчики обладают феноменальной способностью не замечать сложности и находить ключевое звено, обеспечивающее эффективность проекта. Остальные постоянно застревают между неосознанным “как меня учили” и парализующим волю осознанным проектированием.

Работа Александера была попыткой избежать этой проблемы в архитектуре и строительстве и продемонстрировать разницу между эффективностью примитивных культур и почти полной бесконтрольностью современной архитектуры, основанной на принципе

“делай, что хочешь”. Точка равновесия находится где-то между этими крайностями. Нам нужно определить базовые принципы и общие решения, существующие в неосознанной архитектуре, и описать их так, чтобы их можно было осознанно применять в широком спектре контекстов. Необходимо выкристаллизовать здравое зерно из разных решений, достигнутых методом проб и ошибок, и сформулировать концепции, которые кто угодно может изучить, применить в своих проектах и использовать как руководство к действию при проектировании.

Именно для этого необходимы шаблоны проектирования — чтобы кристаллизовать опыт огромного и разумного сообщества. Это краудсорсинг¹ в его наилучшем проявлении. За годы, прошедшие с момента появления книги GoF, общество специалистов по шаблонам выросло, стало крупным и энергичным и достигло больших результатов. Гради Буч (Grady Booch) и Селсо Гонсалес (Celso Gonzalez) опубликовали на своем веб-сайте все шаблоны, которые они смогли найти в индустрии и академической среде [11]. К данному моменту они собрали около двух тысяч шаблонов. Количество проектов, выполненных сообществом, огромно, и хотя об их качестве можно поспорить, более важным сейчас является задача их систематизации.

Даже в полностью проиндексированной, хорошо организованной коллекции качественных шаблонов проектирования очень трудно организовать быстрый и точный поиск информации — ее слишком много. Что еще хуже, студенту, желающему освоить принципы правильного проектирования, чрезвычайно сложно найти подходящий образец. Это все равно что изучать математические основы аeronautики, наблюдая за полетом самолета. Опытный практик, убежденный, что он изобрел новый шаблон проектирования, не может сравнить его с существующими шаблонами, и нет никакого способа создать инструмент, который мог бы ему помочь.

Сообществу разработчиков программного обеспечения необходимо хорошоенько разобраться в том, чем они владеют, выработать методологию, объясняющую, как более точно описать существующие шаблоны проектирования с помощью компонентов и четких принципов, и сделать их доступными для студентов и новых разработчиков. Нам необходимы базовые принципы для литературы по шаблонам проектирования, чтобы их можно было лучше понимать, объяснять и изучать. Эта книга представляет собой основу для разработки этих принципов.

1.1. Размышления о племенах

Эффективность, обеспечиваемая документированием и передачей передового опыта, важна, но причина, по которой нужно использовать такой способ, часто игнорируется нашим сообществом. Говоря прямо, все мы смертны и наши лучшие годы позади. Мы уже потеряли множество светил, заложивших основы нашей индустрии, и вскоре потеряем еще больше. Совершенно очевидно, что мы не готовы к превращению разработки программного обеспечения в научную дисциплину.

¹ Краудсорсинг (от англ. crowdsourcing) — вовлечение большой группы людей в совместную деятельность на основе добровольного участия без заключения договоров (в отличие от аутсорсинга). — Примеч. ред.

Что еще хуже, программное обеспечение функционирует намного дольше, чем ожидали его разработчики. Язык COBOL до сих пор должен поддерживаться всеми коммерческими системами во всем мире. Язык Fortran по-прежнему используется для научных вычислений. В микросхемах современных высокоеффективных компьютерных систем “защит” код, который был написан на ассемблере или языке С тридцать и более лет назад. Вы можете быть почти абсолютно уверены, что среди миллионов строк программного обеспечения, поставленного вместе с новейшим персональным компьютером, найдется фрагмент программы, который уже не понимает никто из живущих на Земле людей.

Мы знаем, что обязаны документировать свое программное обеспечение; мы знаем, что должны его обновлять; мы знаем, что необходимо записывать или рисовать “что”, “как” и “почему”; но мы также знаем, как это неприятно. Это действительно неприятно, поэтому мы этого не делаем. Вместо этого мы накапливаем знания, которые скрыты в головах разработчиков, передаются в авральном порядке, по просьбе и только при необходимости, часто не систематично.

Для такой информации Гради Буч предложил очень точное название — “племенное знание” (“tribal knowledge”) [10]. У этого факта есть несколько неприятных последствий. Культуры, основанные только на устной традиции передачи знаний, не способны обеспечить полную и точную передачу информации, и это при том, что традиции передачи информации являются *строгими*. Культуры, обладающие слабой дисциплиной в отношении достоверности и точности передачи информации, остаются уязвимыми для искаений. Тем не менее строгие устные традиции могут приводить к самым разным результатам.

Сообщество разработчиков также имеет устную традицию передачи информации. Хотя мы в состоянии записать фрагменты программ, которые понимаем, мы часто не выражаем письменно всей полноты своего понимания и не синхронизируем документацию с эволюцией своих систем. Этот провал в документации является повсеместным, и только в ходе расспросов мы можем надеяться заполнить эти пробелы и понять, почему конкретная система работает так, а не иначе.

Честно говоря, это не всегда выглядит так уж плохо. Методология гибкой разработки программного обеспечения отдает предпочтение работающим, а не документированным программам, и против этой точки зрения трудно возразить. Конечно, это до поры до времени. Гибкие системы обладают забавным свойством со временем становиться устаревшими и требовать для поддержания своей работы все более и более крупных коллективов программистов. В конце концов успешный код, начинавшийся как ускоренная разработка, столкнется с множеством проблем, характерных для традиционно разрабатывавшихся систем. Разработчики уходят. Документация устаревает. Знания теряются.

Программное обеспечение, существующее в данный момент, нельзя назвать самодокументированным, а по его исходному коду невозможно понять скрытые причины, по которым нечто в автоматизированной системе было сделано именно так, а не иначе. Это плохо, потому что мало просто иметь программу, нужно ее еще и понимать. Мы хотим иметь обновляемую документацию, когда она нам нужна, но не хотим нагружать себя лишней работой, когда она не нужна. Мы хотим, чтобы наш код был в большей степени самодокументированным или хотя бы автоматически документированным, но

большинство из нас лишено этой роскоши. Итак, мы плывем по течению и надеемся на лучшее. Тем временем наше коллективное понимание системы деградирует. Существующее положение вещей можно назвать очень слабой устной традицией.

В результате племенное знание деградирует до “племенной мифологии”. Теперь на вопрос “Почему?” мы можем ответить лишь “Потому что мы всегда так делали”. Я подозреваю, что, вливаясь в новую команду программистов, вы хотя бы раз уже испытывали моментальную растерянность. Вам, наверное, уже приходилось вести такие разговоры, вероятно, даже чаще, чем вы можете вспомнить.

Племенная мифология — это действие без объяснения причин. Это механическое повторение без понимания. Еще одним признаком племенной мифологии в группе являются фразы “Потому что меня так научили”, “Я не совсем уверен, но Джо говорит, что так надо”, “На этот вопрос могла бы ответить Джейн, но она уволилась в прошлом году, а я просто скопировал то, что было”, “О нет, ничего не изменяйте! Программа перестанет работать, и мы не сможем ее исправить”. Эти фразы отражают непонимание причин действия или по крайней мере нежелание или неумение объяснить эти причины. Со временем этот недостаток понимания породит большую неопределенность и страх изменений. К сожалению, такая ситуация наблюдается в большинстве проектов, которые по иронии относятся к индустрии, развивающейся под влиянием инноваций, изменений и усовершенствований.

Тем не менее оборотной стороной племенной мифологии является племенная мудрость. Она предписывает действие *с пониманием*, объясняет, *как и почему*, и способна адаптироваться к новым условиям, ситуациям и задачам. Она выходит за пределы механического копирования и вносит ясность благодаря точному объяснению причин, лежащих в основе действия. В какой-то момент прошлого почти каждое решение и действие системы кто-то мог объяснить. Продвижение этих решений, даже небольших, может оказаться очень важным. Небольшие решения объединяются в большие системы, а из маленьких проектов образуются крупные. Обеспечив строгую традицию сохранения знаний, облегчающих понимание, мы создаем традицию племенной мудрости.

Коллекционирование шаблонов проектирования необходимо именно для создания племенной мудрости. К сожалению, они часто интерпретируются как часть племенной мифологии, т.е. отвечают на вопрос *как*, не объясняя *почему*.

Если на протяжении вашей карьеры вы еще не попадали в такие ситуации, позвольте мне привести еще один иллюстративный пример. Недавно мы с женой купили наш первый дом, следовательно, у нас появился наш первый двор. Местность, в которой мы живем, известна своими дождями, а значит, и мхами. Теперь я люблю мох. Он зеленый, почти не требует ухода и образует на земле прекрасный мягкий зеленый ковер. Он удовлетворяет всем условиям, предъявляемым ко двору, требует меньше ухода, чем трава, но есть одна странность. Одна часть нашего двора сильно затенена, и туда почти не попадают солнечные лучи. На ней растет сплошной мох и не растут трава и другие растения. Там мало света даже для тенелюбивых трав.

Однако на расстоянии двадцати футов с каждой стороны затененного участка солнечный свет падает практически беспрепятственно. Местами там также рос мох, и поначалу мне казалось, что это прекрасно. Мох и трава отлично сосуществовали, мох не вытеснял

траву, а просто заполнял места, где трава была недостаточно сильной. На самых солнечных участках мха почти не было и росло много травы. На промежуточных участках мох и трава мирно сосуществовали. Что может быть лучше?

К сожалению, старожилы этой местности пришли в ужас от нашей ситуации. “Вы должны избавиться от мха вообще!” Когда я спросил, почему, я услышал в ответ “Потому что он не трава”, “Потому что я *так делаю*”, “Потому что это плохо для газона”. К моему удивлению, никто не мог мне объяснить, *почему* я должен выполоть мох. Мне кажется, что если бы выполол *весь* мох везде, не глядя на конкретные условия, я бы получил проплешины, на которых не растет трава. Это совсем не оптимально.

Еще хуже то, что, как и во многих проектах по созданию программного обеспечения, я попал в ситуацию, в которой не понимал, что делали прежние жильцы для ухода за двором и почему. Не было никаких сведений, которые помогли бы мне понять, что я должен делать с газоном и почему газон оказался в таком состоянии. Поэтому я провел несколько экспериментов. В самых тенистых местах я выдернул немного мха и посейал траву. На остальной части двора я оставил мох, чтобы посмотреть, что получится.

Трава, посеянная на самом тенистом участке, вообще не взошла. Применив племенную мифологию, я бы превратил большую часть этого участка в грязь. Откровенно говоря, мох лучше, чем грязь. Он зеленый и сочный и растет без ухода. Для данного участка мох — наилучшее решение.

На остальной части двора, где тени мало или совсем нет, мох и трава росли *вместе*, более или менее мирно сосуществуя. Трава превосходно росла и мох не мог ей помешать непосредственно. К сожалению, у мха есть побочный эффект. На самых солнечных участках он действует как защитный слой для сорняков, которые прорастают снизу, препятствуя птицам и мышам съедать их семена и обеспечивая достаточную влажность. Мох не мог вытеснить траву, а сорняки могли. Оставив мох расти на солнечных участках, я создал рассадник для сорняков, которые проросли сквозь мох к солнцу и быстро распространились. Кроме того, мох образовал защитный влажный слой для корней сорняков, которые могли свободно прорастать под его прикрытием.

На самых затененных участках этой проблемы не было, потому что там просто очень мало солнца как для травы, так и для сорняков, а вот на солнечных участках ситуация стала ужасной. Через несколько месяцев я уже вел войну со сорняками. Мох сам по себе не является проблемой, но может стать причиной крупных неприятностей.

Зато теперь я знаю, *почему* мох нужно удалять с лужайки. Дело не в самом мхе, а в том, что он создает микроклимат, в котором возникает серьезная проблема. По существу, мох создает новые силы, формирующие новый *контекст*. В этом новом контексте возникает новая проблема, например сорняки. Теперь совет удалить мох имеет смысл, по крайней мере для солнечных участков, на которых он порождает проблемы.

Зная причины, я теперь могу изменить точки приложения своих знаний в соответствии с окружающей средой. На солнечных участках я должен удалить мох и предотвратить его появление, чтобы не появились сорняки. На затененных участках этого делать не следует, потому что возникнет другая проблема — грязь вместо травы.

Зная причины, лежащие в основе совета, я могу уточнить решение “удалить мох” в зависимости от контекста (солнечный участок или затененный) и не только решить

мою исходную проблему, но и предотвратить появление новых. То, что сначала было племенной мифологией, стало племенной мудростью, которую можно распространить, уточнить и применить там и тогда, когда это приемлемо. По существу, это начало нового шаблона.

1.2. Искусство или наука?

Несомненно, шаблоны — бурно развивающаяся тема, имеющая огромную практическую важность. Шаблонам посвящаются академические конференции, методы Аккермана (Ackerman) и Гонсалеса (Gonzalez), основанные на шаблонах, по праву стали основой для новой дисциплины [2] и от промышленных консультантов ожидают свободного владения шаблонами и умения создавать диаграммы UML. Существуют инструменты для создания, демонстрации и извлечения шаблонов. Шаблоны стали компонентом программной инженерии.

Мы просто не совсем уверены в том, правильно ли они используются в программной инженерии и как их согласовать друг с другом. Более широкому подходу к шаблонам мешают два обстоятельства, к сожалению, характерные для индустрии программного обеспечения. Первый фактор — интерпретация шаблонов как застывших элементов, которые следует слепо копировать, второй фактор — запутанная языковая реализация шаблонов, создающая их новые варианты.

1.2.1. Механическая точка зрения на шаблоны

Попросите десять разработчиков дать определение шаблона, и вы получите десять разных ответов. Самым традиционным является определение шаблона как “решение повторяющихся проблем в конкретном контексте”, но вы можете также услышать, что шаблон — это “рецепт” или “пример структуры”, что свидетельствует об узкой точке зрения на шаблоны. Шаблоны предназначены для мутации, трансформации и подгонки, чтобы удовлетворять потребности, возникающие в конкретном контексте, но слишком часто разработчики просто копируют и вставляют образцы кода, взятые из текста шаблонов или с веб-сайта, и объявляют результат успешным применением шаблона. Это путь к неудаче, а не рецепт создания хорошего шаблона.

Слепое копирование структуры шаблона, “потому что так сказали авторитеты”, — это возврат к концепции Александера о неосознанном проектировании. Оно противоречит самому предназначению шаблонов проектирования. Работая с шаблоном, необходимо объяснять не только то, *как нужно действовать*, но и *почему*. Мы должны уметь описывать причины, лежащие в основе шаблона, одновременно объясняя, как его реализовать. Без понимания причин, лежащих в основе описания шаблона, механическое использование часто приводит к неправильным результатам. В лучшем случае результатом станет испорченный шаблон, не соответствующий ожиданиям. В худшем случае мы внедрим в систему ятрогенный шаблон, который выглядит, как правильный, но порождает неправильные результаты, для выявления которых могут понадобиться годы. Это не простое неумение достичь цели. В данном случае мы создаем новую проблему, которая может оказаться хуже прежней. Эти шаблоны относятся к племенной мифологии — действие без понимания.

Традиционная форма шаблона проектирования, определенная в книге *Design Patterns* [21], объясняет причины, лежащие в основе шаблона — мотивы, применимость и последствия, — но читатель должен сам выделить базовые понятия, формирующие шаблон. В некоторой степени эти базовые понятия описаны в разделах “Участники” (что такое части) и “Отношения” (как они связаны), но при механической реализации разработчики часто просто воспринимают их как контрольный список частей решения, а не как описание базовых понятий и абстракций, образующих решение.

1.2.2. Точка зрения, зависящая от языка программирования

Спросите разработчика, насколько важны шаблоны для его работы, и его ответ часто будет зависеть от языка реализации, который он использует. И это не удивительно. Разные языки поддерживают разные концепции и способы их выражения. Способы выражения этих концепций становятся причиной жарких споров между приверженцами разных языков, но игнорирование базовых концепций в большинстве случаев делает эти споры бессмысленными. Заключены ли блоки в фигурные скобки, как в языках семейства C, или выделены пробелами, как в языке Python, практически не имеет никакого значения по сравнению с самой концепцией блоков.

Это значит, что некоторые шаблоны в одних языках легче реализовать, чем в других. Некоторые языки могут настолько сильно упростить понятия, лежащие в основе шаблона понятия, что становятся частью самой реализации языка. Хорошим примером является шаблон *Visitor*.² В разделе, посвященном реализации шаблона *Visitor* [21, р. 338], сказано, что “шаблон *Visitor* достигает [своей цели] с помощью приема, называемого двойной диспетчеризацией. Этот прием хорошо известен. Некоторые языки программирования (например, язык CLOS) поддерживают его явно”. Что это значит? Это значит, что упоминание шаблона проектирования *Visitor* в присутствии программистов, работающих на языке CLOS (Common LISP Object System), заставит их почесать затылки. “Шаблон? Для свойства языка? Зачем?” Шаблон *Visitor* фактически встроен в язык CLOS. Вам не нужен шаблон для того, чтобы наилучшим образом выразить данную концепцию, — он уже включен в язык как его базовое свойство. Однако в большинстве других языков программирования шаблон *Visitor* предоставляет широкое пространство для выражения концепции двойной диспетчеризации.

Это иллюстрирует важную мысль. Если вы в разговоре с теми же самыми программистами, работающими на языке CLOS, упомянете двойную диспетчеризацию, а не шаблон *Visitor*, они поймут, что вы имеете в виду, и будут знать, как использовать двойную диспетчеризацию и когда ее не использовать. Терминология, особенно общепринятая, — очень важный фактор.

Это относится ко всем языкам и шаблонам: одни языки облегчают реализацию некоторых шаблонов, а другие усложняют. Однако ни одному языку в данном случае нельзя отдать однозначного предпочтения. Распространенный миф утверждает, что шаблоны проектирования маскируют дефекты языков программирования, но это не так. Шаблоны

² Пока вам знать шаблон *Visitor* необязательно. Я выбрал его только потому, что он является ярким примером.

проектирования описывают полезные концепции независимо от языка, используемого для его реализации. Включен ли шаблон в свойства языка или должен быть реализован отдельно, не имеет значения. Концепция выражается в ее реализации, и это важное обстоятельство позволяет нам говорить о проектировании программного обеспечения независимо от его реализации. Проектирование — это совокупность концепций; способ выражения этих концепций в данном языке программирования — это реализация.

Нет никаких препятствий реализовать каждый шаблон из книги GoF на языке C, но это было бы очень утомительно. Вы должны были бы создать самые эффективные способы связывания данных и функций в осмысленные семантические единицы, инкапсулировать эти данные, обеспечить их доступность и т.д. Эта работа кажется огромной, но эти концепции считались *настолько* важными, что революционизировали свойства языков программирования для облегчения работы с шаблонами проектирования. Эта революция называется *объектно-ориентированным программированием*.

В объектно-ориентированных языках программирования эти концепции включены как первичные свойства языка, называемые классами, областями видимости и конструкторами. И снова обратимся к книге GoF: “Если бы мы ориентировались на процедурные языки, то могли бы включить такие шаблоны проектирования, как *Наследование, Инкапсуляция и Полиморфизм*”. Авторы считали это высказывание настолько важным, что поместили его в раздел 1.1. Этот фундаментальный пункт, похоже, забытое большинство разработчиков, поэтому позвольте его напомнить.

Шаблоны являются концепцией, не зависящей от языков программирования; они принимают форму и становятся конкретными решениями только в ходе реализации с помощью конкретного языка при заданном наборе языковых свойств и конструкций.

Это значит, что было бы странно говорить о “шаблонах проектирования в языке Java”, “шаблонах проектирования в языке C++”, “шаблонах проектирования в языке WebSphere” и т.д., несмотря на то что мы все же так говорим. Это просто разговорное сокращение фраз “шаблоны проектирования, реализованные с помощью языков Java, C++, WebSphere и других, независимо от языка и интерфейса прикладного программирования”.³

К сожалению, если вы прочитали одну из многочисленных книг о шаблонах проектирования, то, возможно, привыкли ошибочно думать, что между шаблонами, реализованными на языках Java, и, например, Smalltalk, существует эфемерная, но фундаментальная разница. На самом деле это не так. Эти концепции совпадают; отличаются лишь способы их выражения и легкость, с которой программист может их реализовать в конкретном языке.

Нам нужно сосредоточить свое внимание на этих абстракциях при исследовании шаблонов проектирования, и эти абстракции должны быть основными при изучении шаблонов. Если вы будете изучать шаблоны проектирования как языковые концепции, то все сведется к простому заучиванию рецептов, и вы потеряете то, что делает шаблоны действительно полезными.

³ Некоторые шаблоны проектирования характерны только для конкретных языков, поэтому их часто называют *идиомами*. Когда в тексте мы используем термин *шаблоны проектирования*, мы говорим о концепциях, не зависящих от языка.

1.2.3. От мифа к науке

Обстоятельства, упомянутые выше, обнажают базовую проблему, связанную с шаблонами проектирования, при их описании, использовании и анализе. До сих пор слишком часто мы не знаем, почему мы делаем то, что делаем, даже если в нашей программе используется шаблон проектирования. Используя шаблоны проектирования механически, мы просто улучшаем документирование неосознанных фрагментов программ без понимания, которое должно быть следствием систематического анализа этих фрагментов.

Мы владеем искусством. Нам нужна наука. В конце концов, мы постоянно и самозабвенно разбрасываемся терминами *компьютерные науки* и *программная инженерия*. Интерпретация шаблонов проектирования как эталонных кодов искажает их смысл. Шаблоны проектирования позволяют экспериментировать с этими концепциями и распространять, обсуждать и уточнять полученные результаты.

Шаблоны проектирования как механические рецепты относятся к племенной мифологии.

Шаблоны проектирования как концепции являются основой научной дисциплины.

Элементарные шаблоны проектирования являются строительными блоками этой науки.

Коллекция элементарных шаблонов проектирования

Коллекция EDP (Elemental Design Patterns), по существу, представляет собой каталог основных концепций объектно-ориентированного программирования. Ее уникальность обусловлена двумя свойствами. Во-первых, она создана в стиле литературы о шаблонах проектирования. Каждый из шаблонов рассматривается как индивидуальная концепция с конкретным именем. Это позволяет обсуждать его и анализировать до полного понимания. Каждый элементарный шаблон проектирования в книге рассматривается вместе с проблемой, которую он решает, и описанием ситуаций, в которых его следует или не следует использовать. Мы приводим примеры реализаций, комментируем возможные последствия использования шаблонов и указываем родственные шаблоны, которые также следует внимательно рассмотреть. Это естественное определение каждого элементарного шаблона проектирования позволяет ввести в обиход термин, который имеет ясное и точное значение, что позволяет обсуждать его со студентами и другими разработчиками.

Во-вторых, шаблоны проектирования являются описаниями решений типичных проблем, которые могут возникнуть в ходе анализа существующих систем программного обеспечения. Элементарные шаблоны проектирования также являются решениями типичных проблем. Зная, что вы ищете, вы найдете их повсюду. Они имеют настолько универсальный характер, что до сих пор считалось нецелесообразным обстоятельно их описывать. Элементарные шаблоны проектирования изначально составляли полный набор несвязанных одна с другой проектных концепций, относящихся не к программному

обеспечению, а к его формальному описанию. Элементарные шаблоны проектирования возникают из математических основ объектно-ориентированных языков, но, смею вас уверить, что за исключением приложения, помещенного в конце книги, в этой книге нет математики. (Я все же надеюсь, что вы прочитаете это приложение просто из интереса, — оно этого заслуживает.) Мощные, но простые идеи, лежащие в основе элементарных шаблонов проектирования, обеспечивают надежную платформу, на которой можно подвергнуть ревизии основные концепции, дать им новую жизнь и видоизменить для новых приложений. Эта платформа позволяет также использовать элементарные шаблоны проектирования в качестве мельчайших строительных блоков, почти атомов, чтобы ясно и четко описать другие шаблоны проектирования. Из этих атомов вы можете создавать свои миры.

В книге описаны предпосылки для появления элементарных шаблонов проектирования, а также более широкий контекст использования шаблонов проектирования в программировании. Она также дает представление об объектно-ориентированной теории, позволяющей понять все взаимосвязи между шаблонами, но без математических формул, как я и обещал. Мы рассмотрим некоторые идеи, лежащие в основе объектно-ориентированного программирования, с большинством из которых вы уже знакомы. В заключение мы покажем, как эта базовая теория дает начало действительно надежным концепциям, используемым в обычном программировании. В последующих главах будет показано, как улучшить свои проекты с помощью элементарных шаблонов проектирования.

2.1. Основы

Впервые элементарные шаблоны проектирования были идентифицированы в проекте System for Pattern Query and Recognition (SPQR) в Университете Северной Каролины (University of North Carolina) в Чапел-Хилл (Chapel Hill) [35]. SPQR — это исследовательский проект, направленный на идентификацию образцов известных шаблонов проектирования в исходном коде. Эта система могла находить шаблоны проектирования независимо от языка программирования и легко распознавать разные реализации одного и того же шаблона проектирования поциальному определению шаблона. Если вас интересуют принципы работы системы SPQR, можете найти их в работах [35, 37, 38], но для нас достаточно общего представления об элементарных шаблонах проектирования и их определении.

Идея создать систему SPQR возникла, когда я работал профессиональным разработчиком программного обеспечения. Я работал в команде, ответственной за разработку одной из трех библиотек, предназначенных для использования в системе моделирования коммерческих и военных полетов в реальном времени. На одном из общих совещаний сотрудники отдела приложений горячо благодарили нас за новую функциональную возможность, говоря, что это именно то, что им было нужно. Мы отвечали, мол, не за что, это наша работа, и т.д.

После совещания разработчики всех трех библиотек посмотрели друг на друга и спросили, знает ли кто-нибудь, о чем говорили сотрудники отдела приложений. Никто

из нас не разрабатывал функциональную возможность, о которой шла речь, мы даже не понимали, откуда она взялась и как это стало возможным. Мы решили разобраться, полагая, что на это не понадобится много времени.

Затратив около двухсот человеко-часов, мы нашли ответ. В систему был незаметно встроен экземпляр шаблона *Decorator* (Декоратор) одного из шаблонов GoF (*Gang of Four*). Удивительным было то, что он не содержался целиком в какой-то одной из библиотек. Его фрагменты были разбросаны по всем трем библиотекам, а отдел приложений случайно интегрировали в свою систему целиком. Мы были ошеломлены. Начались довольно горячие споры. Действительно ли это шаблон проектирования? В конце концов, его никто *не разрабатывал*, он возник сам по себе.

Честно говоря, большинство программ так и развиваются. Они растут естественным и часто неожиданным образом. В управляемых проектах по разработке программного обеспечения мы можем управлять только тем, что знаем, и знание о существовании такого полезного компонента открыло перед нами возможности его улучшить, уточнить и документировать, чтобы разработчики могли его эффективно использовать. Однако его поиск стал отдельной проблемой. Напомним, что мы были разработчиками этого программного обеспечения, мы были экспертами и, тем не менее, мы потратили очень много времени, чтобы понять, что происходит. Очевидным ответом была автоматизация этого процесса. Как указано в главе 1, на самом деле нам нужен самодокументированный код или по крайней мере система для извлечения этой документации. Такой системой является SPQR.

Для создания системы SPQR необходимо было решить фундаментальную задачу: научить компьютер идентифицировать шаблоны проектирования. Шаблоны — это не рецепты; они являются гибкими и аморфными, т.е. тем, что люди называют концепциями. Большинство исследовательских и промышленных систем, предназначенных для поиска шаблонов проектирования в исходном коде, решают эту задачу, рассматривая шаблоны как конструкции, т.е. как жесткие формы, которые выглядят как реализация, а не как абстрактные концепции. Это вполне разумный подход. Он отражает точку зрения многих разработчиков: “Этот класс имеет поле, а этот подкласс имеет доступ к нему с помощью метода X” и т.д. Проблема в том, что такой подход просто сводит шаблоны проектирования к конкретным реализациям. В этом случае каждый вариант возможной реализации требует нового определения. Здесь мы снова сталкиваемся с методологией “вырезай и копирай” [25, 30].

Я далек от мысли, что эта работа имеет низкое качество. Эти инструменты заслуживают внимания как яркие примеры такого подхода. Они хорошо выполняют свою работу и повышают ценность программы, если они удовлетворяют предъявляемым требованиям. Проблема в том, что определение шаблонов в рамках такого подхода является крайне хрупким. Что если мы назовем метод не X, а Y? Что если подкласс не является прямым потомком, а существует другой класс, который находится между этими классами в иерархии наследования? Что если наш подкласс вообще не является производным подклассом, а использует суперкласс как объект делегата? В подходе к шаблонам проектирования, ориентированном на конструкции, в этих случаях потребуются новые определения. При этом предполагается, что вы работаете только с одним языком реализации. Напомним, что шаблоны не зависят от языковой реализации — проблема намного шире.

Кажется очевидным, что в системе SQPR следует применить другую тактику. Необходимый ответ можно найти, внимательно изучив основы и историю шаблонов проектирования. Чем больше я анализировал книгу GoF и причины, подвигнувшие их на ее создание, тем больше понимал, что истоки следует искать в исходной работе Александра. Когда я впервые прочитал мало цитируемую в среде разработчиков программного обеспечения книгу о шаблонах проектирования *Notes on the Synthesis of Form* [4], я понял нечто важное, что было утрачено: шаблоны являются концепцией, а не конструкциями.

Эта простая истина в корне изменила мой подход к обучению компьютера распознаванию шаблонов. Обучение компьютера распознавать языковые конструкции, связанные в жесткие структуры, никогда не обеспечит необходимой гибкости и желаемой точности. Вместо этого нужно научить систему находить программные концепции, потому что шаблоны проектирования относятся именно к этой категории.

К сожалению, для решения данной конкретной задачи общепринятая литература о шаблонах проектирования не подходила. Она нацелена в основном на описание высокуюровневых абстрактных концепций. Именно в этом заключается ее предназначение — поднять уровень дискуссий до высоких абстракций. Программисты же лучше справляются с абстракциями более низкого уровня, поэтому описывать их в виде шаблонов они не считают нужным. Сообщество специалистов по шаблонам проектирования сосредоточено на документировании своих уроков, которые еще не являются установившимися и общепризнанными. Шаблоны проектирования, описанные сообществом разработчиков программного обеспечения, скорее расширяют их спектр. С другой стороны, компьютеры делают лишь то, чему мы их можем научить. Для системы SPQR мы должны были установить цепочку понятий, которые должны были помочь описать концепции программирования. Мы должны были заполнить пробел между очень жесткими фрагментами реализации и очень гибкими концепциями, которые их описывают.

В процессе работы стало ясно, что нам необходимы элементарные шаблоны проектирования, которые образуют базис для описания проектирования программного обеспечения не только для автоматического анализа, но и для людей. Наша книга представляет собой частичный каталог элементарных шаблонов проектирования, написанных для людей, а не для машин. Она должна помочь разработчикам, студентам и проектировщикам заполнить пробелы в знаниях и дополнить язык, с помощью которого мы обсуждаем процесс разработки программного обеспечения. Как и любой шаблон проектирования, каждый элементарный шаблон проектирования имеет неформальную, или *шаблонную*, спецификацию, если использовать общепринятый термин, а также исходную математическую конструкцию. Благодаря существованию математических конструкций каталог EDP является уникальным. Каждый элементарный шаблон проектирования является абстракцией программирования, что позволяет говорить о еще более мелких компонентах проектов, не зависящих от способов реализации.

В результате возникли два способа описания элементарных шаблонов проектирования: теоретический и практический. В следующем разделе эти два подхода используются одновременно, что позволяет свести к минимуму формальные моменты. Для дальнейшего изложения нам не нужны строгие математические определения, хотя эта тема может заинтересовать некоторых читателей. Если вас интересует внутренняя механика и вы

хотите лучше понять основы взаимосвязей между шаблонами проектирования и теорией языков программирования, можете начать чтение книги с приложения, в котором описано ρ-исчисление, лежащее в основе нашей работы.

2.2. Где, почему и как

Обучение компьютера поиску крупномасштабных абстракций и концепций, называемых шаблонами проектирования, как это принято в существующей литературе, — трудная задача. Что мы делаем, когда сталкиваемся с большой и сложной задачей в компьютерных науках? Мы разбиваем ее на части.

Анализ литературы о шаблонах проектирования — непростое дело. За прошедшие годы было предпринято несколько попыток [12, 17, 32, 40, 41, 43], но они сводились лишь к частичному пересмотру и в глазах большинства разработчиков и исследователей, работавших за пределами этой темы, выглядели чудачеством. Помимо всего прочего, эти более мелкие фрагменты и концепции очевидны, не так ли? Они являются базовыми концепциями, с которыми мы сталкиваемся ежедневно, так зачем же описывать то, что нам и так кажется понятным?

Хорошо, если они так очевидны, то почему еще не задокументированы?

Основные концепции проектирования программного обеспечения являются настолько же “очевидными”, как и “правильный” способ строительства жилищ: все зависит от контекста, опыта и знаний. Всякий, кто обучен функциональному программированию на языке ML, понимает рекурсию совсем не так, как программист на языке С. Эти два разработчика выдвигают разные предположения о рекурсии, местах ее применения и способах использования, даже если они решают одну и ту же проблему. Если вы помните обсуждение стилей строительства зданий в главе 1, то вам уже знакома эта тема.

Ранее в этой главе говорилось, что низкоуровневые концепции программирования не были основным предметом исследования в сообществе специалистов по шаблонам проектирования, потому что, как правило, литература нацелена на документирование концепций, которые либо не являются рутинными, либо недостаточно понятны. Простые концепции кажутся рутинными и поэтому выпадают из их поля зрения, но это не значит, что они хорошо их понимают.

Низкоуровневые концепции, используемые в программировании, являются неосознанными по определению Александера. Мы более или менее успешно заучиваем их в классе и на практике и применяем их, потому что “так они работают”, не понимая осознанных решений. Большинство из нас никогда не изучали принципов, лежащих в основе этих концепций, но они существуют и хорошо понятны на математическом уровне.

Эти концепции просто недостаточно четко представлены на естественном языке, и для этого необходимы элементарные шаблоны проектирования. Кратко говоря, элементарные шаблоны проектирования представляют собой базовые концепции программирования и разработки программного обеспечения, по большей части слабо описанные. Если же некоторые шаблоны все же были подробно описаны, то в отрыве от других шаблонов. Каждый из элементарных шаблонов проектирования является осознанным описанием базовой концепции. Каталог EDP в совокупности связывает элементарные

шаблоны проектирования между собой в целостную концептуальную платформу, которую студент или разработчик может использовать для анализа других шаблонов. Он обеспечивает классификацию и лексикон для описания высокоуровневых абстракций, унифицирует язык и абстракции, так чтобы два разработчика понимали друг друга, например, говоря об элементарном шаблоне проектирования *Extend Method*.

Элементарные шаблоны проектирования предлагают язык, на котором можно размышлять о программном обеспечении, описывать его, говорить о нем на фундаментальных уровнях. Шаблоны проектирования являются частью базы знаний опытных разработчиков, но до сих пор нет аналогичных полных баз знаний для новичков или студентов.

Как же можно выделить эти концепции из традиционной литературы о шаблонах проектирования? Покажем это на примере шаблона *Decorator*.

2.2.1. Декомпозиция шаблона *Decorator*

Сначала договоримся о системе обозначений: имена шаблонов в книге вводятся курсивом и с прописной буквы, например *Extend Method*. Имена типов и классов начинаются с прописных букв и выделяются моноширинным шрифтом. Имена полей и методов выделяются “верблюжьим” стилем и моноширинным шрифтом, например `thisIsAMethod()`. Имена в текстах программ подчиняются тем же правилам.

Каноническое описание шаблона *Decorator* на языке UML (Unified Modeling Language) в соответствии с книгой GoF *Design Patterns* [21] показано на рис. 2.1. Сейчас вам не обязательно понимать, что такое шаблон проектирования *Decorator*; достаточно визуального обзора диаграмм UML. В общем, *Decorator* является популярным и широко используемым шаблоном проектирования, обеспечивающим механизм динамического расширения поведения во время выполнения программы. Его можно представить как встроенный модуль или систему расширения, которые можно найти в любом веб-браузере.

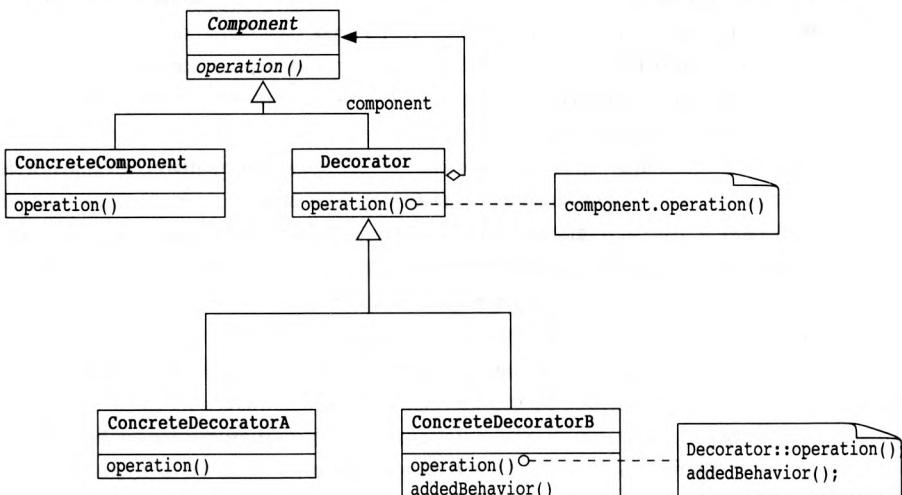


Рис. 2.1. Типичное описание шаблона *Decorator* на языке UML

Допустим, что вы хотите выполнить декомпозицию шаблона *Decorator*, чтобы лучше его понять. В конце концов, он относится к довольно высокому уровню абстракции. Если вы сможете по отдельности освоить более мелкие компоненты шаблона проектирования, то почти наверняка сможете лучше его понять. Что еще лучше, вы сможете использовать эти мелкие компоненты в процессе анализа других шаблонов проектирования, поскольку они описываются аналогично.

Людей всегда интересовало, как работают сложные механизмы, например редкостные автомобили. Вы можете купить один из таких автомобилей, взять инструменты и разобрать всю машину или изучить каждую из ее частей отдельно. Например, вы можете исследовать двигатель внутреннего сгорания или гидравлическую тормозную систему. Затем эти системы можно разобрать на еще более мелкие фрагменты и изучить их по очереди, чтобы лучше понять более крупные части.

В конце концов, зная, как работает двигатель внутреннего сгорания, вы сможете понять не только его роль в работе машины в качестве инкапсулированной абстракции, но и применить эти знания к другим средствам передвижения и даже газонокосилкам, генераторам и другим машинам с двигателем внутреннего сгорания. Декомпозиция шаблона проектирования преследует примерно те же цели: идентификация более мелких частей, играющих отдельную роль, их внимательное изучение и применение в новых ситуациях.

После тщательного изучения литературы, посвященной шаблонам проектирования (напомним, что существуют тысячи таких работ), вы заметите, что компоненты шаблона *Decorator* похожи на компоненты других шаблонов.

Одним из них является шаблон *Objectifier* (Конструктор объектов), впервые описанный Вальтером Циммером (Walter Zimmer) в 1995 году [43] и показанный на рис. 2.2. Шаблон *Objectifier* описывает, как представить интерфейс отдельного объекта с многими конкретными реализациями скрытно от клиента. Когда клиент вызывает метод через интерфейс *Objectifier*, он не знает, какой из двух (или нескольких) конкретных методов будет выполнен.

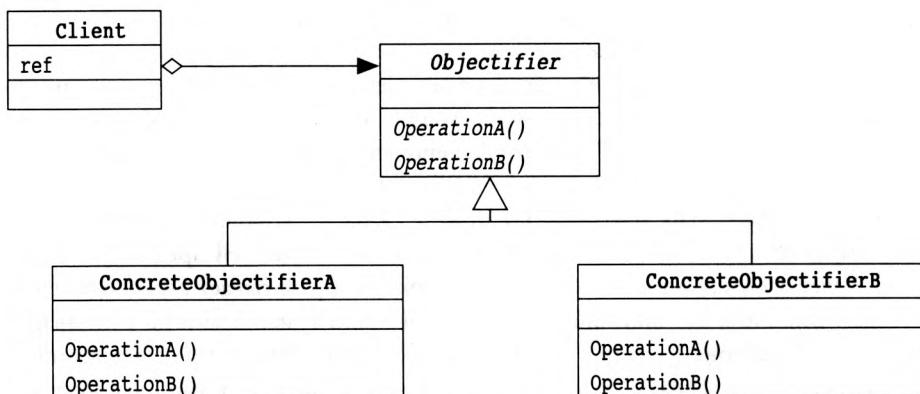


Рис. 2.2. Шаблон Objectifier на языке UML

Другим примером является шаблон *Object Recursion* (Рекурсия объектов) описанный Бобби Вулфом (Bobby Woolf) в 1998 году [41] и показанный на рис. 2.3. Шаблон *Object Recursion* связывает один с другим два объекта, имеющих родственные типы. Глядя на диаграммы UML, представленные на рис. 2.2 и 2.3, можно убедиться, что шаблоны *Objectifier* и *Object Recursion* выглядят, как наборы компонентов шаблона *Decorator*. Хотя они не совпадают в точности, некоторые свойства их UML-структурки достаточно похожи, чтобы сказать, что мы можем описать части шаблона *Decorator* в терминах этих шаблонов, т.е. можно сказать, что шаблон *Decorator* состоит из этих шаблонов.

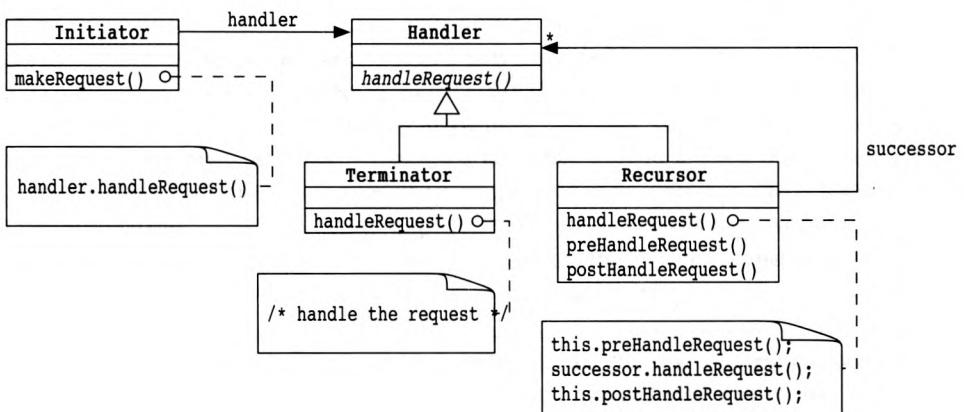


Рис. 2.3. Шаблон *Object Recursion* на языке UML

Более детальный анализ показывает, что шаблон *Objectifier* можно рассматривать как часть шаблона *Object Recursion*. Форма шаблона *Object Recursion* основана на шаблоне *Objectifier*, но в нее добавлена связь между конкретными реализациями и интерфейсом, и она применяется к одному и тому же методу. Иначе говоря, когда класс *Initiator* вызывает метод *handleRequest()*, по замыслу шаблона *Objectifier* на этот вызов может ответить класс *Terminator* или *RecurSOR*. Если на вызов отвечает класс *Terminator*, то запрос выполняется. Если на вызов отвечает класс *RecurSOR*, то создается дополнительный вызов через интерфейс для другого метода *handleRequest()*, и процесс начинается сначала. Эта цепочка продолжается до тех пор, пока обработчиком вызова не станет класс *Terminator*. В этой точке цепочка обрывается, чего и следовало ожидать.

Анализируя серию этих действий, вы можете сказать: “Шаблон *Decorator* использует шаблон *Object Recursion* для обхода цепочки объектов, состоящей предположительно из двух или более звеньев, и в каждом из этих объектов обрабатывает вызов одного и того же метода”. Впрочем, это описание выглядит довольно туманным и не учитывает множество деталей шаблона *Decorator*. Для работы необходима более удачная декомпозиция, но по крайней мере мы знаем, что шаблоны можно описывать с помощью более мелких шаблонов.

2.2.2. Вниз по кроличьей норе

Мы установили, что шаблон *Decorator* можно построить из более мелких частей, но являются ли эти части настолько малыми, насколько возможно? До какой степени можно “измельчить” шаблон, чтобы он оставался шаблоном?

Для ответа на этот вопрос необходимо снова спросить “Что такое шаблон?” Мы знаем, что это концепция или элемент проекта, содержащий определенные основные компоненты. Давайте отложим в сторону вопрос о том, что такая концепция, и рассмотрим две другие составляющие. Как сказано в канонической книге GoF, проект — это способ, с помощью которого части целого взаимодействуют и связываются одна с другой. Это описание указывает, что участники и сотрудники являются главными компонентами проекта, определяющими части проекта и способы их взаимодействия и связи.

Рассмотрим традиционное определение шаблона проектирования: “Общее решение общей проблемы в конкретном контексте”. Как показано в табл. 2.1, составные части канонических спецификаций шаблонов проектирования практически всегда можно разделить на три категории: решение (или реализация), проблема (или описание) и контекст (или среда). Структура, реализация и пример кода, без сомнения, являются решением. Предназначение, мотивация и известные примеры использования описывают предметную область.

Таблица 2.1. Три категории составных частей определения шаблона

Решение	Структура Реализация Пример кода
Проблема	Предназначение Мотивация Известные примеры использования
Контекст	Условия применения Последствия Родственные шаблоны

Контекст, в котором возникает проблема, определяет условия применения шаблона проектирования, очерчивает рамки дискуссии о возможных последствиях и часто позволяет понять, достаточно ли точно подходит данный шаблон и какие родственные шаблоны могли бы быть более удачным выбором.

После разделения этих элементов на категории остаются две части, которые трудно отнести к одной из этих категорий: участники и сотрудники. Очевидно, что они являются частью решения, поскольку во многом его формируют. Однако в то же время они являются частью описания, поскольку почти непосредственно отражают формулировку проблемы. Кроме того, они являются частью контекста, поскольку создаются в ответ на требования среды, в которую помещены. Итак, они находятся на пересечении трех компонентов шаблона проектирования.

Ядро проекта образуют отношения. Проект автомобиля представляет собой нечто большее, чем набор деталей: он описывает, как они согласуются друг с другом. Дом — это нечто большее, чем набор бревен, гвоздей и медных труб; он имеет форму или план, который преодолевает энтропию и, так сказать, сохраняет структуру на более высоком энергетическом уровне, чем обычная гора камней. Список частей дома описывает то, что у вас есть для начала строительства, а связи в проекте говорят о том, как они согласуются между собой.

Давайте немного изменим вопрос: “Какую самую простую связь мы можем определить?” Это простой вопрос. Самым простым является отношение между двумя сущностями. Применим этот принцип для деконструкции шаблонов проектирования. В результате у нас появится ясная цель. Критически рассматривая последующие более мелкие шаблоны проектирования, мы можем спросить: “Имеет ли эта часть больше одного отношения?” Если не имеет, то мы достигли цели. Если имеет, возможно, следует продолжить деконструкцию.

Почему мы, “возможно”, должны продолжать деконструкцию, если в шаблоне есть больше одного отношения? Потому что не все отношения созданы одинаковыми. Одни отношения играют важную роль в обсуждении шаблона, а другие лишь создают контекст. Главным контекстуальным отношением является область видимости, которая проявляется в разных формах.

Объявляя переменную, определяя метод и описывая класс, мы должны помнить, что новая сущность “живет” в своей области видимости. Область видимости делает элемент уникальным по отношению к другим элементам системы. Если у нас есть два класса с именем `Menu`, но один из них определен в пакете с именем `GraphicalUIElements`, а другой — в пакете с именем `RestaurantNecessities`, то можно быть совершенно уверенным, что это разные сущности. Их области видимости свидетельствуют о том, что они являются разными классами, и путаница никогда не возникает. Этот принцип применяется каждый раз, когда вы охватываете *нечто*, имеющее имя, а внутри него определяете новую сущность.

Классы представляют собой область видимости для методов и полей, которые в них определены. Пространства имен и пакеты являются областями видимости для всех сущностей, находящихся внутри них. Методы и функции являются областями видимости для локальных переменных, определенных внутри них. Снова и снова мы видим один и тот же механизм, работающий в рамках разных языковых свойств. Для доступа к конкретному элементу мы точно указываем, какой именно элемент нам нужен, называя области видимости сверху вниз. К сожалению, это не всегда просто. Иногда эти области видимости являются неявными, и мы можем выходить за их пределы, например, ссылаясь на локальную переменную внутри метода или другой член внутри класса. Более того, несмотря на похожее поведение, эти области видимости могут иметь разные синтаксисы, с помощью которых обеспечивается доступ. Например, в языке C++ для доступа к классу `Menu` в пространстве имен `GraphicalUIElements` используется двойное двоеточие: `GraphicalUIElements::Menu`. Однако доступ к списку пунктов меню внутри экземпляра этого класса обеспечивается с помощью точки: `Menu.theItems`. Оба этих способа задают элемент, который мы хотим выбрать из охватывающего элемента, но делают

оны это по-разному. В книге описывается единственный способ, с помощью которого можно работать с областями видимости, независимо от того, как они реализованы в конкретных языках.

Рассмотрим класс A, имеющий метод f, и класс B, имеющий метод g, как показано на диаграмме UML на рис. 2.4 и в листинге 2.1. Класс A имеет поле b типа B. В теле функции main() создается экземпляр a типа A, а затем вызывается метод a.f().

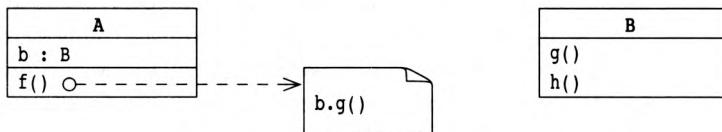


Рис. 2.4. Простой вызов метода на языке UML

Листинг 2.1. Псевдокод простого вызова метода

```

1  class A {
2      B b;
3      f() {
4          b.g();
5      };
6  }
7
8  class B {
9      g() {};
10     h() {};
11 }
12
13 main() {
14     A a;
15     a.f();
16 }
  
```

Область видимости метода f() ограничена объектом a. В объектно-ориентированных языках функции и методы всегда заключены в какую-то область видимости, даже если она является неявной. Даже в языке C++ глобальные функции и поля можно рассматривать как принадлежащие невидимому и неявному объекту, представляющему пространство имен и определяющему способ, которым система выполнения программ с ним работает. Если не верите, прочитайте утверждение 3.3.6 [basic.scope.namespace], параграф 3 документа 2011 C++ ISO Standard Working Draft [6]. Кроме того, в настоящее время использование объектов в языке C++, область видимости которых ограничена файлом (объектов, объявленных как глобальные и статические), коллективная мудрость считает неприемлемым, отдавая предпочтение безымянным пространствам имен, выполняющих ту же работу более прозрачным способом. Иными словами, каждая единица трансляции задает область видимости каждого своего объекта, и тот факт, что область видимости не имеет имени, означает, что ее невозможно использовать за пределами единицы трансляции. Здесь объекты также используются для определения области видимости и сокрытия элементов, ранее считавшихся доступными без ограничений.

Возвращаясь к листингу, отметим, что метод `a.f()`, в свою очередь, вызывает метод `b.g()`, и между этими двумя методами существует отношение. Отношение между объектом `a` и методом `f`, представляющее собой определение области видимости, является контекстным. Оно помогает указать, о каком методе идет речь. Аналогичное отношение существует между объектом `b` и методом `g`. Однако вызов от `a.f()` к `b.g()` является не просто контекстным. Это отношение между двумя методами является *основным*. Иначе говоря, это отношение, которое нас интересует, но требует учета области видимости.

Отношения определения контекста помогают уточнить точку зрения на конкретный элемент проекта. Нам необходимы два элемента проекта, чтобы образовать простое отношение, описанное в коллекции EDP. Может существовать множество отношений определения контекста, влияющих на окончательное единственное отношение, которое нас интересует, но мы пока не будем ими заниматься. Они являются частью описания элементов программирования, содержащих конечные точки отношения, с которым мы хотим работать. Возвращаясь к предыдущей метафоре с архитектурными стилями, можно сказать, что элементы контекста немного напоминают утверждение, что плашка — это лист фанеры размером 2×4 метра и толщиной полдюйма. Это ничего нам не говорит о том, как ориентировать листы фанеры один относительно другого или как забивать в них гвозди. Задание контекста позволяет дать определение элемента, но ничего не говорит о том, как этот элемент связан с другими элементами системы.

Теперь мы можем поставить вопрос о том, достигли ли мы цели декомпозиции: “Содержит ли данное включение более одного интересующего нас отношения?” Мы просто должны определить, какое отношение нас интересует. Выше мы описали отношение определения контекста, включающее в себя владение классом своими методами и полями, а также пространства имен, пакеты и все другие методы группировки, существующие в программировании, и указали, что пока они нас не интересуют. Рассмотрим вместо этого остальные сущности, между которыми мы будем создавать отношения.

Остаются классы, их методы и поля, и кое-что еще. На первый взгляд, из этого списка выпал один пункт: объекты. Ведь, в конце-то концов, мы же занимаемся *объектно-ориентированным* программированием! Мы добавим объекты в этот список из соображений полноты и позднее покажем их центральное место в этой концепции. (Формальные объяснения приведены в приложении.) Пока в нашем списке есть четыре категории программных сущностей: объекты, методы, поля и классы, или типы. Называть классы типами может показаться странным, но на самом деле в этом есть глубокий смысл.

Мы не будем называть их классами, поскольку не во всех объектно-ориентированных языках есть классы, но в каждом объектно-ориентированном языке есть типы. Напомним, что мы хотим описать шаблоны проектирования независимо от языков программирования, поэтому необходимо найти общие свойства и требования, относящиеся ко всем без исключения объектно-ориентированным языкам программирования. Итак, как же перейти от классов, с которыми знакомы большинство программистов, к чистой объектно-ориентированной концепции? На самом деле все просто: достаточно разбить класс на составные части.

Класс — это интересная сущность, в которой так много общего для всех объектно-ориентированных языков, что большинство студентов и программистов думают, будто он является основным и необходимым элементом. На самом деле, если придерживаться строгой объектно-ориентированной точки зрения, это не так. Некоторые языки, такие как Self и Lua, даже не содержат понятия класса. Вместо этого они используют определение прототипов, клонирование и другие действия над объектами, чтобы выполнить те же самые функции. Этим свойством обладают не только такие довольно экзотические языки, использующие объекты там, где большинство языков использует классы. Следы такого подхода можно обнаружить и в языке JavaScript. Даже язык Smalltalk, считающийся прародителем большинства объектно-ориентированных языков, реализует и трактует класс немного нестандартно, хотя термин используется тот же самый.

В общем, класс в современных версиях объектно-ориентированных языков, таких как C++ и Java, выполняет две функции. Во-первых, он описывает элементы, которые должны принадлежать созданному объекту данного класса, — методы и поля *объекта*. Большинство из нас думает, что класс должен использоваться именно так. Во-вторых, он описывает элементы, общие для всех созданных объектов класса, — методы и поля *класса*. Именно это происходит, когда в программе на языке C++ или Java мы объявляем метод или поле с помощью ключевого слова `static`. Первый случай точно соответствует понятию типа в формальном смысле, а второй можно описать с помощью специального объекта. Рассмотрим поле, объявленное как статическое в классе на языке C++, и механизм доступа к нему (листинг 2.2).

Листинг 2.2. Определение и использование полей, экземпляров и пространств имен в языке C++

```
namespace MyNamespace {
2   int aField;
3 }
4
5 class MyClass {
6 public:
7   static int sharedField;
8   int instanceField;
9 };
10
11 main() {
12   MyClass mc;
13   mc.instanceField = 0;
14   MyClass::sharedField = 1;
15   MyNamespace::aField = 2;
16 }
```

Обратите внимание на то, что в функции `main` перед именем поля `sharedField` указано имя класса `MyClass`. Точно такое же обозначение используется для доступа к полю `aField` в определенном нами пространстве имен `MyNamespace`. Сущность, владеющую элементами класса, можно интерпретировать как “живой” объект. Точно так же можно трактовать и пространство имен. Мы указываем имя этого объекта и ту его

часть, которая нам нужна, так, как мы это обычно делаем по отношению к объектам, являющимся экземплярами класса. Если в языке программирования предусмотрена концепция пространства имен, пакета или класса, то возникает возможность создавать эти объекты автоматически.

Вы еще не верите? В языке Smalltalk класс представлен объектом. Буквально он называется *объектом-классом* (class object). Для создания объекта этого класса мы посылаем объекту-классу сообщение — “эквивалент вызова метода объекта” в языке Smalltalk, — чтобы он вернул экземпляр этого класса. Методы и поля класса принадлежат объекту-классу. Это точно соответствует описанному нами сценарию.

Теперь посмотрим на язык Java. В нем объект-класс представляет собой специальный объект, доступный для проверки с помощью отражения и связанный с классом, определенным программистом. В отличие от языка Smalltalk, он не доступен программисту “как обычно”, а резервируется для более сложных действий отражения. Здесь применяются те же базовые принципы, что и в языке Smalltalk, однако новая конструкция MyClass, заимствованная из языка C++, скрывает этот процесс под покровом привлекательного синтаксиса.

В верхней части листинга 2.3 приведен фрагмент кода на языке Java, а в нижней — одна из возможных декомпозиций этого класса. Если вспомнить, что в данном случае обычное выражение new MyClass является синонимом MyClass Object.new(), а доступ к статическим членам осуществляется через объект MyClass_Object с помощью выражений MyClass_Object.sharedData и MyClass_Object.sharedMethod(), то можно получить традиционный механизм, уже существующий в языке Java. Аналогичные механизмы есть и в языке C++, в котором нет отражения.

Листинг 2.3. Класс в языке Java и его возможный эквивалент в виде объекта и типа

```

1 class MyClass {
2     public:
3         static int sharedData;
4         int instanceData;
5         static void sharedMethod() { ... };
6         void instanceMethod() { ... };
7     };
8
9
10    MyClass:
11        int instanceData;
12        void instanceMethod();
13
14    MyClass_Object:
15        int sharedData;
16        void sharedMethod();
17
18    MyClass new();

```

Несмотря на то что детали этого механизма в разных языках варьируют, в любом случае класс эмулируется с помощью типа и объекта. Возможно, в первый раз это звучит странно, но на практике к этому быстро привыкают. Просто помните, что класс

без статических членов (если использовать терминологию языков C++ и Java) является просто типом. Любые определенные члены класса можно перенести в соответствующий объект-класс, как в языке Smalltalk, или в объект-класс, доступ к которому обеспечиваеться отражением, как в языке Java.

Для того чтобы обеспечить основные свойства типизированных языков, необходимы типы, а объекты являются основой объектно-ориентированных языков независимо от свойств их классов. Используя эти два понятия в сочетании, мы можем эмулировать классы из объектно-ориентированных языков, основанных на концепции класса. Это позволит нам свести набор элементов, необходимых для определения шаблонов проектирования, до четырех ранее указанных: объекты, методы, поля и типы.

Как взаимодействуют элементы этого короткого списка? В табл. 2.2 приведен полный список возможных способов, которыми сущности из левого столбца могут взаимодействовать с сущностями в верхней строке. Объекты и типы могут содержать, а следовательно, определять, любой из четырех элементов, но мы хотим избавиться от отношения “содержит” при изучении концепции области видимости. В некоторых языках методы могут так же определять, или охватывать, вложенные методы или типы, и, разумеется, мы можем определять локальные переменные и поля. Кроме того, методы могут вызывать другие методы, использовать нелокальные поля и иметь тип возвращаемого значения. Поля представляют собой более простое понятие, поскольку им можно присваивать значения, возвращаемые методом, или значения других полей. Конечно, они могут иметь тип. Наконец типы могут определять любую из указанных четырех сущностей с помощью области видимости и использовать другие типы в качестве подтипов.

Таблица 2.2. Взаимодействия между сущностями в объектно-ориентированных языках программирования

	Объект	Метод	Поле	Тип
Объект	Определяет	Определяет	Определяет	Определяет <i>или</i> имеет тип
Метод	Нет	Определяет <i>или</i> вызывает метод	Определяет <i>или</i> использует поле	Определяет <i>или</i> имеет тип возвращаемого значения
Поле	Нет	Изменяет состояние	Связывает	Имеет тип
Тип	Определяет	Определяет	Определяет	Определяет <i>или</i> является подтипом

Небольшое количество взаимодействий позволяет их перенумеровать и образовать небольшое количество возможностей. В табл. 2.3 перечислены взаимодействия, которые не предусматривают определения другой сущности.

Поле имеет тип, с которым оно связано; он называется типом поля. То же самое касается объекта. Аналогично методы имеют типы возвращаемых значений. Эти типы используются для определения вида возвращаемых данных, например поля или другого объекта, который может быть параметром или возвращаемым значением другого метода. Как и определение контекста, определение данных описывает сам объект или поле, а не устанавливает отношение между двумя сущностями.

Таблица 2.3. Взаимодействия между сущностями в объектно-ориентированных языках программирования, не определяющие контекст

Объект	Метод	Поле	Тип
Объект			Обладание типом
Метод	Вызов метода	Использование поля	Обладание типом возвращаемого значения
Поле	Изменение состояния	Связность	Обладание типом
Тип			Является подтипом

В качестве примера рассмотрим фрагмент кода в листинге 2.4. Поле `pos` определено как экземпляр типа `Position`. Это определение позволяет понять, что собой представляет объект `pos`. Его размещение внутри класса `Glyph` говорит о том, как его интерпретировать, но при этом у нас нет никакой информации о том, как объект `pos` взаимодействует с другими элементами в системе. Аналогично типы параметров метода `scaleCopy` означают лишь их вид, но не сообщают, как они связаны между собой.

Листинг 2.4. Типизация как контекст

```
1 class Glyph {
2     Position pos;
3     Glyph scaleCopy( float x, float y );
4 }
```

С другой стороны, иные отношения, такие как зависимость одного типа от другого, предоставляют больше информации, чем простое описание элемента. Зависимость одного типа от другого, которая чаще всего наблюдается при наследовании, довольно хорошо документирована. Эта концепция широко известна и хорошо освоена.

Для определения элементарных шаблонов проектирования достаточно всего четырех основных отношений, указанных в центре табл. 2.3, в которых метод или поле зависит от другого метода или значения поля при выполнении своего задания. Этими отношениями являются вызов метода (один метод зависит от другого), использование поля методом (метод зависит от поля), задание поля методом (поле зависит от метода при изменении состояния) и зависимость одного поля от другого, например `a = b + 1`. Как показано в предыдущих таблицах, эти отношения можно назвать вызовом метода, использованием поля, изменением состояния и связностью. Хотите верьте, хотите нет, но почти вся книга посвящена только одному виду отношений — вызову метода. Да, если взглянуть на эту тему под правильным углом, то о простом вызове метода другим методом можно многое сказать.

Вероятно, вы считаете, что вызов метода выглядит как структурный, а не как концептуальный элемент. Это правда. Именно поэтому я начал с обзора зависимостей. Зависимость одного элемента от другого не требует прямой связи. Рассмотрим ситуацию, в которой метод `f` вызывает метод `g` в своем теле. Мы говорим, что метод `f` зависит от метода `g`. Теперь предположим, что метод `g`, в свою очередь, вызывает другой метод `h`, как показано в листинге 2.5. Мы говорим, что метод `g` зависит от метода `h`. Должно быть очевидным, что это отношение является транзитивным — если `f` зависит от `g`, а `g`

зависит от `h`, то `f` также зависит от `h`. Эта зависимость не прямая, но это неважно: методу `f` все равно необходимо, чтобы метод `h` выполнил свою работу, чтобы метод `f` мог выполнить свою.

Листинг 2.5. Псевдокод цепочки вызовов методов

```
function f() {  
2     g();  
};  
4  
function g() {  
6     h();  
};  
8  
function h() {};  
10  
function main {  
12     f();  
};
```

Давайте немного задумаемся. Если мы хотим, чтобы метод `f` зависел от метода `h`, имеет ли значение, будет он вызывать его непосредственно или с помощью промежуточного метода `g`? Не имеет. Поскольку метод `f` как-то зависит от метода `h`, наше требование выполняется. Мы просто переходим от структурной связи, порождаемой вызовом метода, к концептуальной связи, основанной на отношении вызова метода (method call reliance). Это освобождает нас от обсуждения низкоуровневых концепций программирования в структурном стиле и позволяет описать их в концептуальном виде, необходимом для идентификации шаблонов проектирования с помощью системы SPQR и работы с ними. Мы обсудим эту тему более детально в разделе 4.1.1, где покажем, насколько эти концепции важны для работы с элементарными шаблонами проектирования.

В процессе обучения системы SPQR концепциям программирования мы совершаляем переход от жестких структур программирования к гибким и нестрогим способам описания отношения, образующих проект программного обеспечения. Те же самые технологии наделяют программистов и проектировщиков способностью к гибкому мышлению и систематическому и в то же время понятному абстрагированию проекта на основе реализации. Тем самым мы решаем две задачи.

Итак, повторим вкратце вышесказанное. Мы свели список сущностей, интересных с точки зрения программирования, к четырем элементам: объектам, методам, полям и типам (которые в зависимости от языка программирования могут рассматриваться как классы). Другие программные сущности, такие как пространства имен и пакеты, помогают описать искомый элемент, но не создают те виды отношений, которые нас интересуют в процессе анализа нетривиальных проектов. На основе этих четырех элементов можно выделить четыре интересующих нас отношения, или зависимости: “метод–метод”, “метод–поле”, “поле–метод” и “поле–поле”. В книге мы сосредоточимся почти исключительно на зависимости “метод–метод”.

Может ли одна эта зависимость, основанная только на вызовах методов, быть полезной для описания шаблонов проектирования? В правильном контексте все возможно.

2.2.3. Контекст

Почти весь предыдущий раздел был посвящен поиску отношений, заслуживающих нашего внимания. Мы рассмотрели основные зависимости, такие как зависимости между методами и полями, пропустив другие виды зависимостей, такие как “тип–тип”, или наследование.

Настал момент вернуться к контекстуальным зависимостям. Вызов метода связан с тремя видами информации, которая помогает понять цель данной зависимости в конкретном проекте. Они очевидны, если искать вызов метода так, как показано ниже.

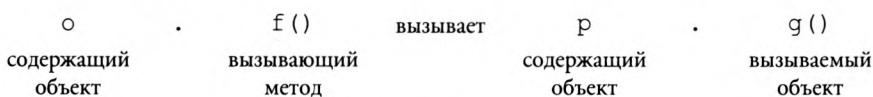


Рис. 2.5. Части вызова метода

В объектно-ориентированной теории на самом деле нет такой сущности, как простой метод или функций. Это не является чем-то необычным. В гибридных объектно-ориентированных языках программирования, таких как C++, вы можете поместить функции в глобальное пространство. В более строгих языках, таких как Java, этого сделать нельзя; каждый метод должен находиться в объекте либо в результате создания экземпляра класса, либо как статический метод уровня класса, что эквивалентно размещению метода в классе-объекте данного класса. Допустим, что каждый метод должен быть погружен в какой-то объект. У каждой зависимости, основанной на вызове метода, существует четыре компонента: вызывающий метод, вызываемый метод и объекты (классы-объекты или экземпляры), содержащие эти методы. В листинге 2.6 показан пример таких классов и объектов для вызова метода в языке C++. На рис. 2.5 показаны четыре компонента зависимости, в которой метод `o.f()` вызывает метод `p.g()`. Эти компоненты существуют в каждом вызове метода.

Листинг 2.6. Простой вызов метода, показанный на рис. 2.5

```

1 class Class2 {
2     void g() {};
3 };
4
5 class Class1 {
6     void f() {
7         Class2 p;
8         p.g();
9     }
10};
11
12 void main() {
13     Class1 o;
14     o.f();
15 };
  
```

На рисунке скрыты три вида информации.

1. Сходство между объектами, содержащими методы.
2. Сходство между типами объектов, содержащих методы.
3. Сходство между вызывающим и вызываемым методами.

Я ввел новое слово: “сходство” (*similarity*). В разговорной речи оно означает именно то, о чём вы подумали: аналогию между двумя вещами. Но что оно означает в контексте вызова метода?

Сходство объектов — это степень, до которой один объект похож на другой. Имеется в виду один и тот же объект? Имеется в виду псевдоним второго объекта, заданный указателем? Эти объекты никак не связаны?

Можно также обсудить отношение сходства между типами объектов. Это один и тот же объект? Это подтип другого объекта? Может быть, это одноуровневые типы с одним и тем же предком?

Сходство методов объяснить сложнее. Необходимо определить, решают ли эти методы сходные задачи. Нам придется учесть множество аспектов, например найти ключевые слова в комментариях или даже выполнить полный анализ тел методов, чтобы понять, что они вычисляют. Однако есть более простой путь, если учитывать только имена методов, или их сигнатуры.

Нет, правда, подумайте об этом. Каким образом программист выражает назначение метода? Именем. Это свойство даже нашло свое отражение в шаблоне *Intention Revealing Selector*, изобретенном Кеном Беком [5] (курсив мой): “Есть два способа именования методов. Первый — назвать метод так, чтобы описать, как он выполняет свою задачу. . . Наиболее важный аргумент против этого стиля именования заключается в его низкой информативности. . . Второй способ — назвать метод так, чтобы было понятно, что он делает, а вопрос “как” оставить для анализа тел методов. Это трудная работа, особенно если у вас есть только одна реализация. Ваш мозг занят мыслями о том, как выполнить задачу, поэтому естественно, что вы называете метод, отвечая на вопрос “как”. Попытки перейти в именовании методов от вопроса “как” в вопросу “что” стоят усилий как в долгосрочной, так и в краткосрочной перспективе. Получившийся код будет более понятным и гибким”. Бек так формулирует свою мысль: “Называйте метод так, чтобы указать, что он делает”.

В книге Роберта Мартина (Robert Martin) *Clean Code* [27] именованию посвящена целая глава, поскольку это очень важный вопрос. Возможно, лучше всего смысл этой главы выражает старая пословица: “Говори, что думаешь. Думай, что говоришь”. Это хорошее правило, и хотя не все группы разработчиков строго ему следуют, оно достаточно широко принято, чтобы сэкономить время и силы. В худшем случае мы можем обратиться к поиску синонимов и перестановке слов, чтобы понять, что имена `makeAString` и `stringCreator` имеют сходный смысл, но достойно удивления то, что в большинстве случаев для решения этой задачи достаточно простого лексикографического сравнения.

Небольшую проблему создают перегруженные методы, имеющие одинаковые имена, но разные списки аргументов, но и в этом случае мы можем понять причины сходства.

Один метод больше похож на другой, имеющий такое же имя, но другие аргументы, чем на метод, имеющий совершенно иное имя. Идеальным совпадением считается такое, в котором методы имеют одинаковые имена и списки аргументов. Однако этот вопрос не относится к теме нашей книги.

Именно этот алгоритм используется в системе SPQR с великолепным результатом. Изначально я выбрал этот алгоритм, потому что его проще всего реализовать, и я даже не надеялся, что он окажется работоспособным. Я планировал заменить его подходящим алгоритмом после первичного тестирования. Однако как только система SPQR стала выдавать результаты и стало ясно, что в большинстве случаев этого метода вполне достаточно, возник вопрос, как это объяснить. В ретроспективе это очевидно, но неожиданные открытия обычно становятся яснее только по прошествии времени.

Нас не должно было удивлять, что этот метод работает. Когда разработчик читает код, пытаясь понять его цель, он сначала ищет имена сущностей, а затем пытается угадать, что они означают. Именно так работает принцип именования Бека: мы *ожидаем*, что сущности будут названы правильно и, главное, согласованно. Мы ожидаем, что сущности, выполняющие сходные функции, будут иметь сходные имена. Все, что мы делаем, лишь усиливает это ожидание. Система SPQR просто была научена делать то, что делаем все мы, читая код: пытаться выяснить, как сущности связаны одна с другой. В свою очередь, этот опыт привел к созданию каталога EDP, позволяющего лучше понять, как люди выполняют эту задачу, и предлагающего простой и естественный способ описания сходства третьего рода.

Следовательно, для любого вызова метода мы можем идентифицировать задействованные объекты, типы этих объектов и сходство методов путем довольно простого анализа, автоматического или ручного. Что это нам дает?

Это дает нам три независимые оси контекста, в котором можно поместить и описать любой вызов метода. Мы можем представить эти три оси как координатные оси трехмерного пространства и приступить к исследованию. Любой вызов метода, в зависимости от оценок сходства по трем координатам контекста, занимает только одну точку пространства. Иначе говоря, все зависимости, представляющие собой вызов метода и находящиеся в одном и том же месте, имеют похожую форму, и мы можем описать эти формы независимо от того, кто ее написал, как он ее написал, в каком программном обеспечении она использована и даже на каком языке программирования реализована.

Мы можем описать такие зависимости на основе того, что они делают и чего мы от них *ожидаем*. Благодаря этому мы можем спорить о том, как их использовать наилучшим образом, как они связаны с другими зависимостями в этом трехмерном пространстве и как превратить одну зависимость “вызов метода” в другую.

Это дает нам первую группу элементарных шаблонов проектирования.

2.2.4. Пространство проекта

Говоря о зависимости “вызов метода”, посмотрим, как может выглядеть упомянутое выше трехмерное пространство. У нас есть три ортогональных и независимых компонента вызова отдельного метода, основанные на трех зависимостях: объектах, содержащих методы, типах этих объектов и самих методах. Возможно, это описание слишком

громоздко, но его легко представить на рисунке. Для простоты начнем с вызовов методов, в которых игнорируется тип объекта, сконцентрировав внимание на сходстве объектов и методов (рис. 2.6).

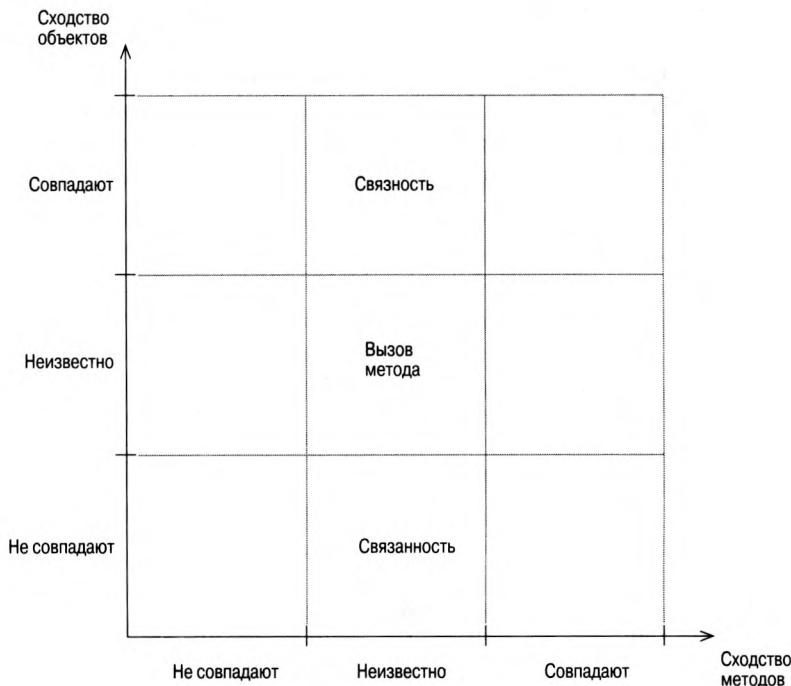


Рис. 2.6. Простое пространство проекта

Получается сетка 3×3 , по осям которой отложены координаты “*Совпадают*”, “*Неизвестно*” и “*Не совпадают*”. Координата “*Неизвестно*” включена по причинам, которые мы вскоре объясним. Совершенно ясно, что центральный квадрат соответствует простому вызову метода как таковому. В этом случае нам ничего не известно ни о задействованных объектах, ни о самих методах. Как ни странно, в большинстве случаев анализ и исследование программного обеспечения не выходят за пределы центрального квадрата. Когда говорят о вызовах методов без указания зависимости между методами, содержащих их объектов или типов этих объектов, то *все* такие вызовы методов находятся в центральном квадрате. Все без исключения.

В верхнем центральном квадрате содержатся вызовы методов со сходными объектами. Для целей дальнейшего обсуждения будем называть сходные объекты просто *эквивалентными*. В таком случае речь идет о методах, принадлежащих одному и тому же объекту и характеризующих *связность* объекта (*object's cohesion*), т.е. то, как плотно он связан со своими методами [42].

В нижнем центральном квадрате находятся непохожие объекты. Пока будем просто считать, что это разные экземпляры объекта. В этом случае вызовы методов образуют основу для *связанности* (*coupling*), т.е. меры независимости между объектами [15].

Верхний и нижний центральные квадраты образуют ядро для широкого спектра исследований [7–9, 13, 16, 22, 23, 31, 34], а вместе с центральным квадратом они определяют всего лишь треть пространства проекта. Что же можно сказать об остальных двух треугольниках? Самые интересные ячейки расположены по углам, где известны и объекты, и методы.

На рис. 2.7 четыре угловых квадрата заполнены концепциями программирования, ассоциированными с ними. Например, если метод объекта вызывает сам себя, то речь идет о шаблоне *Recursion* (Рекурсия). Эту ситуацию можно описать с помощью листинга 2.7, в котором рекурсия возникает в методе `countDown()`. Все это легко и просто, но что можно сказать об остальных трех концепциях?

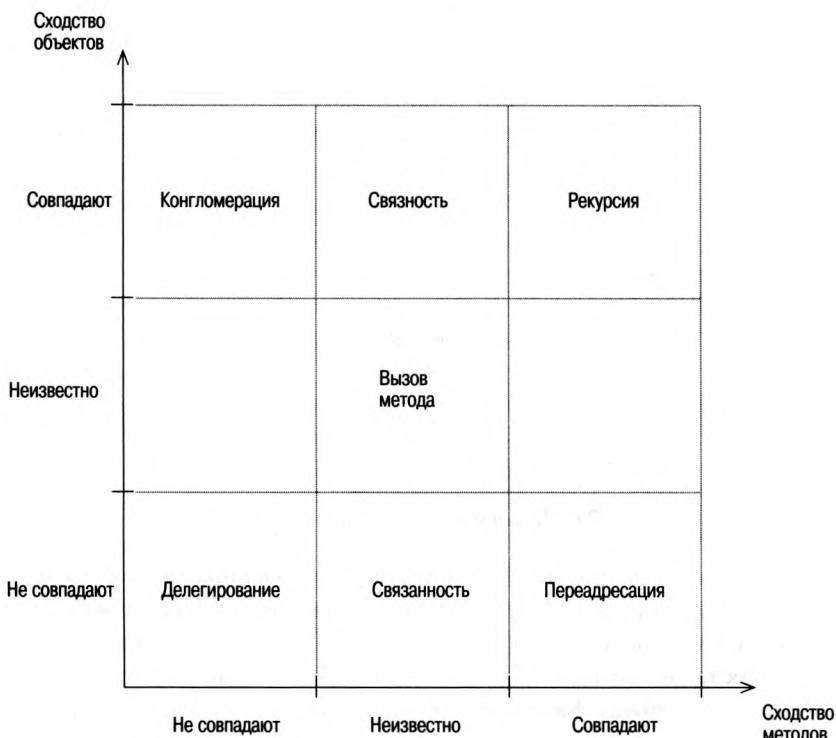


Рис. 2.7. Простое пространство проекта с элементарными шаблонами проектирования

Листинг 2.7. Пример рекурсивного вызова метода в языке Java

```

1  class Timer {
2      public void countDown(int counter) {
3          if (counter > 0) {
4              this.countDown(counter--);
5          } else {
6              // Эту ветвь пока игнорируем
7          }
8      };
9  };

```

В противоположном углу мы видим шаблон *Delegation* (Делегирование), название которого встречается во многих проектах программного обеспечения. Здесь мы дадим точное определение зависимости “вызов метода” между двумя разными объектами и разными методами. Соответствующий пример показан в листинге 2.8. Потому мы использовали слово “делегирование”? Потому что вызов метода представляет собой делегирование части работы другому методу в другом объекте. Это все равно что сказать “Сделай это, чтобы я мог сделать свою работу”. Делегирование — это поручение отдельных заданий, которые могут быть, но часто на самом деле не являются, частями собственного задания метода. Представьте себе генерального директора компании, делегирующего вице-президентам ответственность за работу разных отделов. Генеральный директор имеет задание “руководить компанией”, но у каждого из вице-президентов задание называется по-своему: “управлять кадрами”, “обеспечить финансовое благополучие”, “исследовать и внедрять информационные технологии”. Генеральный директор поручает вице-президентам задания, которые не похожи на его собственное, но каждое из них является очень важным для успешного выполнения задания генерального директора.

Листинг 2.8. Пример вызова метода с помощью делегирования в языке C++

```
1 class VicePresidentOfSales {
2     public:
3         void increaseQuarterlySales();
4     };
5
6 class CEO {
7     VicePresidentOfSales vpOfSales;
8     void increaseProfits() {
9         vpOfSales.increaseQuarterlySales();
10    };
11 }
```

Сравните это с нижним правым квадратом с меткой *Redirection* (Переадресация). Здесь вызывающий метод просит другой объект выполнить *ту же самую* работу, которую должен был выполнить сам. Он переадресует часть своей рабочей нагрузки другому объекту, но эта часть очень похожа на его собственное задание.

Пример переадресации приведен в листинге 2.9. Представьте себе сборочную линию, разветвляющуюся на параллельные ветви, по которым перемещаются многочисленные детали, над которыми выполняется конкретная операция, а затем результаты объединяются в одно целое на основном конвейере для продолжения обработки. Допустим, что ваша компания производит автомобили, а вы отвечаете за их покраску. Из-за продолжительной сушки последовательная покраска автомобилей приводит к большим простоям. Вместо того чтобы красить автомобили по одному, вы нанимаете несколько команд и организовываете пункты покраски для каждой из них. Теперь вам остается только перемещать автомобили к тем пунктам, которые закончили покраску предыдущей машины. Ваша задача — покрасить автомобили. Работа пунктов покраски также заключается в том, чтобы автомобили были покрашены. Их задача, которая является частью вашей

задачи, формулируется точно так же. Ответственность за распределение работы лежит на вас, но это только часть вашей основной задачи: обеспечить покраску автомобиля.

Листинг 2.9. Пример вызова метода с помощью переадресации в языке Objective-C

```

1  @interface Painter {
2      -(void) paintCar: (Car) theCar;
3  @end
4
5  @interface PaintShopManager {
6      Painter subpainter;
7  }
8      -(void) paintCar: (Car) theCar;
9  @end
10
11 @implementation PaintShopManager
12     -(void) paintCar: (Car) theCar {
13         [subpainter paintCar:theCar];
14     }
15 @end

```

Распределение заданий — это часть реализации, а сходство вашего задания и задачий ваших подчиненных заключается в дроблении работы при переадресации в противоположность делегированию. В совокупности шаблоны *Redirection* и *Delegation* образуют основу связности объектов.

Осталось проанализировать верхний левый квадрат с меткой *Conglomeration* (Конгломерация). Это обратная сторона рекурсии, а в совокупности они образуют основу для связности. Конгломерация — это объединение отдельных элементов в одно целое — в данном случае это объединение подзадач, похожих на главную задачу (как в шаблоне *Delegation*), но работу выполняет только один объект. В этом шаблоне нет никаких других объектов, которым можно было бы делегировать или переадресовать работу, поскольку задание предназначено только для одного объекта, но разделено на более мелкие части, которые обрабатываются разными методами одного и того же объекта. Это делается для повышения читабельности, гибкости и группировки атомарных функций в более сложные сущности. В любом случае в этом шаблоне задействован один объект и отдельные методы, как показано в листинге 2.10, который представляет собой переделанный вариант примера *Timer* из листинга 2.7.

Листинг 2.10. Пример вызова метода с помощью конгломерации в языке Java

```

1  class Timer {
2      void goDing();
3
4      public void countDown(int counter) {
5          if (counter > 0) {
6              // Эту ветвь пока игнорируем
7          } else {
8              this.goDing();
9          }
10     };
11 }

```

Эти четыре фундаментальные концепции — *Recursion*, *Delegation*, *Redirection* и *Conglomeration* — являются четырьмя элементарными шаблонами проектирования. Да, они просты. Да, эти шаблоны программисты ежедневно используют рефлексорно, но в этом-то и дело, не так ли? Они используют их механически, без размышлений, и каждый из них сталкивается с последствиями для дальнейших реализаций и проектирования. Если вы новичок в программировании, эти концепции еще не стали для вас рефлексорными, еще не усвоились и у вас может быть много вопросов о них: “Когда их использовать?”, “Чем они хороши?”, “С какими другими концепциями они связаны?” Это именно тот тип племенной мудрости, которую призваны выражать шаблоны проектирования. Более подробное обсуждение этих четырех элементарных шаблонов проектирования содержится в каталоге, прилагаемом к этой главе. Он не повредит и опытным программистам — они найдут в нем много неожиданного.

Что можно сказать о двух оставшихся ячейках нашей сетки, в которых сходство объектов неизвестно, а сходство методов известно? Ну, этого никто не может сказать определенно. Если вы найдете программную концепцию или аспект проектирования, связанные с двумя этими ячейками, то сможете добавить их в коллекцию EDP. В этом нет ничего невозможного. Среди новых путей, по которым идут разработчики, используя готовые шаблоны, некоторые пути могут оказаться полезными и привести к новой семантике. Если эти новые методы окажутся достаточно полезными, то в новых поколениях языков программирования они обязательно станут базовыми свойствами языка. Вполне возможно, что открывателем новых путей окажетесь именно вы. А пока мы просто отбросим центральный столбец таблицы, свойства которого еще исследуются, и центральную строку, о которой вообще ничего невозможно сказать. В итоге остается четыре ячейки в таблице 2×2, как показано на рис. 2.8.

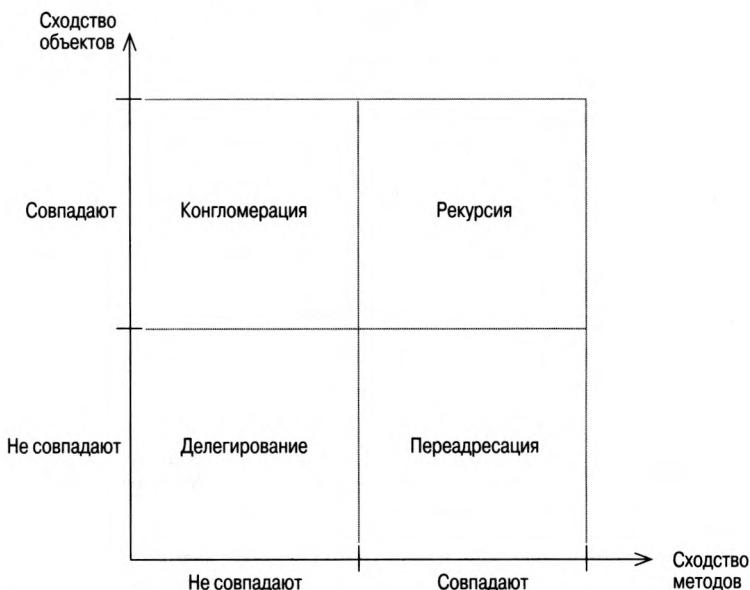


Рис. 2.8. Четыре первых элементарных шаблона проектирования

Интересно было бы вывести эти простые концепции из основных принципов теории программирования. Как только мы начнем экспериментировать с типами объектов, концепции станут намного сложнее, хотя теория останется довольно простой. Посмотрим, что произойдет, если мы добавим третью ось.

На рис. 2.9 показаны три оси. Там, где нас до сих пор вообще не интересовал тип объекта, мы теперь можем работать с четырьмя ячейками, определенными ранее для оси сходства типов объектов. В данном случае ячейка, соответствующая неопределенной ситуации, не представляет интереса, поэтому мы будем ее игнорировать. Сосредоточим свое внимание на срезе справа, определенном с помощью фиксации сходства методов на значении “совпадают”, как показано на рис. 2.10. Остальная часть пространства будет полнее рассмотрена в главе 5.

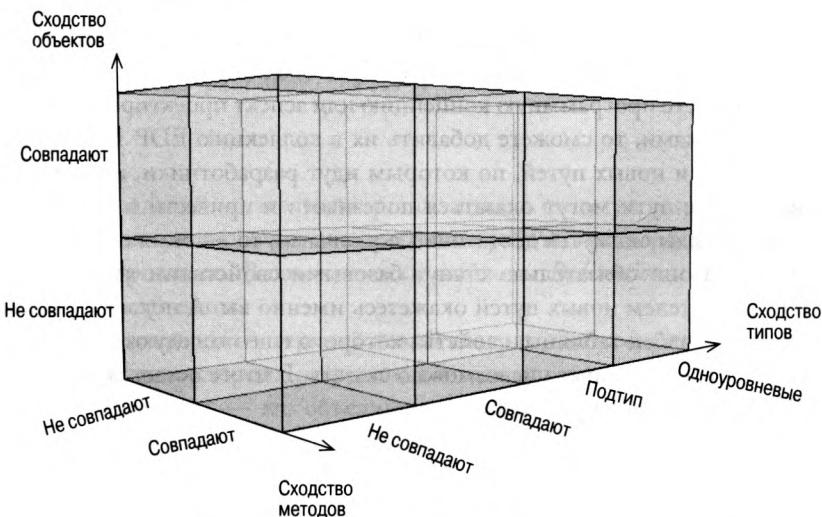


Рис. 2.9. Трехмерное пространство шаблонов проектирования

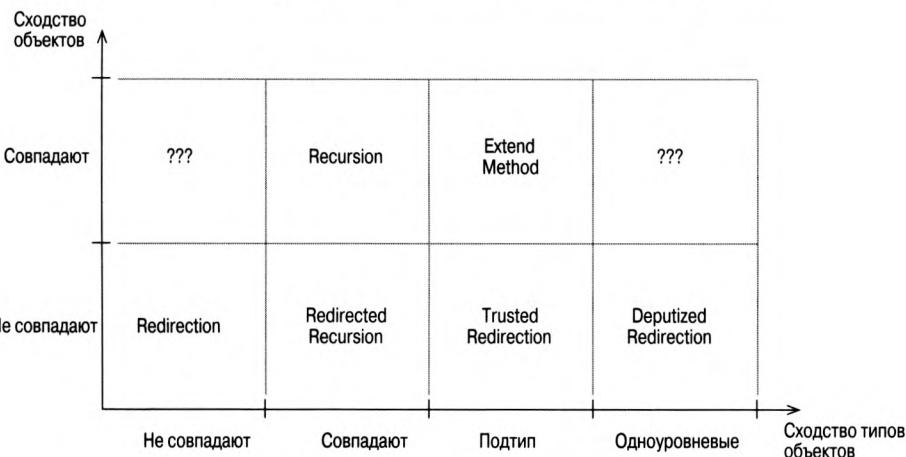


Рис. 2.10. Пространство проектирования с фиксированной координатой сходства методов

Мы можем найти место для шаблона *Recursion* в этой таблице, учитывая, что тип объекта не может *полностью* отличаться от типа аналогичного объекта, вызывающего метод.¹ Однако определение шаблона *Recursion* гласит, что должен вызываться только тот же самый метод, т.е. и объекты, и типы должны быть одинаковыми.

Аналогично мы можем найти правильное место для шаблона *Redirection* в данной таблице, поняв, что он лучше всего описывает ситуации, в которых и объекты, и типы являются разными. Это наиболее общая форма шаблона *Redirection*. Что произойдет, если мы сделаем еще один шаг и сохраним различия между объектами, но сделаем их однотипными? Это обычное проектное решение, в котором один объект передает запрос другому объекту того же типа и формирует цепочку таких объектов, разделяя между ними задачу. Поскольку при этом используется сочетание аспектов шаблонов *Redirection* (другой объект) и *Recursion* (тот же самый тип), мы придумали чрезвычайно точное название — *Redirected Recursion* (Переадресованная рекурсия)².

Продолжим исследование шаблона *Redirection* и немного изменим отношение типа. Если мы переадресуем задачу, выполняемую методом, другому объекту, имеющему другой тип, который в то же время является подтипом типа текущего объекта, мы сможем образовать целое семейство типов, которое обрабатывает вызов полиморфно. Здесь полиморфизм упоминается в нашем изложении впервые. Ранее мы говорили о вызовах между однородными (одинаковыми) типами и неоднородными (разными) типами, но не уточняли последнюю категорию. Теперь на сцене появляется новый уровень объектно-ориентированного проектирования.³ Этот тип отношений мы назовем просто *Subtype* (Подтип).⁴

Для разных типов можно ввести еще одну категорию и задать зависимость *Sibling* (Одноуровневые) между типами. Отношение *Sibling* возникает, когда типы задействованных объектов имеют общий подтип и ни один из них не является супертипов другого. Мы перемещаем вызов метода вверх по иерархии типов, а затем переходим по ограниченной надежной ветви вниз по этому дереву. Этот вариант шаблона называется *Deputized Redirection* (Делегированная переадресация).

Вернемся к шаблону *Recursion* и переместимся на один квадрат вправо, в котором экземпляры объектов совпадают, но между типами установлено отношение подтипа. Прочитайте это еще раз. Действительно целесообразно требовать, чтобы объект был тем же самым и вызов метода передавался от объекта к нему самому, но в этом вызове метода задействовать два типа объектов. Фактически это супертип.

В частности, супертип открывает доступ к своей реализации метода из подтипа. В языке Java для этого используется ключевое слово `super`. В языке C++ используется явный механизм разрешения видимости типов, имеющий синтаксис `Supertype::`. В других языках

¹ Должен ли объект быть одним и тем же? Нет! Этую ситуацию мы рассмотрим позднее.

² Мы могли бы использовать название *Recursive Redirection*, но для большинства людей этот термин уже означает цикл между двумя объектами. Кроме того, существует формальная причина, которую вы можете найти в приложении.

³ Если вам не знаком термин “полиморфизм”, начните с элементарного шаблона проектирования *Inheritance*. Это позволит вам лучше понять данную концепцию.

⁴ Напомним, что зависимости направлены от вызывающего метода к вызываемому; в данном случае вызывающий метод определен в подтипе типа, содержащего вызываемый метод.

используются разнообразные механизмы, но все они делают одно и то же — предоставляют объекту доступ к реализации метода в супертипе. Насколько это полезно? Рассмотрим ситуацию, в которой вы хотите расширить функциональность метода, но не полностью заменить ее новой функциональностью. Вы можете вывести подкласс типа, заместить метод, а затем завернуть вызов в исходный метод в дополнительном коде. Как и следовало ожидать, такой элементарный шаблон проектирования называется *Extend Method* (Расширение метода).

Обратите внимание на то, что в предыдущем изложении мы взяли уже существующую, хорошо известную концепцию и сделали всего один шаг, меняя то одну, то другую деталь. Результат такой простой модификации позволил создать удивительно широкую коллекцию концепций и элементов проектирования. Посмотрите на множество UML-диаграмм для предыдущих восьми элементарных шаблонов проектирования, и вы убедитесь, что они охватывают многие основные концепции. Рассмотрим идеи, связанные с шаблонами *Recursion* и *Deputized Redirection*, приведенными на рис. 2.11 и 2.12. Они выглядят совершенно непохожими, поскольку в пространстве проектирования отдельного вызова метода находятся друг от друга в трех шагах.

И снова в нашем пространстве остаются два загадочных квадрата. На этот раз мы не будем их игнорировать, а обсудим, что они означают концептуально. Первый квадрат, расположенный слева, соответствует одинаковым (или сходным) объектам, но на каждом конце зависимости “вызов метода” находятся совершенно разные типы. Как это может быть? Не знаю. Здесь можно искать какие-то эзотерические зависимости между типами, но это не имеет большого смысла. Пока это совершенно неизвестная область, но, возможно, в будущем в каком-нибудь языке программирования это сочетание получит какой-то смысл. Например, язык ВЕТА [26] имеет уникальную конструкцию, которая создает зеркальное отражение супертипа, создавая сходство с супертипов, а не с подтипов. Однако этот механизм крайне редкий и в нашем изложении не используется. Возможно, мы добавим его в дальнейшем, если он окажется интересным и полезным элементом проектирования.

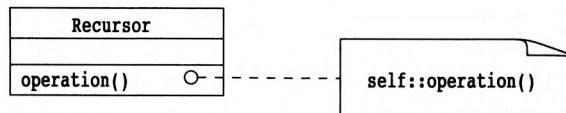


Рис. 2.11. UML-диаграмма для шаблона Recursion

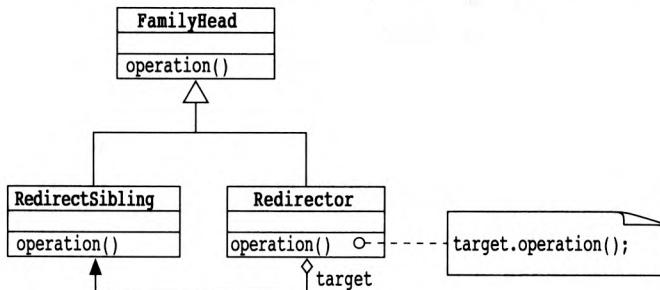


Рис. 2.12. UML-диаграмма для шаблона Deputized Redirection

Аналогичная ситуация возникает в квадрате, расположеннном далеко справа. Он также соответствует двум одинаковым (или сходным) объектам, но теперь между ними существует зависимость типа *Sibling*. И вновь приходится констатировать, что ни один язык программирования, известный мне, не предоставляет такой возможности. Эта область открыта для интерпретаций и экспериментов. Самое приятное заключается в том, что мы можем предсказывать свойства этих неописанных ячеек в пространстве проектирования, как в периодической таблице элементов Менделеева.

Мы описали больше половины элементарных шаблонов проектирования, связанных с вызовом методов. Они будут представлены полностью в справочном разделе. Просто удивительно, как много концепций можно выразить всего лишь одним вызовом метода! Всего один вызов метода. Это все, о чем мы говорили, а какой широкий спектр компонентов проектирования с этим связан!

2.3. Главные элементарные шаблоны проектирования

До сих пор мы обсуждали только вызовы методов, но существуют еще три формы зависимости, с которыми можно работать: использование поля, изменение состояния и связность. Что можно сказать о них? В настоящий момент они хорошо исследованы, детально описаны и задокументированы. Вызовы методов были рассмотрены в первую очередь, потому что они образуют самое маленькое и самое простое пространство проектирования. Остальные пространства проектирования немного сложнее. Однако следует рассмотреть несколько фундаментальных концепций, образующих ядро объектно-ориентированного программирования, как с формальной, так и с pragматической точек зрения.

Для начала представим себе идею “создания объектов”. Это одна из характерных особенностей объектно-ориентированного программирования, которая легла в основу отдельной парадигмы программирования. Для справки обратитесь к элементарному шаблону проектирования *Create Object* (Создать объект). С высоты прошедших трех десятилетий сейчас даже странно, что когда-то эта идея встречала возражения, но она должна была преодолеть много препятствий. Одно из возражений заключалось в том, что все, что можно реализовать в объектно-ориентированном программировании, можно реализовать и в процедурном программировании. Несмотря на то что формально это правда, все, что осуществимо в любой высокоуровневой системе, является осуществимым в процедурном языке, ассемблере и даже в двоичном коде; в противном случае программы просто бы не работали — объектно-ориентированное программирование позволило легко внедрить определенные принципы. Один из них реализован в шаблоне *Create Object*, создающем зависимость между отдельным объектом и отдельным типом в один и тот же момент времени.

Как, создав объекты, заставить их общаться между собой? Очевидно, с помощью вызовов методов. Хорошо, но как они найдут один другого? Мы могли бы статически установить все связи между объектами, но это слишком неудобно, поскольку в таком случае система не сможет реагировать на ввод данных в ходе выполнения программы.

Программисты должны предвидеть все возможные варианты использования системы, включая такие детали, как обеспечение максимального объема памяти, необходимого для хранения всех данных независимо от фактически доступного объема. Для того чтобы решить эту проблему, мы должны позволить объектам получать доступ к другим объектам в ходе выполнения программы. Этую задачу решает элементарный шаблон проектирования *Retrieve* (Найти), динамически создающий связи между объектами.

Для того чтобы создать объект, необходимо знать его тип. Часто типы, доступные нам, подходят не полностью или не совсем точно. Вместо переписывания типа с самого начала мы хотели бы использовать определенную заготовку. Возможность повторного использования типов стала возможной благодаря элементарному шаблону *Inheritance*, который относится к ядру коллекции EDP. Наследование — это разновидность зависимости между типами, которую иногда называют порождением подтипа (subtyping). Она создает связи между базовым типом и типом, зависящим от него и использующим его основные функциональные свойства (методы) и состояние (поля). Это позволяет создать тип для правильного повторного использования крупных частей логики и данных.

Другим элементарным шаблоном проектирования, связанным с зависимостями между типами, является шаблон *Abstract Interface* (Абстрактный интерфейс). Он образует зависимость между двумя типами, которая немного отличается тем, что заранее известен тип только на одном из ее концов. Он использует метод, как мост между двумя типами. Если тип объявляет метод абстрактным, он не обязан — а в строгом смысле ему даже запрещено — создавать реализацию метода. Этот элементарный шаблон проектирования подразумевает, что в определенной точке другой тип унаследует его методы и поля и создаст соответствующую реализацию метода. Этот тип еще неизвестен, но готов к использованию. До тех пор этот тип считается неполным.

Четыре элементарных шаблона проектирования — *Create Object*, *Retrieve*, *Inheritance* и *Abstract Interface* — позволяют создавать объекты с определенными гарантиями, связывать один с другим во время выполнения программы и декларировать обещания, касающиеся будущего, еще неопределенного типа. В совокупности они образуют основу объектно-ориентированного программирования. Вместе с элементарными шаблонами проектирования, связанными с вызовом метода, такими как *Delegation* и *Recursion*, они обеспечивают надежную точку опоры для развития проектирования как научной дисциплины, а также для правильного и систематического использования надежных строительных блоков. Почему же мы не видим, что можно сделать с помощью всего нескольких шаблонов?

2.4. Заключение

В этой главе содержится введение в элементарные шаблоны проектирования. В ней кратко описаны проблемы, которые решают эти шаблоны, и их источники. Здесь показано, как такие шаблоны, как *Decorator*, можно представить в терминах более мелких шаблонов. Это привело нас к заключению, что для того, чтобы лучше описать более абстрактные шаблоны проектирования, изложенные в стандартной литературе, необходимо разработать более детальные шаблоны. Мы представили теорию объектно-

ориентированного программирования в минималистской форме и показали, как могут возникать немногочисленные зависимости. Эти зависимости образуют основу для определения наименьших шаблонов. Одна из этих зависимостей — зависимость “вызов метода” — является предметом исследования в нашей книге. Мы увидели, что каждый вызов метода имеет контекст, определенный тремя простыми видами информации — сходством между объектами, типами и методами. Они порождают пространство проектирования, в котором находятся хорошо известные концепции программирования. Три остальных вида зависимостей — использование поля, изменение состояния и связность — образуют собственные пространства проектирования, которые также являются предметом отдельных исследований.

Пространство проектирования, связанное с вызовом метода, исследовано слабо. Для многих элементарных шаблонов проектирования приведены примеры программ и показана связь между ними. В заключение мы описали ядро элементарных шаблонов проектирования, определяющее множество фундаментальных концепций объектно-ориентированного программирования.

Теперь у нас есть солидная база знаний, позволяющая понять важность и полезность элементарных шаблонов проектирования, и мы готовы приступить к работе с ними. В главе 3 будет показан полезный способ графического изображения шаблонов, позволяющий визуализировать их взаимодействия.

Описание шаблонов

Прежде чем перейти к изучению действий над элементарными шаблонами, рассмотрим новую графическую систему обозначений Pattern Instance Notation (PIN). Система PIN помогает визуализировать некоторые из обсуждаемых концепций.

В этой главе приведены неформальное описание системы PIN и способ ее использования в книге. Если вас заинтересовала эта информация или применение системы PIN в проектировании программного обеспечения, прочтайте работу “The Pattern Instance Notation: A Simple Hierarchical Visual Notation for the Dynamic Visualization and Comprehension of Software Patterns” [36], в которой система PIN описана полностью.

3.1. Основы

Система PIN предназначена для визуального представления концепций и идей, лежащих в основе шаблонов. Она позволяет быстро и просто описывать шаблоны проектирования и способы их взаимодействия. Выбор названия “PIN” обусловлен ее полезностью для демонстрации экземпляров шаблонов или концепций в других системах графических обозначений, таких как UML. Именно в этих диаграммах они использовались изначально.

В главе 2 уже упоминался раздел “Участники” спецификации шаблона проектирования. Этот раздел является неотъемлемой частью любого шаблона проектирования.

Иногда участников называют *ролями* шаблона проектирования, поскольку каждый участник играет определенную роль в организации шаблона. Когда мы говорим “участник *ConcreteDecorator* шаблона *Decorator*”, мы имеем в виду “класс реализации, играющий роль *ConcreteDecorator* в шаблоне *Decorator*”. Роли также являются абстракциями, поскольку они присваивают имя и устанавливают ограничения на реализацию или свойства шаблона, действующие как участники.

Шаблон проектирования можно сравнить с пьесой, например “Гамлет”. Гамлет — это такая же роль, как и Офелия, Розенкранц и Гильденстerner. Эти роли описывают обязанности актеров и указывают, как им действовать. Актёр — это участник постановки спектакля, и все актеры сотрудничают друг с другом при его формировании.

Если шаблон проектирования похож на пьесу, то экземпляр шаблона — на спектакль по этой пьесе. Спецификация шаблона похожа на сценарий: она описывает роли, а также их диалоги и взаимодействия. Участники сценария аналогичны актерам, пробующимся на эти роли.

В языке UML есть специальный графический компонент — элемент взаимодействия (*collaboration element*), описывающий такие ситуации. Пример его использования для описания шаблона *Decorator* показан на рис. 3.1. Это неплохой и очень гибкий элемент, но он страдает от нескольких недостатков.

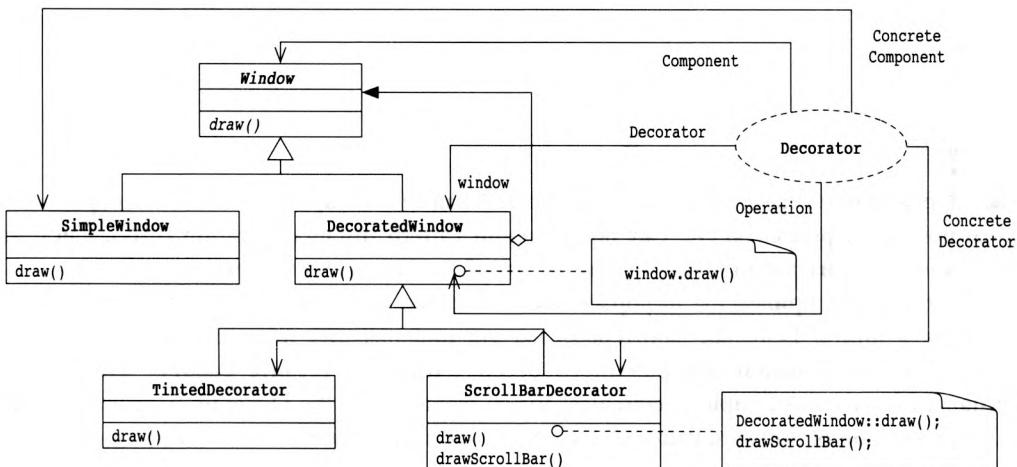


Рис. 3.1. Диаграмма взаимодействий в UML

Во-первых, на практике его довольно трудно использовать в каких-либо диаграммах, кроме тривиальных. Он просто добавляет информацию в диаграмму UML. Чем чаще вы его используете, тем сложнее становится диаграмма. Такие абстракции, как шаблоны проектирования, призваны уменьшить сложность и уровень детализации, чтобы проектировщик мог оперировать понятиями на более высоком уровне абстракции. Камнем преткновения для взаимодействий в диаграммах UML является то, что они не способны уменьшать их сложность. Они не могут уменьшить объем отображаемой информации, как это должно быть на более высоком уровне абстракции.

Во-вторых, элемент взаимодействия также требует, чтобы диаграмма UML была полной. Может показаться странным, что мы называем это недостатком, но оказывается, что существует множество ситуаций, в которых необходимо обсуждать способы взаимодействия между шаблонами, вообще не создавая диаграмм для всех элементов, участвующих в их реализации.

Другим распространенным обозначением в шаблонах проектирования является конструкция “шаблон:роль”, предложенная в книге GoF [21]. Она является аннотацией, которую следует применять в дополнение к UML, и использует внешний флаг, имя которого состоит из шаблона и роли. При необходимости эту аннотацию можно применять к классу, пакету, полу или методу, как показано на рис. 3.2. Эта конструкция очень гибкая и прозрачна, но также не лишена недостатков.

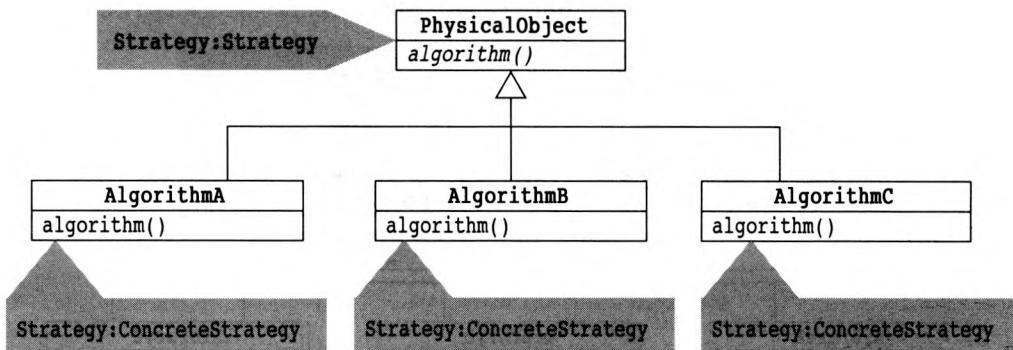


Рис. 3.2. Стратегия в виде дескриптора “шаблон:роль” в UML

Этот подход не так уж плох, если в диаграмме содержится только один шаблон, но что произойдет, если в диаграмме скрыт шаблон, состоящий из сотен и даже тысяч частей? Полюбуйтесь рис. 3.3.

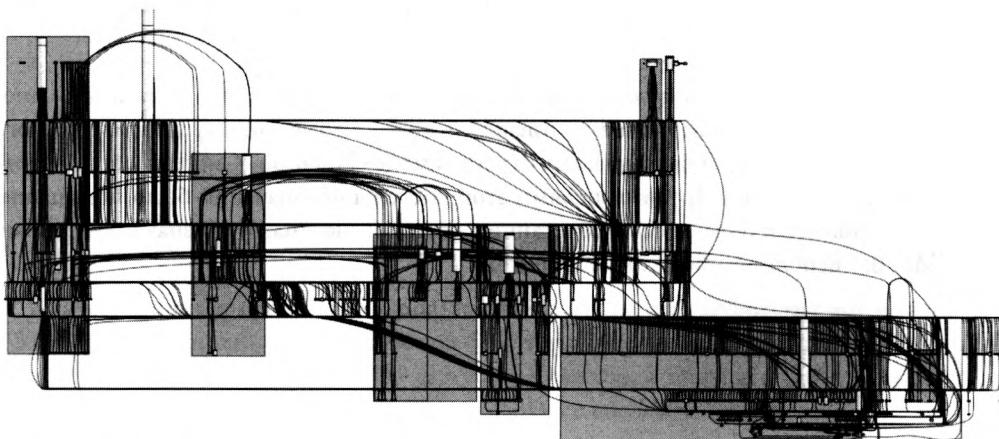


Рис. 3.3. Огромная диаграмма UML для относительно небольшой системы

На рис. 3.3 представлена реальная диаграмма UML из реального проекта. В данном масштабе ее невозможно прочитать, но, честно говоря, ее практически невозможно прочитать в любом масштабе. Если бы вы захотели ее распечатать, используя шрифт размером 10 пунктов, то вам понадобился бы лист бумаги размером 86×32 фута. Это много: примерно 3 290 листов бумаги стандарта Letter. В малом масштабе эту диаграмму невозможно прочитать, а в большом масштабе ее невозможно просмотреть. В этой диаграмме записано огромное количество шаблонов, информацию о которых можно извлечь из справочного документа системы.

В качестве более конкретного примера рассмотрим диаграмму UML, представленную на рис. 3.4. В этой диаграмме содержатся два экземпляра шаблона *Strategy*, но с помощью конструкции “шаблон:роль” мы не можем указать, какой фрагмент относится к какому экземпляру шаблона, даже в таком маленьком примере.

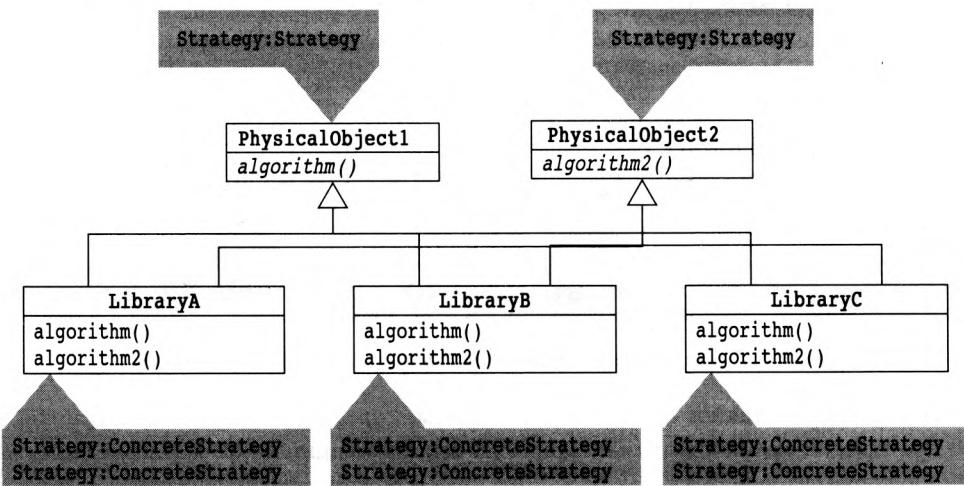


Рис. 3.4. Множественные экземпляры шаблона *Strategy* в виде дескрипторов “шаблон:роль” в UML

Это было первопричиной создания системы PIN: позволить ясно указывать множественные экземпляры шаблонов в отдельной диаграмме в виде сущностей первого класса. Поэтому система и была названа Pattern Instance Notation (Система обозначений экземпляров шаблонов). Другими целями этой системы обозначений было повышение простоты, гибкости и полезности в сочетании с другими системами обозначений, такими как UML, без использования их компонентов.

3.2. Компонент PINbox

Ядром системы обозначений PIN является компонент PINbox. Он изображает отдельный экземпляр шаблона и позволяет выбирать уровень детализации при его представлении. Начнем с простейшей формы компонента PINbox, а затем перейдем к более интересным вариантам.

3.2.1. Свернутый компонент PINbox

Свернутый компонент PINbox представляет собой просто метку с именем и полуожиженной двойной границей, как показано на рис. 3.5. Внутренняя граница представляет собой прямоугольник, а внешняя — прямоугольник с закругленными углами. По возможности граница закрашивается серым цветом. Метка внутри прямоугольника представляет собой имя шаблона.

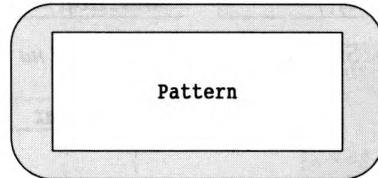


Рис. 3.5. Свернутый компонент PINbox

Это как раз то, что нам нужно, — простейшая форма компонента PINbox. Не волнуйтесь, мы его еще развернем. Даже сейчас мы можем сделать с ним полезную работу. Мы можем использовать его как аннотацию к диаграмме UML или какой-нибудь другой диаграмме, просто нарисовав стрелку от компонента PINbox к тому элементу диаграммы, который с ним наиболее тесно связан, как на рис. 3.6. Как показывает опыт использования графических систем обозначения, прямоугольники и линии являются простейшими элементами.

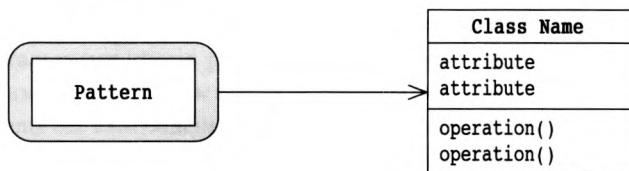


Рис. 3.6. Свернутый компонент PINbox в качестве аннотации

В качестве примера реального использования компонента PINbox рассмотрим рис. 3.7 и 3.8. Пример использования шаблонов *Singleton* и *Abstract Factory* показывает, как применить систему обозначений PIN в диаграмме классов, а пример с шаблоном *Template Method* демонстрирует его применение в диаграмме последовательностей.

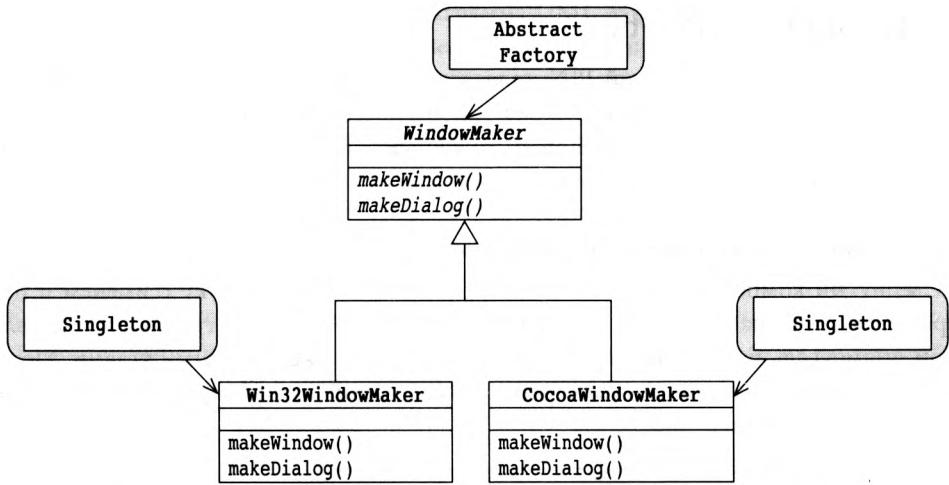


Рис. 3.7. Шаблоны Singleton и Abstract Factory на диаграмме классов

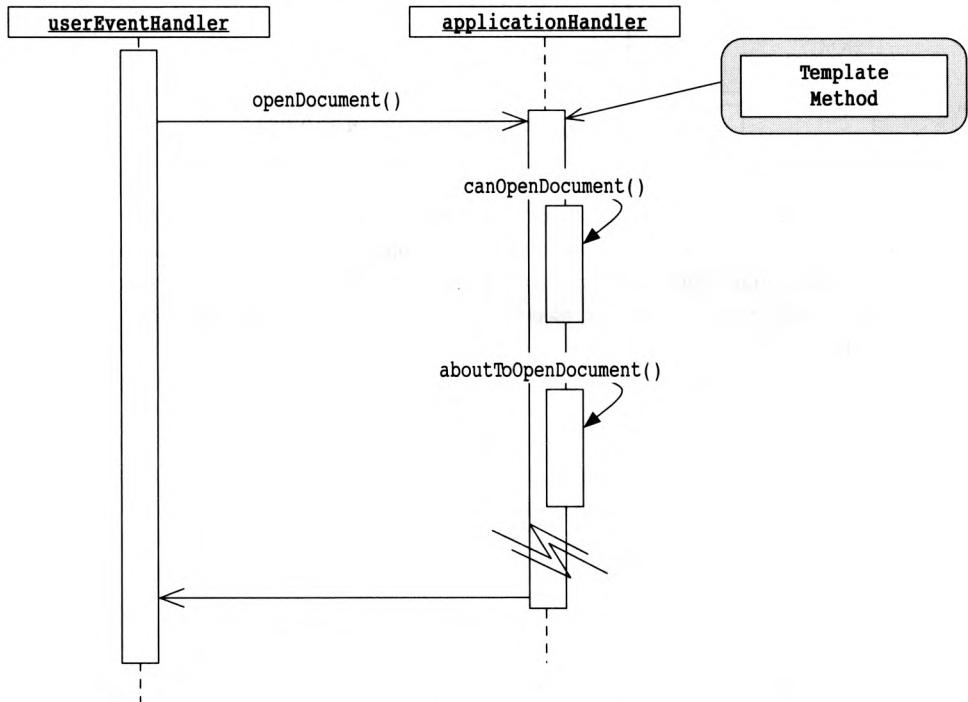


Рис. 3.8. Шаблон Template Method на диаграмме последовательностей

Свернутые компоненты PINbox лучше всего использовать для мнемонического напоминания о существовании в системе шаблона с единственным определенным функциональным свойством. Это неформальный способ обозначения, но он полезен, особенно при создании эскизов нового проекта.

3.2.2. Стандартный компонент PINbox

Поднявшись на один уровень абстракции, мы получим стандартный компонент PINbox. Мы расширим полужирную серую границу свернутой формы, чтобы добавить в нее текст. В частности, мы добавим роли, образующие шаблон, названный в середине, как показано на рис. 3.9. Роли названы в соответствии с правилами, приведенными в разделе 2.2.1. Добавив роли, можно выразить значительно более детальные связи. Например, поскольку каждый шаблон имеет точно определенный и уникальный набор ролей, мы можем соединить каждую роль с элементами диаграмм классов и последовательностей UML, как показано на рис. 3.10 и 3.11.



Рис. 3.9. Стандартный компонент PINbox

Выбор расположения имен ролей вокруг границы предоставляет проектировщику; однако лучше всего они выглядят на последней диаграмме. Упорядочение имен облегчает чтение последней диаграммы.

Этот компонент выглядит так же устрашающе, как элементы взаимодействия на диаграмме UML, представленной на рис. 3.1, но компоненты PINbox могут делать то, чего не могут делать элементы взаимодействий. Обратите внимание на то, что на рис. 3.11 экземпляр шаблона *Flyweight* имеет роль *AbsFlyweight*, которая ни с чем не связана. В данном случае это очень хорошо, потому что мы просто пытаемся описать взаимодействия между остальными тремя ролями. Если бы мы попытались сделать это с помощью элементов взаимодействия и захотели сохранить всю информацию о ролях шаблона *Flyweight*, мы были бы вынуждены нарисовать стрелку, ссылающуюся на несуществующий компонент с неиспользуемым именем роли или, что еще хуже, направленную в пустоту. Ни один из этих вариантов не оптimalен. Благодаря компоненту PINbox мы не потеряем информацию и не загромоздим диаграмму без необходимости. Рассмотрим другой пример: допустим, что в проекте программного обеспечения есть два экземпляра шаблонов и необходимо указать, что один и тот же класс, выполняющий роль в одном экземпляре шаблона, выполняет другую роль в другом экземпляре шаблона, объединяя эти два экземпляра шаблона в общую структуру.

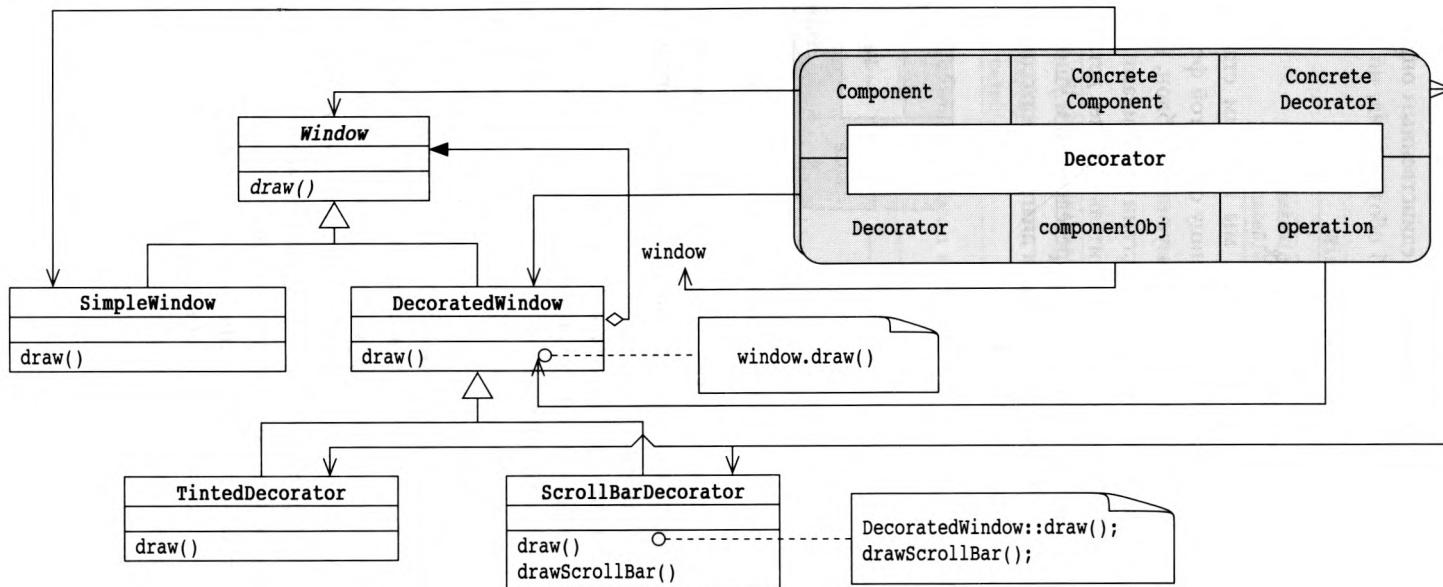


Рис. 3.10. Использование компонента PINbox в диаграмме классов UML

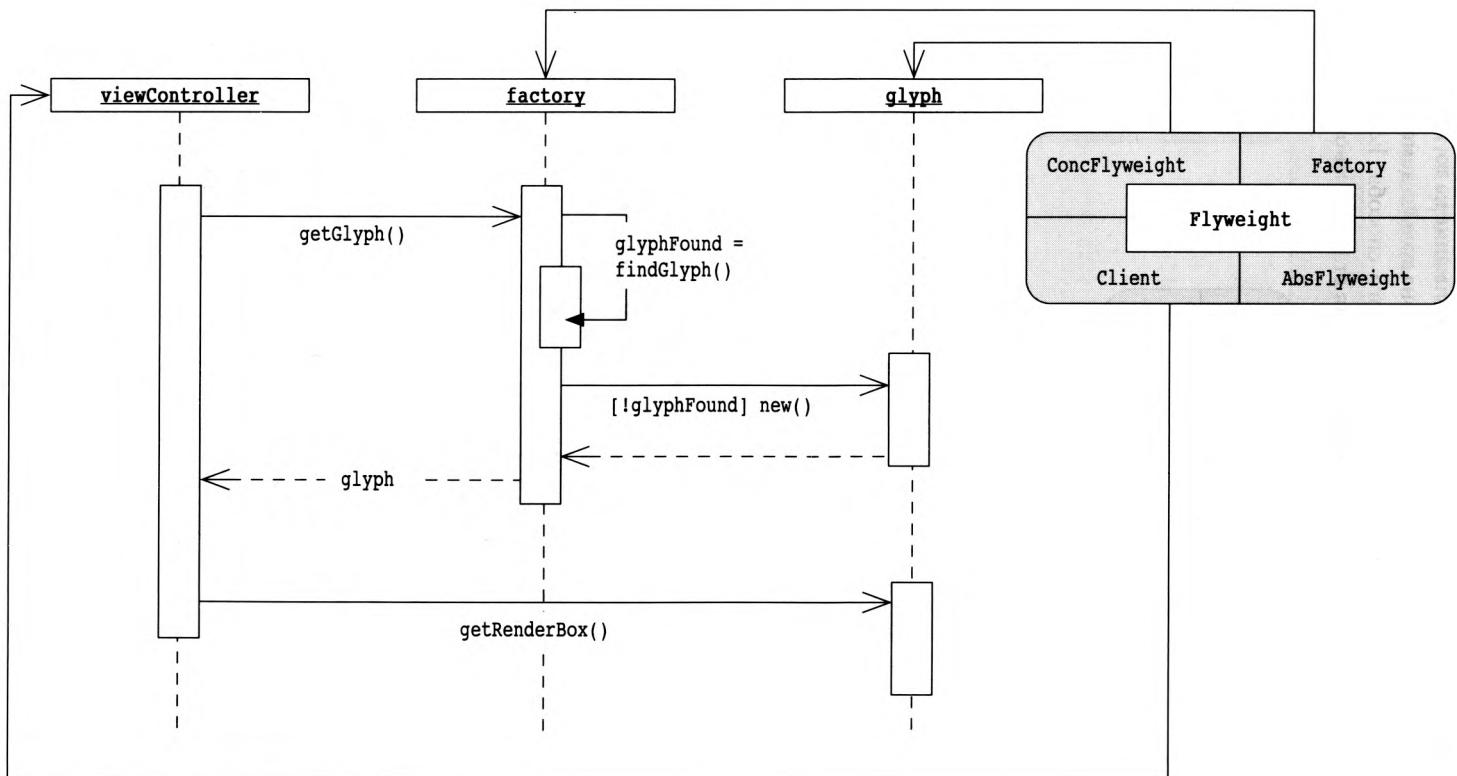


Рис. 3.11. Использование компонента PINbox в диаграмме последовательностей UML

Мы попытались сделать это на рис. 3.12. Обратите внимание на то, что эта диаграмма никак не использует систему обозначения языка UML; она только демонстрирует соединение между двумя шаблонами с помощью конкретного способа. Если мы соединим один с другим несколько компонентов PINbox, ситуация станет намного интереснее.

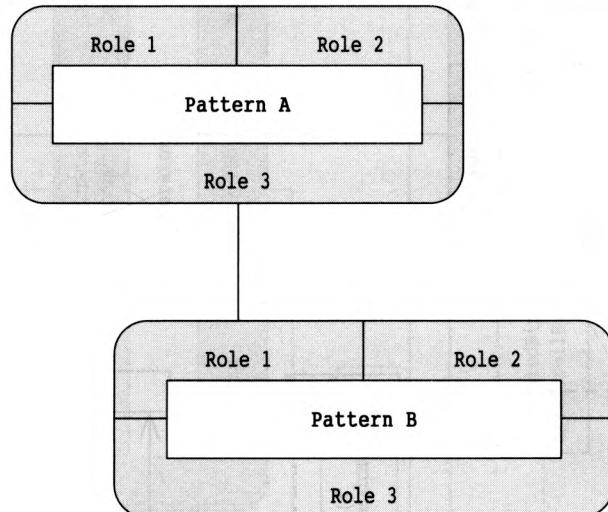


Рис. 3.12. Стандартные связи между ролями в диаграмме PIN

Здесь нет компонентов UML.

Здесь нет кода.

Здесь нет классов, методов, полей. Вообще нет ничего, кроме отношения между двумя шаблонами, т.е. двумя *концепциями*, и это отношение связывает их определенным образом. В следующей главе мы используем эту возможность для демонстрации того, как шаблоны можно объединять в новую концепцию.

Однако мы можем сделать больше.

3.2.3. Раскрытый компонент PINbox

Если мы проделаем с компонентом PINbox небольшую манипуляцию, он станет более гибким и мощным. Сначала распахнем прямоугольник в центре, где написано имя шаблона. Мы увидим нечто похожее на рис. 3.13. Новую пустую область можно использовать для рисования дополнительных компонентов PINbox. Зачем это нужно? Посмотрите на рис. 3.14. Здесь мы заполнили канву. Мы видим, что данный шаблон *Pattern* имеет пять ролей и может быть разделен на два более мелких подшаблона: *Subpattern A* и *Subpattern B*. Далее, мы ясно видим, как роли, указанные за пределами кольца, отображаются в роли внутренних подшаблонов. Если роль *Role 3* в подшаблоне *Subpattern A* исполняется той же сущностью, что и роль *Role 1* в подшаблоне *Subpattern B*, то эти два экземпляра можно заменить одним экземпляром шаблона *Pattern*. Иначе говоря, мож-

но немного повысить уровень абстракции. Вместо двух экземпляров можно управлять одним экземпляром шаблона.

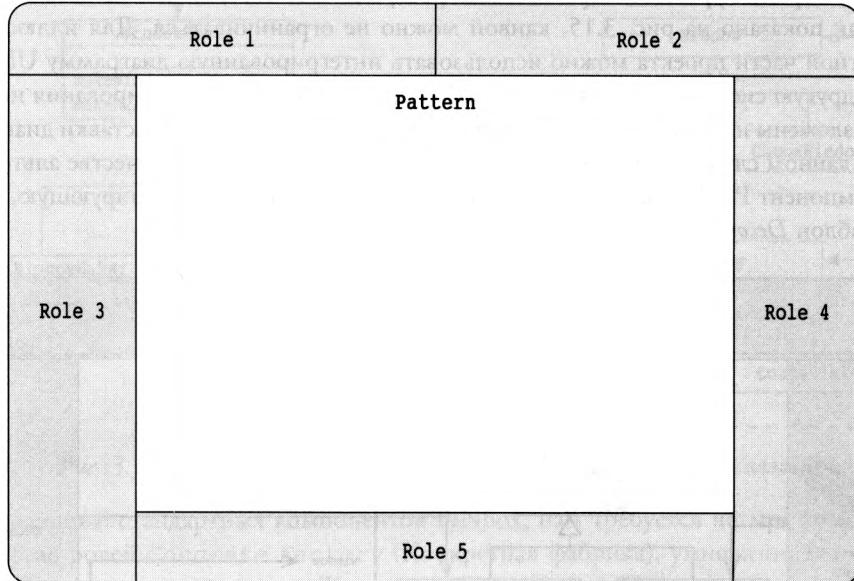


Рис. 3.13. Пустой раскрытый экземпляр PINbox

С другой стороны, совершенно очевидно, что вместе с экземпляром шаблона *Pattern* мы получаем экземпляр каждого из подшаблонов. Они могут быть скрытыми, но мы знаем об их существовании.

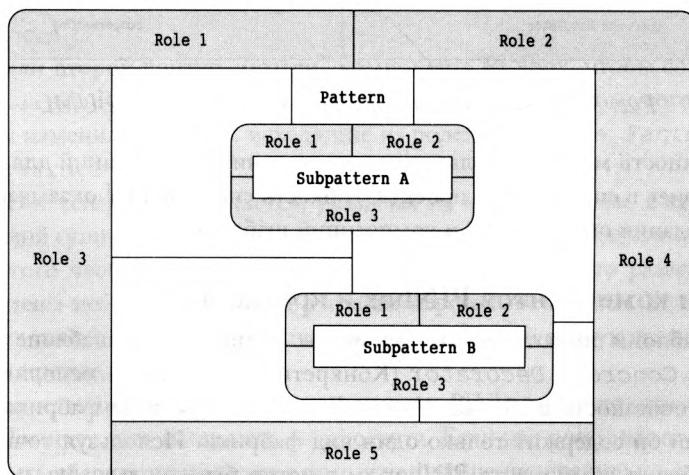


Рис. 3.14. Раскрытый экземпляр PINbox

Раскрытий компонент PINbox позволяет выявить иерархическую природу шаблонов до той степени, до которой мы сами пожелаем. Мы можем оставить один шаблон на самом верхнем уровне абстракции или раскрыть его, чтобы показать детали. Кроме того, как показано на рис. 3.15, канвой можно не ограничиваться. Для иллюстрации конкретной части проекта можно использовать интегрированную диаграмму UML или любую другую систему обозначений. Элементарные шаблоны проектирования не могут быть разложены на более мелкие подшаблоны, поэтому возможность вставки диаграммы UML в данном случае позволяет создавать прямые определения. В качестве альтернативы в компонент PINbox можно поместить диаграмму UML, демонстрирующую, например, шаблон *Decorator*.

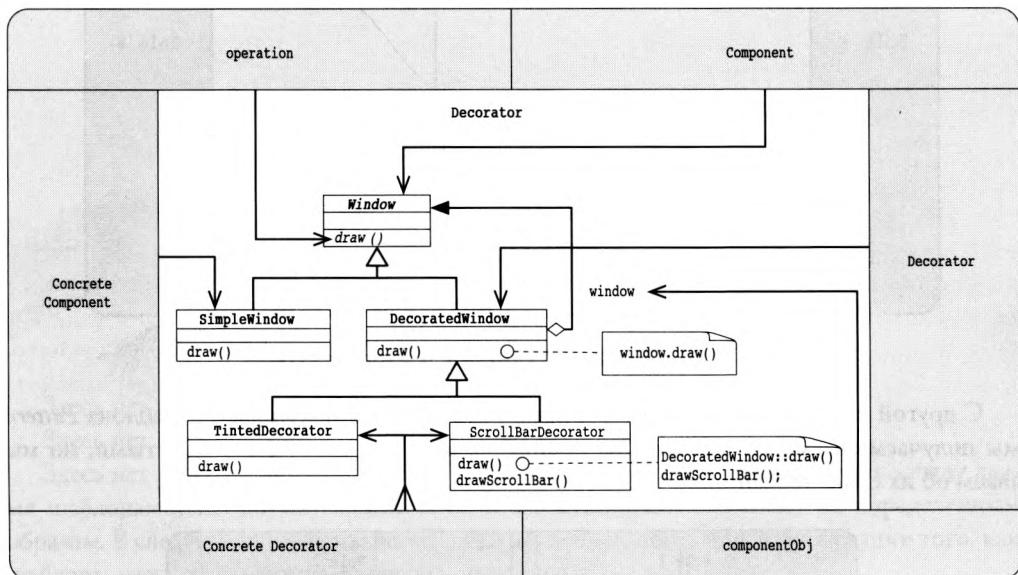


Рис. 3.15. Раскрытий экземпляр PINbox на языке UML

Эту возможность можно использовать для создания иллюстраций для студентов или для комментариев в справочнике проекта. Гибкость системы PIN оказывается очень полезной при создании определений и композиций шаблонов.

3.2.4. Стеки компонентов PINbox и кратность

Многие шаблоны имеют кратные элементы. Например, в шаблоне *Decorator* (Декоратор) роль *Concrete Decorator* (Конкретный декоратор) исполняют несколько участников, а полезность шаблона *Abstract Factory* (Абстрактная фабрика) значительно снижается, если он содержит только один вид фабрики. Используя точную интерпретацию определения компонента PINbox, мы могли бы использовать по одному компоненту PINbox для каждой комбинации элементов. Взгляните на базовую диаграмму шаблона *Abstract Factory*, приведенную на рис. 3.16. Для того чтобы правильно описать

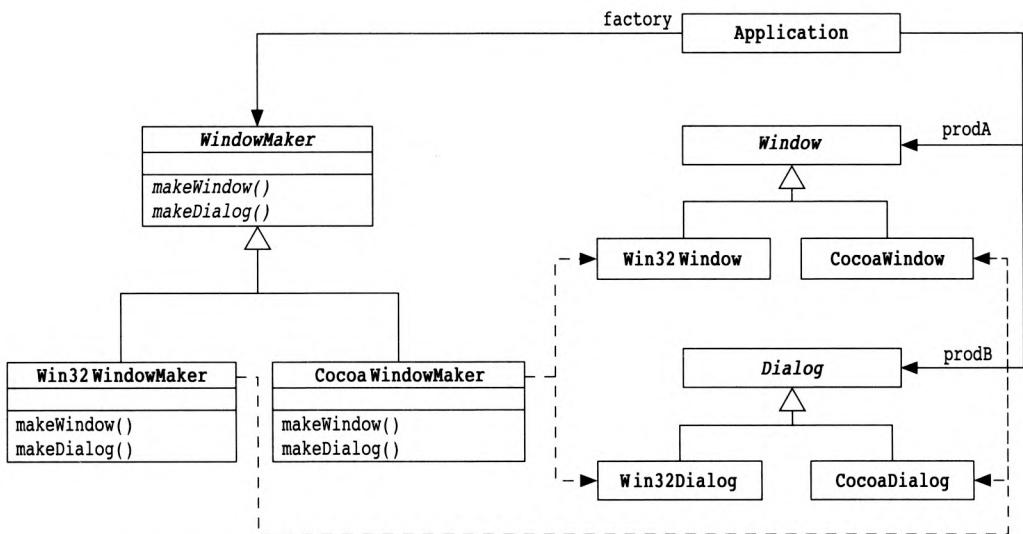


Рис. 3.16. Необходимость компонентов PINbox с кратными связями

его с помощью стандартных компонентов PINbox, нам требуется четыре компонента: количество ролей *Concrete Factory* (Конкретная фабрика), умноженное на количество ролей *Concrete Product* (Конкретная продукция). Пока мы говорим только о двух фабриках и двух видах продукции. А представьте себе, скажем, четыре фабрики и десять видов продукции. Среди сорока компонентов PINbox очень легко запутаться. Однако мы должны сделать так, чтобы любой проектировщик, просматривающий наши диаграммы, знал, что эти отдельные шаблоны проектирования просто являются частями некоторого шаблона проектирования более высокого уровня и тесно связаны между собой. Эти связи можно отобразить с помощью стека компонентов PINbox, как показано на рис. 3.17.

Мы добавили второй контур “позади” компонента PINbox, чтобы создать впечатление, что компоненты PINbox уложены в стек, как колода карт. Кроме того, обратите внимание, что мы изменили стрелки, исходящие из ролей *Concrete Factory*, *Concrete Product* и *Abstract Product*.

Конец стрелы теперь разветвляется, указывая на кратные связи, а затем линии расходятся к каждой сущности, которые ранее были представлены отдельным компонентом PINbox. Для того чтобы проиллюстрировать, где происходит это разветвление, в диаграмму добавлены небольшие кружки в точках пересечения линий. Если точка пересечения не имеет такого индикатора, значит, несколько ролей обслуживают один и тот же элемент, как и прежде.

В принципе, кратные связи внутри компонентов PINbox следует использовать с осторожностью. Помните, что наша цель — простота. Если две роли имеют кратные связи, мы должны быть абсолютно уверены в том, что отношение правильно определено с помощью обозначаемой абстракции. В данном случае три роли с кратными связями имеют смысл. Каждая роль *Concrete Factory* должна быть связана с ролью *Concrete*

Product, а роли *Abstract Product*, очевидно, должны быть связаны с соответствующими подклассами. В данном случае трудно запутаться.

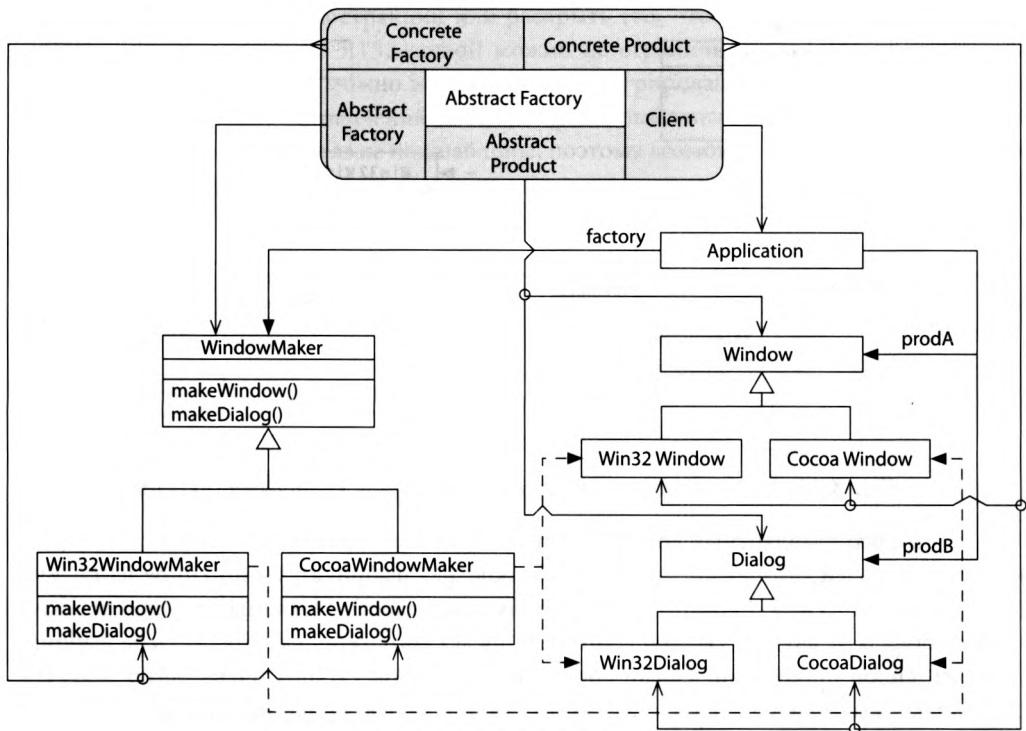


Рис. 3.17. Стек компонентов PINbox

Однако если существует несколько сущностей *Abstract Factory*, то каждая из них может породить новый стек компонентов PINbox. Вдумайтесь: если шаблон можно изобразить с помощью свернутого компонента PINbox, то его следует представлять как стек компонентов PINbox с несколькими кратностями *только в том случае*, если его основная роль в свернутом виде *не имеет* кратности. Иначе говоря, если существует настолько важная роль, что она имеет значение даже для свернутого компонента PINbox, то ее можно использовать как “стержень” стека компонентов PINbox.

Основываясь на этих соображениях, мы создали рис. 3.19, представляющий собой переработанный вариант рис. 3.4 с несколькими экземплярами шаблона *Strategy*, которые трудно отличить один от другого. Теперь у нас есть два кластера *Strategy*, каждый из которых ярко выражен. Для указания на то, что каждому экземпляру шаблона соответствуют несколько конкретных стратегий, используется стек. Это, конечно, прекрасно, но что если мы захотим выразить идею об использовании нескольких кластеров *Strategy* вместе, т.е. не отдельного экземпляра шаблона *Strategy*, а нескольких взаимодействующих вместе. Иначе говоря, ситуация с несколькими экземплярами шаблона *Strategy* может оказаться настолько распространенной, что понадобится ее точное и ясное описание. В таком случае можно использовать рис. 3.19. Мы используем диаграмму UML,

тесно связанную с нашей реализацией, и абстрагируем соединение между кластерами *Strategy* простым и точным способом. Эта диаграмма намного проще, чем ее исходный вариант, показанный на рис. 3.4. При этом она описывает более быстрый и простой механизм совместного использования классов и перекрестной коммуникации: “Существует несколько экземпляров шаблона *Strategy*, причем для каждого из них используются одни и те же классы”. Возможно, эту ситуацию даже стоило бы описать в виде отдельного шаблона.

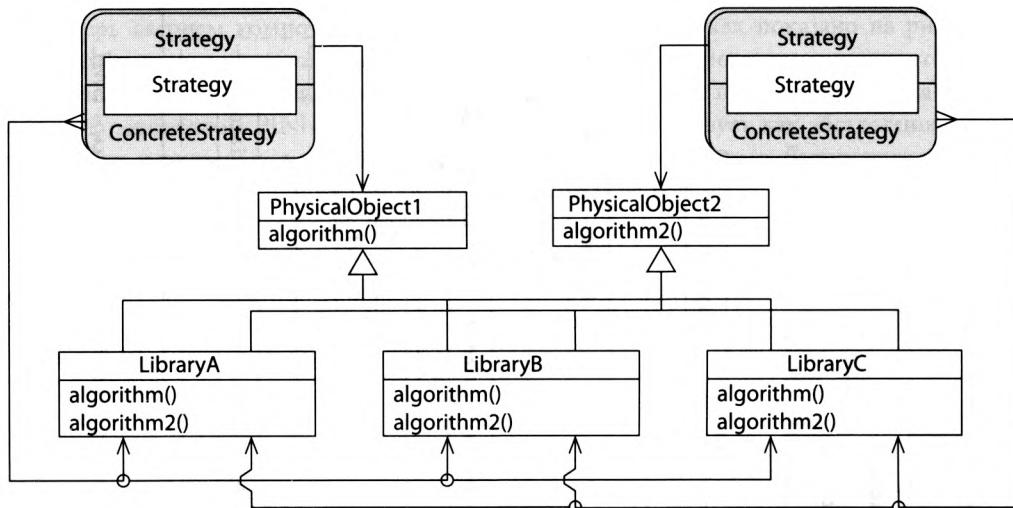


Рис. 3.18. Множественные экземпляры шаблона Strategy в виде компонентов PINbox

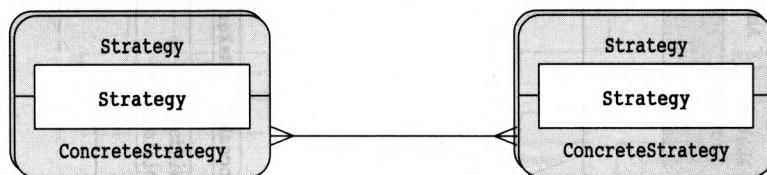


Рис. 3.19. Изображение взаимодействия между множественными компонентами PINbox в шаблоне Strategy

3.2.5. Пилинг и конденсация

Рассмотрим последний трюк с раскрытым компонентом PINbox, о котором стоит знать, несмотря на то что мы не будем часто использовать его в этой книге. Рассматривая систему обозначений UML, мы указывали, что в диаграмму можно лишь добавлять (но не удалять) информацию, и поэтому ее неудобно использовать для крупномасштабного абстрагирования.

Предположим, мы анализируем систему, представленную на рис. 3.17. Мы аккуратно идентифицировали стек компонентов PINbox для нескольких экземпляров шаблона проектирования *Abstract Factory*, но не смогли упростить диаграмму. Польза от этого невелика.

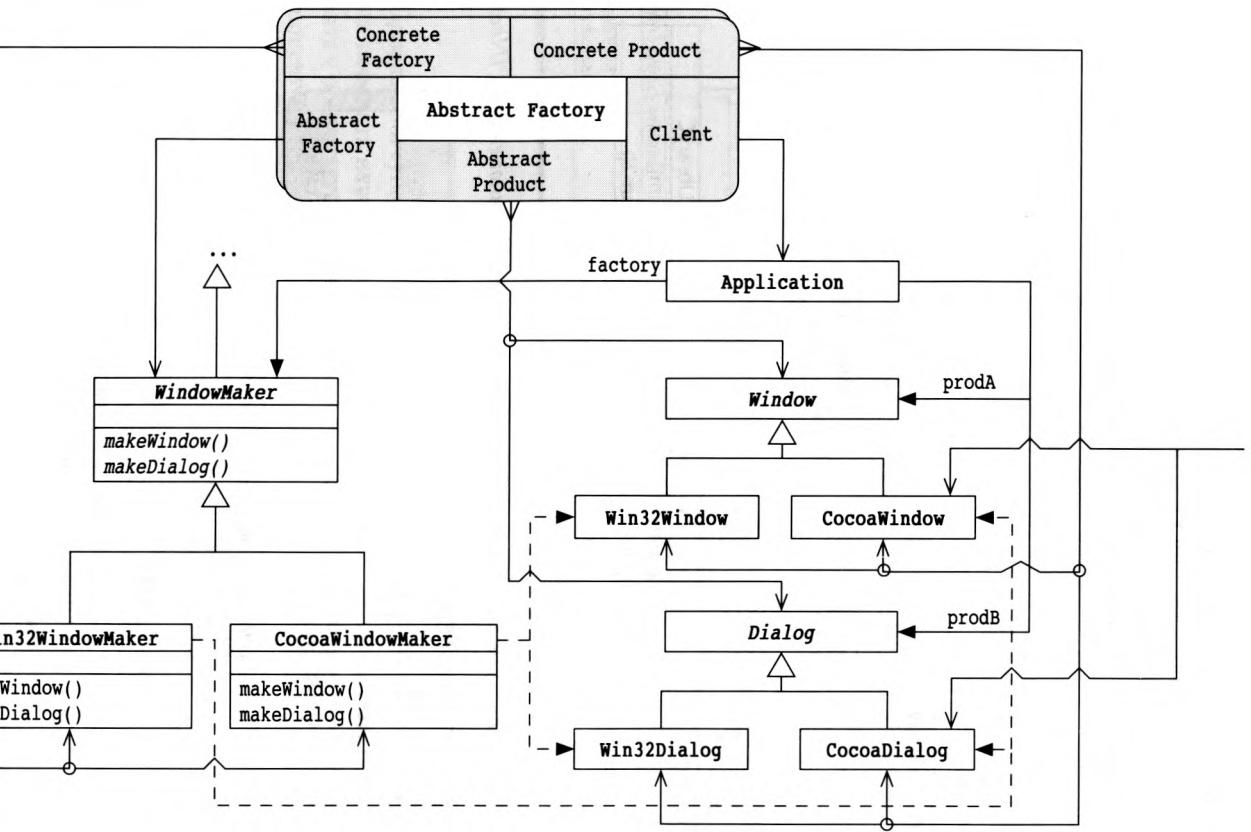


Рис. 3.20. Шаблон Abstract Factory как часть более крупной диаграммы UML

Представьте себе, что фрагмент обозначений UML на рис. 3.17 является лишь частью более крупной диаграммы. Добавим несколько соединений, оставив их неоднозначными, как показано на рис. 3.21. Трудно увидеть, что же мы добавили, не так ли? Теперь вставим фрагмент диаграммы UML, показанный на рис. 3.17, *внутрь* раскрытоого компонента PINbox, как показано на рис. 3.21. Информация осталась неизменной, но теперь в качестве прокси-механизма прокси мы используем компонент PINbox. Все, что было присоединено к внешней границе роли, соединяется с контактами, присоединенными к внутренней границе роли. Серую границу роли можно считать переходным слоем.

Теперь свернем компонент PINbox в стандартный вид, как показано на рис. 3.22. Диаграмма значительно упростилась, не так ли? Теперь можно точно понять, что собой представляют новые связи. Все детали экземпляра шаблона проектирования заменены одним компонентом PINbox. Этот компонент теперь действует как абстракция, скрывая детали и упрощая, а не усложняя ситуацию. Шаблон *Abstract Factory* описан в разделе 7.1.1. Там же продемонстрировано использование раскрытоого компонента PINbox для представления внутренней структуры шаблона.

Для того чтобы увидеть детали, достаточно раскрыть компонент PINbox снова. Полное удаление компонента PINbox и восстановление связей между внутренними и внешними сущностями называется пилингом (peeling). Идея пилинга описана в статье [36], упомянутой в начале главы.

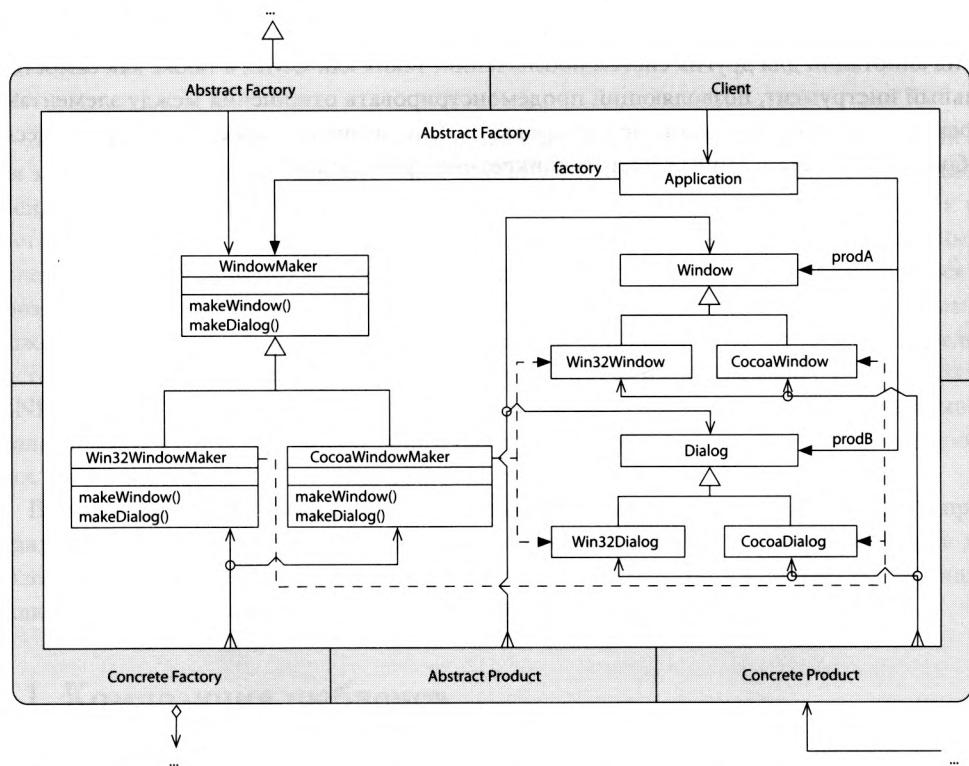


Рис. 3.21. Шаблон *Abstract Factory*, погруженный в раскрытоый компонент PINbox

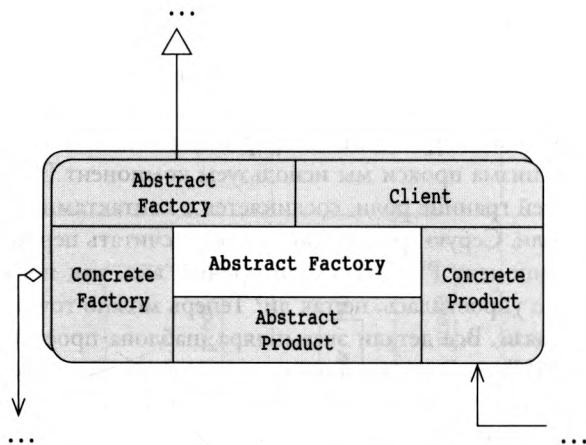


Рис. 3.22. Конденсированный компонент PINbox

3.3. Заключение

Система PIN предназначена для упрощения и повышения гибкости визуального описания элементов проектов программного обеспечения. Ее можно использовать для создания аннотаций для других систем обозначений, таких как UML, а также как самостоятельный инструмент, позволяющий продемонстрировать отношения между элементами проекта, такими как шаблоны проектирования. Она позволяет обеспечить практически любой уровень детализации с учетом конкретных требований.

Работа с элементарными шаблонами проектирования

До сих пор мы говорили об элементарных шаблонах проектирования (EDP), используя метафоры, например “строительные блоки”, и сравнения с периодической таблицей Менделеева. Эти метафоры и сравнения мы выбрали потому, что они относятся к областям знаний, в которых фундаментальные элементы объединяются в более крупные и полезные конструкции. То же самое можно сказать об элементарных шаблонах проектирования. Иначе говоря, мы предполагаем, что элементарные шаблоны проектирования можно объединять в более крупные осмысленные фрагменты. В главе 3 была введена система обозначений Pattern Instance Notation (PIN) и было показано, как компонент PINbox можно внедрить в диаграмму. Кроме того, было продемонстрировано, как могут взаимодействовать между собой многочисленные экземпляры шаблонов проектирования, в том числе EDP.

В этой главе мы систематизируем информацию и покажем, что шаблоны EDP представляют собой не просто отдельные небольшие концепции. Они образуют базис для более крупных абстракций и могут эффективно использоваться в множестве ситуаций, включая проектирование, реализацию и переработку систем.

4.1. Композиция шаблонов

Вернемся к нашей дискуссии о реконструкции шаблона *Decorator* из раздела 2.2.1. Мы выяснили, что при этом использовался шаблон *Object Recursion* (Рекурсия объекта) и что шаблон *Objectifier* (Конструктор объектов) является частью шаблона *Object Recursion*.

Для выполнения деконструкции необходимо глубоко знать существующие шаблоны проектирования. Но и при этом его результат является неопределенным и не дает полного понимания шаблона *Decorator*. Для получения более полной картины мы будем использовать шаблоны EDP.

Начнем с одного из основных элементарных шаблонов объектно-ориентированного программирования — *Abstract Interface* (Абстрактный интерфейс). Как уже говорилось, он означает, что в будущем будет создан подкласс, реализующий указанный метод. Сделаем еще один шаг вперед и объединим элементы этого шаблона с диаграммами UML. Поскольку речь идет о подклассе и шаблоне *Abstract Interface*, нужен шаблон *Inheritance*. Эта ситуация изображена на рис. 4.1.

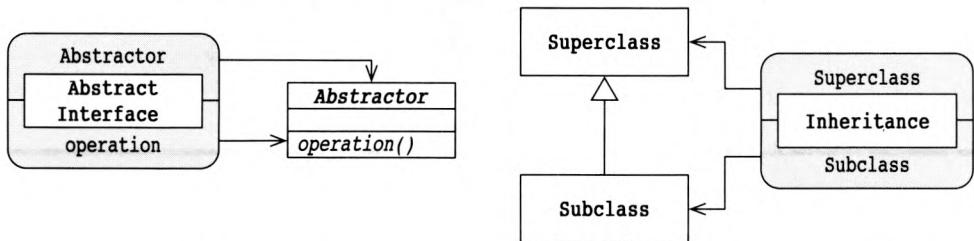


Рис. 4.1. Элементарные шаблоны проектирования *Abstract Interface* и *Inheritance* в виде диаграммы UML

Теперь соединим эти элементарные шаблоны проектирования особым способом. Известно, что шаблон *Abstract Interface* подразумевает “неопределенный подкласс” класса *Abstractor*. Это значит, что класс, исполняющий роль *Abstractor*, является суперклассом. Покажем, что роль *Superclass* в шаблоне *Inheritance* ссылается и на класс *Abstractor*. В результате две диаграммы UML объединяются в одну (рис. 4.2). Применим прием, описанный в разделе 3.2.2, соединив два элементарных шаблона проектирования и получив более крупный проект. В эту диаграмму была добавлена еще одна порция информации: роль *Subclass* шаблона *Inheritance* формулирует конкретное определение роли оператора из шаблона *Abstract Interface*. Эта ситуация является новой и еще не описана ни одним из шаблонов EDP.

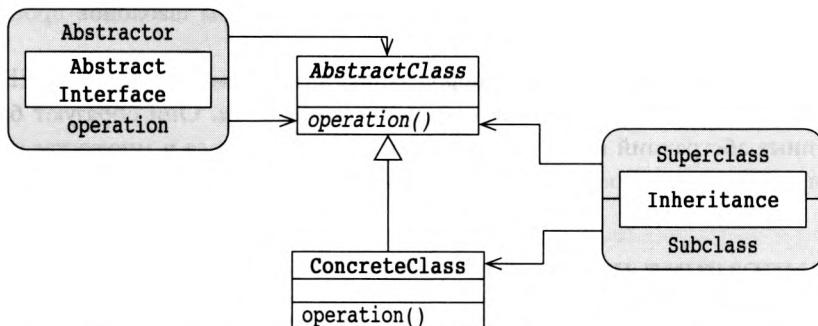


Рис. 4.2. Внутреннее определение шаблона Fulfill Method в виде диаграммы UML

Это новое определение выполняет контракт, являющийся концептуальной особенностью шаблона *Abstract Interface* и заключающийся в том, что подкласс обязан обеспечить реализацию абстрактного метода, и указывает имя нового шаблона проектирования: *Fulfill Method* (Выполнение метода). Эта концепция еще не появлялась в нашей книге, хотя я уверен, что вы ее применяли в объектно-ориентированном программировании. Мы уже ссылались на эту композицию шаблонов в конце раздела 3.2.2 и демонстрировали ее на рис. 3.12, но еще не анализировали ее. Так давайте разберемся.

Перерисуем рис. 4.2, не используя диаграмму UML, и ограничимся лишь системой обозначений PIN. На рис. 4.3 детали реализации скрыты и оставлено только отношение между двумя элементарными шаблонами проектирования. Это позволяет четко аBSTРАГИРОВАТЬ отношения, содержащиеся в каждом из элементарных шаблонов проектирования. Остается простая диаграмма, которая позволяет четче показать связь между этими двумя концепциями. Затем можно свернуть новую упрощенную диаграмму в компонент PINbox под названием *Fulfill Method*, как показано на рис. 4.4.

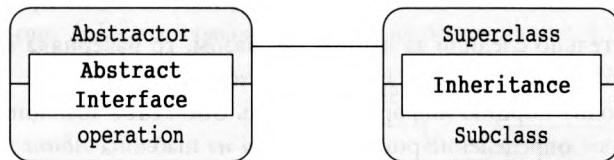


Рис. 4.3. Представление шаблона Fulfill Method в виде простых компонентов PINbox

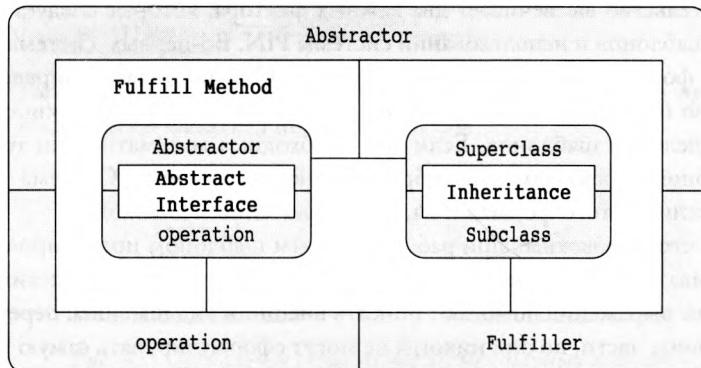


Рис. 4.4. Представление шаблона Fulfill Method в виде развернутого компонента PINbox

Реальное преимущество такой системы обозначений показано на рис. 4.5, на котором мы свернули раскрытый компонент PINbox и перешли на более высокий уровень абстракции. Эта диаграмма эквивалентна каждой из двух предыдущих, но состоит из одного простого компонента PINbox, который можно использовать для аннотирования диаграммы UML или, как будет показано позже, как часть определения более крупного шаблона проектирования. Эта практически фрактальная структура компонентов PINbox

и шаблонов проектирования постоянно проявляется при работе с шаблонами. Шаблоны проектирования состоят из более мелких шаблонов, которые состоят из элементарных шаблонов проектирования.

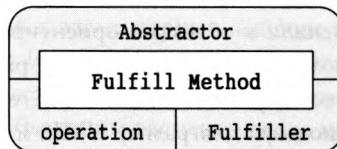


Рис. 4.5. Представление шаблона Fulfill Method в виде стандартного компонента PINbox

В свою очередь, все шаблоны проектирования, начиная с элементарных, используются как строительные блоки более крупных шаблонов. На каждом уровне детализации для точного описания рассматриваемой концепции можно использовать компоненты PINbox.

Если вы внимательно следили за нашим рассказом, то наверняка заметили, что мы пропустили один блок информации. Описывая рис. 4.2 я утверждал: “В эту диаграмму мы добавили еще одну порцию информации: роль *Subclass* шаблона *Inheritance* формулирует конкретное определение роли оператора из шаблона *Abstract Interface*. Эта ситуация является новой и еще не описана ни одним из шаблонов EDP”. Однако конкретного определения этой операции — т.е. определения функционального свойства *Fulfill Method* — на рис. 4.3 нет.

Это обстоятельство высвечивает два важных фактора, которые следует иметь в виду при изучении шаблонов и использовании системы PIN. Во-первых, система PIN не предназначена для формального описания шаблонов. Она не обязана определять каждое мелкое свойство шаблона. Она разработана для того, чтобы по возможности упростить описание определений шаблонов. Если вам необходима математически точная формализация шаблонов проектирования, обратитесь к приложению. Система PIN является лишь приближением этого формализма, предназначенным для людей.

Во-вторых, что еще важнее, при работе с любым шаблоном проектирования никогда не следует забывать о его словесном описании, которое является сердцевиной шаблона. Математические выражения помогают описать внешний вид шаблона, перечислить и назвать его составные части, но они никогда не могут сформулировать самую главную идею шаблона. Они не могут ответить на вопросы “*почему*”, “*где*” и “*когда*”. В лучшем случае они могут ответить на вопрос “*что*”. Это хорошо, но недостаточно для того, чтобы говорить о мудрости. Мудрость, т.е. понимание предназначения шаблона, заключается в ответах на любые вопросы, за исключением вопроса “*что*”. Если в какой-то момент вы перестали понимать шаблон, способ его применения, его предназначение или основные концепции, обратитесь к каноническому документу, описывающему этот шаблон, — спецификации. При изучении шаблонов и их применения такие системы обозначений, как PIN или даже формальные системы исчисления, никогда не могут полностью заменить словесное описание. Они служат лишь мнемоническими схемами.

В предыдущем примере это обстоятельство почти не заметно, но оно очень важно для понимания того, как устроены шаблоны EDP и как они формируют более крупные шаблоны. Мы взяли два элементарных шаблона проектирования, в каждом из которых определено отношение между двумя сущностями, и сшили их один с другим, создав немного более крупный шаблон. Мы просто продемонстрировали главный акт качественного проектирования — осмысленное объединение маленьких и понятных фрагментов, относящихся к решаемой задаче. Причем следует отметить, что этот процесс не имеет конца. Каждый раз, добавляя новое отношение в программу, вы изменяете проект. Дело лишь в том, чтобы понять, какие изменения являются полезными, а какие порождают проблемы.

В таком небольшом примере, как шаблон *Fulfill Method*, это почти очевидно. Вы можете сказать “А, собственно, как еще мы могли соединить эти два элементарных шаблона проектирования?” Ну, попробуйте сделать это так, как показано на рис. 4.6. Это те же самые элементарные шаблоны проектирования, но мы поменяли местами классы и подклассы. Теперь мы *не можем* добавить определение метода, как мы хотели!¹ Для того чтобы это было проще увидеть, перерисуйте соединение между компонентами PINbox, как показано на рис. 4.7. Этот граф совершенно не похож на рис. 4.3.

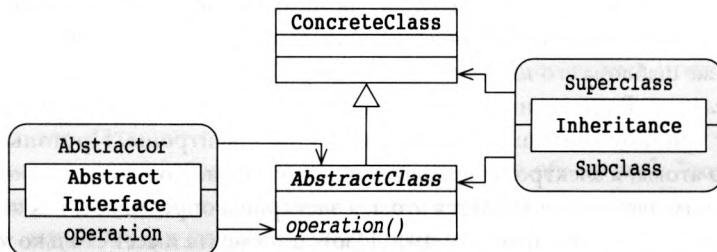


Рис. 4.6. Зеркальное отражение элементарных шаблонов проектирования в шаблоне *Fulfill Method*. Ой!

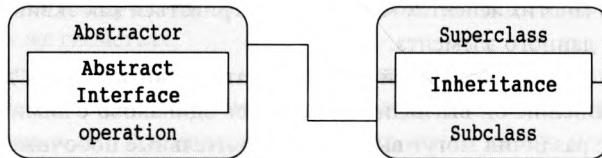


Рис. 4.7. Зеркальное отражение элементарных шаблонов проектирования в виде компонентов PINbox

К нашей цели ведет только одна определенная комбинация концепций. В этом и заключается магия шаблонов проектирования: с помощью метода проб и ошибок можно найти наилучшие практические решения решаемых нами проблем. Мы можем записать

¹ Стого говоря, это не так: некоторые языки программирования позволяют удалять методы с помощью позднего абстрагирования, но мы не хотим вникать в эти подробности, поскольку это связано со значительным риском и выходит за рамки выбранной нами темы. Кроме того, такое удаление было бы совершенно другой концепцией, не так ли? Мы же хотим выполнить обещанную реализацию метода, а не закрыть доступ к его предыдущим реализациям.

эти решения в виде комбинации более мелких концепций и создать из них новые формы, сохранив смысл исходных концептуальных соединений.

Элементарные шаблоны проектирования позволяют построить наилучшие практические решения из первичных блоков, не прибегая к методу проб и ошибок. Кроме того, рис. 4.2 демонстрирует абсолютный *минимум* обозначений UML, которого достаточно для элементарных шаблонов проектирования в компонентах PINbox, но не для *точного* решения. Это не *искомая* реализация; это просто *какая-то* реализация. Напомним, что мы планируем говорить о проектирования в терминах шаблонов, не ограничиваясь конкретной и строго определенной реализацией. Мы можем вернуться к отдельным элементарным шаблонам проектирования, из которых состоит шаблон *Fulfill Method*, и посмотреть, насколько их реализация может отличаться от концепции, не выходя за пределы допустимого.

4.1.1. Изотопы

Существует огромное количество способов и концепций, с помощью которых можно реализовать и описать любой шаблон проектирования, каким бы маленьким, как элементарный, или крупным, как *Decorator*, он ни был. Эта гибкость является главным преимуществом шаблонов проектирования. Мы будем называть эти разные реализации одного и того же шаблона его *изотопами* (*isotopes*). Это имя будем считать эквивалентом слова *элементарный*. В химии изотопом называется вариант атома конкретного элемента. Атомы имеют три компонента: электроны, протоны и нейтроны. Протоны и нейтроны образуют ядро атома, а электроны врачаются вокруг ядра. Количество протонов в ядре определяет, каким элементом является атом, а электроны определяют, как атом соединяется и взаимодействует с другими атомами. Изотоп элемента имеет столько же протонов, сколько и другие атомы этого элемента, а также столько же электронов на орбите. Благодаря этим двум свойствам с химической точки зрения он ведет себя так же, как и другие изотопы этого элемента. Он соединяется с другими атомами точно так же, образует такие же соединения и во многих аспектах может рассматриваться как эквивалент любого другого изотопа атома данного элемента.

Однако с точки зрения внутренней структуры атом изотопа имеет другое количество нейтронов в ядре. Внешне он выглядит и действует одинаково с химической точки зрения, но внутренние различия могут вызвать незначительные побочные эффекты, например расщепление ядра. Из-за этого они могут быть неустойчивыми.

В программном обеспечении такой неожиданный распад изотопа называется крахом.

Изотоп шаблона проектирования в программном обеспечении определяется аналогично. Изотоп представляет собой реализацию шаблона проектирования, в котором его основные концепции — то, что отличает его от других шаблонов проектирования, — а также внешний интерфейс остаются теми же.

Иначе говоря, роли, которые должны быть исполнены, не отличаются от ожидаемого описания шаблона проектирования.² Роли напоминают электронную оболочку в

² Если роли отличаются, значит, произошло фундаментальное изменение шаблона проектирования. Это изменение называется *вариантом* шаблона и является подсказкой о том, что в шаблоне изменилось что-то важное, например решаемая задача или контекст.

метафоре атома и явно указываются в компоненте PINbox. Все внешние сущности, такие как другой экземпляр шаблона или элемент кода, видят роли, окружающие “ядро” определения.

Количество протонов в атоме отличает его от всех других элементов и определяет его место в периодической таблице Менделеева. Для шаблонов проектирования наблюдается аналогичная ситуация, в которой задача, решение и контекст являются уникальными. Что делает шаблон проектирования, такую задачу он решает и другие характеристики являются его отличительными свойствами. Благодаря им он является уникальным и отличается от всех других шаблонов проектирования. Измените задачу, и вы получите другой шаблон проектирования. Измените контекст, и снова нужно будет адаптировать шаблон проектирования и создать нечто другое. Если это небольшое изменение, то результат будет выглядеть похожим на оригинал и получится вариант; если изменения большие, то получится нечто совершенно иное.

Для элементарных шаблонов проектирования аналогия с атомами и периодической таблицей еще сильнее, потому что оси контекста проектирования, описанные в разделе 2.2.4, создают точно определенное пространство, в котором каждый элементарный шаблон проектирования занимает четко определенное место. Контекст совершенно аналогичен химической периодической таблице, которая создает точно определенное пространство, в котором находятся элементы и которое позволяет предсказывать его свойства.

Положение элементарного шаблона проектирования в пространстве проектирования “метод–зависимость” (определенном осями “сходство объектов”, “сходство типов” и “сходство методов” в разделе 2.2.4) является абсолютным. Измените значение одной из координат на этих осях — и положение шаблона в пространстве изменится. Оригинальный элементарный шаблон проектирования должен переходить в другой, и можно предсказывать свойства нового шаблона. Однако измените одно из свойств элементарного шаблона проектирования, например то, как он выражается в конкретном экземпляре, — и шаблон EDP останется прежним, но его реализация изменится, т.е. получится другая реализация при тех же свойствах.

Изотоп, в химии или в шаблонах проектирования, отличается внутренней структурой. В атоме изменяется количество нейтронов. В экземпляре шаблона проектирования изменяется реализация его экземпляра. Это может иметь далеко идущие последствия, даже если такое изменение не распознается извне немедленно. Аналогично тому, как в любой момент нестабильный атом может распасться в зависимости от количества нейтронов, в любой момент плохо реализованный экземпляр шаблона проектирования может вызвать крах системы, даже если внешне все выглядит прекрасно.

Идея отделения реализации от внешнего интерфейса является традиционной и естественной для объектно-ориентированного программирования. Для изотопов идея отделения реализации от интерфейса *концепции* является новой. Напоминаем, что каждый шаблон проектирования имеет ряд ролей, перечисленных в разделе участников, которые должны быть исполнены частями реализации. Эти роли являются внешним интерфейсом, с помощью которого шаблон проектирования взаимодействует с другими концепциями и шаблонами.

В разделе участников описывается, как эти роли взаимодействуют внутри шаблона на концептуальном уровне, но эти взаимодействия можно описать и обсудить независимо от реализации. В этом месте включается в работу формализм, описанный в приложении и обеспечивающий огромную гибкость, а концепцию здесь можно выразить в примере кода. В разделе 2.2.2 был уже приведен простой пример транзитивности между методами, а здесь мы проанализируем это отношение более подробно.

Начнем с левой части следующего примера кода, в котором объект `f` содержит метод `foo`, вызывающий метод `bar` из объекта `b`. Мы говорим, что метод `f.foo` зависит от метода `b.bar`. Сходство объектов, сходство типов и сходство методов между этими конечными точками точно определены и определяют соответствующий элементарный шаблон проектирования. Если реализация изменится, например, путем внедрения нового объекта в цепочку вызовов методов, то конструкция изменится, но отношение между конечными точками останется прежним, как показано в правой части примера. Исходное отношение является интактным, как и три координаты схожести. Очевидно, возникли новые зависимости, но отношения, являющиеся неотъемлемыми частями исходного шаблона, остаются неизменными. Новая реализация является изотопом исходного элементарного шаблона проектирования: с внешней точки зрения он выглядит по-прежнему и действует так же, несмотря на то, что его внутреннее устройство изменилось.

```

1 class F {
2     B b;
3     void foo() {
4         b.bar();
5     };
6 };
7 F f;
f.foo();

```

`f.foo()` зависит от `b.bar()`

```

class F {
    G g;
    void foo() {
        g.goo();
    };
}
class G {
    B b;
    void goo() {
        b.bar();
    };
}
F f;
f.foo();

```

`f.foo()` по **прежнему** зависит от `b.bar()`

Abtractor
attribute : int
operation2()
operation()

Abtractor
attribute : int
attribute2 : string
...
attribute562: float
operation()
operation2()
...
operation1023()

Abtractor
attribute : T
operation()
operation2()

Рис. 4.8. Альтернативные классы, которые могут реализовать элементарный шаблон проектирования Abstract Interface

Еще более важно то, что это позволяет разработчикам обсуждать проект системы, не упоминая детали реализации. Мы можем осуждать проекты на более высоком уровне абстракции.

Для примера рассмотрим еще раз конструкцию шаблона *Fulfill Method* и покажем, как изотопы могут повысить гибкость даже в таком небольшом масштабе. Завершая описание этой конструкции, я упомянул, что диаграмма UML, показанная на рис. 4.2, использует минимальное подмножество языка UML, а не всю систему обозначений. Если бы мы использовали при описании шаблона *Fulfill Method* весь язык UML, то любое изменение в реализации шаблона *Fulfill Method* потребовало бы его нового определения. Поскольку существует множество вариантов, с помощью которых шаблон *Fulfill Method* может проявляться в системе, это привело бы к большому количеству определений для достаточно простой концепции. Это не оптимально.

Класс, исполняющий роль *Abstractor* в шаблоне *Abstract Interface*, может иметь любое количество других полей и методов, ассоциированных с ним. Просто хотя бы один метод должен быть абстрактным. На рис. 4.8 показано несколько классов UML, с каждым из которых может ассоциироваться экземпляр шаблона *Abstract Interface*. Любой из них можно использовать для того, чтобы продемонстрировать существование экземпляра шаблона *Abstract Interface*. Не имеет значения, сколько атрибутов и полей содержится в этом классе, сколько методов он имеет и является ли он шаблонным или обобщенным. В этом классе один метод должен быть абстрактным, и все.

Иначе говоря, экземпляр шаблона *Abstract Interface*, показанный в виде компонента PINbox на диаграмме шаблона *Fulfill Method* на рис. 4.4, может иметь сколько угодно разных реализаций. С помощью одних лишь компонентов PINbox невозможно создать все диаграммы UML или реализации кода.

Аналогично двухклассная минимальная версия шаблона *Inheritance*, показанная на рис. 4.2, не является единственным средством его представления. Как показано на рис. 4.9, в цепочке наследования между двумя классами может быть любое количество классов. Глубина дерева наследования значения не имеет, поскольку подкласс остается подклассом.³ И снова экземпляр шаблона *Inheritance*, изображенный с помощью компонента PINbox на рис. 4.4, может считаться представителем всех возможных реализаций.

Концепцию изотопа можно развить немного дальше. Напомним, что не все языки содержат классы в явном виде. Этот вопрос рассматривался в разделе 2.2.2. А как работает механизм подклассов в этих языках? Мы рассмотрим его в главе 5 в процессе анализа шаблона *Abstract Interface*, но даже такое изменение, сильно зависящее от языка программирования, можно инкапсулировать с помощью изотопа. Напомним, что мы хотели бы отвлечься от вопросов, связанных с реализацией шаблона в языке программирования. Работая на уровне элементарных шаблонов проектирования и компонентов PINbox, мы освобождаемся даже от семантики языка, который может быть использован для реализации. Возможность абстрагироваться от языка программирования является очень важной. В большей или меньшей степени мы обеспечиваем здесь полиморфизм концепций проектирования.

³ Это еще один пример транзитивности и основное свойство ρ-исчисления, описанного в приложении, которое придает элементарным шаблонам проектирования особую мощь. Видите, я же говорил, что буду слегка подталкивать читателей к изучению формальных моментов.

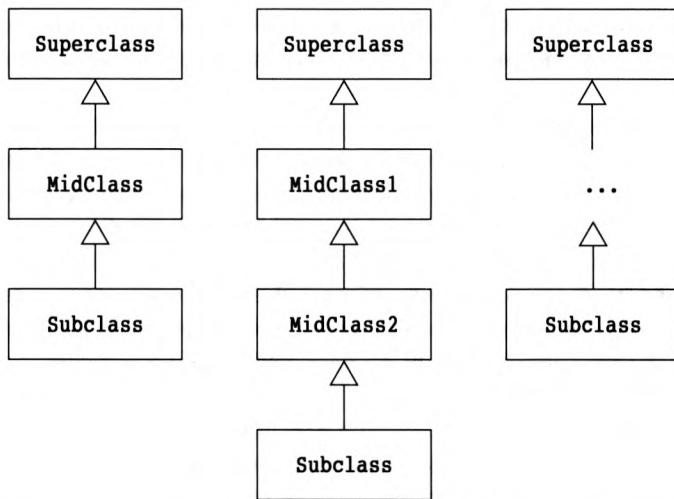


Рис. 4.9. Альтернативные структуры, которые могут выполнить работу элементарного шаблона проектирования Inheritance

Имея так много возможностей для реализации данного шаблона, вы, вероятно, задаете вопрос “Как же выбрать правильное определение шаблона проектирования?” Как уже говорилось, любая концепция, связанная с проектированием, может иметь множество реализаций. Сообщество специалистов по шаблонам проектирования до сих пор выбирает и описывает каноническую форму, которую могли бы использовать все без исключения, но какие критерии следует учитывать при выборе этой формы?

Проще говоря, то, что описано выше, представляет собой самую простую из всех возможных реализаций и множество участников и взаимодействий. Финальная форма не содержит ничего лишнего и выражает суть рассматриваемой концепции. Все, что выходит за пределы этого простого ядра, лишь затемняет основную идею и запутывает урок. Изотопы почти всегда вносят в каноническое описание дополнительные элементы, при этом основная концепция остается прежней.

Шаблоны проектирования создаются и редактируются так, чтобы выразить суть задачи, решения, контекста и максимально облегчить обучение и использование. В конце концов, наилучшее описание — это самое простое описание. Пусть изотопы учитывают изменения в реализации.

4.2. Воссоздание шаблона *Decorator*

До сих пор мы видели, как лишь очень небольшое количество концепций и вопросов проектирования позволяет выразить более крупные концепции. Мы также видели, как эти концепции инкапсулируют реализацию, так что вопросы проектирования можно упростить с помощью абстракции. Мы собираемся объединить эти элементы, чтобы создать более удобное и надежное определение шаблона *Decorator*, который мы начали обсуждать в главе 2.

Начнем с пересмотра первоначальной диаграммы UML для шаблона *Decorator*, представленной на рис. 4.10 в UML-версии шаблона *Fulfill Method* на рис. 4.11.

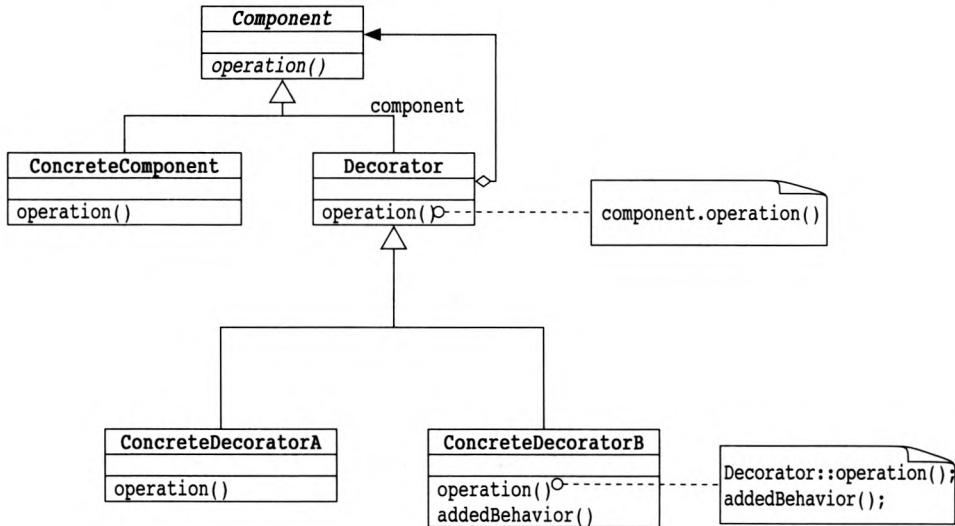


Рис. 4.10. Обычная диаграмма UML для шаблона Decorator

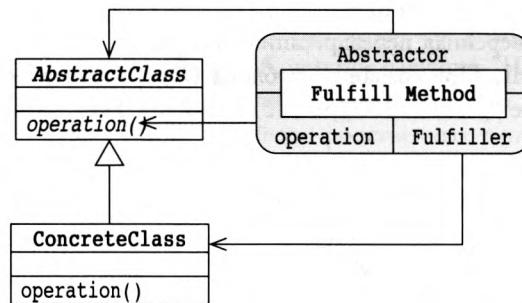


Рис. 4.11. Определение шаблона Fulfill Method с помощью аннотированной диаграммы UML

Еще раз взглянув на шаблон *Decorator* (рис. 4.10), можно увидеть два места, в которых проявляется структура шаблона *Fulfill Method*, и класс *Component*, который в обоих местах исполняет роль *Abstractor*. Впрочем, во втором месте он больше похож на шаблон *Objectifier* из рис. 2.2, не так ли? У вас зоркий глаз! Это действительно шаблон *Objectifier* — расширение одиночного шаблона *Fulfill Method* на несколько конкретных подклассов, как показано на рис. 4.12. Здесь для описания шаблона *Fulfill Method* мы снова использовали стековую форму PIN (см. раздел 3.2.4), чтобы подчеркнуть, как базовые концепции согласуются друг с другом. Мы просто определили шаблон проектирования, уже существующий в литературе, но используя первичные принципы и создавая его из маленьких и понятных блоков. На рис. 4.2—4.5 мы свернули шаблон *Fulfill Method*

в компонент PINbox. Эту же операцию можно выполнить над шаблоном *Objectifier*, используя рис. 4.12 как его внутренность и пропустив промежуточные диаграммы. В результате мы представим шаблон *Objectifier* в виде компонента PINbox.

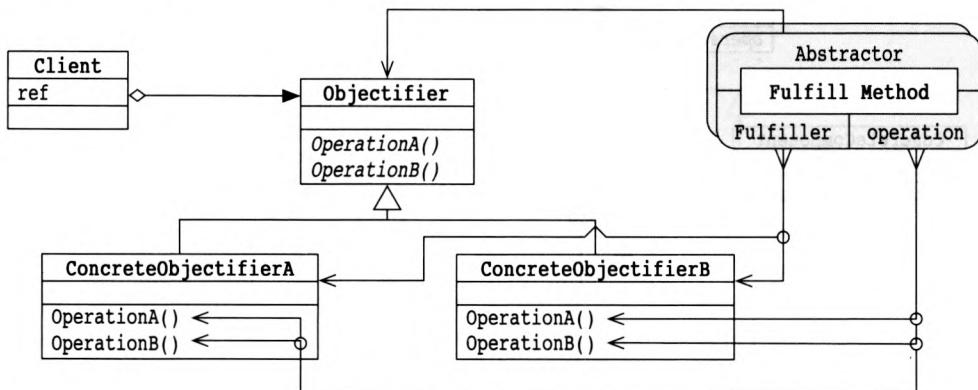


Рис. 4.12. Диаграмма UML для шаблона Objectified, дополненная обозначениями системы PIN

Далее можно определить шаблон *Object Recursion*. В разделе 2.2.1 сказано, что шаблон *Objectifier* является компонентом шаблона *Object Recursion*, но нас сейчас это не интересует. Теперь мы можем идентифицировать его как элементарный шаблон проектирования *Trusted Redirection* (Доверенная переадресация). На рис. 4.13 эти два шаблона показаны в виде диаграммы UML. При создании шаблона *Object Recursion* классы *FamilyHead* и *Objectifier*, а также *ConcreteObjectifierB* и *Redirector* объединяются.

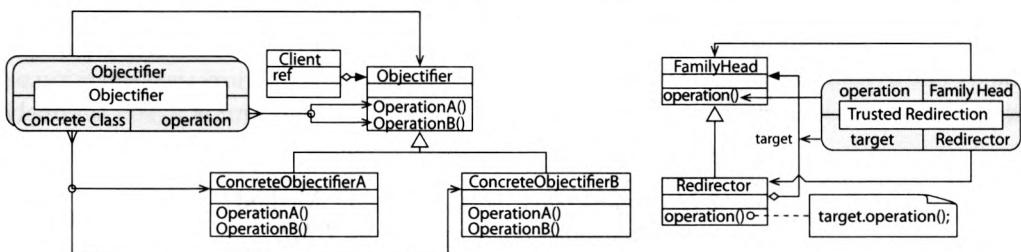


Рис. 4.13. Шаблоны Objectifier и Trusted Redirection

На рис. 4.14 шаблон *Object Recursion* имеет немного более четкий вид, чем на исходном рис. 2.3, который теперь дополнен компонентами PINboxes для шаблонов *Objectifier* и *Trusted Redirection*. Роль *Objectifier* из шаблона *Objectifier* и роль *Family Head* из шаблона *Trusted Redirection* теперь исполняются одной и той же сущностью, классом *Handler*. Аналогично классы *ConcreteObjectifierB* и *Redirector* объединены в класс *Recursor*. Более лаконично это показано диаграммой PIN, представленной на рис. 4.15.

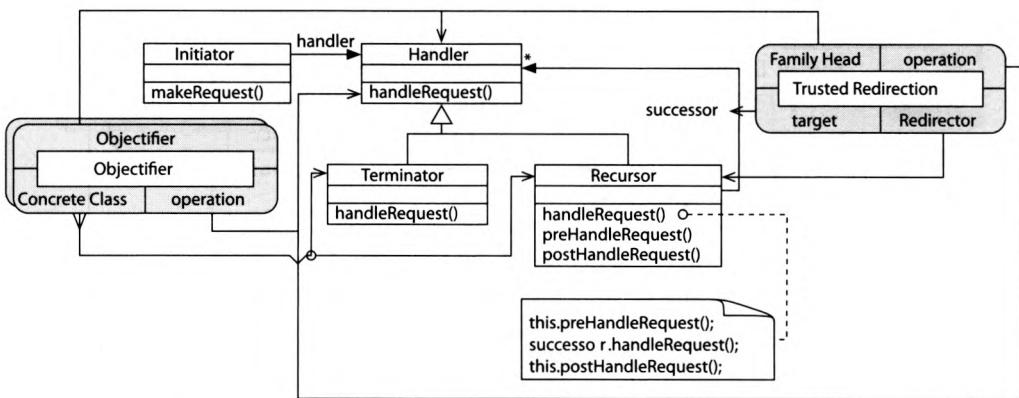


Рис. 4.14. Шаблон Object Recursion, дополненный обозначениями системы PIN

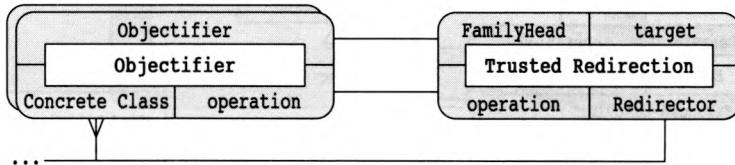


Рис. 4.15. Шаблон Object Recursion, представленный с помощью только системы PIN

Теперь мы можем обсудить шаблон *Object Recursion* как композицию более мелких шаблонов. Это намного более точный способ, чем предыдущие. “Шаблон *Object Recursion* использует полиморфизм с помощью шаблона *Objectifier*, чтобы во время выполнения программы определять, какой тип из семейства родственных типов должен обработать конкретный вызов. Применяя шаблон *Trusted Redirection* хотя бы в одной из возможных реализаций, он также связывает два или более объектов из этого семейства, так что они могут обработать тот же самый вызов по очереди, используя собственную реализацию”. Это утверждение является точным, ясным и не использует понятия, связанные со структурой шаблона. Более того, если базовая концепция кому-то осталась непонятной, он может изучить полные спецификации шаблонов для каждого из подклассов. Можно также взять эти спецификации и определения из литературных источников и создать собственные описания на высоком уровне абстракции, и тогда шаблоны станут намного понятнее.

Мы уже создали верхнюю часть шаблона *Decorator*, и осталось только закончить его определение. Оставшаяся концепция называется *Extend Method*. Она заполняет нижнюю часть шаблона *Decorator*, расширяя шаблон *Trusted Redirection*, как показано на рис. 4.16 и 4.17. Исходные функциональные свойства из шаблона *Extend Method* и рекурсор из шаблона *Object Recursion* объединяются в шаблоне *Decorator*, связывая эти два более мелкие шаблоны вместе.⁴ Мы можем упростить эту схему, изобразив ее в виде компонентов PINbox, как показано на рис. 4.18.

⁴ Классы `ConcreteDecoratorA` и `Decorator` образуют еще одну ветвь шаблона *Inheritance*, но этот момент мы для простоты пока пропустим.

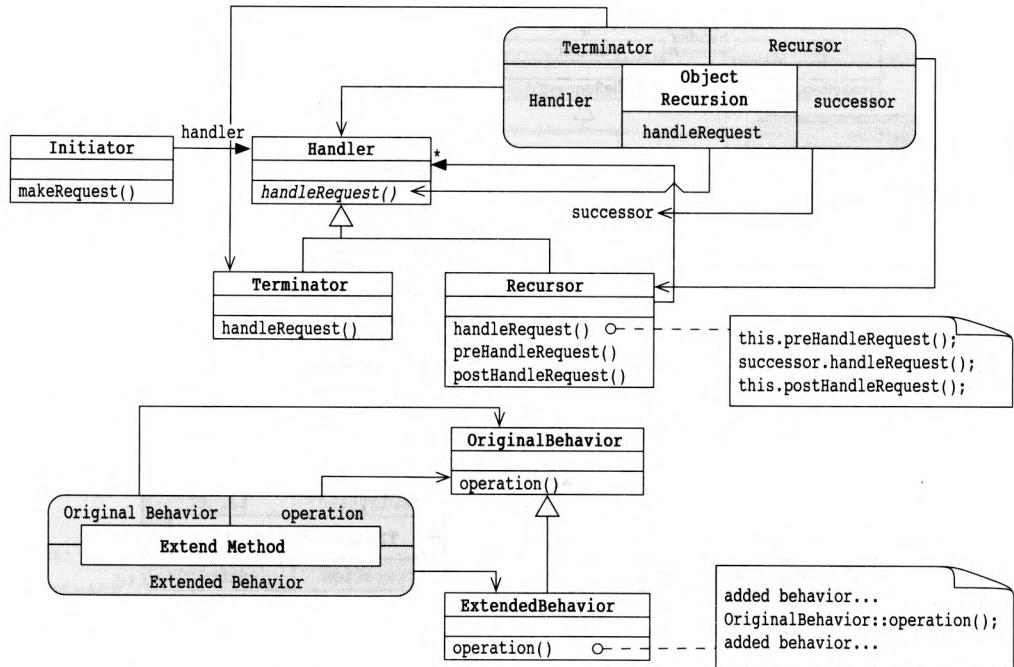


Рис. 4.16. Шаблоны Object Recursion и Extend Method

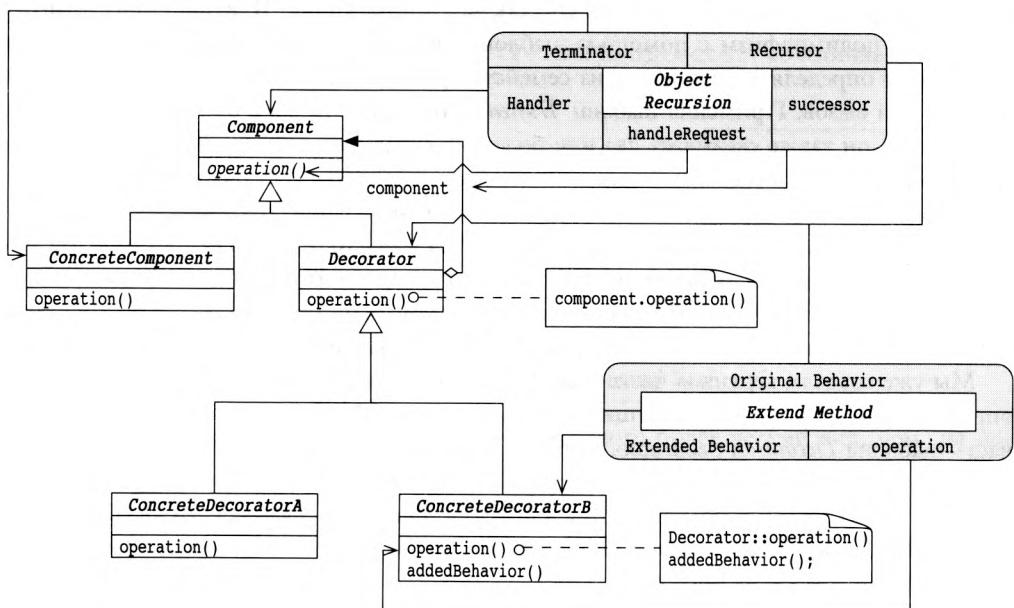


Рис. 4.17. Шаблон Decorator, дополненный обозначениями системы PIN

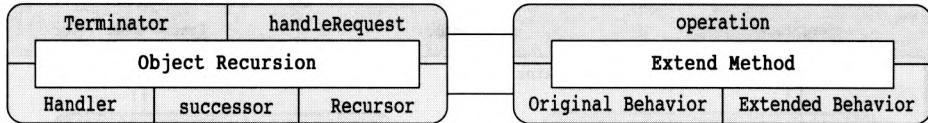


Рис. 4.18. Шаблон Decorator, представленный с помощью только системы PIN

Более того, мы можем сократить и свернуть эту схему в один компонент PINbox, обозначив этот экземпляр как *Decorator* (рис. 4.19). Имена ролей взяты непосредственно из раздела “Участники” спецификации шаблона *Decorator* из книги GoF [21]. Помимо этого, сделаны два дополнения: операция и объект *componentObj*. Они неявно обсуждаются в разделе “Участники”, но мы для ясности представили их явно.

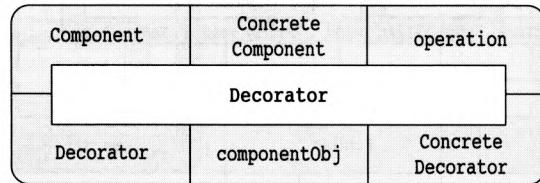


Рис. 4.19. Представление шаблона Decorator в виде компонента PINbox

Теперь у нас есть простая и точная система обозначения для шаблона *Decorator*. Однако и здесь существует альтернатива. Мы можем использовать раскрытое компоненты PINbox для того, чтобы увеличить уровень детализации в шаблоне *Decorator* и показать базовую иерархию концепций. На рис. 4.20 показан шаблон *Decorator* в виде раскрытоого компонента PINbox, демонстрирующего свое внутреннее устройство. На рис. 4.21 и 4.22 показано внутреннее устройство шаблонов *Object Recursion* и *Objectifier* соответственно, а на рис. 4.23 шаблон *Fulfill Method* раскрывается до уровня элементарных шаблонов проектирования, на котором декомпозиция заканчивается. Теперь шаблон *Decorator* полностью раскрыт.

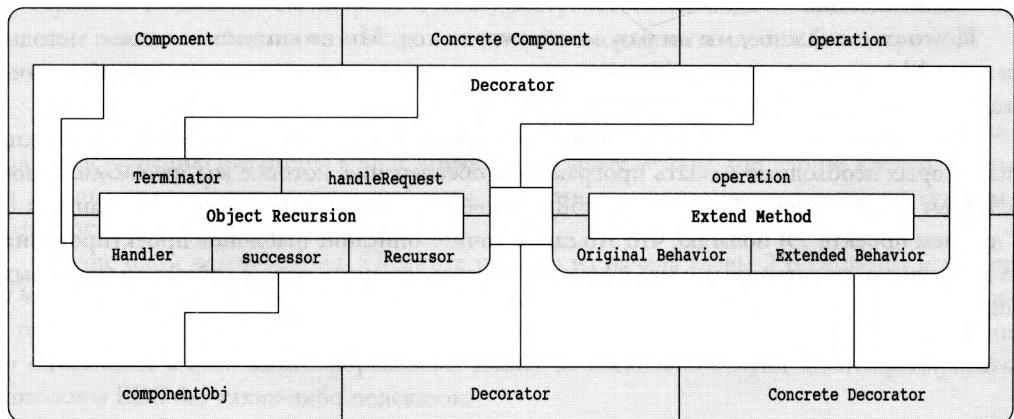


Рис. 4.20. Раскрытый шаблон Decorator: один уровень

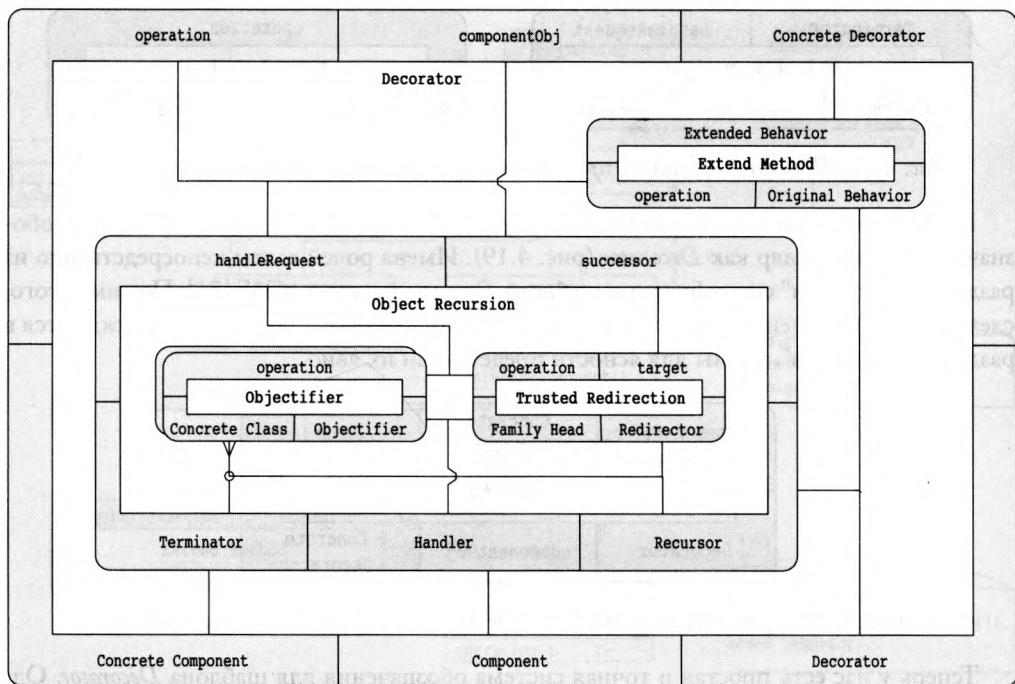


Рис. 4.21. Раскрытий шаблон Decorator: два уровня

Все эти диаграммы эквивалентны одна другой. Это шаблон *Decorator*, который мы составили всего из четырех элементарных шаблонов проектирования: *Abstract Interface*, *Inheritance*, *Trusted Redirection* и *Extend Method*. Каждый из них является простой концепцией, но вместе, связанные в особом порядке, они описывают высокуюровневую абстракцию, широко распространенную в системах программного обеспечения. Кроме того, мы продемонстрировали, что для овладения шаблоном *Decorator* можно изучать промежуточные концепции.

И что самое важное, мы ни разу не обсуждали код. Мы не вникали в классы, методы и поля. Мы говорили только о концепциях и идеях, хотя и создали платформу, обеспечивающую методологическую основу и точность.

Элементарные шаблоны проектирования представляют собой строительные блоки, из которых необходимо создать программное обеспечение, которое мы *понимаем*. В последнем разделе книги GOF [21, р. 358]⁵ приведена цитата Кристофера Александера о “лучшем проекте”. Я полагаю, что это самое точное описание шаблонов проектирования в виде плотно подогнанных один к другому и переплетенных между собой элементарных шаблонов проектирования.

⁵ В издании на русском языке — с. 340. — Примеч. ред.

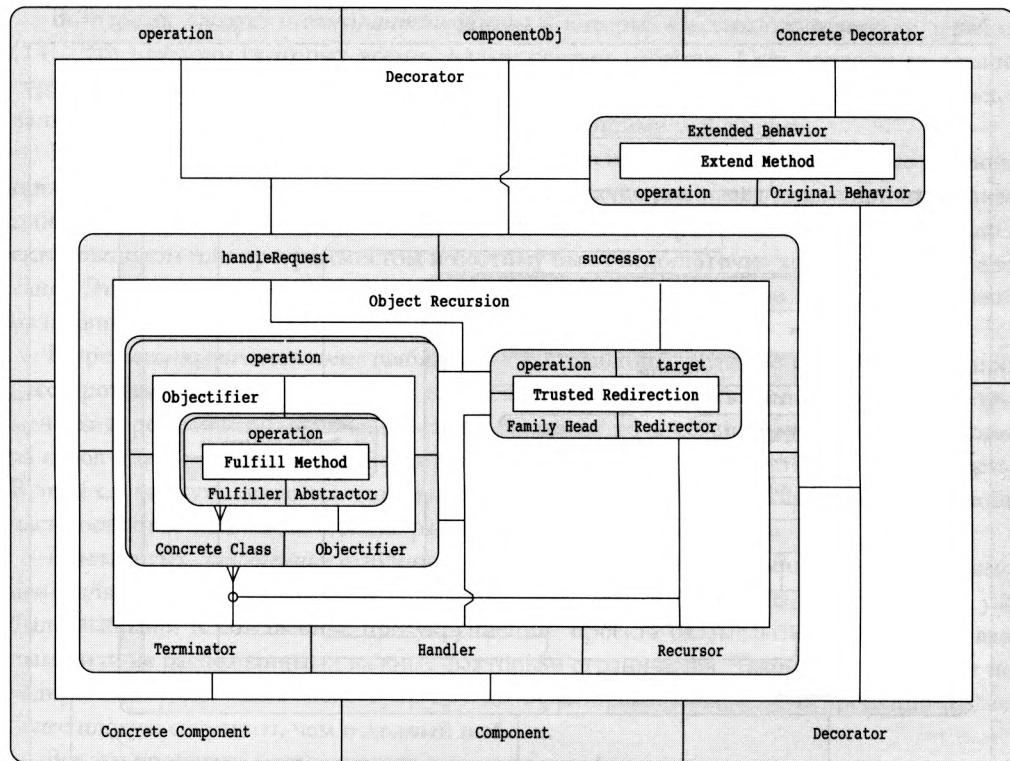


Рис. 4.22. Раскрытий шаблон Decorator: три уровня

Можно строить здание, произвольно нанизывая шаблоны. Такое здание представляет собой простую совокупность шаблонов. Оно не компактное. Оно не основательное. Но можно объединять шаблоны таким образом, что многие из них перекрываются в одном и том же физическом пространстве: такое здание очень компактное; в нем на небольшом пространстве реализовано много идей; и благодаря этой компактности оно становится основательным [3, р. xli].

Литература о шаблонах проектирования наполнена основательными идеями благодаря представлению шаблонов в виде композиций более мелких концепций в компактном и точном виде. Элементарные шаблоны проектирования позволяют выявить эту компактность со всей четкостью и глубиной.

И еще один комментарий. Помните пример из начала главы 2 о скрытом шаблоне *Decorator* в промышленном коде, который стимулировал создание системы SPQR? Система SPQR, используя элементарные шаблоны проектирования, метод декомпозиции и описанный в этой книге формализм, может за несколько секунд идентифицировать шаблоны EDS без каких-либо подсказок.

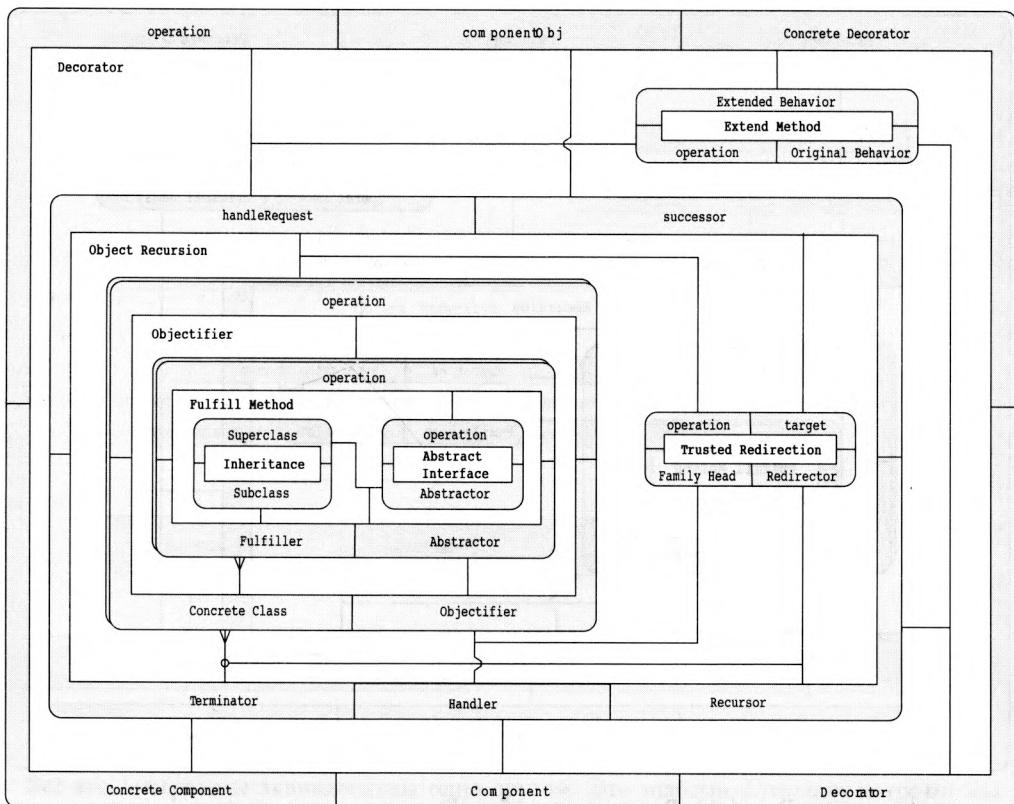


Рис. 4.23. Раскрытий шаблон Decorator: четыре уровня

4.3. Рефакторинг

Обсуждая изотопы в разделе 4.1.1, я утверждал, что внешне экземпляр изотопа шаблона может выглядеть точно так же, как другой экземпляр этого шаблона, и при этом иметь неудачную реализацию, которая в будущем создаст проблемы. Однако еще хуже то, что отдельный плохо реализованный шаблон проектирования взаимодействует с другими многочисленными шаблонами проектирования, которые по отдельности имеют осмысленную и рациональную реализацию, но в совокупности порождают проблемы. Напомним, что основным стимулом для создания системы SPQR, описанной в разделе 2.1, был тот факт, что шаблоны проектирования появляются естественным образом. Точно так же легко могут возникать некачественные шаблоны, а поскольку способов написать плохую программу больше, чем создать хороший проект, плохие шаблоны появляются довольно часто.

Когда такая проблема возникает среди шаблонов проектирования, некоторые из плохих проектов можно классифицировать.

Во-первых, следует назвать *антишаблоны*, о которых вы, возможно, читали, в работе [14]. Это шаблоны, которых всеми силами следует избегать. Они воплощают распространенные ошибки, а не лучшие приемы программирования. Если они появились в вашей системе, их следует удалить и заменить хорошими шаблонами.

Во-вторых, существуют вполне приемлемые, но неправильно примененные шаблоны проектирования. Возможно, они были просто неудачно реализованы или при их реализации были потеряны важные компоненты. Возможно, они просто были плохо поняты проектировщиком или программистом и поэтому были недостаточно хорошо сконструированы. Эти *частичные* шаблоны проектирования часто имеют одно или два отклонения от правильного шаблона.

В-третьих, *злокачественные* шаблоны — это шаблоны, которые растут, но этот процесс протекает неправильно. Даже если злокачественные шаблоны когда-то были применены и реализованы правильно, при изменении кода со временем система выходит из строя или превращается в систему, которая больше не соответствует исходной цели. В этом случае хуже всего то, что в документации, если она существует, на эти шаблоны часто остаются ссылки, вводящие разработчиков в заблуждение.

В-четвертых, *ятрогенные* шаблоны — это шаблоны, которые были правильно применены для решения реальной проблемы, распознанной только тогда, когда решение уже было выбрано. К сожалению, при укрупнении проекта оказываются проигнорированными или не распознанными важные факторы и ограничения. Такой шаблон решает исходную задачу, но, взаимодействуя с другими факторами контекста, непременно создает более плохую ситуацию, чем исходный шаблон.

Все эти проблемы можно решить с помощью рефакторинга кода, при котором приходится в основном переносить функциональные свойства из одного места в другое, чтобы улучшить контуры проекта. Функциональные свойства и выполняемые модули системы изменяются незначительно, но способ взаимодействия между ее частями модифицируется существенно. При этом сущность кода не изменяется, но расширяются его потенциальные возможности.

Улучшая проект системы, вы повышаете ее понятность, т.е. упрощаете для программиста ее анализ и модификацию. Добавление новых функциональных свойств, исправление ошибок и выполнение другой полезной работы окажется более простым и полным.

Рефакторинг — хорошо разработанная дисциплина. Особого внимания заслуживают две книги: Martin Fowler *Refactoring* [19] и Joshua Kerievsky *Refactoring to Patterns* [24].⁶ Они содержат набор хорошо согласованных между собой рецептов по преобразованию кода из одной конфигурации в другую. Обе эти книги достойны внимательного изучения. Если вы когда-либо использовали интегрированные среды разработки (IDE), такие как Eclipse, Visual Studio или Xcode, то, возможно, использовали рефакторинг с помощью возможностей, предоставляемых этими инструментами. Они автоматизируют этот

⁶ Фаулер М. Рефакторинг. Улучшение существующего кода. — М.: Символ Плюс, 2003. — 432 с.; Керивеский Д. Рефакторинг с использованием шаблонов. — М.: Вильямс, 2006. — 400 с. — Примеч. ред.

процесс, выполняя простые и полезные задания. Для большинства этих функциональных возможностей отправной точкой является команда **Refactor**. Чтобы лучше понять, как эффективно использовать эти автоматизированные инструменты, следует прочитать указанные книги.

Книга Фаулера заложила основы для создания каталога небольших дискретных действий, позволяющих улучшить структуру кода. Например, она начинается с “дурных запахов”, т.е. с ситуаций, которые вам не нравятся, несмотря на то что код работает правильно. Он описывает эти интуитивные догадки как возможные индикаторы подсознательной тревоги относительно структуры кода и объясняет, как их понять и ясно выразить. Чем больше у вас опыта в программировании и знаний принципов правильного проектирования, тем лучше вы осознаете свои инстинктивные ощущения в таких ситуациях.

Для каждой такой ситуации Фаулер предлагает ряд действий для решения проблемы. Эти действия очень разнообразны и многочисленны, но я назову только три из них. Например, *Extract Method* — это команда рефакторинга, которую можно использовать для устранения сомнительных ситуаций, связанных с использованием слишком длинного и запутанного метода. В своей простейшей форме эта команда предлагает выделить самодостаточный фрагмент длинного метода в виде отдельного метода и вызывать его. Большое значение придается названию метода, что вас уже не должно удивлять.

Это значение отражено в команде **Rename Method**, используемой для прояснения предназначения метода в ситуациях, когда его полезность или применимость не вполне понятны.

Аналогично команда **Move Method** используется для выяснения связей между методами. По мнению Фаулера, такое выяснение необходимо тогда, когда “метод используется или будет использоваться многими свойствами другого класса, а не класса, в котором он определен”. Иначе говоря, команду **Move Method** следует выполнять, когда по каким-то причинам метод оказался в неподходящем классе.

Сейчас это может показаться тривиальным, но каждому из нас приходилось не раз убеждаться в том, что советы Фаулера по праву стали классическими. И главное: Фаулер использует эти маленькие аккуратные действия как строительные блоки для выполнения более крупного рефакторинга. Кериевский принял эту концепцию и развил ее, используя каталог Фаулера как отправную точку для проведения более мощного рефакторинга, показывая, как методически правильно и точно перейти от одной реализации шаблона проектирования к другой. Это что-то вам напоминает?

Здесь прослеживается интересная параллель. Мы можем проследить ее, рассмотрев команду **Extract Method**. Смысл этого рефакторинга заключается в создании нового метода и его нового вызова. Метод, из которого извлекается фрагмент кода, является вызываемым, а новый метод —зывающим. Очевидно, что эти два метода не могут совпадать,⁷ поэтому можно считать, что они являются разными. Кроме того, поскольку новый метод создается в том же самом классе/типе — в соответствии с определением

⁷ Это еще один пример транзитивности и основное свойство ρ -исчисления, описанного в приложении, которое придает элементарным шаблонам проектирования особую мощь. Видите, я же говорил, что буду слегка подталкивать читателей к изучению формальных моментов.

рефакторинга, — тип и объект остаются теми же самыми. Это уже обсуждалось в разделе 2.2.4. Элементарный шаблон проектирования с тем же самым объектом и типом, но с другим методом называется *Conglomeration*.

Теперь перейдем к анализу команды *Rename Method* и покажем, что она может означать для элементарных шаблонов проектирования. Напомним, что сходство между методами (частично) определяется их именами. Изменив имя, мы могли бы перейти от элементарного шаблона проектирования со схожими методами к его аналогу с неподходящими методами. Например, рассмотрим экземпляр шаблона *Delegation*, показанный на рис. 4.24. Если целевой метод `called()` переименовать в `caller()`, то он превратится в точку вызова, а шаблон *Delegation* станет шаблоном *Redirection*, как показано на рис. 4.25. Конечно, переименовать можно что угодно, и эта трансформация элементарного шаблона проектирования происходит, только если новое имя обеспечивает сходство. Итак, можно сказать, что *Delegation + Rename Method* (для сходства) = *Redirection*.

Иначе говоря, *Redirection + Rename Method* = *Delegation*.

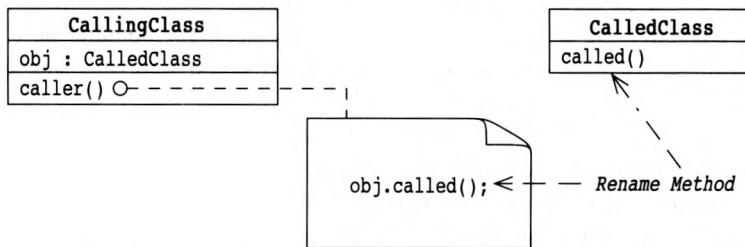


Рис. 4.24. Шаблон Decorator до выполнения команды рефакторинга *Rename Method*

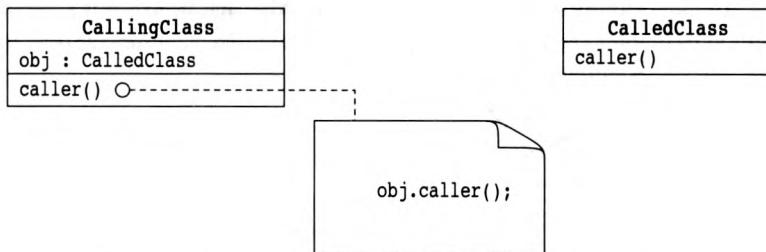


Рис. 4.25. Шаблон Decorator после выполнения команды рефакторинга *Rename Method*

При каждом изменении проекта (не важно, насколько малом) мы изменяем элементарные шаблоны проектирования, из которых он состоит. В свою очередь, элементарные шаблоны проектирования влияют на более крупные шаблоны, которые они образуют, и так далее вверх по композиционной иерархии. Зная точно, как изменения, указанные в литературе по рефакторингу, влияют на проект на разных уровнях детализации и как эти изменения распространяются по системе, можно точнее предсказать, как будут разветвляться более крупные планируемые модификации проекта.

Однако большие изменения не обязательно приводят к существенному изменению проекта или порождают мультиплексионный эффект. Мы трансформировали шаблон *Delegation* в *Redirection*, применив команду *Rename Method* так, чтобы она обеспечила сходство методов, которые ранее были непохожими. Попробуем выполнить обратное преобразование, начав с шаблона *Trusted Redirection*, который используется для создания шаблона *Decorator* из раздела 4.2. Как вы думаете, что произойдет, если мы применим команду рефакторинга *Rename Method* к методу, используемому в шаблоне *Trusted Redirection*?

Все просто. Экземпляр шаблона *Decorator* перестанет существовать.

Переименовав метод независимо от глубины его вложенности в реализации, что возможно благодаря изотопам, мы удаляем шаблон *Trusted Redirection* и заменяем его аналогом с непохожими методами, т.е. шаблоном *Trusted Delegation*. Удаляя шаблон *Trusted Redirection*, мы удаляем главную и необходимую часть шаблона *Decorator*. Во что же он превратился? Хороший вопрос. В некоторой степени новый проект выглядит, как шаблон *Strategy*, еще один шаблон проектирования GoF, имеющий определенное сходство с шаблоном *Decorator*, но в любом случае шаблона *Decorator* больше нет. Если это не предусмотрено в системной документации, возникает несоответствие между описанием системы и ее реализацией. Это источник ошибок.

Рассмотрим теперь команду *Move Method* и посмотрим, что произойдет с зависимостями между перемещаемым методом и вызывающими его методами после его переноса. Сначала нужно понять, что перемещение метода означает его перенос в новый класс/тип. Его появление внутри нового типа означает, что любой вызывающий его метод должен быть уточнен, чтобы сохранить работоспособность после перемещения вызываемого метода. Начнем с простого примера, показанного на рис. 4.26. Напомним, что это вызов метода между двумя несходными методами, содержащимися в разных объектах, имеющих разные не связанные один с другим типы. Мы знаем, что применение команды *Rename Method* изменяет сходство методов. А как изменится элементарный шаблон проектирования при перемещении метода?

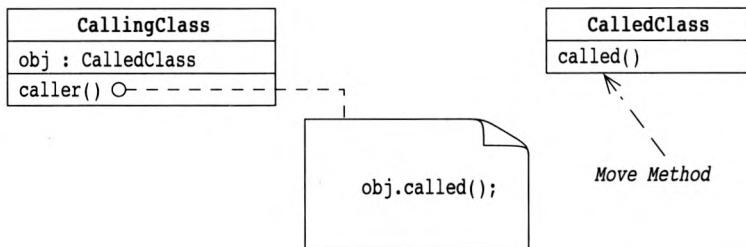


Рис. 4.26. Шаблон Decorator до выполнения команды рефакторинга Move Method

Эта команда имеет множество возможных последствий, зависящих от способа ее реализации. Количество этих последствий довольно велико, поэтому настало время коротко ознакомить вас с остальными элементарными шаблонами проектирования. Напомним (см. раздел 2.2.4) двухмерную сетку элементарных шаблонов в плоскости

“метод–сходство” (см. рис. 2.10). На рис. 4.27 показан ее аналог, в котором координата “Сходство методов” зафиксирована и имеет значение “Разные”. Ось “Сходство объектов” повернута зеркально относительно первого рисунка, потому что она аналогична рассмотренному ранее случаю с точностью до сходства методов.

При перемещении метода его чаще всего переносят в родственный тип, как показано на рис. 4.28. Это позволяет сохранить в целости экземпляр шаблона *Delegation*, но это не очень интересно. Рассмотрим лучше, что произойдет, если перенести метод в класс, которому принадлежит вызывающий метод, как показано на рис. 4.29. Конечно, теперь вызов метода `called()` объекта `obj` не имеет большого смысла.

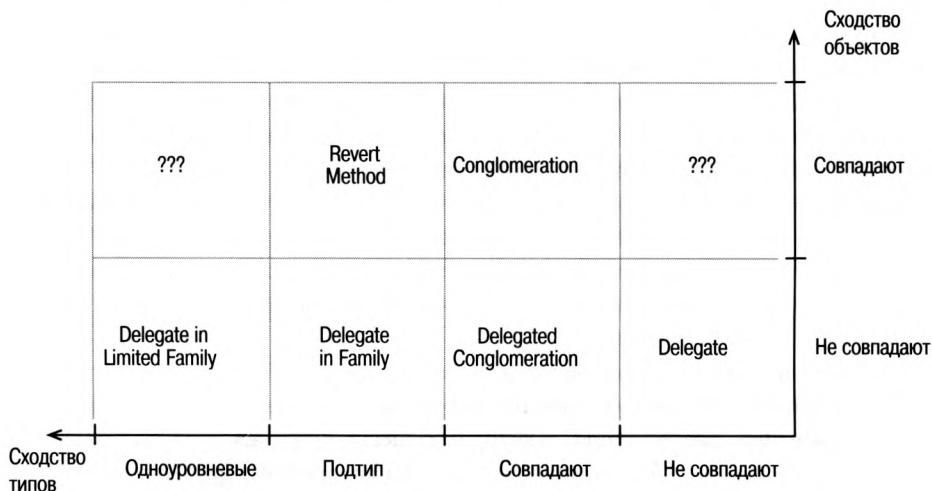


Рис. 4.27. Пространство проектирования, в котором координата “Сходство методов” зафиксирована и имеет значение “Разные”

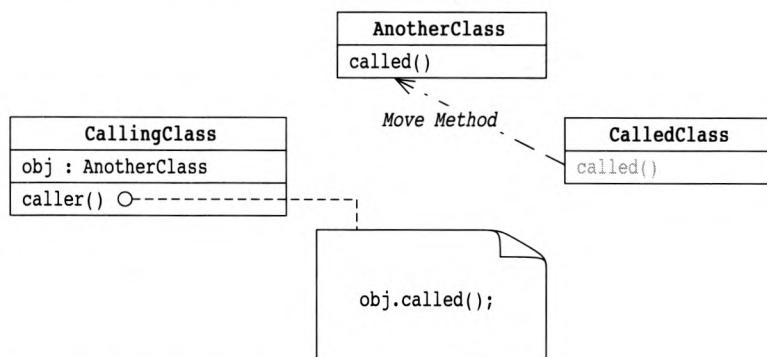


Рис. 4.28. Шаблон Delegation после выполнения команды рефакторинга *Move Method*: неинтересный вариант

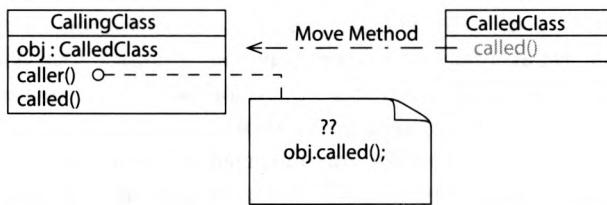


Рис. 4.29. Шаблон Delegation после выполнения команды рефакторинга Move Method: перенос в тот же самый тип

Существуют две возможности устраниТЬ несходство в методе `caller()`. Во-первых, этот метод может изменить тип объекта `obj` на целевой тип. Иначе говоря, объект `obj` становится полем экземпляра того же типа, что и тип, в котором он определен. Во-вторых, метод `caller()` может вообще исключить объект и вызвать собственный экземпляр, используя объект `self`. Выбор действия зависит от требований, предъявляемых к использованию данных в методе `called()`. Если метод `called()` работает в основном с данными, передаваемыми ему через метод `caller()`, и этими данными является экземпляр данных в объекте, инкапсулирующем метод `caller()`, то доступ к этим данным можно обеспечить непосредственно через метод `called()` и не передавать их как параметры. В данном случае лучшим является второй вариант. В остальных случаях следует предпочесть первый.

Какие элементарные шаблоны проектирования представляют эти два случая?

Первый вариант, в котором изменяется тип объекта, описывается элементарным шаблоном проектирования “вызов метода”, в котором существуют разные методы в разных объектах одного и того же типа. Анализируя рис. 4.27, на котором изображены разные объекты одного и того же типа, можно увидеть пример шаблона *Delegated Conglomeration*. Как и шаблон *Redirected Recursion*, он содержит два объекта одного и того же типа, работающие согласованно. Однако в отличие от шаблона *Redirected Recursion* в этом случае два метода отличаются один от другого. Эта ситуация изображена на рис. 4.30.

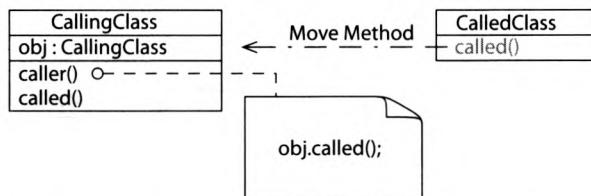


Рис. 4.30. Шаблон Delegation после выполнения команды рефакторинга Move Method: шаблон Delegation Conglomeration

Альтернатива заключается в простой замене вызываемого объекта объектом `self`, исключая тем самым необходимость в объекте поля, как показано на рис. 4.31. В результате возникает отношение “разные методы”, но теперь с одинаковыми типами и объектами. На рис. 4.27 показано, что это шаблон *Conglomeration*, рассмотренный в разделе 2.2.4.

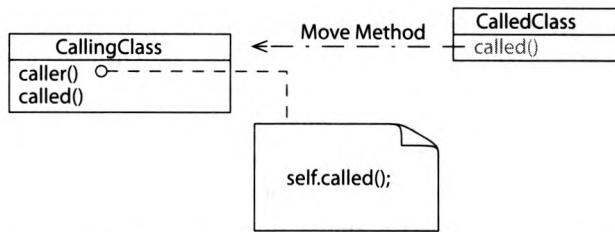


Рис. 4.31. Шаблон Delegation после выполнения команды рефакторинга Move Method: шаблон Delegation Conglomeration

Аналогичная двойственность возникает и в других вариантах, связанных с выбором типа. Если перемещаемый метод переносится в подтип класса CallingClass, то возникает либо шаблон *Trusted Delegation*, либо шаблон *Revert Method*, в зависимости от того, сохраняет ли вызываемый объект новый тип или исключается и заменяется объектом суперкласса. На рис. 4.32 и 4.33 показаны оба варианта. Шаблон *Trusted Delegation* чрезвычайно широко распространен и возникает в ситуациях, в которых делегирование должно быть поручено доверенной группе родственных типов и выполнено соответствующим образом.

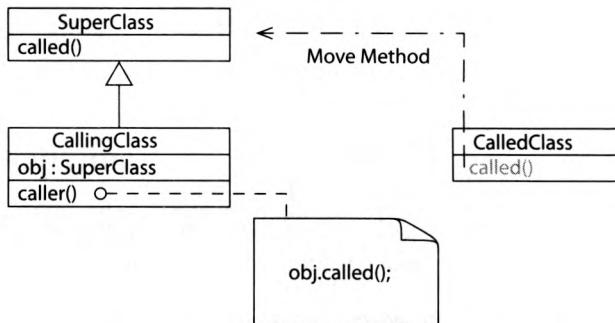


Рис. 4.32. Шаблон Delegation после выполнения команды рефакторинга Move Method: шаблон Trusted Delegation

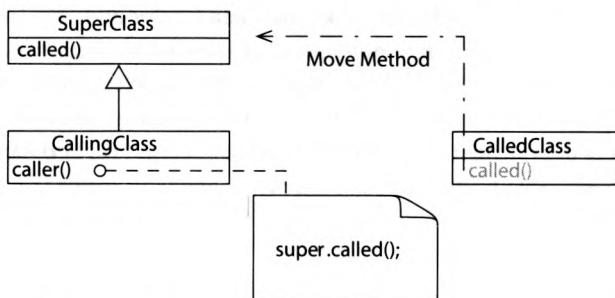


Рис. 4.33. Шаблон Delegation после выполнения команды рефакторинга Move Method: шаблон Revert Method

Проще всего это сделать с помощью полиморфного вызова, тип которого определяется суперклассом вызывающей сущности. Суперкласс обеспечивает доверенную группу типов, а полиморфизм — соответствующую обработку вызова.

Перемещение вызываемого метода в другой класс, находящийся на том же самом уровне иерархии, что и *CallingClass*, приводит к ситуации, показанной на рис. 4.34. Как и следовало ожидать, она описывается шаблоном *Deputized Delegation*. Здесь доверенная группа возможных обрабатывающих типов уточнена путем ограничения полиморфного корня классом, который находится на том же самом уровне иерархии, что и вызывающая сущность.

В заключение следует отметить, что каждая из указанных трансформаций имеет обратную. Можно легко вернуться к шаблону *Delegation* из любого преобразованного элементарного шаблона проектирования, а затем перейти к любому другому шаблону EDP. Таким образом, можно обойти весь рис. 4.27. Итак, с помощью простой команды рефакторинга *Move Method* можно обойти все шесть описанных элементарных шаблонов проектирования.

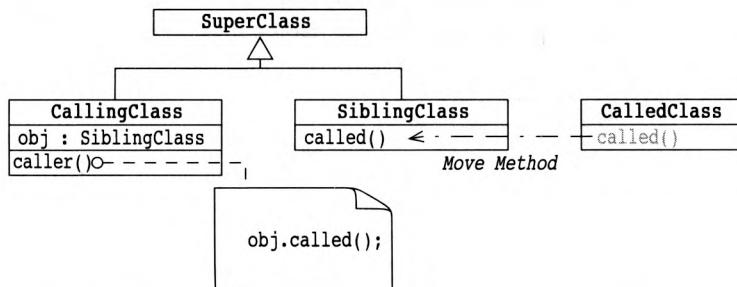


Рис. 4.34. Шаблон *Delegation* после выполнения команды рефакторинга *Move Method*: шаблон *Deputized Delegation*

Легко видеть, что рефакторинг может сильно повлиять на проект, даже если его действия кажутся тривиальными. Это важно помнить, принимаясь за рефакторинг своего проекта, а элементарные шаблоны проектирования помогут реализовать ваш план.

Описанные действия и трансформации с помощью элементарных шаблонов проектирования показаны на рис. 4.35. На нем показано, как с помощью команды *Move Method*, начиная с шаблона *Delegation*, можно получить все пять остальных шаблонов EDP, выделенных в левой части диаграммы полужирным шрифтом. Здесь также указаны остальные возможные трансформации, включая обратные. Использование команды рефакторинга *Rename Method* для перемещения влево и вправо свидетельствует о наличии симметрии, которая будет рассмотрена в следующем разделе.

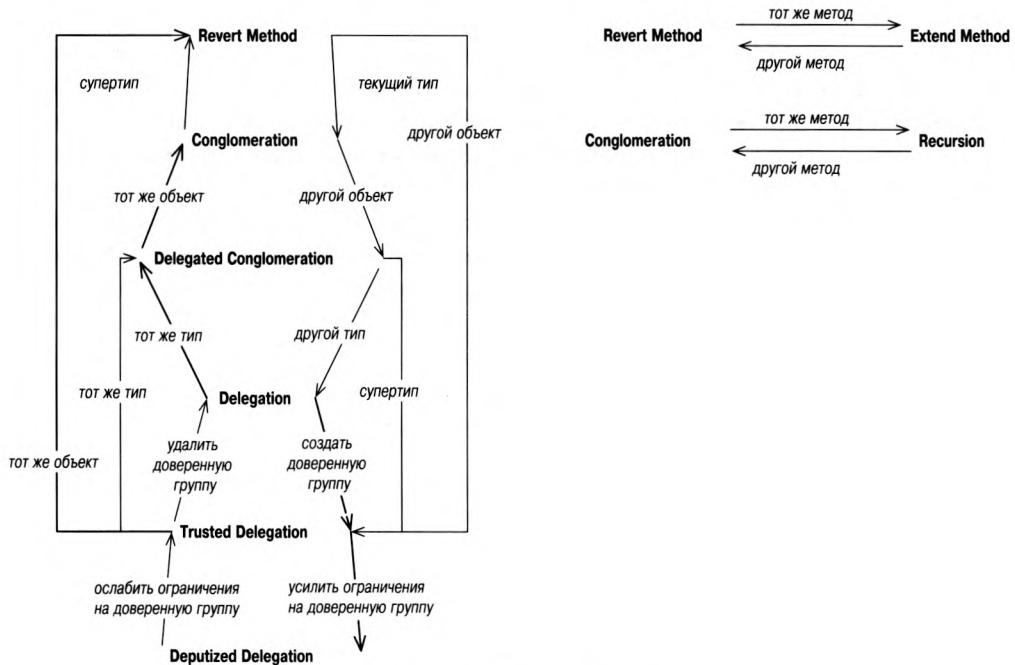


Рис. 4.35. Эффекты рефакторинга

4.4. Общая картина

До сих пор эта глава представляла собой ряд уроков по работе с элементарными шаблонами проектирования. Вы узнали, как составлять из элементарных и других шаблонов проектирования более крупные абстракции, имеющие солидные и пригодные для работы определения. Вы научились распознавать известные вам шаблоны, изучая новые шаблоны. Были рассмотрены изотопы, позволяющие отделить шаблоны проектирования от их реализации и тем самым обеспечивающие инкапсуляцию концепций проектирования и гибкость их выражения в коде. Вы даже получили представление о широко распространенных библиотеках инструментов для рефакторинга, увидели, как они сочетаются с элементарными шаблонами проектирования и как четко определенные отношения между элементарными шаблонами проектирования могут облегчить планирование рефакторинга.

Мы рассмотрели широкий круг тем, а теперь попробуем суммировать их в виде простых диаграмм. Следующие рисунки позволяют лучше понять, как элементарные шаблоны проектирования связаны между собой. Некоторые имена шаблонов окажутся новыми для вас. Мы их еще не обсуждали подробно, но вы их найдете в каталоге, который будет рассмотрен в следующих главах.

На рис. 4.36 показано, какие элементарные шаблоны проектирования используются другими шаблонами EDP, и отдельно выделены шаблоны, которые являются ядром, используемым в элементарных шаблонах проектирования, связанных с вызовом метода. На этом рисунке выделены группы шаблонов, образующие концептуальные комбинации. В некоторых случаях они используются неявно. Например, шаблон *Redirection* неявно использует шаблон *Retrieve*, но поскольку шаблон *Redirection* оперирует двумя объектами, шаблон *Retrieve* должен проявить себя в какой-то точке, чтобы сделать один объект доступным для другого при вызове метода из него. С другой стороны, шаблон *Inheritance* явно использует шесть элементарных шаблонов проектирования, в которых есть подклассы.

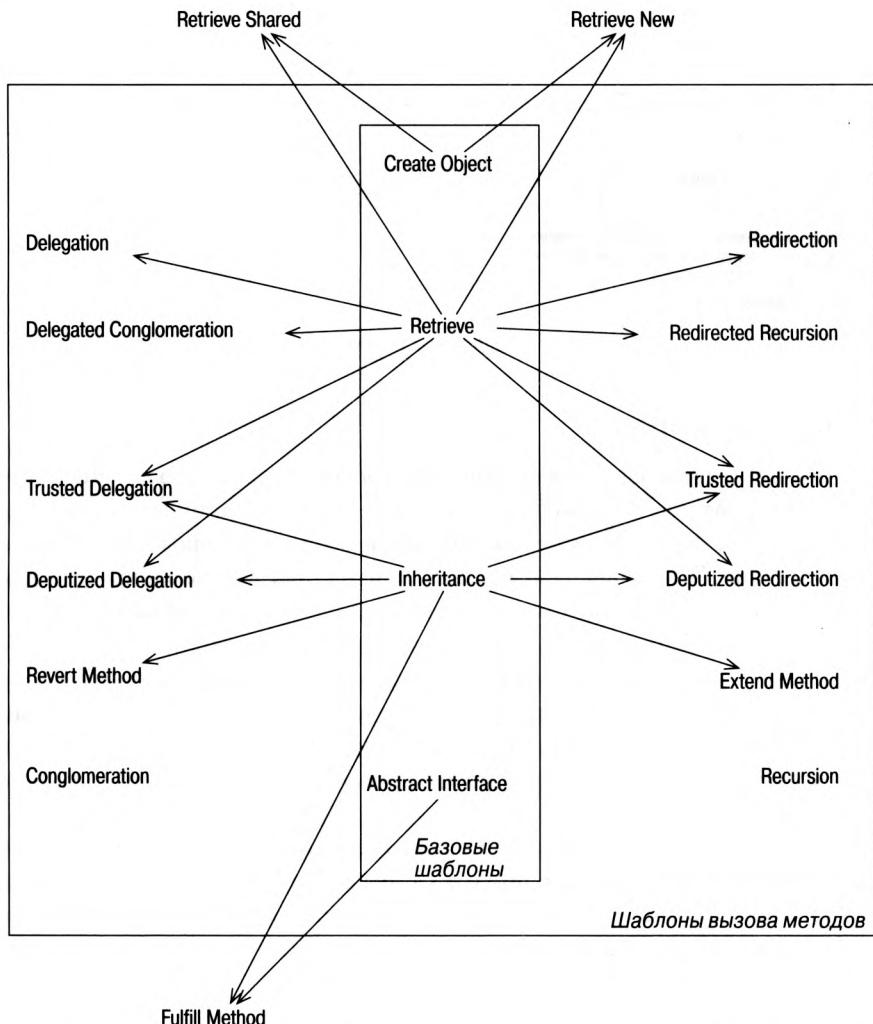


Рис. 4.36. Неявно используемые отношения между элементарными шаблонами проектирования и некоторыми другими шаблонами

Во-первых, обратите внимание на то, что шаблон *Retrieve* интенсивно используется в верхней части диаграммы. Элементарные шаблоны проектирования, использующие шаблон *Retrieve*, как и следовало ожидать, содержат два объекта. Эти шаблоны относятся к категории элементарных шаблонов вызова метода с разными объектами (*dissimilar-object method-call*). Ниже этих шаблонов расположены четыре элементарных шаблона проектирования.

Во-вторых, шаблон *Inheritance* доминирует в средней части диаграммы. Элементарные шаблоны проектирования, связанные с вызовом метода и использующие шаблон *Inheritance*, относятся к категории шаблонов, в которых типы объектов являются подклассами или классами одного уровня.

Обратите внимание на симметричность диаграммы по отношению к элементарным шаблонам вызова метода. Слева расположены элементарные шаблоны проектирования, содержащие разные методы, а справа — элементарные шаблоны вызова одного и того же метода.

В заключение отметим, что шаблоны, расположенные за пределами ядра и области шаблонов вызова метода, состоят из двух или более элементарных шаблонов проектирования. Хорошо известный шаблон *Fulfill Method*, например, использует шаблоны *Abstract Interface* и *Inheritance*.

В разделе 2.2.4 была показана часть пространства проектирования для элементарных шаблонов вызова метода. На рис. 4.37 и 4.38 это пространство показано полностью. На каждом из этих рисунков изображена половина пространства проектирования. Используя его как концептуальную схему, вы сможете понять, как элементарные шаблоны связаны один с другим.

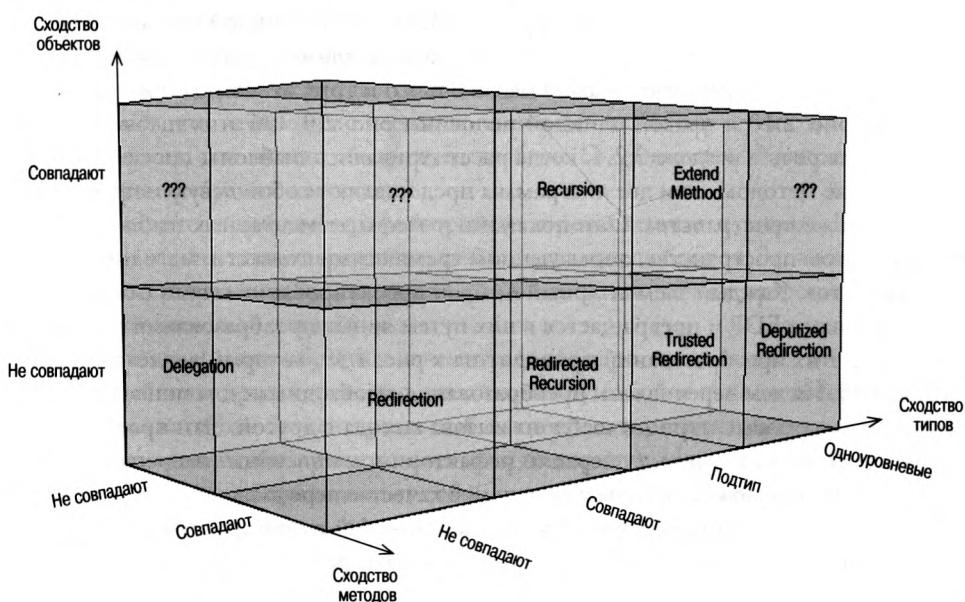


Рис. 4.37. Полное пространство элементарных шаблонов вызова метода: разные методы

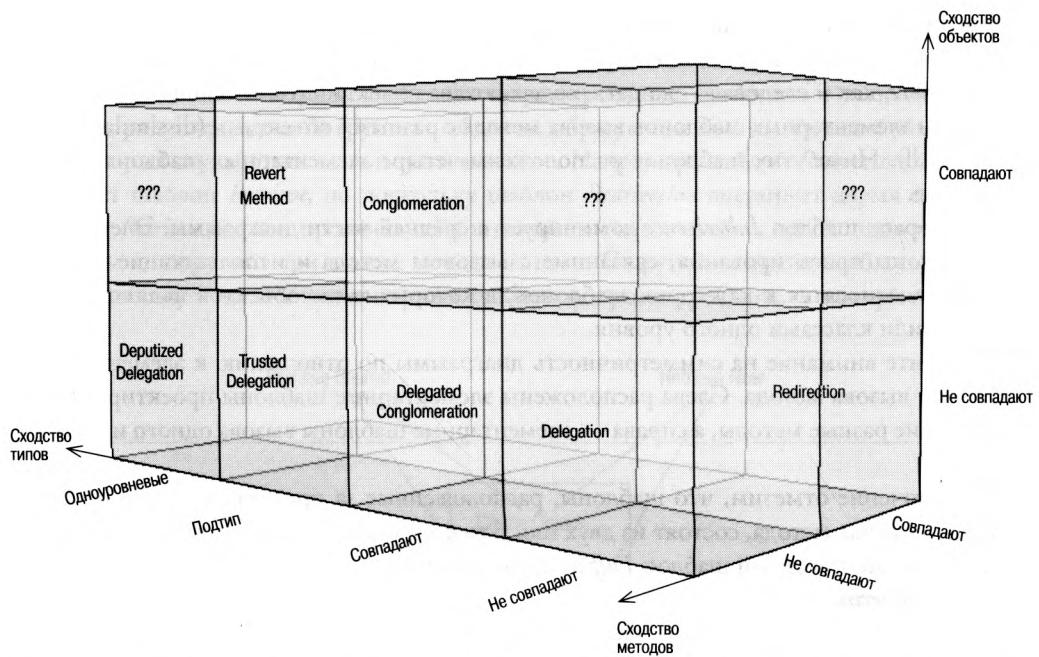


Рис. 4.38. Полное пространство элементарных шаблонов вызова метода: одинаковые методы

Эти две диаграммы соответствуют левой и правой частям рис. 4.36. На рис. 4.37 показаны все элементарные шаблоны вызова метода с разными методами; этот рисунок представляет собой комбинацию рис. 4.27 из раздела 4.3 и левой длинной половины рис. 2.9. Повернув этот параллелепипед, можно увидеть правое длинное ребро, как показано на рис. 4.38 с элементарными шаблонами вызова одного и того же метода. Рис. 4.38 — это комбинация рис. 2.10 и правой длинной половины рис. 2.9. Об этих шаблонах проектирования шла речь в разделе 2.2.4, когда рассматривались шаблоны проектирования с одним и тем же методом. Эти две диаграммы представляют собой левую и правую части одного и того же пространства. Они показывают место элементарных шаблонов проектирования в этом пространстве, определенном тремя осями схожести: методов, объектов и типов объектов. Каждый элементарный шаблон проектирования связан с окружающими его шаблонами EDP и превращается в них путем явных преобразований вдоль осей.

Описание этих преобразований приводит на к рис. 4.39, который является расширением рис. 4.35. На нем перечислены преобразования, необходимые для пошаговой трансформации одного элементарного шаблона вызова метода в другой. Эти преобразования представляют собой атомарные операции рефакторинга, описанные в предыдущем разделе. Эта диаграмма может оказаться полезной в качестве перечисления требований к изменениям системы. Зная конечную точку преобразований в пространстве элементарных шаблонов проектирования, можно точнее предсказать результат рефакторинга. Левая

часть диаграммы соответствует рис. 4.37 с разными методами, а правая — рис. 4.38 с одним и тем же методом. Если различие между методами исчезает, вы перемещаетесь слева направо, и наоборот.

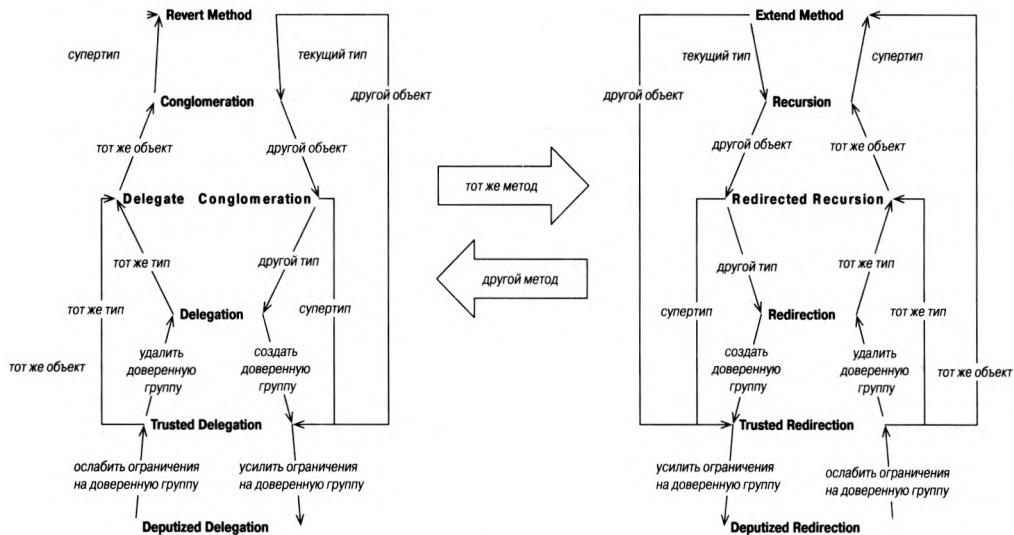


Рис. 4.39. Рефакторинг элементарных шаблонов вызова методов

Например, если вы реализовали экземпляр шаблона *Recursion* и должны распределить задачу между несколькими объектами, то стрелки, помеченные как “другой объект”, приведут прямо к шаблону *Redirected Recursion*. Если же вы обнаружили, что шаблон *Recursion* следует разделить на несколько разных подзадач, то вам придется ввести новые методы и сходство методов сохранить не удастся. В таком случае большая стрелка с меткой “другой метод” означает, что необходимо переместиться в левую часть рисунка, в соответствующую точку дерева; там можно обнаружить шаблон *Conglomerate*. Каждая точка решения приводит к требуемой концепции.

Применяя другие действия рефакторинга, перечисленные в книгах Fowler [19] и Kerievsky [24], можно также перемещаться по диаграмме шаблонов. Заранее зная, как изменяются элементарные шаблоны проектирования, можно точнее предсказать результат и избежать потенциальных проблем.

Эти четыре диаграммы образуют концептуальное ядро, позволяющее понять, как элементарные шаблоны проектирования связаны между собой и какие изменения они допускают. Читая каталоги элементарных шаблонов проектирования, можно возвращаться к этим диаграммам, чтобы лучше понять общую картину. Несмотря на то что каждый шаблон проектирования представляет собой самодостаточную концепцию, между ними существуют разные отношения, образующие более богатую систему для работы и анализа в целом.

4.5. Зачем читать приложение

Ну, я немного схитрил, сказав, что вы не обязаны понимать математические основы элементарных шаблонов проектирования, для того чтобы в них разбираться и правильно их использовать. Однако сейчас я собираюсь сформулировать несколько формальных утверждений, а вы можете либо поверить мне на слово либо прочитать приложение, чтобы убедиться самостоятельно в правильности моих слов. Я надеюсь на ваш скептицизм.

Во-первых, подчеркнем, что в совокупности элементарные шаблоны проектирования охватывают все возможные схемы в объектно-ориентированном программировании. Напомним, что эта книга касается только четверти всех существующих элементарных шаблонов проектирования. Практически невозможно написать объектно-ориентированную программу, не используя элементарных шаблонов проектирования. Да, я знаю, что это слишком смелое заявление. Но, пожалуйста, прочтите его формальное доказательство. Подсказка: в главе 2 мы установили, что бинарные отношения являются наименьшими отношениями, которые можно создать. Если имеется конечное количество сущностей, между которыми можно установить отношения, то должно существовать конечное количество возможных отношений. Насколько малым может быть это множество, если рассматривать только объекты, методы, поля и типы? Если вы ответите “Его размер примерно равен количеству элементарных шаблонов проектирования”, возьмите с полки пирожок. Именно так действуют элементарные шаблоны проектирования: они описывают все возможные наименьшие отношения в объектно-ориентированной программе на естественном языке, чтобы мы могли распространять их, понимать и более эффективно использовать. Эта книга — только начало; я надеюсь, что вы присоединитесь к нашей дискуссии.

Во-вторых, формализм, определенный ρ -исчислением, по существу, является источником элементарных шаблонов проектирования. Большинство шаблонов проектирования создается путем выявления повторяющихся решений с помощью анализа существующих систем программного обеспечения. Такой подход позволяет описать все, что мы делали раньше. Это очень важно, но элементарные шаблоны проектирования уникальны тем, что их можно вывести из первичных принципов, а не из опыта. Они всегда есть в существующих системах. Именно их экземпляры мы используем, чтобы выявить предназначение и написать спецификацию любых шаблонов. Однако пространство проектирования EDP, определенное ортогональными осями контекста, определяет каркас для заполнения скрытых пробелов и устанавливает отношения между элементарными шаблонами проектирования.

Разница заключается в том, что в прошлом механики использовали чертежи, созданные методом проб и ошибок, а современные инженеры могут смоделировать полный проект, не изготовив ни единой детали. В первом случае чертежи были описанием полученного опыта, а во втором прогнозирование основывается на формальных рассуждениях и моделировании. Элементарные шаблоны проектирования позволяют точно описывать систему и предсказывать последствия ее малейших изменений. Более того, как вы могли видеть при реконструкции шаблона *Decorator*, элементарные шаблоны проектирования дают возможность работать с более крупными абстракциями, варьируя уровень

детализации в соответствии с требованиями момента. Благодаря элементарным шаблонам проектирования можно описывать новые шаблоны проектирования, комбинируя их определения в терминах существующих абстракций. Они открывают действительно бесконечное поле возможностей.

Эти свойства означают, что элементарные шаблоны проектирования позволяют делать две чрезвычайно важные вещи, касающиеся систем программного обеспечения. Во-первых — определение и описание любого компонента объектно-ориентированного программного обеспечения так, чтобы это было понятно людям независимо от языка реализации. Мы можем выявлять эти концепции в коде и выражать их в доступном для людей виде. Мы показали, что эти небольшие концепции быстро объединяются в крупные, более интересные концепции, решающие реальные проблемы, например в шаблоны GoF. Эти концепции можно описывать в документах с разным уровнем детализации, а затем выбирать уровень абстракции в зависимости от требований момента.

Во-вторых, что еще более увлекательно, благодаря тому, что эти концепции и отношения между ними формализованы, процесс их извлечения, выявления, композиции и описания можно автоматизировать. Именно эту проблему решает система SPQR.

И это только начало.

4.6. Расширенные возможности

В этом разделе описаны некоторые современные идеи, которые могут оказаться интересными для читателей. Ни одна из них не нужна для понимания элементарных шаблонов проектирования или чтения каталога. При желании можете пропустить этот раздел (надеюсь, что вы этого все же не сделаете).

4.6.1. Специализированная документация и обучение

Система SPQR предназначена для автоматического документирования существующих реализаций в программном обеспечении и выявления концепций и абстракций, описанных в литературе по шаблонам проектирования. Выявление шаблонов проектирования помогает разработчикам создать более точную ментальную модель системы и документацию, которую никто не любит писать. Однако по мере разработки этой системы стало ясно, что мы шли по неверному пути.

Шаблоны проектирования понятны всем. Они хорошо документированы, широко распространены и встречаются во многих системах.

Итак, если вы нашли шаблон в своем программном обеспечении, отлично! Вы используете самые эффективные приемы для решения задач, которые другие уже решили. Примите поздравления!

Однако в целом задача документирования еще не решена. Вы все еще должны объяснить, как вы реализовали выбранный вами шаблон и как он взаимодействует с остальной системой. Помимо этого, тот факт, что вы решаете задачу, которую кто-то уже решил, *не делает* вашу систему уникальной. Шаблоны — это просто скелет, на который вы называете функциональные свойства и фрагменты кода, делающие вашу систему более привлекательной для клиентов и обеспечивающие ей место на рынке. Эти уникальные

разделы представляют собой фрагменты вашего программного обеспечения, которые необходимо описать документально. Ваши коллеги и те, кто унаследует вашу программу, поблагодарят вас за это.

Иначе говоря, части вашей системы, которые можно описать шаблонами проектирования, не обязательно являются именно теми частями, которые важно описать в документе. Разумеется, необходимо полностью выполнить процесс создания документации. Однако очерчивание частей систем, которые описываются шаблонами проектирования, позволяет понять, на что не следует тратить время при разработке документации. Это фрагменты кода, которые *не требуют интенсивного обучения* новых специалистов, которые можно *не учитывать* в ходе анализа последствий изменения системы и которые *не обязаны* оставаться в неприкосновенности, чтобы удовлетворять потребности клиентов.

К фрагментам, которые можно описать с помощью высокоровневых шаблонов проектирования, относится инфраструктура. Это объясняется тем, что именно инфраструктура обычно уже подробно разработана другими. Для того чтобы понять, где начинается и заканчивается ваш вклад, можно ограничиться только наиболее важными аспектами.

И разумеется, остальная часть документации будет на самом высоком уровне, потому что ее уже кто-то написал. Вам нужно просто на нее сослаться.

4.6.2. Метрики

В любой инженерной дисциплине для обеспечения обратной связи, измерения соответствия поставленным целям и понимания системы необходимы количественные показатели. Программная инженерия не является исключением, и элементарные шаблоны проектирования и соответствующий им формализм позволяют по-новому измерить то, что мы делаем.

Ясность выражения. Все мы сталкивались с программным обеспечением, которое было менее понятным, чем хотелось бы. Если нам везло, то нам попадалась программа, которую было очень легко читать. Почему? Что одно программному обеспечению оказывается понятным, а другое повергает в ужас?

Читабельность не сводится к выбору правильных имен и удобному форматированию. Читабельными считаются фрагменты программного обеспечения, четко выражающие концепции, которые они реализуют. Если чтение программы не требует от программиста больших умственных усилий, она считается ясной. Если программное обеспечение содержит правильные концепции, но они скрыты в клубке других сущностей, значит, будет трудно понять, чего хотел его разработчик. Это качественный процесс. Мы узнаем ясный код, когда его видим.

Как же сделать так, чтобы концепция была как можно более ясной, а ее реализация максимально понятной? Напомним, что мы говорили об изотопах. Мы можем измерить относительную ясность выражения концепции, оценив, насколько точно она соответствует наименьшим, недвусмысленным и самым прозрачным формам. Как правило, это относится к каноническим спецификациям конкретного шаблона проектирования. Если эта каноническая форма еще не стала ясной или не была сведена к простейшим формам, ее рано передавать на суд общественности. Спецификация шаблона проектирования, помимо всего прочего, должна быть ясной.

Каждый дополнительный шаг в цепочке зависимостей или лишний фрагмент в реализации элементарного шаблона проектирования затрудняет работу разработчика по выявлению отношений между конечными точками. Следовательно, можно измерить расстояние от оптимального варианта, подсчитав количество изотопных компонентов, которые не нужны для выражения концепции, но содержатся в реализации. В сущности, мы подсчитываем количество лишних компонента, необходимых для реализации отдельного шаблона или концепции.

Возможно, эти ненужные компоненты являются необходимыми и полезными в какой-то другой системе, но их наличие затемняет выражение концепции в нашей конкретной системе. Мы можем использовать эту метрику как показатель ясности выражения концепции в коде. Мы можем измерить концептуальную читабельность и сделать это автоматически. Более того, мы можем дать советы по повышению ясности этих концепций, предложив действия по рефакторингу на уровне EDP–EDP. Проектировщики должны решить, какие концепции являются самыми важными и требуют максимальной ясности, но при оценке плана рефакторинга им все равно требуются количественные оценки отклонения от оптимального варианта.

Плотность абстракций. Изотопы и элементарные шаблоны проектирования отлично подходят для точной оценки конкретных концепций, отраженных в коде, но не дают полной картины. Качество кода можно оценить иначе, перейдя на крупный масштаб. Абстракции позволяют представить систему, состоящую из компонентов, как нечто целое, охватив все детали сразу. Например, если сказать, что разделом кода является цикл `for`, то это будет более быстрым и ясным выражением концепции цикла, чем его описание на машинном уровне шаг за шагом. Аналогично каждый шаблон проектирования или концепция является абстракцией, которая содержит огромное количество информации при минимальном количестве выразительных средств. Кроме того, поскольку, как мы показали, крупные системы и высокоуровневые абстракции можно создавать из маленьких, можно описывать достаточно большие разделы системы сразу, если найти для них подходящие абстракции.

Следовательно, описав тысячу строк кода одной абстракцией, а не сотней абстракций, мы облегчаем себе работу. Мы можем работать с тысячами строк одновременно и не беспокоиться об их внутренней структуре. Таким образом, можно рассматривать, анализировать, понимать и, возможно, подвергать рефакторингу крупные разделы системы.

Кроме того, при прочих равных условиях реализация, которую можно описать с помощью небольшого количества высокоуровневых абстракций, предпочтительнее реализации, для описания которой требуется очень много компонентов. Это качество можно назвать *плотностью абстракций* (*abstraction density*) в коде. Она измеряется количеством абстракций, выраженных в коде с учетом их состава. Иначе говоря, абстракция, инкапсулирующая шесть небольших компонентов, считается более важной и предпочтительной, чем шесть небольших абстракций плюс еще одна маленькая абстракция. В каждом из этих случаев есть семь абстракций, но в первом случае для разработчика важна только одна из них, а во втором он должен работать со всеми семью абстракциями.

Ключевая фраза в предыдущем абзаце — “при прочих равных условиях”. Абстракция может состоять из многих порций информации, но не все порции информации являются

компонентами осмысленной абстракции. Получить приблизительную оценку плотности абстракций в коде можно, измерив количество его контекстно-свободных зависимостей. Напомним, что каждая зависимость представляет собой связь между двумя сущностями в коде; следовательно, каждая зависимость — это связь, которую разработчик обязан отследить, но абстракции позволяют этого избежать. Этот приблизительный показатель мы назовем *плотностью информации* (information density), которая выражается количеством зависимостей в расчете на строку кода.

Вычислив эту дробь, можно определить количество абстракций на единицу информации и вычислить *относительную плотность абстракций* (relative abstraction density — RAD). Этот анализ можно выполнить на уровне методов, классов, файлов и модулей. Он помогает распознать проблемные места и очаги потенциальных неприятностей, предупредив менеджера проекта о возможных последствиях. Излишне говорить, что такой код является неправильным.

Чем ниже относительная плотность абстракций, тем выше вероятность того, что данный фрагмент кода станет проблемным. Он может быть эффективным, оптимизированным, корректным и не содержать ошибок. Мы просто не можем считать его надежным, если потребуются изменения. Вероятно, это основное требование, которое инженеры предъявляют к коду в данном случае. Знание мест, в которых могут возникнуть проблемы, позволяет правильно спланировать свои действия.

С другой стороны, модуль может иметь очень высокую относительную плотность абстракций, но содержать ошибки. В этом случае можно решить двигаться вперед и “затоптать жучков”, поскольку высокий показатель RAD придает уверенность в том, что код является понятным и в нем легко найти ошибки. Вероятно, в этом случае расходы на обучение будут довольно низкими и можно будет привлечь к этой работе больше людей. Как видим, все зависит от получения правильной информации для принятия правильного решения о проекте.

4.6.3. Процедурный анализ

Выше мы придерживались негласного предположения о том, что языки реализации являются объектно-ориентированными. Кроме того, наши формализмы, диаграммы UML и шаблоны проектирования являются объектно-ориентированными, не так ли? Нет, не совсем так. Шаблоны проектирования должны рассматриваться как простые концепции, которые не отличаются от других. Помните, что некоторые из них, например шаблоны *Create Object* и *Inheritance*, можно прекрасно выразить в процедурных языках. Кроме того, именно в этих языках они впервые появились. Объектно-ориентированное программирование возникло потому, что оно воплотило наилучшие приемы, используемые в процедурных языках.

Оказывается, что при правильных предположениях процедурный код можно проанализировать на наличие шаблонов проектирования так же, как и объектно-ориентированный. Помните описание модели глобальных сущностей в языке C++ в разделе 2.2.2? В языке С все сущности являются глобальными, но к ним применяются те же приемы. В компании IBM Research мы научились успешно генерировать осмысленные диаграммы UML для систем, содержащих сотни тысяч строк кода, написанных исключительно на языке С.

Средством для этого стали элементарные шаблоны проектирования. Создание достаточно точно детализированного набора концепций позволило осуществить подробный анализ соответствия процедурных идиом объектно-ориентированным предположениям. В результате возникли совершенно новые перспективы для визуализации и анализа унаследованного кода.

4.7. Выводы

Я надеюсь, что наше быстрое введение в элементарные шаблоны проектирования было интересным. Я надеюсь также, что оно было поучительным. Программное обеспечение является одним из наиболее важных современных изобретений, требующих чрезвычайно глубокого анализа. В настоящее время мы представляем собой скорее алхимиков, чем инженеров, добывая самородки золота и не зная периодической таблицы. Элементарные шаблоны проектирования и связанные с ними исследования позволяют заложить основы для более строгого подхода к разработке программного обеспечения, которое все еще сильно зависит от человеческого фактора. Программное обеспечение представляет собой средство и результат взаимодействия между человеком и машиной, а попытка оптимизировать его для одной из сторон этого взаимодействия обязательно создает последствия для другой. Я предлагаю машинам предоставить обработку конструкций, а людям — обработку концепций, используя в качестве моста между этими “конечными точками” элементарные шаблоны проектирования.

Теперь можно приступить к анализу остальной части книги, и я надеюсь, что вы найдете в ней много интересного, как нашел я, когда ее писал. Если вы научитесь думать о системах с точки зрения их предназначения и фокусировать внимание на концепциях, образующих проект, а не на конструкциях, которые мы используем для общения с компилятором, вы сможете сконцентрироваться на задачах, которые мы решаем лучше машин.

Мы проектируем. Мы сотрудничаем. Мы общаемся. Мы объясняем. Мы строим.

Мы подгоняем современный мир.

Так давайте переведем его на новый уровень.

where $\rho_{\text{m}} = \rho_{\text{m}}(T)$ is the density of the melt at temperature T , $\rho_{\text{c}} = \rho_{\text{c}}(T)$ is the density of the crystalline state at temperature T , and $\rho_{\text{f}} = \rho_{\text{f}}(T)$ is the density of the frozen state at temperature T .

The second term in Eq. (1) is the enthalpy of crystallization per unit volume, ΔH_{c} , which is given by

$$\Delta H_{\text{c}} = \frac{\partial H}{\partial T} = \frac{\partial H}{\partial T} \left(\frac{\partial T}{\partial \mu} \right)_{V, P} = \frac{\partial H}{\partial \mu} \left(\frac{\partial T}{\partial V} \right)_{H, P} = \frac{\partial H}{\partial \mu} \left(\frac{\partial T}{\partial \mu} \right)_{V, H}^{-1} \quad (2)$$

where $H = H(\mu, T, V)$ is the free energy of the system, μ is the chemical potential, and P is the pressure.

The third term in Eq. (1) is the entropy of crystallization per unit volume, ΔS_{c} , which is given by

$$\Delta S_{\text{c}} = \frac{\partial S}{\partial T} = \frac{\partial S}{\partial T} \left(\frac{\partial T}{\partial \mu} \right)_{V, P} = \frac{\partial S}{\partial \mu} \left(\frac{\partial T}{\partial V} \right)_{H, P} = \frac{\partial S}{\partial \mu} \left(\frac{\partial T}{\partial \mu} \right)_{V, H}^{-1} \quad (3)$$

where $S = S(\mu, T, V)$ is the entropy of the system.

The fourth term in Eq. (1) is the enthalpy of fusion per unit volume, ΔH_{f} , which is given by

$$\Delta H_{\text{f}} = \frac{\partial H}{\partial T} = \frac{\partial H}{\partial T} \left(\frac{\partial T}{\partial \mu} \right)_{V, P} = \frac{\partial H}{\partial \mu} \left(\frac{\partial T}{\partial V} \right)_{H, P} = \frac{\partial H}{\partial \mu} \left(\frac{\partial T}{\partial \mu} \right)_{V, H}^{-1} \quad (4)$$

where $H = H(\mu, T, V)$ is the free energy of the system, μ is the chemical potential, and P is the pressure.

The fifth term in Eq. (1) is the entropy of fusion per unit volume, ΔS_{f} , which is given by

$$\Delta S_{\text{f}} = \frac{\partial S}{\partial T} = \frac{\partial S}{\partial T} \left(\frac{\partial T}{\partial \mu} \right)_{V, P} = \frac{\partial S}{\partial \mu} \left(\frac{\partial T}{\partial V} \right)_{H, P} = \frac{\partial S}{\partial \mu} \left(\frac{\partial T}{\partial \mu} \right)_{V, H}^{-1} \quad (5)$$

where $S = S(\mu, T, V)$ is the entropy of the system.

The sixth term in Eq. (1) is the enthalpy of vaporization per unit volume, ΔH_{v} , which is given by

$$\Delta H_{\text{v}} = \frac{\partial H}{\partial T} = \frac{\partial H}{\partial T} \left(\frac{\partial T}{\partial \mu} \right)_{V, P} = \frac{\partial H}{\partial \mu} \left(\frac{\partial T}{\partial V} \right)_{H, P} = \frac{\partial H}{\partial \mu} \left(\frac{\partial T}{\partial \mu} \right)_{V, H}^{-1} \quad (6)$$

where $H = H(\mu, T, V)$ is the free energy of the system, μ is the chemical potential, and P is the pressure.

The seventh term in Eq. (1) is the entropy of vaporization per unit volume, ΔS_{v} , which is given by

$$\Delta S_{\text{v}} = \frac{\partial S}{\partial T} = \frac{\partial S}{\partial T} \left(\frac{\partial T}{\partial \mu} \right)_{V, P} = \frac{\partial S}{\partial \mu} \left(\frac{\partial T}{\partial V} \right)_{H, P} = \frac{\partial S}{\partial \mu} \left(\frac{\partial T}{\partial \mu} \right)_{V, H}^{-1} \quad (7)$$

where $S = S(\mu, T, V)$ is the entropy of the system.

The eighth term in Eq. (1) is the enthalpy of sublimation per unit volume, ΔH_{s} , which is given by

$$\Delta H_{\text{s}} = \frac{\partial H}{\partial T} = \frac{\partial H}{\partial T} \left(\frac{\partial T}{\partial \mu} \right)_{V, P} = \frac{\partial H}{\partial \mu} \left(\frac{\partial T}{\partial V} \right)_{H, P} = \frac{\partial H}{\partial \mu} \left(\frac{\partial T}{\partial \mu} \right)_{V, H}^{-1} \quad (8)$$

where $H = H(\mu, T, V)$ is the free energy of the system, μ is the chemical potential, and P is the pressure.

The ninth term in Eq. (1) is the entropy of sublimation per unit volume, ΔS_{s} , which is given by

$$\Delta S_{\text{s}} = \frac{\partial S}{\partial T} = \frac{\partial S}{\partial T} \left(\frac{\partial T}{\partial \mu} \right)_{V, P} = \frac{\partial S}{\partial \mu} \left(\frac{\partial T}{\partial V} \right)_{H, P} = \frac{\partial S}{\partial \mu} \left(\frac{\partial T}{\partial \mu} \right)_{V, H}^{-1} \quad (9)$$

where $S = S(\mu, T, V)$ is the entropy of the system.

Спецификации элементарных шаблонов проектирования

Одновременно с изучением материала предыдущей главы, можно приступить к изучению первой части каталога EDP. Первые 16 шаблонов, представленных здесь, — это 4 фундаментальных шаблона объектно-ориентированного программирования и 12 элементарных шаблонов вызова метода. Однаждать из них мы уже рассмотрели, осталось еще пять.

Формат описания шаблона проектирования стал почти стандартным. Впервые он был использован в книге Gang of Four (GoF) [21], и я также придерживаюсь его в своей книге. Каждый шаблон состоит из нескольких разделов, начиная с имени, так что на него можно ссылаться. В разделе “Назначение” объясняется предназначение шаблона. В следующем разделе, “Мотивация”, излагается проблема, которую решает шаблон, а в разделе “Применимость” объясняется, как его следует (или не следует) использовать. Затем приводится пример структуры как в виде диаграммы UML, так и в виде раскрытоого компонента PINbox (там, где это возможно). Эту структуру не следует использовать механически. Помните, что это лишь пример. Если необходимо изменить реализацию, не изменив отношений, которые вас интересуют, то вы можете смело это делать. Основные концепции останутся неизменными.

В разделах “Участники” и “Отношения” излагаются концепции, указанные в их названиях. Затем спецификация шаблона содержит разделы “Результаты” и “Реализация”. Здесь приводится пример кода, но его следует рассматривать только как ориентир. Ваш язык реализации почти наверняка сильно повлияет на выражение шаблона проектирования.

Там, где это возможно, обсуждение концепций сопровождается примерами на разных языках программирования, чтобы показать разные способы реализации шаблонов проектирования, не изменяя их основ. Каждая спецификация завершается обсуждением связи шаблона с другими шаблонами проектирования.

Элементарные шаблоны проектирования разделяются на три категории: Object Elements (Элементы объектов), Type Relation (Связи между типами) и Method Invocation (Вызов метода).

Шаблоны Object Elements связаны с созданием и определением объектов: *Create Object* (Создать объект) и *Retrieve* (Извлечь). Шаблон *Create Object* описывает, когда, как и почему создаются объекты, в чем их преимущества над процедурными системами и почему они не являются просто синтаксическими конструкциями для облегчения чтения программ. Шаблон *Retrieve* описывает, как и почему объекты используются как поля внутри других объектов.

Категория Type Relation содержит два простых шаблона: *Inheritance* (Наследование) описывает основной способ эффективного повторного использования информации о типе и определений методов в объектно-ориентированных системах, а шаблон *Abstract Interface* (Абстрактный интерфейс) описывает способ отложенной реализации определений типов.

Последняя группа, Method Invocation, содержит остальные 12 шаблонов.

Несмотря на то что шаблоны, включенные в коллекцию, невелики и точно определены, они играют важную роль, поскольку повсеместно встречаются в объектно-ориентированном программировании и необходимы для формализации проектирования, в отличие от более абстрактных шаблонов проектирования. Каждый программист ежедневно использует эти шаблоны, как правило, почти не задумываясь. Поскольку одно из предназначений шаблонов проектирования заключается в переводе интуитивных и рефлексорных решений в область сознательных концепций, эти спецификации должны способствовать этому процессу и упрощать обсуждение основных принципов и их использование.

Кроме того, элементарные шаблоны проектирования являются основой для более точного описания и обсуждения наиболее передовых приемов использования шаблонов проектирования вообще, некоторые из которых остаются непонятными или забытыми в современной литературе. Эти элементарные шаблоны проектирования являются строительными блоками знаний, позволяя значительно изменить характер дискуссий о шаблонах проектирования.

Create Object

Порождающий шаблон

Назначение

Гарантирует, что вновь размещенные в памяти структуры данных соответствуют набору предположений и предварительных условий, прежде чем они будут использованы остальной частью системы, и что их можно использовать только заранее определенными способами.

Синоним

Instantiation (Реализация)

Мотивация

В любом сеансе выполнения программы на компьютере необходимо выделять участок памяти, чтобы хранить и обрабатывать данные. Данные без функций не порождают никаких действий. Функции без данных бессмысленны. Главный талант программиста заключается в том, чтобы связать данные и функции между собой так, чтобы они имели концептуальный смысл и правильно выражались в исходном коде. Созданный объект решает две задачи: установление корректного состояния данных по умолчанию и определение набора связанных между собой действий над этими данными.

В процедурных языках данные и действия не зависят друг от друга. Данные выражаются в виде структур, а действия — в виде функций. Данные — это не просто выделенная память: это еще и набор предположений о том, как их интерпретировать, задавать и обрабатывать. Кроме того, эти предположения определяют действия, которые можно применять к этим данным.

Если в программе требуются данные, они занимают фрагмент памяти, который во многих языках программирования просто резервируется и предоставляется в распоряжение программиста. Эти данные могут быть ассоциированы с типом, но могут и не иметь типа.

В большинстве случаев они не инициализируются, т.е. в памяти не хранится никакого значения по умолчанию. Программист должен самостоятельно заполнить выделенный фрагмент памяти начальными значениями или рискнуть и оставить его заполненным случайными данными, записанными там ранее. Этот процесс выполняется постоянно и каждым программистом. Разумеется, целесообразно было бы инкапсулировать эту инициализацию в отдельную вызываемую функцию, но это лишь перенесло бы проблему на новый уровень и теперь разработчик должен был бы помнить о необходимости вызывать функцию инициализации. Если этого не сделать, то позднее функции, оперирующие данным фрагментом данных, могут получить неправильные значения. И все же это требование обычно не автоматизируется и не регламентируется строго.

В листинге 5.1 показан этот сценарий на языке С. Допустим, что вы пишете низкоуровневую графическую библиотеку, создающую окна для приложения. В программе

предусмотрена функция инициализации, но до ее вызова в строках 31 и 32 данные, по существу, являются случайными. Я говорю “по существу”, поскольку некоторые реализации языка С устанавливают целые и действительные числа равными нулям по умолчанию, но это происходит только в *некоторых* реализациях на *определенном* аппаратном обеспечении. Как правило, значения просто являются остатками предыдущих данных. Поскольку листинг 5.1 является законченной программой, ее можно выполнить и увидеть, как ведет себя операционная система.

Листинг 5.1. Неинициализированные данные

```

1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4
5 struct WindowData {
6     int xPosition;
7     int yPosition;
8     int width;
9     int height;
10    char* title;
11 };
12
13 void
14 initializeWindowData( struct WindowData* wd ) {
15     wd->xPosition = 0;
16     wd->yPosition = 0;
17     wd->width = 600;
18     wd->height = 800;
19     // Выделяем память, достаточную для хранения строки
20     // (плюс 1 для завершающего NULL)
21     wd->title = (char*)malloc(
22         strlen("Стандартный заголовок") + 1) * sizeof(char));
23     strcpy(wd->title, "Стандартный заголовок");
24 }
25 int
26 main(int argc, char** argv) {
27     struct WindowData wd;
28
29     // Обычно эти функции выводят случайные данные
30     printf("width: %d\n", wd.width);
31     printf("title: %s\n", wd.title);
32     initializeWindowData(&wd);
33
34     // Эти функции всегда выводят 0 и "Значение по умолчанию"
35     printf("width: %d\n", wd.width);
36     printf("title: %s\n", wd.title);
37 }
38

```

Это приводит ко второй проблеме, связанной с обработкой правильно организованных данных правильно организованными способами: дело не только в том, что данные вначале могут быть некорректными, но и в том, что существует несколько ограничений, регламентирующих работу функций с этими данными. Функции, не предназначенные

для работы с конкретным фрагментом данных, могут работать неправильно из-за нарушения предположений о данных, хранящихся в памяти. Кроме того, неправильные предположения функции о данных могут их испортить. Ранее правильно организованные данные в результате могут принять неверный тип.

Часто данные и связанные одна с другой функции поставляются в виде библиотеки. Однако библиотека представляет собой всего лишь слабо определенную группу; система на ее организацию практически не влияет. Для улучшения ситуации в некоторых процедурных языках можно связать функции с данными непосредственно, тем самым инкорпорируя эти функции в структуры данных.

В языке С это можно сделать с помощью указателей на функции. Указатель на функцию, которую необходимо ассоциировать со структурой, можно непосредственно включить в структуру, а затем передать вместе с данными.¹ Это удобный способ указать, какое действие связано с конкретными данными, но он не защищает данные от вмешательства разработчика.

Для защиты данных необходима *инкапсуляция*, скрывающая данные от внешнего мира и предоставляющая четко определенные и ограниченные средства для доступа к данным и манипулирования ими. Данные, скрытые с помощью инкапсуляции, называются *закрытыми* (private).

Инкапсуляция предотвращает непосредственные манипуляции данными со стороны разработчика и ограничивает спектр функций, которые можно связать с этими данными. В процедурных языках также существует возможность реализовать инкапсуляцию, например, с помощью указателей. Этот метод называется *указателем на реализацию* (pointer-to-implementation — pimpl), *d-указателем* (d-pointer), *непрозрачным указателем* (opaque pointer) или *чеширским котом* (Cheshire cat).

Сочетание этих методов обеспечивает надежную защиту данных и согласованный набор действий над ними. Однако связывание и инкапсуляция функций допускаются лишь в нескольких процедурных языках. Кроме того, эти способы довольно запутанны, уязвимы для ошибок и требуют тщательного анализа и реализации. Что еще хуже, эти способы не регламентированы и не навязываются автоматически компиляторами или языками.

Листинг 5.2. Фиксированные начальные значения

```
1 struct WindowData {
2     int xPosition = 0;
3     int yPosition = 0;
4     int width = 600;
5     int height = 800;
6     char title[] = "Стандартный заголовок";
7 };
```

¹ Именно так выглядели первые системы С++, в которых основным инструментом был компилятор cfront. Если вас интересуют отличия объектно-ориентированных и процедурных языков программирования, это хороший объект для изучения.

С другой стороны, объект задается языком и представляет собой отдельную неделимую единицу, содержащую данные и применимые к ним методы, концептуально связанные между собой типом объекта. Методы, являющиеся частью типа объекта, должны иметь осмысленную связь один с другим и с данными. Данные легко защитить, потребовав, чтобы объект находился в связанном и точно определенном состоянии до того, как мы приступим к действиям над ним, причем можно гарантировать, что к этому объекту будут применяться только точно определенные операции.

В некоторых процедурных языках разработчик также может эмулировать инкапсуляцию и связывание методов в объекте, но не может гарантировать, что эти данные всегда будут связанными или что операции над объектом будут подчиняться заданным ограничениям. Почти всегда существует *какой-то* прием, компрометирующий описанные выше способы решения этой задачи.

Кроме того, что особенно важно, практик не может гарантировать, что в момент размещения записи в памяти ее содержание будет соответствовать *всем* установленным предположениям. Возвращаясь к предыдущему примеру, мы могли бы задать значения по умолчанию для структуры WindowData, как показано в листинге 5.2.

Теперь мы не обязаны вызывать функцию initializeWindowData(). Каждый раз, когда мы будем определять новую переменную типа WindowData, она уже будет заполнена правильными значениями.

Этот подход работает при условии, что значения по умолчанию никогда не изменяются. Изменение начальных значений требует редактирования и перекомпиляции всего кода. К сожалению, это нежелательно. Рассмотрим, что произойдет, если мы откроем новое окно для документа в большинстве современных приложений с графическим пользовательским интерфейсом. Новое окно открывается немного правее и ниже текущего окна, если оно было открыто, и, как правило, имеет заголовок "Untitled". Но если окно с таким именем уже существует и не было переименовано, тогда в заголовке появляется номер, например "Untitled 1" и "Untitled 2". Это поведение интерфейса необходимо обеспечить с помощью библиотеки, поэтому было бы хорошо иметь информацию, которая автоматически определяется при открытии нового окна типа WindowData.

Листинг 5.3. Динамическая инициализация

```
1 void
  initializeWindowData( struct WindowData* wd ) {
3     wd->xPosition = currentWindow()->xPosition + 10;
4     wd->yPosition = currentWindow()->yPosition + 10;
5     wd->width = currentWindow()->width;
6     wd->height = currentWindow()->height;
7     char * currTitle = currentWindow()->title;
8     int counter = currentUntitledCounter();
9     if (counter == 0) {
10         // Задаем как "Untitled": 8 символов + 1 NULL
11         wd->title =
12             (char*)malloc(9 * sizeof(char));
13         strcpy(wd->title, "Untitled");
14     } else {
```

```

15     // Добавляем символы и две цифры
16     // Не более 100 окон Untitled одновременно
17     wd->title =
18         (char* )malloc(12 * sizeof(char));
19     strcpy(wd->title, "Untitled ");
20     char num[3];
21     sprintf(num, 3, "%d", counter + 1);
22     strcat(wd->title, num);
23 }
};

```

Но мы не можем этого сделать. Описанные выше настройки, заданные по умолчанию, зависят от существующего состояния приложения и его документов, а мы не можем знать это, когда записываем или компилируем код. Разумеется, можно реализовать сбор информации и ее присваивание с помощью функции, как показано в листинге 5.3, но затем нужно будет вновь напомнить разработчику о том, чтобы он вызвал эту функцию. Использование статических значений, заданных по умолчанию, не решает нашу проблему так, как хотелось бы.

Эффективным решением было бы наличие инициализирующей функции, но это условие не является обязательным. Оно зависит от принятых правил, документации и дисциплины проектирования. Ни один из этих факторов не является точным или надежным. Следовательно, проектировщик не может гарантировать выполнение предположений относительно вновь размещенных в памяти данных. Злонамеренный, небрежный или ленивый программист может разместить структуру в памяти, а затем не вызвать процедуру инициализации, тем самым вызвав потенциально катастрофические последствия.

Альтернативой являются системы, основанные на объектах и классах. Когда объект размещается в памяти во время выполнения программы, он корректно инициализируется с учетом языка и среды. Все объектно-ориентированные среды обеспечивают аналогичный механизм в качестве фундаментальной части своей реализации. Этот механизм представляет собой точку крепления, в которой программист может создать функцию (которая обычно называется инициализатором или конструктором), выполняющую соответствующую настройку объекта.

Таким образом, еще до использования данных остальной частью системы гарантируется выполнение *любых* конкретных предположений, включая те из них, которые основаны на информации, доступной только во время создания объекта.²

Пользователь объекта не может обойти этот механизм, поскольку он предусмотрен языком и средой выполнения. Гипотетически злонамеренный, небрежный или ленивый программист теперь обезврежен и не может допустить ошибку. Поскольку ошибки, связанные с отсутствием инициализации, чрезвычайно трудно отследить и выявить, желательно их вообще предотвратить.

² Один из популярных языков является вопиющим исключением из этого правила: язык Objective-C не предусматривает одновременное выделение памяти и инициализацию, как большинство языков. Вместо этого он реализует идиому alloc-init, позволяющую разработчикам раздельно управлять памятью и инициализацией данных в объектах. Несмотря на то что это решение открывает возможности для оптимизации, оно может привести к ошибкам при разработке программ.

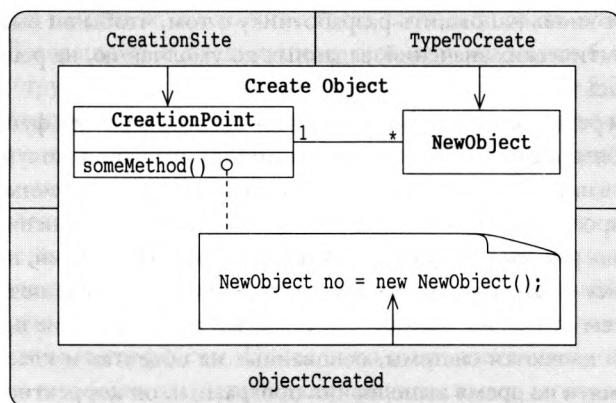
Внедрение лучших практических приемов в процедурных языках позволяет создать жесткий механизм на языковом уровне и придает объектам дополнительную полезность, помимо удобства.

Применимость

Шаблон *Create Object* используется в следующих ситуациях.

- Когда необходимо обеспечить представление данных и ограничить круг операций, которые можно выполнять над конкретными экземплярами.
- Когда необходимо обеспечить размещение экземпляров структур данных в памяти и гарантировать, что они не нарушают предварительные условия.

Структура



Участники

CreationSite

Объект, запрашивающий создание нового объекта типа `objectCreated`.

objectCreated

Объект, который должен быть создан.

TypeToCreate

Тип объекта `objectCreated`.

Отношения

Экземпляр объекта `CreationSite` отправляет запрос на создание нового экземпляра объекта `objectCreated` типа `TypeToCreate`. Точный механизм этого запроса зависит от языка программирования, но в целом он предусматривает запрос типа “самого” объекта. В действительности запрос отправляется в среду выполнения программы, при этом тип объекта задается, как аргумент, но синтаксис большинства языков создает впечатление, что запрос относится непосредственно к типу объекта. Внимательный читатель может заметить, что это циклическое определение: прежде чем создать новый

объект, необходимо иметь объект `CreationSite`. Начальный объект, запускающий всю эту цепочку, передается механизму выполнения программ на используемом языке. Например, в языке Java пользователь сообщает виртуальной машине Java (JVM — Java Virtual Machine), в каком классе искать метод `main()`, чтобы запустить всю систему. Это статический метод, поэтому он принадлежит классу, но поскольку он также доступен всем объектам этого класса, наше требование не нарушается.

Результаты

Большинство объектно-ориентированных языков программирования не допускают создание структур данных с помощью любого другого метода, но не требуют от разработчика определения процедуры инициализации. Как правило, существует стандартная процедура инициализации, выполняющая минимальные настройки.

Некоторые языки, хотя и являются объектно-ориентированными, допускают создание необъектных структур данных. Примерами являются языки C++ и Objective-C, являющиеся наследниками императивного языка С. Языки Python, Perl 6 и другие имеют аналогичные исторические причины, по которым они позволяют такое поведение.

Как только объект создан, корректными операциями над ним считаются только те методы, которые были предусмотрены разработчиком исходного класса. Изменить это положение вещей можно, например, с помощью шаблона *Inheritance*.

Когда объекты больше не нужны, их удаляют. Для очистки памяти используется специальная функция, которая называется деструктором и вызывается до того, как память будет полностью освобождена от данных. Эта функция позволяет задать любые постусловия, относящиеся к данным и ресурсам объекта. Хотя, кроме этого, желательно иметь хорошо оформленную и четную последовательность действий по удалению объектов, с вычислительной точки зрения освобождение памяти является лишь вопросом удобства. Если бы в распоряжении разработчика был неограниченный объем ресурсов, то объекты могли бы продолжать свое существование без использования неограниченное время и мы никогда бы к ним не вернулись. Только потому, что наши ресурсы ограничены, удаление объектов в большинстве систем является важным делом. Однако создание объектов вводит их в среду выполнения программ, где они могут использоваться в вычислениях, а значит, это действие является необходимым, а не просто удобным. С теоретической точки зрения некоторые типы объектов могут учитывать процесс удаления объектов (например, содержать фиксированное количество элементов в множестве), но этот вопрос относится к компетенции разработчика класса.

Реализация

В языке C++

В листинге 5.4 содержится класс, демонстрирующий инкапсуляцию, связанные функции и состояние, заданное по умолчанию. Прекрасные советы по разработке классов на языке C++ приведены в книгах Scott Meyers *Effective C++* [28] и *More Effective C++* [29].³

³ Перевод на русский язык: Мейерс С. Эффективное использование C++. — М.: ДМК, 2000; Мейерс С. Наиболее эффективное использование C++. — М.: ДМК, 2000.

Листинг 5.4. Реализация шаблона *Create Object*

```

1 class ThreadCount {
2 public:
3     // Конструкторы
4     ThreadCount() {
5         numThreads = getNumberOfRunningThreads();
6     };
7     ThreadCount(int newData) {
8         numThreads = newData;
9     };
10
11     // Методы доступа
12     int getNumThreads() {
13         return numThreads;
14     };
15     ThreadCount setNumThreads(int newData) {
16         if (newData > 1) numThreads = newData; }
17     };
18 private:
19     int numThreads;
20 };
21 int
22 main(int argc, char** argv) {
23     // Создает объект
24     ThreadCount mc;
25
26     // Уже настроен и готов к работе
27     // Выводит на печать количество выполняемых потоков
28     cout << mc.getPrivateData() << endl;
29
30     // Не изменяет данные, значение некорректное
31     mc.setPrivateData(-1);
32     // Изменяет данные, значение корректное
33     mc.setPrivateData(100);
34
35     // Выводит число 100
36     cout << mc.getPrivateData() << endl;
37 };
38

```

Родственные шаблоны

Шаблон *Create Object* является главной концепцией объектно-ориентированного программирования и встречается повсюду. Любой шаблон проектирования, предназначенный для создания или распределения объектов, основывается на этом элементарном шаблоне проектирования. К таким шаблонам относятся шаблоны *Retrieve New* (Найти новый) и *Retrieve Shared* (Найти совместно используемый), описанные в следующей главе, а также шаблоны создания объектов, описанные в книге GoF: *Abstract Factory* (Абстрактная фабрика) *Builder* (Строитель), *Factory Method* (Фабричный метод), *Prototype* (Прототип) и *Singleton* (Одиночка) [21].

Retrieve

Структурный шаблон

Назначение

Предназначен для использования объекта из другого нелокального источника в локальной области видимости, т.е. для создания отношения и связи между объектом в локальной области видимости и нелокальным источником.

Мотивация

Объекты представляют собой проверенный и понятный механизм инкапсуляции общих данных и методов и установления правил, как показано в описании шаблона *Create Object*. Однако отдельные объекты имеют очень ограниченное применение. Фактически, если в системе есть только один объект и он не связан с внешним источником, программу можно считать процедурной — все данные и методы являются локальными и полностью доступны друг для друга. Необъектные типы данных можно имитировать в любой объектно-ориентированной среде, допускающей использование функций-объектов и классов без методов. Следовательно, есть насущная потребность в хорошо продуманной методологии транспортировки объектов через границы объектов. Существуют две похожие ситуации, в которых эта методология необходима.

Простейшая ситуация возникает, когда внешний объект должен открыть доступ к своему полю. В более сложной ситуации внешний объект содержит вызываемый метод, а значение, возвращаемое этим методом, используется во внутренней области видимости.

Простейшая ситуация описана в листинге 5.5, содержащем программу на языке Java.

Листинг 5.5. Шаблон *Retrieve* с обновлением

```

1  public class SoundSettings {
2      public int volume;
3      // Настройка громкости плеера
4      public int offset;
5  };
6  public class MusicPlayer {
7      public void setVolume( SoundSettings ss ) {
8          // Это экземпляр шаблона Retrieve
9          this.settings.volume = ss.volume;
10     };
11    private SoundSettings settings;
12  };
13 }
```

Если использование внешних данных происходит в середине временного выражения, как в методе *adjustVolume1* в листинге 5.6, можно рассмотреть его эквивалент, например метод *adjustVolume2*.

Листинг 5.6. Шаблон *Retrieve* во временной переменной

```

1  public class MusicPlayer {
2      public void adjustVolume1( SoundSettings ss ) {
```

```

3      // Это тоже шаблон Retrieve
4      this.settings.volume =
5          this.settings.offset + ss.volume;
6      };
7  public void adjustVolume2( SoundSettings ss ) {
8      // Здесь шаблон Retrieve представлен явно
9      int tempVar = ss.volume;
10     this.settings.volume =
11         this.settings.offset + tempVar;
12     };
13 }

```

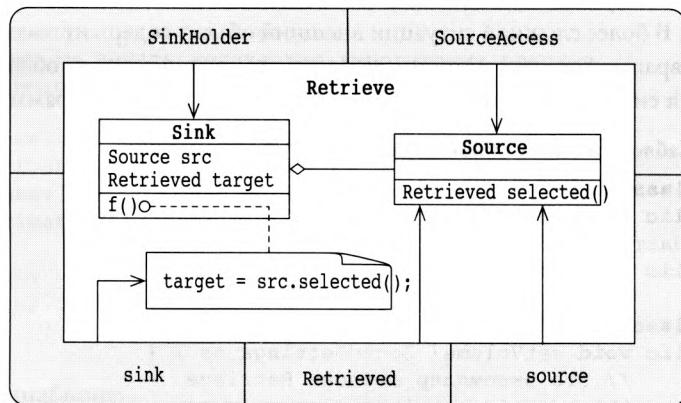
Применимость

Шаблон *Retrieve* используется в следующих ситуациях.

- Нелокальный объект предоставляет доступ к объекту, запрашиваемому для локальных вычислений, или сам объект
- с помощью значения, возвращаемого методом, или
- доступного поля объекта.

Структура

Обратите внимание на то, что имя `selected` может означать и метод, и открытое поле.



Участники

SourceAccess

Тип (или класс) объекта, содержащий элемент `selected`.

SinkHolder

Тип (или класс) объекта, содержащий элемент `target`, который должен принять новое значение.

Retrieved

Тип обновления значения и возвращаемого значения.

sink

Поле, принимающее новое значение.

source

Метод (или поле), порождающий новое значение.

Отношения

Это простое отношение состоит из двух объектов и двух методов. Отличительными факторами являются передача возвращаемого значения в пространство локального объекта и обновление локального поля с помощью извлеченного объекта. Этим локальным полем может быть определенное поле или временное значение в середине выражения.

Результаты

Связывание двух объектов и/или типов происходит постоянно, но это нельзя делать механически. Каждый раз связывая два объекта таким способом, вы создаете зависимость: целевой объект теперь зависит от объекта-источника. Убедитесь, что вам это необходимо.

В языках с динамическим контролем типов, таких как Python, Ruby или JavaScript, роли *SourceAccess*, *SinkHolder* и *Retrieved* не во всех ситуациях могут проявляться явно. Кроме того, разработчик может обеспечить строгий контроль типов, а может отказаться от этого.

Реализация

В языке C++

```

1  class SourceAccess {
2      public:
3          Retrieved source();
4      };
5
6  class SinkHolder {
7      Retrieved sink;
8      SourceAccess srcobj;
9  public:
10     void operation() { sink = srcobj.source(); }
11 };

```

В языке Java

```

1  public class SourceAccess {
2      public Retrieved source();
3  };
4
5  public class SinkHolder {
6      private Retrieved sink;
7      private SourceAccess srcobj;
8      public void operation() {
9          sink = srcobj.source();
10     };
11 };

```

В языке Python (обратите внимание на динамический контроль типов)

```
1  class SourceAccess(object):
2      def source(self):
3          pass // Возвращаемое значение
4
5  class SinkHolder(object):
6      def void operation(self):
7          self.sink = self.srcobj.source()
```

Родственные шаблоны

Retrieve — это элементарный шаблон проектирования, который встречается в любом другом шаблоне, содержащем два или более объектов, связываемых один с другим в ходе выполнения программы. (См. также шаблоны *Retrieve New* и *Retrieve Shared* в главе 6, описывающие владение объектом и его создание.)

Inheritance Relation

Отношения между типами

Назначение

Предназначен для повторного использования, добавления или модификации интерфейса другого класса, его реализации и функций.

Синонимы

IsA, Type Reuse

Мотивация

Часто существующий класс очень удобно использовать в качестве основы для создания нового класса. Интерфейсы этих классов могут почти совпадать, существующие методы могут *почти* полностью обеспечивать все потребности нового класса, а существующий класс может быть концептуально близким с тем, что вам требуется.

В таких ситуациях полезно и эффективно повторно использовать существующий класс, а не создавать новый класс “с нуля”. Это можно сделать, например, копируя и вставляя код в новый класс. Этот прием используется довольно часто, но имеет массу недостатков. Если в исходном коде содержалась ошибка, она будет продублирована. Если одна копия кода была улучшена, то любой программист, работающий с другой копией в другом месте, должен об этом знать. Метод копирования и вставки внешне выглядит быстрым и простым, но связывает друг с другом не копии кода, а *коллективы разработчиков*. Кроме того, во многих ситуациях копировать весь код неудобно, например при использовании коммерческих графических пользовательских интерфейсов.

Для повторного использования лучше всего применить шаблон *Inheritance*. Этот способ повторного использования кода поддерживают все объектно-ориентированные языки. Обычно он входит в число основных примитивов таких языков. В самых общих чертах этот шаблон создает отношение между *суперклассом*, или *базовым классом*, и *подклассом*, или *производным классом*. Суперкласс (будем называть его *Superclass*) — это класс, существующий в системе и предоставляющий минимальный интерфейс для концепций методов и/или структур данных.

Второй класс можно определить как *производный* от класса *Superclass*; назовем его *Subclass*. Класс *Subclass* наследует интерфейс и реализации всех методов и полей класса *Superclass*, тем самым создавая для программиста отправную точку для работы над классом *Subclass*.

Предположим, что вы пишете библиотеку графического интерфейса, которая может стать основой для разработки игрового движка. Каждая фигура, независимо от ее вида, должна содержать какую-то информацию. К параметрам этой фигуры относятся координаты на экране, цвет, линии границы и т.д. Вы могли бы добавить эту информацию и реализовать методы для работы с этими данными в каждом классе, реализующем фигуру. Или же вы можете использовать шаблон *Inheritance*, показанный в листинге 5.7.

Класс Shape содержит основную информацию для каждой фигуры в системе и базовый набор реализаций методов для работы с этими данными.

Обратите внимание на то, что в классе Square мы повторно объявили метод экземпляра `setWidth:andHeight`. Это пример *замещения (overriding)*, которое часто сопровождает шаблон *Inheritance*. Замещение позволяет настраивать некоторые методы из базового класса, создавая его новое определение. Вы можете использовать замещение для того, чтобы изменить поведение существующего метода, либо полностью заменить его, либо расширить его возможности.

Листинг 5.7. Пример использования шаблона *Inheritance* на языке Objective-C

```

1  @interface Shape
{
3      int xPos;
4      int yPos;
5      int lineWidth;
6      Color* fillColor;
7 }
8 - (void) setPosWithX: (int) x andY: (int) y;
9 - (void) setColor: (Color*) c;
10 - (void) setLineWidth: (int) lw;
11 @end

13 @interface Circle : Shape
{
14     int radius;
15 }
16 - (void) setRadius: (int) r;
17 @end

19 @interface Rectangle : Shape
20 {
21     int width;
22     int height;
23 }
24 - (void) setWidth: (int) w andHeight: (int) h;
25 @end

27 @interface Square : Rectangle
28 {
29 }
30 - (void) setWidth: (int) w andHeight: (int) h;
31 - (void) setSize: (int) s;
32 @end
33 
```

Листинг 5.8. Замещение реализации

```

1  @implementation Square
2  - (void) setWidth: (int) w andHeight: (int) h
3  {
4      if (w == h) {
5          [super setWidth: w andHeight: h];
6      }
7  } 
```

```

    } else {
7     printf("ERR: Width != height\n");
    }
9 }
- (void) setSize: (int) s
11 {
12     [self setWidth: s andHeight: s];
13 }
@end

```

В листинге 5.8 показано замещение метода `Square`. Мы хотели, чтобы клиенты методов `Square` и `Rectangle` по-прежнему могли использовать интерфейс `setWidth:andHeight`, поскольку квадрат — это лишь особая разновидность прямоугольника, и теперь мы можем сделать длины сторон прямоугольника одинаковыми. Мы добавляем проверку данных, а затем вызываем существующий метод из суперкласса. Полностью заменять реализацию необязательно, поскольку мы просто “заворачиваем” ее в процедуру проверки корректности данных.⁴

Иногда возникает необходимость сохранить неизменным исходное поведение кода, использующего класс с ошибочным методом, одновременно создав новый код с исправленным вариантом метода. Для этого можно создать новый класс, наследующий исходный, и заместить в нем неправильный метод его правильной версией. Существующий код может по-прежнему использовать базовый класс вместе с ошибкой, которая в нем содержится, но при этом новый код содержит исправленную версию. После проверки и тестирования старого кода можно полностью перейти на новую версию.

Например, рассмотрим фрагмент кода на языке Java, приведенный в листинге 5.9. Это совершенно искусственный пример, призванный продемонстрировать так называемую *ошибку поста охранения* (*fencepost error*). Сколько постов охранения вам потребуется, если длина вашей ограды — сто метров, а посты охранения расставлены через каждые десять метров? Если вы ответите “десять”, то окажетесь в хорошей компании. Многие забывают, что на нулевой отметке также нужен пост охранения. Это удивительно широко распространенная ошибка, известная также как *ошибка завышения или занижения на единицу*. Реализация метода `fencepostHeights` просто возвращает высоту требуемого поста охранения.

Листинг 5.9. Нарушение требования реализации

```

public class Fence {
2     int[] fencepostHeights;
3     public int getHeightOfPost( int post ) {
4         return fencepostHeights[post];
5     };
6 };
7 ...
Fence fence;
10 // Объект класса Fence заполняется в некоторой точке
...

```

⁴ Кстати, такой вызов версии одного и того же метода из суперкласса представляет собой пример применения элементарного шаблона проектирования *Extend Method*. Этот пункт заслуживает более глубокого изучения.

```

12 Scanner scanner = new Scanner (System.in);
    System.out.println(
14     "Введите номер поста:");
    int p = scanner.nextInt ();
16 System.out.format("Post # %d has height of: %d%n",
    p, fence.getHeightOfPost(p - 1));
18 // Исправление ошибки

```

Проблема в том, что массив номеров в языке Java нумеруется с нуля. Это значит, что первый элемент имеет индекс 0, второй элемент имеет индекс 1 и т.д. Дело в том, что люди обычно не нумеруют посты охранения таким образом. Допустим, что клиентский код запрашивает у пользователя, какую высоту должен иметь пост охранения, и передает эту величину объекту класса Fence с помощью вызова fence.getHeightOfPost(p). Клиентский код должен уточнить значение p, вычитя из него единицу, прежде чем посыпать его дальше, иначе высота будет присвоена посту, который расположен *за* требуемым постом. Клиентский код должен решить эту проблему. Кстати, легко доказать, что ошибка завышения или занижения на единицу — это вовсе не ошибка, а особенность языка Java. Во многом это правда. Точно так же работают языки С и С++. В коде ошибки нет; просто существует взаимное непонимание между программистом, реализовавшим класс, и разработчиками, которые используют этот класс. Программист, реализовавший класс, руководствовался разумными предположениями. Клиент, использующий класс, также не сделал никаких ошибок. Они просто пользуются *разными* предположениями. Посмотрим, что произойдет, если кто-то обнаружит это несоответствие в листинге 5.9 и решит исправить код (листинг 5.10).

Листинг 5.10. Очевидный, но вряд ли легко реализуемый способ

```

public class Fence {
2     int[] fencepostHeights;
    public int getHeightOfPost( int post ) {
4         return fencepostHeights[post - 1];
        // Исправление ошибки занижения на единицу ^^^^
6     };
}

```

Листинг 5.11. Исправление ошибки с сохранением старого кода

```

1 public class MendedFence extends Fence {
2     public int getHeightOfPost( int post ) {
3         return fencepostHeights[post - 1];
4     };
5 }

```

Это очевидный, прямой и простой способ исправления ошибки, при котором в каждый клиентский код вносится изменение, уточняющее значение, посыпаемое методу getHeightOfPost. Теперь клиентский код будет использовать высоту поста, расположенного *слева* от ожидаемого. И все же это не совсем то, что требуется. Если клиентов много или они имеют разные расписания работы, что происходит почти *всегда*, практически невозможно скоординировать все команды, чтобы они одновременно

исправили ошибку. В этом случае ошибка обычно остается неисправленной, и в каждый клиентский код необходимо инкорпорировать временное решение. Такое решение очень ненадежно.⁵

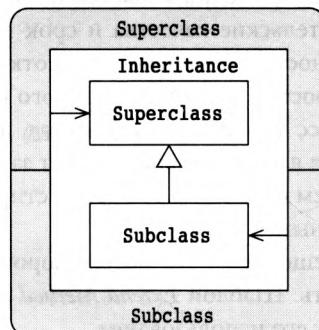
Намного лучше создать исправленную версию класса Fence, которую будут использовать новые клиенты, и позволить старым клиентам перейти на нее, когда они этого захотят. Это исправление показано в листинге 5.11, который основан на исходном классе Fence из листинга 5.9. Этот класс можно использовать в новом коде. Он намного проще и не требует от клиента помнить о необходимости исправить номер поста. Старый клиентский код может продолжать использовать старый класс, пока не произойдет переход на класс MendedFence. Как только новый класс станут использовать все клиентские программы, его можно будет объединить со старым классом и предоставить для всех. Обратите внимание на то, что это исправление может работать и в том случае, когда исходный код класса Fence недоступен. Обычно это происходит тогда, когда вы используете библиотеку, разработанную кем-то другим.

Применимость

Шаблон *Inheritance* используется в следующих случаях.

- Существующий класс предоставляет интерфейс, реализацию и данные, не полностью удовлетворяющие потребности нового класса.
- Копирование и вставка кода исходного класса либо нежелательны, либо невозможны, поскольку исходный код недоступен.

Структура



Участники

Superclass

Существующий класс в системе, который используется для создания нового класса.

Subclass

Вторичный класс, использующий интерфейс и реализацию первого класса.

⁵ Этой ошибке я посвятил один из моих автомобильных заказных номеров FEATURE, установленный на Фольксваген-жук 1960-х годов выпуска, когда я жил в Сиэтле. Я считаю, что любая ошибка, которая достаточно долго оставалась неисправленной, становится свойством (feature), на которое кто-то будет полагаться.

Отношения

Класс *Superclass* создает базовый набор интерфейсных методов, дополненных реализациями вспомогательного метода. Класс *Subclass* наследует все элементы интерфейса класса *Superclass* и по умолчанию все реализации, которые выборочно можно заменить новыми реализациями.

Результаты

Класс *Inheritance* — это мощный механизм, имеющий интересные ограничения и результаты работы. С одной стороны, из-за тонкостей объектно-ориентированной теории подкласс не может удалять методы и поля данных. Неформально говоря, наследование создает отношение *IsA* (“является разновидностью”) между подклассом и суперклассом.

Подкласс буквально является специализированной версией суперкласса. Интерфейс — это часть определения класса, которая описывает, чем класс *является*, поэтому, когда мы говорим, что суперкласс определяет интерфейс, а затем в подклассе часть этого интерфейса удаляется, значит, подкласс больше не является разновидностью суперкласса. Да, все это довольно сложно, но такова объектно-ориентированная теория. Лучше придерживаться более простой концепции: если подкласс невозможно описать как разновидность суперкласса, значит, это не подкласс.

Во многих языках программирования предусмотрено *множественное наследование* и допускается, что подкласс может иметь несколько суперклассов. Это действительно позволяет определить подкласс как разновидность каждого из суперклассов. Например, молоко — это и жидкость, и еда. Если класс *Milk* является наследником класса *Fluid*, то его можно описать с помощью свойств класса *Fluid*, например вязкости и точки замерзания. Если класс *Milk* одновременно является наследником класса *FoodItem*, то можно обсуждать его потребительские качества и срок годности. Однако в таких ситуациях возникает много сложностей, и многие языки отказываются от множественного наследования в пользу более простой модели одиночного наследования.

В некоторых языках подкласс может *скрывать* метод, делая его закрытым, но это не универсальное решение. Во всех языках подкласс может замещать метод и просто представлять пустую реализацию, тем самым, по существу, стирая функциональные свойства и сохраняя интерфейс неизменным.

Может показаться, что замещение — это потеря хорошего кода в базовом классе, и во многих случаях это так и есть. Шаблон *Extend Method* решает эту проблему, замещая метод, но при этом продолжает его использование.

В некоторых случаях желательно наследовать не весь существующий класс и его интерфейс, а только небольшую часть его функциональных свойств. Например, программист может быть не уверен в надежности методов, если исходный код класса недоступен, или, например, не желать интеграции большого класса из-за его небольшого сегмента, или иметь другие причины.

Рассмотрим использование шаблона *Redirection* при наследовании лишь части интерфейса. Экземпляр класса, который мы хотим повторно использовать, размещается в новом классе с помощью шаблона *Create Object*, а часть интерфейса, который мы хотим сохранить, дублируется, как показано на рис. 5.12. Здесь из класса *Square* был удален метод *setWidth:andHeight*, разработанный ранее. Недостатком такого подхода

является то, что необходимо реализовать оболочку для каждого *метода*, который нужно сохранить, и сделать это для всей иерархии наследования. Например, в листинге 5.12 методы из класса *Shape* завернуты в экземплярах шаблона *Redirection* (Переадресация).⁶ На самом деле мы удалили неиспользуемые методы исходного класса, но при этом потеряли преимущества полиморфизма в языках со статическим контролем типов. Языки с динамическим контролем типов, в частности те из них, которые позволяют определение метода в ходе выполнения программы, такие как Objective-C и JavaScript, хуже позволяют использовать типы более свободно, тем самым обеспечивая эффективный полиморфизм между классами, не связанными отношением наследования. Языки, основанные на прототипах, такие как JavaScript, Self и Lua, обеспечивают практически универсальный динамический контроль данных и полиморфизм безо всяких ограничений.

Листинг 5.12. Использование шаблона *Redirection* для скрытия части интерфейса

```

1  @interface Square
2  {
3      Rectangle rect;
4  }
5  - (void) setPosWithX: (int) x andY: (int) y;
6  - (void) setColor: (Color) c;
7  - (void) setLineWidth: (int) lw;
8  - (void) setSize: (int) s;
9  @end
10 @implementation Square
11 - (void) setPosWithX: (int) x andY: (int) y
12 {
13     [rect setPosWithX: x andY: y];
14 }
15 - (void) setColor: (Color) c
16 {
17     [rect setColor: c];
18 }
19 - (void) setLineWidth: (int) lw
20 {
21     [rect setLineWidth: lw];
22 }
23 - (void) setSize: (int) s
24 {
25     [rect setWidth: s andHeight: s];
26 }
27 @end

```

Сравните эту ситуацию с ситуацией, в которой мы хотим повторно использовать реализацию и/или данные, хранящиеся в классе, но не удовлетворены интерфейсом и хотим создать новый. В этом случае можно использовать объект в новом классе, как это делается в шаблоне *Redirection*, а доступ организовать с помощью элементарного шаблона проектирования *Delegation*. Тем самым мы уничтожаем исходный интерфейс и создаем новый, одновременно используя исходные данные и реализации. Это свойство обычно

⁶ Можно даже сказать, что шаблон *Inheritance* можно рассматривать как объединение нескольких экземпляров шаблона *Redirection*, одновременно использующих один объект *Redirect*.

используется в классах, играющих роль фасада, который должен осуществить адаптацию интерфейсов.

Эти приемы, позволяющие обойти ограничения шаблона *Inheritance* с помощью шаблонов *Redirection* и *Delegation*, получили настолько широкое распространение, что во многих языках стали поддерживаться естественным образом, например в языке C# с помощью ключевого слова `delegate`.

Реализация

Механизм создания отношения наследования в разных языках реализуется по-разному, но языки со статическим контролем типов всегда предоставляют четкий синтаксис для наследования. Языки с динамическим контролем типов не всегда явно выражают это отношение.

В языке C++

```
1 class Superclass {
2 public:
3     Superclass( );
4 };
5 class Subclass : public Superclass {
6 public:
7     Subclass( );
8 };
```

В языке Python

```
1 class Superclass(object):
2     def __init__(self):
3         pass
4 class Subclass(Superclass):
5     def __init__(self):
6         Superclass.__init__(self)
```

Родственные шаблоны

Шаблон *Inheritance* используется повсеместно. Это главный компонент любого элементарного шаблона проектирования, использующего сходство подклассов или одноуровневых классов. В частности, см. шаблоны *Trusted Delegation* (Доверенное делегирование), *Deputized Delegation* (Замещенное делегирование), *Trusted Redirection* (Доверенная переадресация), *Deputized Redirection* (Замещенная переадресация), *Revert Method* (Обратный метод) и *Extend Method* (Расширенный метод). Кроме того, шаблон *Inheritance* используется в шаблоне *Fulfill Method* (Выполнение метода). Поскольку каждый из этих шаблонов является основой для создания других составных шаблонов, шаблон *Inheritance* заслуживает самого пристального внимания.

Дальнейшую информацию об использовании шаблонов *Delegation* и *Redirection* для преодоления ограничений шаблона *Inheritance* можно найти в разделах, посвященных этим шаблонам.

Abstract Interface

Зависимость между типами

Назначение

Предназначен для создания общего интерфейса для семейства типов объектов без реализации реальных операций. В этом сценарии подклассы обязаны осуществлять соответствующие реализации методов, поскольку их реализаций “по умолчанию” не существует.

Синонимы

Virtual Method (Виртуальный метод), *Polymorphism* (Полиморфизм), *Defer Implementation* (Отложенная реализация)

Мотивация

Если существует иерархия классов, использующая шаблон *Inheritance* и соответствующая концептуальному проекту, часто возникает ситуация, в которой просто невозможно реализовать осмысленную реализацию метода в корне иерархии классов. Нам известно, что он должен делать *в принципе*, но мы не готовы уточнить его детали.

Допустим, что мы моделируем поведение животных. Грубо говоря, все животные имеют определенные повадки и потребности. Они едят, растут и (как правило) передвигаются. Что они едят, зависит от вида животных, но это действие почти всегда связано с глотанием. Некоторые животные на самом деле не глотают пищу, но эти варианты являются скорее исключением из правила. На протяжении своей жизни животные проходят ряд стадий, например созревание, юность и взросление. Детали и временные характеристики этих стадий очень сильно зависят от вида, но старение животных можно смоделировать в принципе.

В то же время моделирование передвижения создает проблемы. Мы можем указать конечный пункт передвижения животного, но реализовать эту функцию сложнее: рыбы, птицы и сухопутные животные двигаются по-разному. Для них невозможно реализовать какую-то универсальную функцию, которая моделировала бы их движения, не потребовав *массу* уточняющих деталей, зависящих от вида животного. Однако, реализуя метод moveTo, мы планируем в будущем уточнить его в подклассах. Эта ситуация иллюстрируется в листинге 5.13.

Листинг 5.13. Почти все животные передвигаются, но делают это по-разному

```
1 public class Animal {
2     public void eatFood( FoodItem f ) {
3         this.ingest(f);
4         this.digest(f);
5     };
6     public void matureTo ( TimeDuration age ) {
7         if (age > gestation) {
8             this.beAJuvenile(age);
9         } else if (age > maturation) {
10            this.beAnAdult(age);
```

```

11         } else if (age > longevity) {
12             this.die();
13         }
14     };
15     public void moveTo( Location dest ) {
16         // Хм... а что же здесь написать?
17     };
18 };

```

Это значит, что не следует предлагать сразу все реализации, поскольку, как бы мы их ни написали, они, скорее всего, окажутся ошибочными.

Мы хотим гарантировать, чтобы каждый объект класса *Animal*, независимо от подкласса, который его описывает, можно было *заставить* двигаться, но мы не в состоянии описать заранее, *как* он должен двигаться так, чтобы это относилось ко всем или почти ко всем животным. Мы хотим предоставить *интерфейс* — способ активизации действий, — но без описания самих действий.

Напомним, что в шаблоне *Inheritance* действия в типах описываются методами, определенными внутри этого типа. Следовательно, для того чтобы описать действие, относящееся ко всем подклассам, необходимо создать определение метода, но при этом мы не можем создать его реализацию.

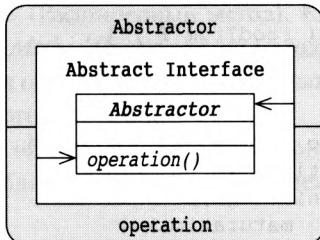
Если оставить пустую реализацию метода, как показано в листинге 5.13, то на самом деле мы создадим реализацию по умолчанию, которая просто ничего не делает. Если подкласс не выполнит замещение метода, то бедное животное, описанное этим подклассом, не сможет двигаться вообще. Несмотря на то что в природе существуют неподвижные животные, они являются исключением из правила. Следовательно, в таких случаях необходимо предусмотреть напоминание о том, что в подклассах необходимо создать реализацию. Для этого необходимо создать *абстрактный интерфейс* этого метода.

Применимость

Шаблон *Abstract Interface* используется в следующих ситуациях.

- Реализация метода либо неизвестна в момент определения класса, либо не имеет разумного обоснования.
- Интерфейс метода определить *можно*.
- Функциональные свойства метода уточняются в подклассах.

Структура



Участники

Abstractor

Класс, объявляющий интерфейс для операции метода `operation`.

operation

Абстрагируемый метод: определение метода задавать нельзя.

Отношения

Класс `Abstractor` определяет интерфейс для метода, который будет реализован во всех подклассах. (Между классом `Abstractor` и пока неизвестным подклассом здесь возникает неявное отношение.)

Результаты

Шаблон *Create Object* позволяет создавать объекты, а шаблон *Retrieve* показывает, как заполнить поля во вновь созданном объекте. Шаблон *Abstract Interface* отличается тем, что указывает *отсутствие* реализации метода; вместо демонстрации способа заполнения метода он показывает способ переноса определения метода в другую точку проекта. Остальная часть решения описана в шаблоне *Fulfill Method*.

В данном случае метод объявляется для того, чтобы определить соответствующий интерфейс, удовлетворяющий наши потребности в принципе, но тело метода остается неопределенным. Это *не значит*, что мы просто определяем пустой метод, который ничего не делает; этот метод *не имеет определения вообще*. Это критический момент, который начинающие программисты, использующие объектно-ориентированные языка, часто не понимают. Такое определение в разных языках реализуется по-разному, как показано в разделе “Реализация”.

Разные языки также по-разному используют этот шаблон. В языке C++, например, предполагается, что любой класс, содержащий хотя бы один метод, который является *абстрактным интерфейсом*, является *абстрактным классом* и, следовательно, не может иметь объектов. Только подклассы могут содержать определение для абстрактного метода с помощью шаблона *Fulfill Method*, который можно включить в объекты с помощью шаблона *Create Object*.

Язык Java реализует еще более строгую концепцию. Как и в языке C++, абстрактный класс может содержать один или несколько абстрактных методов. Однако, если класс содержит только абстрактные методы, в языке Java предусмотрена конструкция `interface`. Здесь термин *интерфейс* означает сущность, похожую на класс, которая содержит только абстрактные методы, а не интерфейс для отдельного метода. Как и абстрактный класс, интерфейс в языке Java не предполагает создания объектов. Однако в отличие от абстрактных классов, которые могут иметь как обычные, так и абстрактные методы, интерфейс в языке Java не может содержать обычных определенных методов вообще. Классы должны содержать хотя бы один определенный метод, а интерфейсы состоят только из абстрактных методов. Хотя в языке Java существует только одиночное наследование классов, он допускает множественное наследование интерфейсов, благодаря чему существует возможность создавать композиции новых интерфейсных классов любой степени детализации.

Модели наследования более широко описаны в разделе *Inheritance*.

В языке Python поддержка абстрактных методов добавлена только с версии 2.6 стандартной библиотеки в виде модуля `abc`, реализующего абстрактный базовый класс. Версия Python 3.0 использует немного более удобную систему обозначений, которую можно увидеть в разделе “Реализация”. В отличие от языков Java и C++, язык Python допускает реализацию по умолчанию, которую могут вызывать из подклассов, чаще всего с помощью элементарного шаблона проектирования *Extend Method*. Однако подклассы по-прежнему обязаны замещать методы и предусматривать их реализацию.

Реализация

В языке C++ (метод, приравненный к нулю, называется *чисто виртуальным*)

```

1  class AbstractOperations {
2  public:
3      virtual void operation() = 0;
4  };

6  class DefinedOperations :
7      public AbstractOperations {
8  public:
9      void operation();
10 };

12 void
13     DefinedOperations::operation() {
14 // Выполняет свою работу
15 };

```

В языке Java (метод включается в *интерфейсный* или *абстрактный класс*)

```

1  public interface AbstractOperations {
2      public void operation();
3  };

5  public abstract class SemiDefinedOperations {
6      public abstract void operation2();
7      public void operation3() {};
8  };

9  public class DefinedOperations
10     extends SemiDefinedOperations
11     implements AbstractOperations {
12     public void operation() {
13         // Выполняем соответствующую работу
14     }
15     public void operation2() {
16         // Выполняем соответствующую работу
17     }
18 };
19 
```

В языке Python 3.x

```
1 class AbstractOperations(metaclass=ABCMeta):
2     @abstractmethod
3     def operation(self, ...):
4         // Допускается определение по умолчанию
5         return
6
7 class DefinedOperations(AbstractOperations):
8     def operation(self, ...):
9         // Выполняем соответствующую работу
10        pass
```

Родственные шаблоны

Для того чтобы создать подкласс, обеспечивающий реализацию метода, используемого в шаблоне *Abstract Interface*, очевидно, следует применить шаблон *Inheritance*. Полное описание этой конструкции приведено в разделе *Fulfill Method* в главе 6.

Delegation Behavioral

Поведенческий шаблон

Назначение

Шаблон предназначен для того, чтобы переслать, или делегировать, часть работы другому методу или объекту.

Синонимы

Messaging (Передача сообщений), *Method Invocation* (Вызов метода), *Calls* (Вызовы), *The Executive* (Выполнение)

Мотивация

При работе с объектами часто возникает ситуация, в которой часть функциональных возможностей может предоставить “какой-то другой объект”. Шаблон *Delegation* представляет собой наиболее общую форму вызова метода из одного объекта в другом объекте. Он позволяет одному объекту посыпать сообщение другому объекту, чтобы тот выполнить часть работы. Объект, получающий вызов, может вернуть данные, но может и не вернуть.

В качестве примера из реальной жизни рассмотрим работу корпорации. Цель генерального директора — успешная работа компании. Он распределяет между подчиненными разные задания, причем очень редко два подчиненных выполняют одно и то же поручение. Например, финансовый директор обеспечивает соответствие финансовых отчетов государственным стандартам, технический директор обеспечивает выполнение технологического процесса и т.д. Каждое задание отличается от остальных, но в целом все они представляют собой части общей задачи — успешного функционирования компании. В листинге 5.14 показана модель такой ситуации.

Листинг 5.14. Генеральный директор делегирует ответственность

```
public class CEO {  
    2     FinanceExec vpFinance;  
    3     ResearchExec vpResearch;  
    4     TechnologyExec cto;  
    5     public void runCompany () {  
    6         vpFinance.ensureFinancialCompliance();  
    7         vpResearch runResearchDivision();  
    8         cto.manageTechnology();  
    9     };  
10 }
```

В листинге 5.14 заслуживает внимания пример *синхронного* вызова метода. В этой программе, написанной на языке Java, при вызове метода *runCompany* финансовому директору поручается руководить финансами, и *только когда он закончит отчет*, директор по исследовательской работе получает команду *начать выполнение* своего задания. Эти

задачи синхронизированы с помощью вызовов. Сначала один вызов, потом — другой, и выполнение задачи происходит в определенном порядке.

В реальной жизни генеральный директор так не делает. Он просит своих заместителей выполнять задания одновременно. Затем, когда каждое из заданий выполнено, ему сообщают о результатах. Это пример *асинхронного* выполнения задания. Делегаты получают команду выполнять свои задания параллельно.

Такую асинхронную, или параллельную, работу в большинстве объектно-ориентированных языков программирования трудно реализовать, хотя некоторые из них имеют потоковые библиотеки, такие как API FutureTask в языке Java. Другие языки, особенно функциональные, такие как Go и Erlang, имеют естественную поддержку параллелизма с помощью асинхронных вызовов, но их синтаксис совершенно другой. Как бы ни называлась эта тема, “асинхронные вызовы” или “параллельное программирование”, она выходит далеко за рамки нашей книги. Однако стоит подчеркнуть, что каждый из описанных в книге элементарных шаблонов проектирования может применяться для асинхронного вызова.

Делегирование довольно часто встречается в проектах. В языке C# оно обеспечивается с помощью ключевого слова `delegate`. Это свойство языка инкапсулирует функцию в объекте, чтобы ее можно было передать, как обычный объект. В таком случае этот объект можно вызывать, как обычный метод. Вложенный объект, по существу, является невидимым.⁷ Дело в том, что на инкапсулируемый метод не накладывается никаких ограничений, кроме того, что он должен соответствовать заранее заданному типу аргумента и типу возвращаемого значения. Вызываемый объект-делегат, очевидно, отличается от вызывающего объекта, типы объектов не должны взаимодействовать, а инкапсулируемый метод может называться как угодно. Это самый общий вариант шаблона *Delegation*.

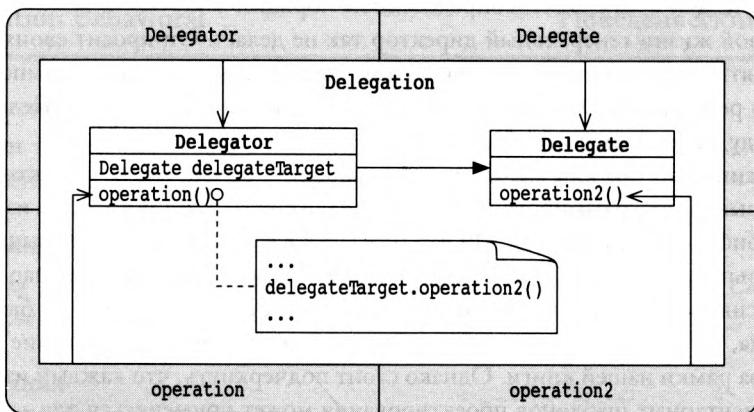
Применимость

Шаблон *Delegation* применяется в следующих ситуациях.

- Другой объект может выполнить часть работы, порученной текущему методу.
- Другой объект не обязан иметь доступ к данным текущего объекта для выполнения своей работы.
- Между двумя объектами нет отношений.

⁷ В языке C++ существует аналогичное понятие под названием *функция*, основанное на механизме вызова функции `operator()`, но язык C# представляет собой шаг вперед, потому что позволяет выполнять настраиваемую упаковку существующих методов в объект для передачи другим объектам.

Структура



Участники

Delegator

Тип объекта, посылающего сообщение объекту типа *Delegate*.

operation

Метод в классе *Delegator*, который должен выполняться при передаче сообщения, — точка вызова метода *operation2*.

Delegate

Тип объекта, получающего сообщение и содержащего вызываемый метод.

operation2

Вызываемый метод.

Отношения

Это простое бинарное отношение: один метод вызывает другой, как в обычных процедурных системах, со всеми преимуществами и недостатками таких вызовов.

Однако, поскольку мы работаем в объектно-ориентированной среде, у нас есть ряд дополнительных потребностей. Нам необходимо, чтобы вызываемый объект был видимым из точки вызова либо в качестве локальной переменной, либо с помощью шаблона *Retrieve*. Кроме того, очевидно, что вызываемый метод должен быть видимым для других объектов.

Результаты

Все операции между двумя объектами можно описать с помощью шаблона *Delegation*, но намного больший интерес представляет описание дополнительных атрибутов данного отношения. Уточнения шаблона *Delegation*, настроенные на специальные задачи и потребности, можно найти в описании элементарного шаблона проектирования *Method Invocation*. Этот шаблон представляет собой обобщенную форму вызова и является ключевой концепцией более общих абстракций, таких как *Bridge* или *Adapter* в их объектно-

ориентированной форме, в которой необходимо обеспечить точно определенный и эффективный переход между двумя интерфейсами, которые во всех других аспектах никак не связаны один с другим. Если интерфейсы связаны между собой систематическими отношениями, такой шаблон использовать нецелесообразно. Для шаблона *Delegation* интерфейсы и типы должны быть независимыми друг от друга.

Реализация

Шаблон *Delegation* является наиболее обобщенной формой вызова метода в объектно-ориентированном программировании. Он описывает, как два объекта взаимодействуют один с другим как отправитель и получатель сообщений, выполняя работу и возвращая значения.

В языке C++

```

1  class Delegator {
2  public:
3      Delegatee target;
4      void operation() {
5          // Первая часть задания...
6          target.operation2();
7          // Вторая часть задания...
8      };
9  };
10
11 class Delegatee {
12 public:
13     void operation2();
14 };

```

Родственные шаблоны

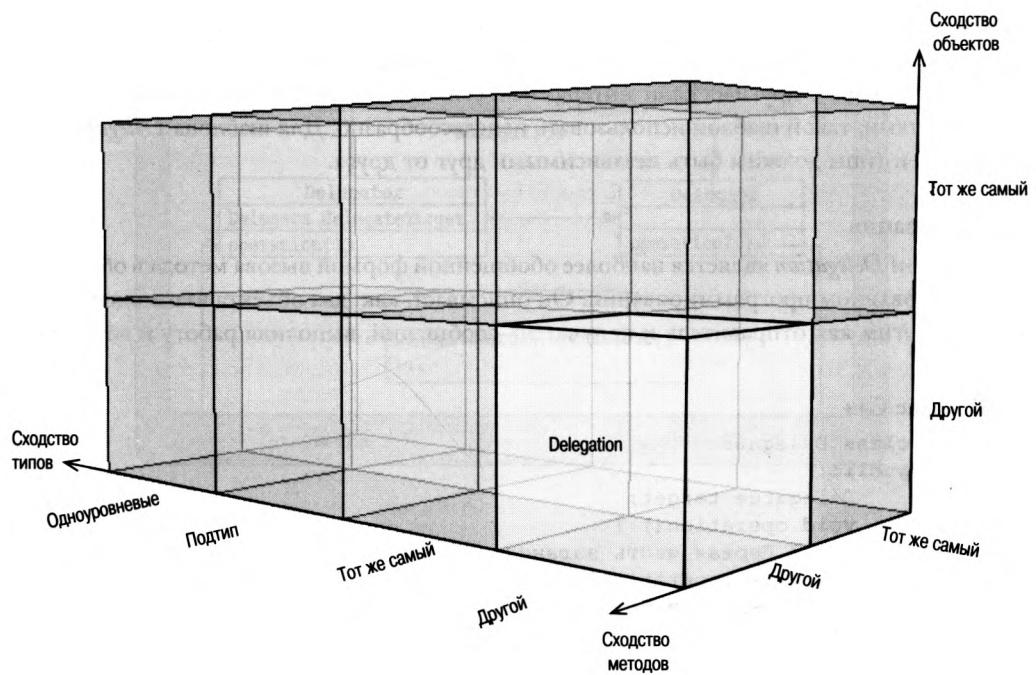
Шаблон *Delegation* повсеместно встречается в объектно-ориентированном программировании. За исключением других шаблонов вызова метода, этот шаблон проектирования представляет собой наиболее общую форму, которую можно использовать по умолчанию. К самым близким к нему шаблонам относятся соседи шаблона *Delegation* в пространстве проектирования. Изменив значение сходства методов на “тот же самый”, мы получаем шаблон *Redirection*. Изменив свойства сходства типов на “тот же самый”, получаем шаблон *Delegated Conglomeration* (Делегированная конгломерация). Изменение одного лишь сходства типов нецелесообразно, поскольку при этом отдельный объект будет иметь два разных типа одновременно. Шаблон *Delegation* часто используется в сочетании с шаблоном *Retrieve*, описывающим вызываемый объект.

Классификация вызова метода

Объект: другой

Тип объекта: другой

Метод: другой



Redirection

Поведенческий шаблон

Назначение

Предназначен для создания запроса, по которому другой объект должен выполнить подзадачу, тесно связанную с главной задачей, или саму задачу.

Синонимы

Tom Sawyer (Том Сойер), *Shop Foreman* (Мастер)

Мотивация

Шаблон *Redirection*, являющийся небольшим уточнением шаблона *Delegation*, учитывает тот факт, что похожие задания часто называются одинаково. Мы можем воспользоваться этим, чтобы уточнить назначение шаблона в более общей форме и продемонстрировать ситуации, в которых его можно применять.

Примером применения этого шаблона является *Tom Sawyer*, литературный персонаж, прославившийся тем, что убедил друзей выполнить работу вместо него [39]. В отличие от шаблона *Executive*, который является синонимом шаблона *Delegation*, Том попросил, чтобы его друзья выполнили именно то задание, которое было поручено ему. Тетушка попросила Тома покрасить забор. Он не хотел делать это сам и попросил друзей выполнить эту работу вместо него. Он мог бы попросить одного друга развести краску в ведре, другого — подготовить забор к покраске и так далее, и рабочий поток выглядел бы так, как распределение работ начальником. Вместо этого он решил разделить работу, чтобы каждый его друг *выполнил* одно и то же: покрасил забор. Том убедился, что у каждого есть кисти и краска и что работа выполнена, но он не делал это самостоятельно. Каждый из друзей, покрасивших забор, выполнил одну и ту же работу, о которой попросил Том, — просто это была маленькая часть общего задания.

Именно так работает шаблон *Redirection*. Он используется в ситуациях, когда работу можно разделить на более мелкие подзадачи, имеющие ту же основную мотивацию и цель, что и основная задача. Обратите внимание на то, что действия методов могут быть абсолютно разными, но цель должна быть одной и той же. Том не красил забор (он только держал кисти и краски), но его целью была покраска забора. Аналогично целью людей, работавших вместе с ним, также была покраска забора, просто они красили меньшие секции. В листинге 5.15 приведен пример на языке Java: Том показывает каждому из друзей, где красить забор.

Листинг 5.15. Том красит забор с помощью друзей

```
1  public class Friend {  
2      public void paintTheFence(int beg, int end) {  
3          // Покраска забора  
4      };  
5  };
```

```

public class TomSawyer {
    // Друзья собраны в одном месте
    java.util.ArrayList<Friend> friends;
}
10
// Покраска забора от пункта beg до пункта end
12 public void paintTheFence(int beg, int end) {
    int fenceLength = end - beg;
    int subfence = fenceLength / friends.size();
    int friendBeg = beg;
    for (Friend f : friends) {
        int friendEnd = friendBeg + subfence - 1;
        f.paintTheFence(friendBeg, friendEnd);
        // ^--- Переадресация
        friendBeg = friendEnd + 1;
    }
22    };
}

```

Эта схема похожа на то, что мы видели в разделе 2.2.4, в котором описано, как мастер красил машины. Обратите внимание, что ни в этом, ни в том случае вызываемый метод на самом деле не выполняет требуемую работу, хотя это вполне возможно. Исполняет ли вызывающий метод роль диспетчера, мастера или менеджера или сам выполняет работу, совершенно не важно. В другой ситуации вызывающий объект может иметь несколько иное представление о работе, которую следует выполнить, и может обратиться к кому-нибудь еще. В таком случае вызывающий объект должен выполнить предварительную работу или уборку после выполнения работы. Если вы красите стены в комнатах, у вас есть две возможности выполнить эту работу. Вы можете броситься в этот процесс с головой и начать красить стены кистью, надеясь, что вы не разбрызгаете слишком много краски на переключатели, плинтусы, розетки и т.п. Или вы можете тщательно подготовить комнату, постелив на пол брезент, аккуратно заклеив детали до начала окраски и так же бережно отклеив ленту по завершении работы.

Одни мастера умеют просто быстро красить стены, а другие всегда готовят комнату к покраске и делают уборку после ремонта. И в том, и в другом случае мастер красит стены комнаты, но в первом случае он старается сделать это как можно быстрее, а во втором — как можно качественнее. Было бы здорово совместить эти цели. В таких случаях в бригаду рабочих входит помощник, занимающийся подготовкой и уборкой, и маляр, умеющий быстро красить. Пока маляр красит стены в одной комнате, помощник может перейти в другую. Этот процесс можно моделировать синхронно, как показано в листинге 5.16.

Листинг 5.16. Подготовительная работа и уборка имеют значение

```

1 class SloppyFastPainter {
2     public:
3         void paintRoom( Room r ) {
4             // Окрашиваем стены
5         };
6     };
7
8     class CarefulPainter {
9

```

```

9 SloppyFastPainter sloppy;
10 public:
11     void paintRoom( Room r ) {
12         // Расстилаем брезент
13         // Выносим инструменты
14         sloppy.paintRoom( r ); // Переадресация
15         // Заменяем инструменты
16         // Делаем уборку комнаты
17     };
18 }

```

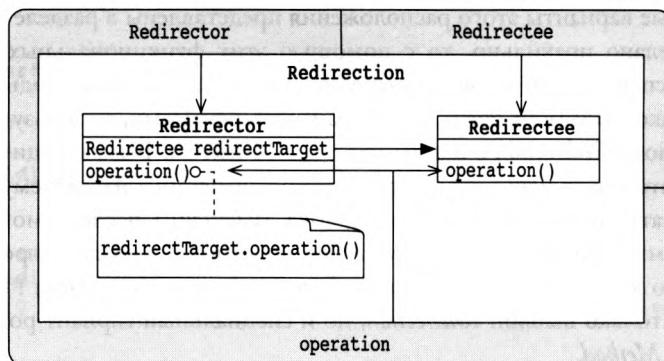
К сожалению, здесь возможна путаница. В общепринятом смысле слово *делегирование* означает перепоручение части работы кому-то другому. Именно в этом смысле сформулировано общее определение элементарного шаблона проектирования *Delegation*. Каждый шаблон вызова метода является специализированным вариантом делегирования, но шаблон *Redirection* ошибочно считать разновидностью шаблона *Delegation*. Эта ошибка вполне объяснима, поскольку термин *делегирование* очень часто встречается в описании процесса разработки программного обеспечения. Просто помните, что в языке C# термин *delegate* используется в самом общем смысле и означает метод, который должен быть передан в любую другую точку пространства проектирования. Шаблон *Delegation* используется для передачи обработки *любому* другому методу, а шаблон *Redirection* является более специальным.

Применимость

Шаблон *Redirection* используется в следующих ситуациях.

- Другой объект может выполнить ту же самую работу, что и текущий.
- Другой объект не обязан иметь доступ к закрытым данным текущего объекта для выполнения работы.
- Между типами объектов нет релевантных отношений.
- Целевой объект содержит метод, имеющий такое же предназначение, выраженное с помощью сигнатуры на основе шаблона Кена Бека *Intention Revealing Selector*.

Структура



Участники

Redirector

Источник вызова метода, содержащий метод *operation*, выполняющий подзадачу, часть которой должна быть передана другому объекту. Этот объект, *redirectTarget*, является полем типа *Redirectee*.

Redirectee

Тип получателя сообщения, выполняющего порученную подзадачу.

operation

Метод *operation* из класса *Redirector* вызывает метод *redirectTarget.operation* для выполнения части работы, т.е. вызывает метод *operation* класса *Redirectee*.

Отношения

Как и шаблон *Delegation* (и на самом деле любой другой элементарный шаблон вызова метода), шаблон *Retrieve* описывает бинарное отношение между двумя объектами и их вложенными методами. В этом случае он определяет отношение между двумя разными типами, используя вызов одинаковых методов. Основу данного отношения образует сходство имен, а значит, и целей.

Результаты

Несмотря на то что шаблон *Retrieve* практически идентичен шаблону *Delegation*, требование сходства методов имеет важные последствия. При описании последующих шаблонов мы увидим, что этот шаблон в сочетании с другими элементарными шаблонами проектирования и с информацией о типах позволяет быстро сформировать и просто описать сложные взаимодействия.

Учитывая тот факт, что оба метода имеют одинаковые имена и что существует общепринятый способ декларирования предназначения метода, можно прийти к выводу, что оба метода имеют одно и то же назначение. Более того, становится очевидным, что существует удобный способ указать, что исходный метод, совершающий вызов, требует от другого метода, чтобы он выполнил часть работы, и что эта работа тесно связана с основными функциональными возможностями исходного метода.

Вызываемый объект может находиться как внутри, так и вне вызывающего метода, а также объекта, содержащего вызываемый метод, или поступать извне с помощью шаблона *Retrieve*. Разные варианты этого расположения представлены в разделе *Реализация*.

Если все сделано правильно, то с помощью этих функциональных возможностей можно создать специальный вариант шаблона *Inheritance*, соединив один с другим два объекта с одинаковыми (или очень похожими) интерфейсами, используя несколько экземпляров шаблона *Redirection* с одним целевым объектом. Вызывающие методы могут просто передавать вызов через класс *Redirectee*, имитируя наследуемую реализацию, или реализовывать собственные тела, имитируя замещение наследуемой реализации и нарушая тем самым схему шаблона *Redirection*. Если в обоих классах реализуется свой метод и его тело вызывает соответствующий целевой метод из класса *Redirectee*, то мы получим не только шаблон *Redirection*, но и специальный вариант родственного ему шаблона *Extend Method*.

Реализация

В языке Java

```

1  public class Foo {
2      public void operation();
3  };
4
5  public class Bar {
6      Foo f;
7      public void operation() {
8          Foo f2;
9          f.operation(); // Переадресация
10         f2.operation(); // Переадресация
11     };
12 };

```

Покажем использование шаблона *Retrieve* в языке Objective-C для получения ссылки на объект класса *Redirectee*.

```

1  @interface Foo
2  {
3  }
4  -(void) operation;
5  @end
6
7  @implementation Foo
8  -(void) operation {
9      // Выполняем работу
10 };
11 @end
12
13 @interface Goo
14 {
15     Foo* f;
16 }
17 -(Foo*) getFoo;
18 @end
19
20 @implementation Goo
21 -(Foo*) getFoo {
22     return f;
23 };
24 @end
25
26 @interface Bar
27 {
28     Goo* g;
29 }
30 -(void) operation;
31 @end
32
33 @implementation Bar
34 -(void) operation {
35     [[g getFoo] operation]; // Переадресация на Retrieve
36 };
37 @end

```

Родственные шаблоны

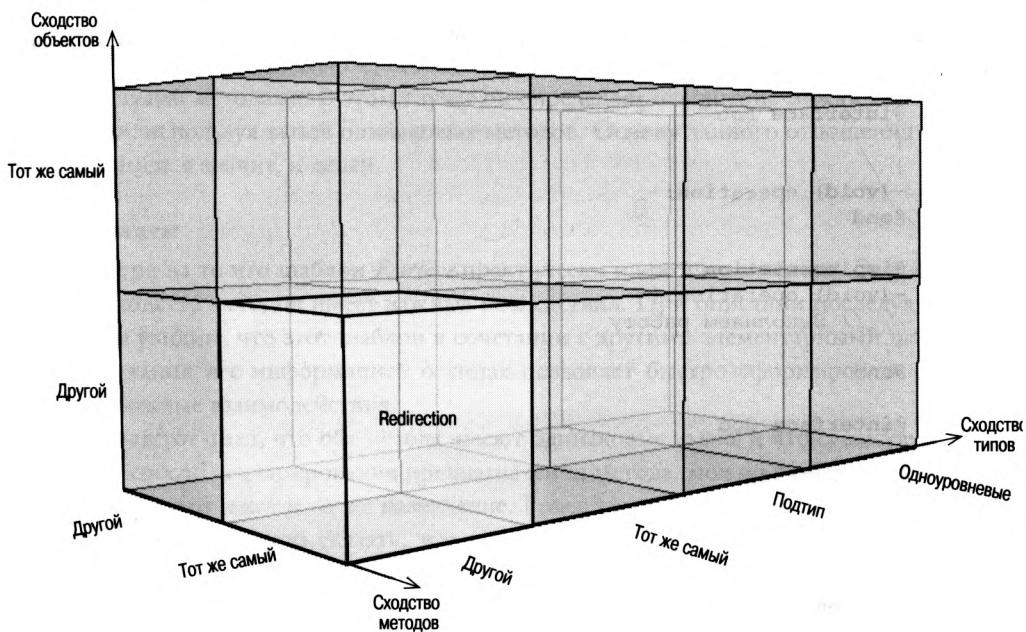
Элементарный шаблон проектирования *Redirection* используется очень часто. Он отличается от шаблона *Delegation* только тем, что в нем методы имеют одинаковые цели и имена. За исключением этого сходства, он совпадает с шаблоном *Delegation*. Как позволить изменение целевого объекта в ходе выполнения программы, ужесточив условие сходства типов, показано в описании шаблона *Retrieve*. Как и в шаблоне *Delegation*, если объекты одинаковы, а их типы не одинаковы, возникает неопределенность.

Классификация вызовов метода

Объекты: разные

Типы объекта: разные

Методы: одинаковые



Conglomeration

Поведенческий шаблон

Назначение

Предназначен для объединения в одно целое, т.е. для конгломерации, разных операций и действий для выполнения более сложной задачи в одном объекте.

Синонимы

Decomposing Message (Сообщение декомпозиции) и *Helper Methods* (Вспомогательные методы)

Мотивация

Объект часто получает команду выполнить задачу, слишком большую или громоздкую для выполнения в одном методе. Обычно в этом случае целесообразно разбить задачу на более мелкие задачи, чтобы решить их по отдельности разными методами, а затем объединить эти решения в одно целое в методе, ответственном за решение большой задачи. Кент Бек называет этот шаблон *Decomposing Message* [5]. Кроме того, связанные подзадачи можно объединить в одном методе, повторно использовав код в *отдельном* объекте.

Объединение в одно целое мелких действий в шаблоне *Conglomeration* создает несколько преимуществ. Оно позволяет уточнить детализацию действий, чтобы каждый метод выполнял меньший объем работы. Кроме того, оно облегчает сопровождение программ, позволяя повторное использование методов в разных местах без тиражирования кода. Другие объекты могут использовать часть детализированных методов по-новому, если они являются открытыми.

Рассмотрим пример из листинга 5.17, основанный на шаблоне *Redirection*. Допустим, мы поняли, что метод, описывающий покраску комнаты, очень длинный; он состоит из слишком большого количества операций, и код будет проще, если разбить его на части. Методы, решающие подзадачи, должны быть открытыми, потому подготовка к ремонту и уборка комнаты не зависят от процесса покраски стен. Метод *CarefulPainter* может выполнить подготовительные работы до того, как будет смонтирован подвесной потолок, и выполнить уборку после ремонта. То, что было неотъемлемой частью метода *paintRoom*, теперь разделено на более мелкие действия.

Листинг 5.17. Подготовительные работы и уборку можно разделить

```
1 class SloppyFastPainter {
2     public:
3         void paintRoom( Room r ) {
4             // Красим стены
5         };
6     };
7
8     class CarefulPainter {
```

```
9     SloppyFastPainter sloppy;
10    Room currentTarpRoom;
11    Room currentHardwareRoom;
12    public:
13    void paintRoom( Room r ) {
14        this->laydownTarps( r ); // Конгломерация
15        this->removeHardware( r ); // Конгломерация
16        sloppy.paintRoom( r );
17        this->replaceHardware(); // Конгломерация
18        this->cleanUp(); // Конгломерация
19    };
20    void laydownTarps( Room r ) {
21        this->currentTarpRoom = r;
22        // Расстилаем брезент
23    };
24    void cleanUp() {
25        // Очистка currentTarpRoom
26        // Освобождение currentTarpRoom
27    };
28    void removeHardware( Room r ) {
29        this->currentHardwareRoom = r;
30        // Вынос инструментов
31    };
32    void replaceHardware() {
33        // Замена инструментов в currentHardwareRoom
34        // Освобождение currentHardwareRoom
35    };
36};
```

Поступая так, необходимо разрешить пересылку информации о комнатах в виде параметра всем методам, выполняющим подзадачи. Это обеспечивает высокую гибкость. Но предположим, что по финансовым причинам один и тот же человек должен выполнить подготовительные и завершающие работы, например, чтобы предотвратить потерю инструментов при их передаче от одного рабочего другому и т.д. Каждый человек должен закончить работу, которую начал. Однако невозможность отслеживать эти данные приводит к путанице.

Если неорганизованный менеджер создаст сразу несколько объектов класса CarefulPainter и должен будет распределить между ними комнаты, он может случайно отправить команду провести подготовительные работы в одной комнате, а выполнить уборку — в другой. Вместо этого мы решили сделать информацию о комнатах закрытой для каждого метода CarefulPainter. Это напоминает ситуацию, когда менеджер говорит рабочим выполнить метод laydownTarps и дает им записку с номером комнаты. Эта записка хранится в кармане рабочего, и, получив команду cleanUp, он посмотрит в записку, чтобы убедиться, что уборка будет выполнена в правильной комнате. Выполнив задание, рабочий выбрасывает записку и готов идти в новую комнату. Данные из записи известны только рабочему и связывают между собой разные задачи.

Шаблон *Conglomeration* является мощным механизмом, но он может вызвать проблемы, если понимать его слишком буквально. Можно было бы прийти к нелогичному выводу о том, что целесообразно поместить каждую инструкцию в отдельный метод,

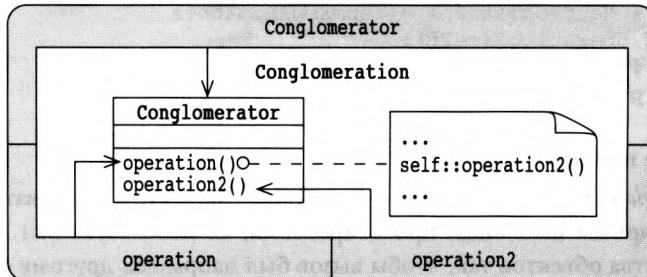
но очевидно, что это плохая идея. При использовании шаблона *Conglomeration* следует соблюдать баланс между необходимостью тонкой детализации с одной стороны и простотой и эффективностью кода — с другой. Поиск “атомарных действий” требует понимания, насколько большим должен быть атом.

Применимость

Шаблон *Conglomeration* используется в следующих ситуациях.

- Большую задачу можно разбить на подзадачи.
- Подзадачи должны выполняться в отдельных объектах, обычно благодаря совместно используемым закрытым данным или состоянию.
- Несколько подзадач можно объединить в теле отдельного метода.

Структура



Участники

Conglomerator

Тип объемлющего объекта.

operation

Главный контролирующий метод, выполняющий разбиение на подзадачи.

operation2

Вспомогательный метод, выполняющий конкретную подзадачу.

Отношения

В специализации шаблона *Delegation* объект вызывает свой метод. Методом вызова является метод *operation*, а вызываемым является метод *operation2*.

Результаты

Как и шаблон *Delegation*, этот шаблон связывает два метода отношением, в котором метод *operation* зависит от поведения и реализации метода *operation2*. В данном случае, в отличие от шаблона *Delegation*, может возникнуть побочный эффект, связанный с совместно используемыми данными, хранящимися внутри одного и того же объекта.

Реализация

В языке Java

```

1  public class Conglomerate {
2      public void operation() {
3          // Необязательная подготовительная работа
4          operation2();
5          // Необязательная завершающая работа
6      };
7      public void operation2() {};
8  }

```

В языке Python

```

1  class Conglomerate:
2      def operation(self):
3          # Необязательная подготовительная работа
4          self.operation2();
5          # Необязательная завершающая работа
6      def operation2(self):
7          # Требуемое поведение
8          pass

```

Родственные шаблоны

Шаблон *Conglomeration* связывает методы в одном объекте и, следовательно, является первым элементарным шаблоном проектирования, не использующим шаблон *Retrieve*. Изменение сходства объектов так, чтобы вызов был направлен другому объекту, при сохранении сходства типов приводит к шаблону *Delegated Conglomeration*. Если же сохранить сходство объектов и изменить сходство типов на отношение подтипа, можно получить шаблон *Revert Method*. Это может звучать странно — один объект может иметь разные типы, — но если прочитать спецификацию шаблона *Revert Method*, то можно понять, что это не только возможно, но и очень полезно. С другой стороны, сохранение сходства объектов и изменение отношения между типами на “совершенно разные”, приводит к неопределенности. И наконец изменение сходства методов так, чтобы вызов направлялся тому же самому методу, приводит к шаблону *Recursion*.

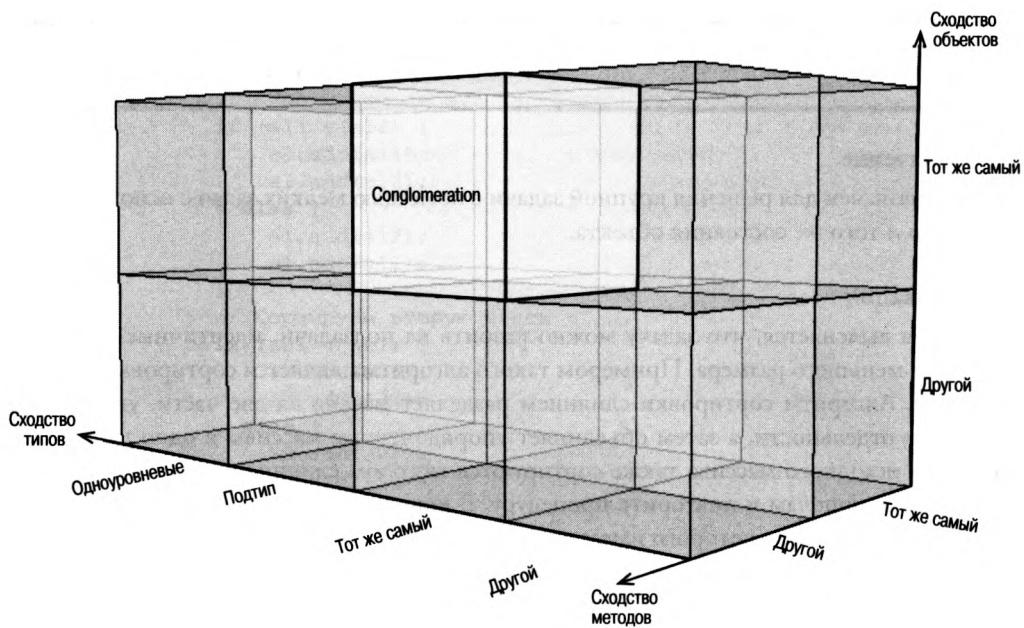
Шаблон *Conglomeration* появляется всюду, где объект разделяет задачу на подзадачи, но существует особый случай его использования в шаблоне *Template Method* (см. главу 7).

Классификация вызовов метода

Объекты: сходные

Типы объектов: одинаковые

Методы: разные



Recursion

Поведенческий шаблон

Назначение

Предназначен для решения крупной задачи с помощью мелких задач с использованием одного и того же состояния объекта.

Мотивация

Иногда выясняется, что задачу можно разбить на подзадачи, идентичные исходной задаче, но меньшего размера. Примером такого алгоритма является сортировка массива слиянием. Алгоритм сортировки слиянием разделяет массив на две части, упорядочивает их по отдельности, а затем объединяет упорядоченные массивы в одно целое. Две половины исходного массива также сортируются методом слияния, поэтому и их можно разделить пополам и повторить процедуру. В конце концов массив будет содержать только один элемент, с которого начнется слияние.

Процесс вызова методом самого себя называется *рекурсией*. Он постоянно используется в программировании. Тот же самый принцип применяется и в объектно-ориентированном программировании, но мы должны установить дополнительное ограничение, что объект должен вызывать сам себя явно или неявно с помощью оператора `self`. Мы уверены, что вы знаете эту концепцию, но она играет особую роль в контексте каталога EDP.

Вообще говоря, рекурсия — это способ сворачивания большой вычислительной задачи в небольшую. Допустим, нужно упорядочить массив и мы решили разделить его на две части, чтобы упорядочить их по отдельности. Слияние будет простым: если первый элемент массива A меньше первого элемента массива B, то первый элемент массива A копируется в новый, более крупный массив; в противном случае в него копируется первый элемент массива B. Копируемый элемент удаляется из соответствующего массива, и процесс повторяется до тех пор, пока оба массива не будут объединены.

Поскольку схема сортировки слиянием основана на соответствующей сортировке подмассивов, мы правильно полагаем, что можно применить этот алгоритм к подмассивам, разделяя, сортируя и объединяя каждый из них по очереди. На каждом шаге мы решаем одну и ту же задачу, поэтому, продолжая, достигнем наименьшего массива, содержащего только один элемент. В этой точке начинается процесс слияния упорядоченных массивов.

Допустим, что класс `Array` содержит обычные методы `add` и `remove`. Если начальный массив содержит четыре элемента, то всю сортировку можно описать как псевдокод, приведенный ниже, помня, что в объектно-ориентированной системе по умолчанию это происходит в классе или в объекте.

```
class ArrayOfLength4Library {
    2     Array sort_merge(Array a) {
        Element a11 = a[0];
        4         Element a12 = a[1];
```

```

6         Element a21 = a[2];
7         Element a22 = a[3];
8         Array a1, a2, res;
9         // Сортируем первую часть
10        if (a11 < a12) {
11            a1.add(a11);
12            a1.add(a12);
13        } else {
14            a1.add(a12);
15            a1.add(a11);
16        }
17        // Сортируем вторую часть
18        if (a21 < a22) {
19            a2.add(a21);
20            a2.add(a22);
21        } else {
22            a2.add(a22);
23            a2.add(a21);
24        }
25        // Слияние
26        if (a1[0] < a2[0]) {
27            res.add(a1[0]);
28            a1.remove(0);
29        } else {
30            res.add(a2[0]);
31            a2.remove(0);
32        }
33        if (a1[0] < a2[0]) {
34            res.add(a1[0]);
35            a1.remove(0);
36        } else {
37            res.add(a2[0]);
38            a2.remove(0);
39        }
40        if (a1.length == 0) {
41            res.add(a2[0]);
42            res.add(a2[1]);
43        }
44        if (a2.length == 0) {
45            res.add(a1[0]);
46            res.add(a1[1]);
47        }
48        if (a1.length == 1 && a2.length == 1) {
49            res.add(a1[0]);
50            res.add(a2[0]);
51        }
52        return a;
53    };
54};

```

Несмотря на высокую эффективность, это решение имеет очевидный недостаток: оно ограничено только массивами, содержащими четыре элемента. Более гибкое решение

должно содержать цикл. Проиллюстрируем реализацию цикла `for`, предполагая для простоты, что длина массива является степенью двойки.

```

1  class ArrayOfPower2LengthLibrary {
2      Array sort_array(Array a) {
3          // Разделение на подмассивы
4          subarray[0][0] = a;
5          for (i = 1 to log_2(a.length())) {
6              for (j = 0 to 2^i) {
7                  prevarray = subarray[i-1][floor(j/2)]
8                  subarray[i][j] = prevarray.slice(
9                      ((j mod 2) *
10                         (prevarray.length() / 2)),
11                      ((j mod 2) + 1) *
12                         (prevarray.length() / 2))
13          }
14      }
15      // Сортируем подмассивы и объединяем
16      for (i = log_2(a.length()) to 1) {
17          for (j = 2^i to 0) {
18              subarray[i-1][j] =
19                  max(subarray[i][j],
20                      subarray[j][i]);
21          }
22      }
23      return subarray[0][0];
24  };

```

Это более гибкое решение, но из-за слишком громоздкого механизма код стал нечитабельным. Обратите внимание на то, что цикл сворачивается в базу (длина массива сходится к единице), состояние массива на каждом шаге сохраняется, а затем цикл разворачивается до предыдущего состояния. Ввод, хранение, извлечение и разворачивание можно выполнить с помощью вызова функции. Этот факт позволяет значительно упростить реализацию алгоритма с помощью шаблона *Recursion*.

```

class ArrayLibrary {
2     Array sort (Array a) {
3         return sort_array(a,
4                           a.beginIndex(), a.endIndex());
5     }
6     Array sort_merge(Array a, int beg, int end) {
7         if (a.length() > 1) {
8             Array firstHalf =
9                 sort_array(a, beg, end-beg/2);
10            // ^--- Рекурсия (неявный оператор self)
11            Array secondHalf =
12                this->sort_array(a,
13                                  (end-beg/2) + 1, end);
14            // ^--- Рекурсия (явный оператор self)
15            return merge(firstHalf, secondHalf);
16        } else {
17            return a;
18        }
19    }
20}

```

```

18      }
19  };
20
21  Array merge(Array a, Array b) {
22      Array res;
23      while (b.length() > 1 || a.length() > 1) {
24          // Если массивы a или b пустые, то a[0] или b[0]
25          // содержат минимальное значение
26          if (a[0] < b[0]) {
27              res.add(a[0]);
28              a.delete(0);
29          } else {
30              res.add(b[0]);
31              b.delete(0);
32          }
33      }
34      return res;
35  };
36 }

```

Эта версия концептуально проще и выполняет ту же самую задачу, что и циклический вариант. Мы перегружаем систему, но в качестве компенсации больше не ограничены длиной массива. Код стал не только более простым, но и более общим.

В некоторых ситуациях, например в высокоеффективных встроенных системах, замедление работы нежелательно. В таких случаях следует развернуть рекурсии в циклы и не применять их в последовательном коде. Однако это крайности, которые характерны для высокоспециализированных систем. В большинстве ситуаций выгода от шаблона *Recursion* — концептуальная ясность и простота кода — намного превышают небольшие потери в скорости.

Применимость

Шаблон *Recursion* используется в следующих ситуациях.

- Задачу можно разделить на аналогичные подзадачи.
- Подзадачи должны выполняться тем же самым объектом, обычно из-за необходимости обеспечить доступ к совместно используемому состоянию для вызывающего и вызываемого методов.
- Небольшая потеря эффективности компенсируется простотой.

Участники

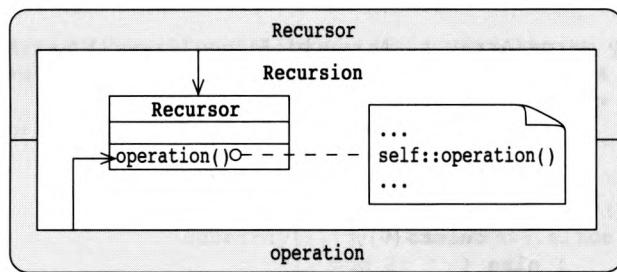
RecurSOR

Класс *RecurSOR* содержит метод, вызывающий сам себя в одном и том же объекте.

operation

Метод *operation* явно или неявно вызывает сам себя с помощью инструкции *self.operation()*.

Структура



Отношения

Метод *operation* класса *RecurSOR* взаимодействует сам с собой, выполняя на каждом шаге все меньшую задачу, пока не будет достигнута база, в которой промежуточные результаты объединяются в окончательный результат.

Результаты

Слабым местом шаблона *Recursion* является зависимость от базы, в которой заканчивается процесс рекурсии. В большинстве современных систем практически невозможно рассчитывать на неограниченные ресурсы, а из-за неправильной базы рекурсия может стать бесконечной.

Реализация

В языке Java

```

1  public class Recursor {
2      public void operation() {
3          // Предварительная работа
4          self.operation();
5          // Завершающая работа
6      ;
7  }
  
```

Родственные шаблоны

Шаблон *Recursion*, вероятно, является наименее общим элементарным шаблоном вызова метода, потому что в нем объекты, типы объектов и методы должны быть одинаковыми. Изменив одно из этих условий, можно получить новый элементарный шаблон проектирования. Шаблон *Conglomeration* получается, если в одном и том же объекте при одинаковых типах вызываются разные методы. (Метод *sort_merge* на заключительном этапе сортировки слиянием является примером шаблона *Conglomeration*, содержащего вызов методов *sort* метода *sort_merge* и вызов методом *sort_merge* метода *merge*.) Сохранение сходства методов при возможном вызове методов из других объектов того же типа приводит к шаблону *Redirected Recursion*, который используется в цепочках объектов. Вероятно, наиболее интересным соседом элементарного шаблона проектирования

Recursion является шаблоном *Extend Method*, который возникает, когда сохраняется сходство методов и объектов при отношении подтипа.

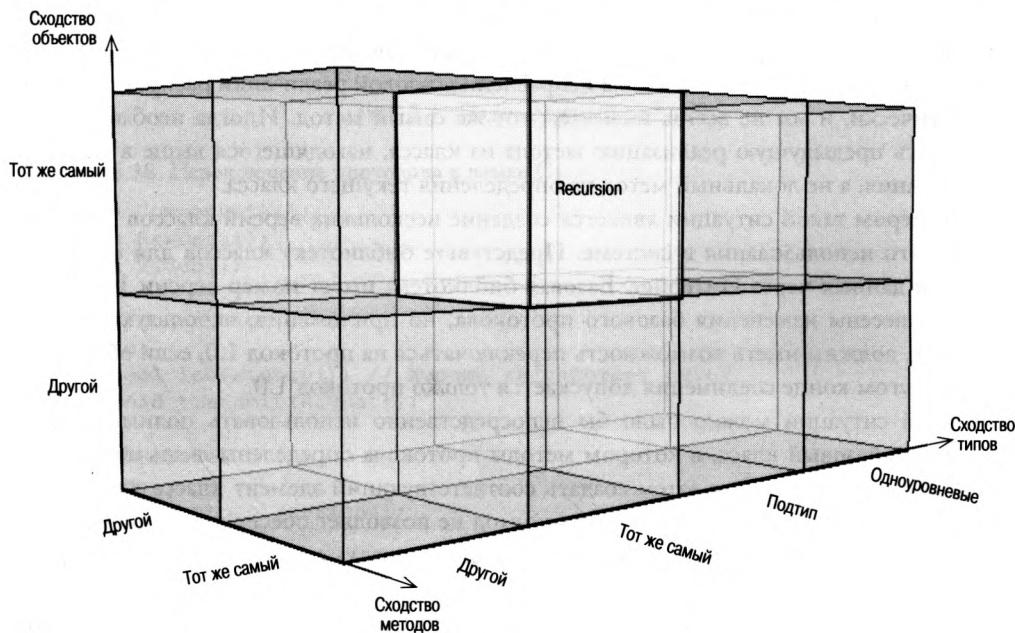
В заключение отметим шаблон *Delegation*: два объекта взаимно вызывают друг друга через два разных метода, имея отдельные закрытые данные. Метод *f()* из объекта А вызывает метод *g()* из объекта В, а метод *g()* вызывает метод *f()* из объекта А. Это явление называется *взаимной рекурсией*, поскольку каждый метод неявно вызывает сам себя. Хотя эта схема отличается от шаблона *Recursion*, она очень похожа на него и заслуживает упоминания.

Классификация вызовов методов

Объекты: одинаковые

Типы объектов: одинаковые

Методы: одинаковые



Revert Method

Поведенческий шаблон

Назначение

Обходит текущую реализацию метода в классе и вместо нее использует реализацию из суперкласса, возвращаясь к менее специализированной версии метода.

Мотивация

Полиморфизм иногда вреден. В спецификации шаблона *Inheritance* мы видели, что его основной мотивацией является упрощенная поддержка повторного использования реализации. Модификации, вносимые в базовую реализацию метода, автоматически распространяются на подклассы, но подклассы могут либо заместить существующие методы, либо добавить новую функциональную возможность с помощью шаблона *Extend Method*. Однако иногда нежелательно, чтобы исправления базовой реализации распространялись автоматически, и мы не всегда вызываем тот же самый метод. Иногда необходимо использовать предыдущую реализацию метода из класса, находящегося выше в иерархии наследования, а не локальный метод из определения текущего класса.

Примером такой ситуации является создание нескольких версий классов для одновременного использования в системе. Представьте библиотеку классов для протокола передачи данных через Интернет. Базовая библиотека имеет номер версии 1.0. В версии 1.1 внесены изменения базового протокола, но приложения, использующие протокол 1.1, должны иметь возможность переключаться на протокол 1.0, если обнаружат, что на другом конце соединения допускается только протокол 1.0.

В этой ситуации можно было бы непосредственно использовать полиморфизм и определить базовый класс, в котором методы протокола определены лишь абстрактно, как показано на рис. 5.1, а затем создать соответствующий элемент класса для идентификации протокола. К сожалению, этот подход не позволяет обеспечить динамическую гибкость и модифицировать протокол, чтобы адаптировать его к изменяющимся требованиям.

Несмотря на то что предлагаемый способ упрощает настройку соединений для пользователей старого протокола, он затрудняет восстановление связи в случае ошибок. Допустим, что вы передаете видеинформацию, сеть работает хорошо и задержка невелика. Тогда можно применить новый алгоритм сжатия данных, интенсивно использующий процессор. Если же задержка велика, то затраты времени, необходимые для распаковки данных на другом конце соединения, могут привести к простоям. Если вы заметите, что сеть перегружена, то вернетесь к предыдущему алгоритму сжатия данных.

Вы могли бы создать объекты для каждого типа протокола и при необходимости переключаться на старые версии, но в этом случае могут возникнуть проблемы с состоянием протокола. Переход от одного класса протокола к другому требует создания нового объекта нового класса и последующего копирования всех закрытых данных при каждом переключении протокола. В лучшем случае эти данные могут храниться только в классе *ProtocolBase*, но даже в этом случае при копировании нарушается принцип

инкапсуляции закрытых данных. Кроме того, информация о протоколах распределяется между классами Protocol и Controller, как показано в листинге 5.18. Старый класс реализует его реальное поведение, а новый должен знать, какое поведение требуется в данный момент.

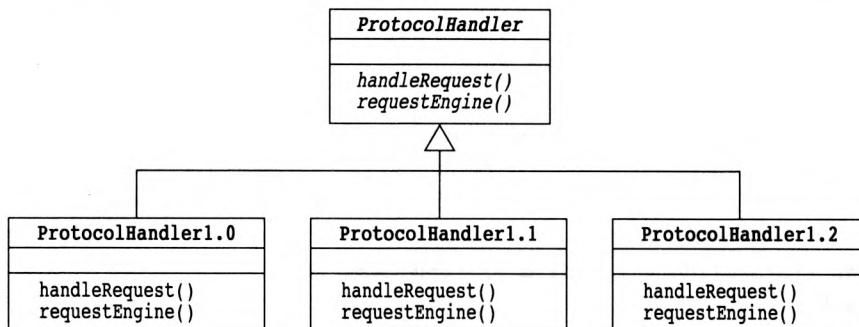


Рис. 5.1. Полиморфный подход

Листинг 5.18. Переключение протокола в языке C+

```

1  class CommonData{};
2  class RawData{};
3  class Video{};

5  class Connection{
6  public:
7      bool lowLatency(); // Хорошо ли работает сеть?
8      void transmit(RawData);
9  };
11 class RemoteEnd {
12 public:
13     int protocolVersion();
14 };
15
16 // Классы выполнения протокола:
17 class ProtocolBase {
18 protected:
19     CommonData data;
20     int d_version;
21     Connection connection;
22 public:
23     virtual
24         Connection initConnection(RemoteEnd otherSide);
25     Connection getConnection() {
26         return connection;
27     };
28     virtual void sendData(Video vid) {
29         connection.transmit(compress(vid));
30     };
31     int version() { return d_version; };
32     CommonData getData() { return data; };
  
```

```
33     virtual RawData compress(Video vid) = 0;
34 };
35
36 class Protocol1dot0 : public ProtocolBase {
37 public:
38     Protocol1dot0() {
39         d_version = 1.0;
40     }
41     Protocol1dot0(ProtocolBase* base) {
42         d_version = 1.0;
43         this->data = base->getData();
44     }
45     RawData compress(Video vid);
46 };
47
48 class Protocol1dot1 : public ProtocolBase {
49 public:
50     Protocol1dot1() {
51         d_version = 1.1;
52     }
53     Protocol1dot1(ProtocolBase* base) {
54         d_version = 1.1;
55         this->data = base->getData();
56     }
57     RawData compress(Video vid);
58 };
59
60 class Controller {
61     ProtocolBase* handler;
62 public:
63     void setupConnection(RemoteEnd otherSide) {
64         if (otherSide.protocolVersion() == 1.0) {
65             handler = new Protocol1dot0();
66         } else {
67             handler = new Protocol1dot1();
68         }
69         handler->initConnection(otherSide);
70     };
71     void sendData(Video vid) {
72         bool networkGood =
73             handler->getConnection().lowLatency();
74         if (networkGood &&
75             handler->version() == 1.0) {
76             // Переключение на новую версию
77             handler = new Protocol1dot1(handler);
78         } else if (!networkGood &&
79             handler->version() == 1.1) {
80             // Переключение на старую версию
81             handler = new Protocol1dot0(handler);
82         }
83         handler->sendData(vid);
84     };
85 };
```

В каждом из методов класса Controller предусмотрен огромный объем лишних операций. Мы могли бы попытаться перенести всю управляющую логику в класс ProtocolBase, и, на первый взгляд, это могло бы сработать. Однако в этом случае необходимо, чтобы класс ProtocolBase знал обо всех подклассах, что является признаком неудачного проекта. Кроме того, это означало бы, что класс ProtocolBase необходимо модифицировать при каждом определении нового класса. Если этот класс поставляется клиентам в составе скомпилированной библиотеки, то они не смогут дополнить библиотеку новыми версиями протокола.

Кроме того, этот подход затрудняет повторное использование существующей реализации. Если протокол 1.1 является всего лишь небольшой модификацией протокола 1.0, то было бы целесообразно повторно использовать функции протокола 1.0, перенеся общий код в базовый класс и разрешив его вызывать из каждого подкласса. Этот подход мог бы хорошо работать с двумя подклассами. Однако если классов, версий протокола и особых вариантов больше двух, то ситуация может запутаться. В этом случае вероятность выделить общее ядро кода снижается.

Совершенно очевидно, что этот подход плохо поддерживает динамическое переключение вверх и вниз по иерархии классов.

Вместо этого мы можем вывести подкласс новой версии протокола из предыдущей версии протокола, чтобы реализовать только необходимые изменения. Это позволит работать только с одним классом, чтобы выбирать оптимальный протокол, имея при этом возможность вернуться к предыдущей версии. Расширенный вариант этого подхода, включая несколько версий, описан на рис. 5.2.

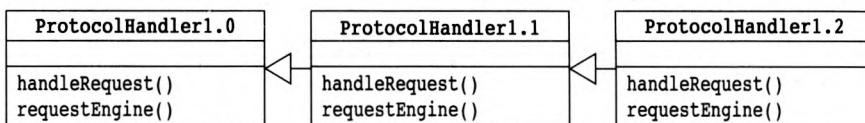


Рис. 5.2. Поход на основе наследования

Пример этого подхода с двумя классами описан в листинге 5.19. Вспомогательные классы описаны в листинге 5.18. В данном случае мы можем создать объект, относящийся к последнему классу в цепочке, а при необходимости условные инструкции в коде передадут протокол по цепочке к требуемой версии. Это значительно упрощает сопровождение кода, поддерживающего этот протокол, потому что при модификации библиотеки необходимо изменить только экземпляр (или экземпляры) объекта, реализующего протокол, а все остальное будет изменено автоматически.

Листинг 5.19. Автоматическое перемещение по иерархии классов с помощью шаблона *Revert Method*

```

1 class Protocol1dot0 {
2     public:
3         virtual
4         void initConnection(RemoteEnd otherSide) {
5             connection = otherSide.connect(1.0);
6         };
7     
```

```

7   virtual
8     void sendData(Video vid) {
9       RawData rawData = compress(vid);
10      connection.transmit(rawData);
11    };
12  protected:
13    virtual
14      RawData compress(Video);
15      Connection connection;
16      CommonData data;
17  };

19 class Protocol1dot1 : public Protocol1dot0 {
20  public:
21    virtual
22      void initConnection(RemoteEnd otherSide) {
23        connection = otherSide.connect(1.1);
24      };
25    virtual
26      void sendData(Video vid) {
27        RawData rawData;
28        if (connection.lowLatency()) {
29          rawData = compress(vid);
30        } else {
31          rawData = Protocol1dot0::compress(vid);
32          // ^--- Шаблон Revert Method
33        }
34        connection.transmit(rawData);
35      };
36  protected:
37    virtual
38      RawData compress(Video);
39  };

```

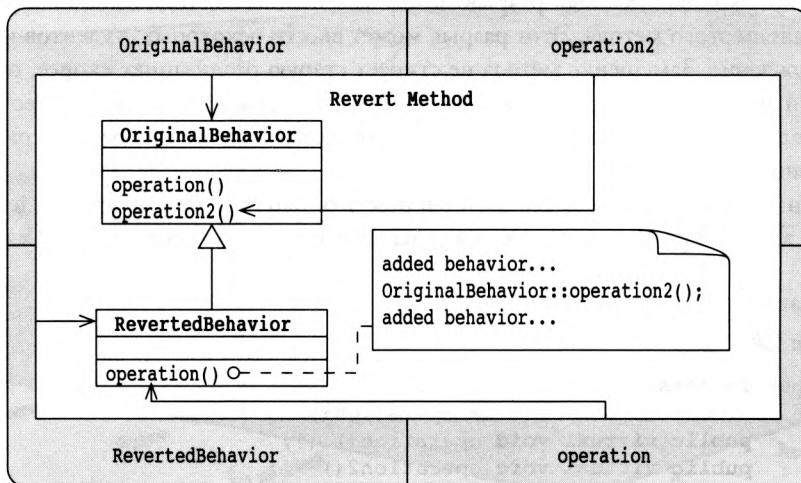
Теперь мы исключили необходимость в классах Controller и ProtocolBase. Любой клиент может просто создать новую работоспособную версию протокола, а любое изменение в иерархии классов произойдет автоматически. Это код намного понятнее и точнее выражает намерения разработчика.

Применимость

Шаблон *Revert Method* используется в следующей ситуации.

- Если класс хочет *восстановить* исходную реализацию метода из суперкласса, замещенную в подклассе.

Структура



Участники

OriginalBehavior

Базовый класс, определяющий метод *operation2*.

operation

Operation — это вызывающий метод, определенный в классе **RevertedBehavior**.

operation2

Operation2 — это вызываемый метод, определенный в классе **OriginalBehavior**. Он должен быть также определен в подклассе класса **OriginalBehavior** либо в классе **RevertedBehavior**, либо в классе, определенном между ними в иерархии классов. Второе определение маскирует реализацию из класса **OriginalBehavior** и не позволяет использовать ее по умолчанию в классе **RevertedBehavior**, вынуждая применять предыдущую реализацию.

RevertedBehavior

Подкласс класса **OriginalBehavior**, в котором замещены методы *operation* и *operation2*. Метод *operation* вызывает реализацию метода *operation2* из класса **OriginalBehavior** в ситуациях, в которых ожидается вызов его собственной реализации этого метода.

Отношения

В большинстве случаев подкласс замещает метод родительского класса, чтобы заменить его функциональные свойства, но два определения метода могут работать совместно, чтобы расширить его возможности. Класс **RevertedBehavior** зависит от класса **OriginalBehavior**, в котором осуществляется основная реализация.

Результаты

Существует концептуальный разрыв между замещением метода из базового класса и использованием этого метода. Этот разрыв может ввести некоторых студентов и практиков в заблуждение. Замещение метода не стирает старую реализацию; скорее, оно скрывает старый метод от объектов подкласса. Объект по-прежнему знает о существовании методов родительского класса и может вызывать их как обычно, не демонстрируя свое знание внешнему миру.

Шаблон *Revert Method* является разновидностью шаблона *Conglomeration*, в который добавлены знание о реализации метода в суперклассе и возможность доступа к ней.

Реализация

В языке C#

```

1  public class OriginalBehavior {
2      // Определение метода operation()
3      public virtual void operation() {};
4      public virtual void operation2() {};
5  };

7  public class RevertedBehavior : OriginalBehavior {
8      public override void operation() {
9          // Предварительные операции
10         if (oldBehaviorNeeded) {
11             OriginalBehavior::operation2();
12         } else {
13             operation2();
14         }
15         // Заключительные операции
16     };
17     // Если метод не замещен, нет необходимости
18     // возвращаться к его реализации в суперклассе
19     public override void operation2();
}

```

Родственные шаблоны

Как указывалось ранее, шаблон *Revert Method* тесно связан с шаблоном *Conglomeration*, за исключением того, что он использует реализацию метода супертипа, который участвует в конгломерации.

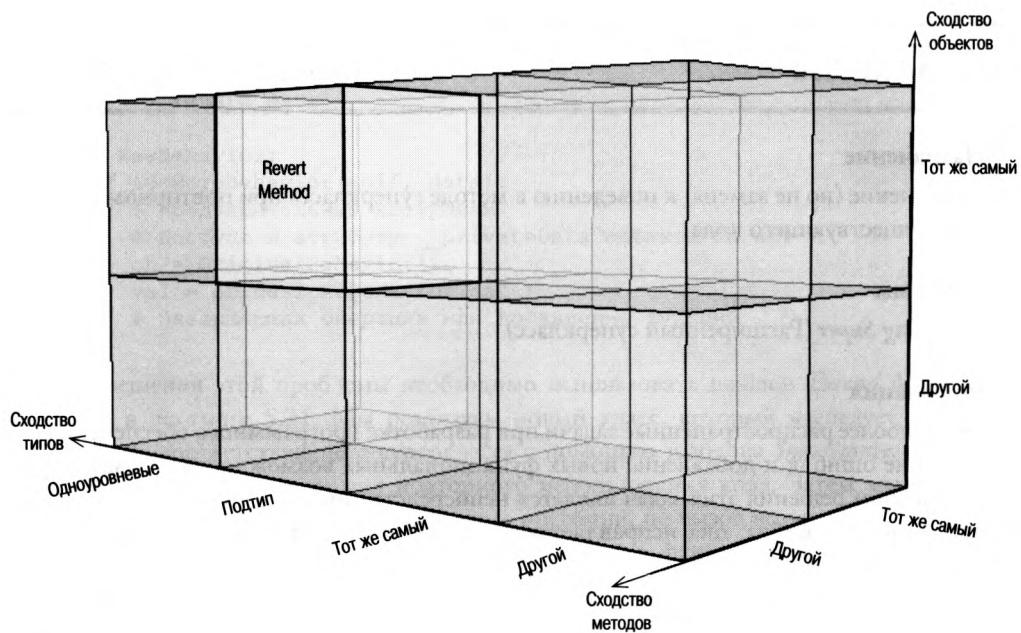
Если вызывается один и тот же метод, то получается шаблон *Extend Method*. Если функциональные возможности, которые должны обеспечиваться другими объектами супертипа, определены в классе *OriginalBehavior*, то получается шаблон *Trusted Delegation*.

Классификация вызовов метода

Объекты: одинаковые

Тип объектов: подтип

Методы: разные



Extend Method

Поведенческий шаблон

Назначение

Дополнение (но не замена) к поведению в методе суперкласса при повторном использовании существующего кода.

Синоним

Extending Super (Расширенный суперкласс)

Мотивация

Две наиболее распространенные задачи при разработке программного обеспечения — исправление ошибок и добавление новых функциональных возможностей. Самым простым способом решения этих задач является непосредственное изменение кода. Там, где обнаруживается ошибка, она исправляется. Там, где требуется новая функциональная возможность, она добавляется.

Однако это возможно не всегда. Часто исходный код, который необходимо исправить или улучшить, недоступен, поскольку он скомпилирован в библиотеке. Иногда исходный код и функции необходимо сохранить, чтобы использовать в будущем. В любом случае иногда исходный код изменять нельзя.

Разумеется, если исходный код доступен, можно просто скопировать и вставить старый код в новый метод. Однако, как показано в спецификации шаблона *Inheritance*, при этом возникает масса проблем, включая несогласованность методов и сложности в сопровождении. Намного удобнее определить каждую реализацию в одном месте, а затем повторно использовать существующий код, выдергивая из него фрагменты кода или результатов по мере необходимости. Перед передачей данных исходной реализации данные можно сжать, а результаты работы исходной реализации использовать для выполнения дополнительных операций. Следуя этим правилам, можно упростить сопровождение и повысить ясность кода.

Если исходный код недоступен, операции копирования и вставки становятся невозможными и следует найти другой способ повторного использования кода. Можно, конечно, создать объект-делегат с исходным поведением и вызывать его при необходимости. Этот подход реализован в виде шаблона проектирования *Redirection*, но в некоторых случаях этот шаблон не работает, как показано в листинге 5.20. Если дополнительные функции должны иметь доступ к данным, инкапсулированным в оригинальном коде, а сам этот код недоступен для внешних клиентов, то такая тактика просто невозможна. Но даже когда она возможна, иногда трудно понять, что делает новый код и как он связан с оригинальным кодом.

Листинг 5.20. Использование шаблона *Redirection* в языке Python для добавления функциональных возможностей

```
1 class OriginalBehavior:  
2     __privateData = True
```

```

4      # Добавляйте перед закрытым атрибутом префикс __
5  def desiredBehavior(self, data):
6      # Нечто интересное
7      return data
8
9  class NewBehavior:
10     def addMoreBehavior(self, data):
11         # Предварительные настройки
12         # Доступа к атрибуту __privateData объекта ob нет
13         ob = OriginalBehavior()
14         val = ob.desiredBehavior(data)
15         # Завершающая операция или добавление функций

```

Для решения этой проблемы необходимо использовать шаблон *Extend Method*, как показано в листинге 5.21. Мы реализуем новый класс, который наследует желаемую функцию из соответствующего суперкласса с помощью шаблона *Inheritance*, поскольку он предоставляет механизм для повторного использования кода. Затем мы замещаем оригинальный метод, добавляем требуемые операции, но вызов возвращаем реализации метода из суперкласса, когда потребуются именно его функциональные возможности. Это упрощает сопровождение и повторное использование кода, а также инкапсуляцию измененного поведения.

Листинг 5.21. Использование шаблона *Extend Method* для добавления функциональных возможностей

```

class OriginalBehavior:
1     __privateData = True
2     # Добавляйте перед закрытым атрибутом префикс __
3
4     def desiredBehavior(self, data):
5         # Нечто интересное
6         return data
7
8  class NewBehavior(OriginalBehavior):
9      def desiredBehavior(self, data):
10         # Предварительные настройки
11         # Есть доступ к __privateData!
12         superDelegate = super(NewBehavior, self)
13         superDelegate.desiredBehavior(data)
14         # ^--- Extend Method
15         # Завершающая операция или добавление функций

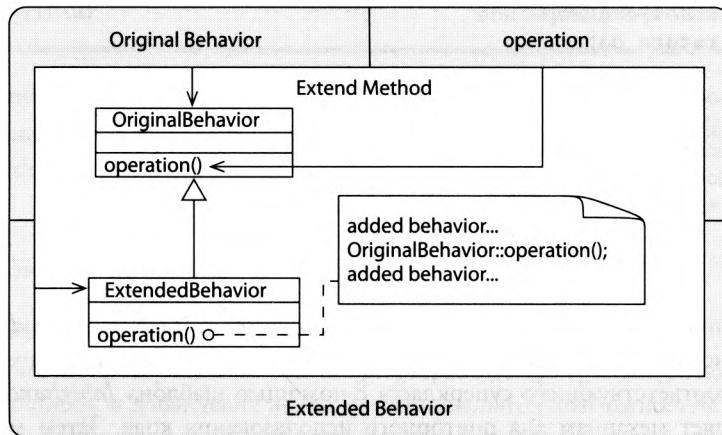
```

Применимость

Шаблон *Extend Method* используется в следующих ситуациях.

- Необходимо расширить, но не заменить существующее поведение метода.
- Из-за отсутствия исходного метода желательно или необходимо повторно использовать код.
- Использовать шаблон *Redirection* невозможно или нецелесообразно, потому что новое поведение требует доступа к данным, которые закрыты в исходной реализации.

Структура



Участники

Original Behavior

Определяет интерфейс и содержит реализацию метода *operation* с требуемыми функциональными возможностями.

operation

Метод, определенный в классе `OriginalBehavior` для реализации базового поведения. Замещается в классе `ExtendedBehavior` с целью создания новой реализации, использующей оригинальную реализацию.

Extended Behavior

Использует интерфейс класса `OriginalBehavior`, затем заново реализует метод *operation*, чтобы включить в него вызов кода из базового класса, окруженный дополнительным кодом и/или функциями.

Отношения

В большинстве случаев, когда подкласс замещает метод родительского класса, требуется заменить функциональные возможности исходного метода. Однако два определения метода могут работать совместно, расширяя возможности родительского класса, а не заменяя их. Класс `ExtendedBehavior` зависит от интерфейса и реализации класса `OriginalBehavior`.

Результаты

Как и в шаблоне *Revert Method*, концепция вызова замещенной версии метода может создать путаницу. В шаблоне *Extend Method* путаница может оказаться еще сильнее, потому что вызывающий метод, на первый взгляд, вызывает метод-привидение.

Использование этого шаблона оптимизирует повторное использование кода, но метод *operation* в классе `OriginalBehavior` становится несколько ненадежным — его

поведение теперь зависит от инвариантности метода `ExtendedBehavior::Operation`. Поведение расширяется полиморфно и прозрачно для клиентов класса `OriginalBehavior`.

Реализация

В языке Java

```

1  public class OriginalBehavior {
2      public void operation() {
3          // Основная работа
4      };
5  };
6  public class ExtendedBehavior
7      extends OriginalBehavior {
8          public void operation() {
9              // Предварительная работа
10             super.operation();
11             // Завершающая работа
12         };
13 };

```

В языке C++

```

1  class OriginalBehavior {
2  public:
3      virtual void operation() {
4          // Основная работа
5      };
6  };
7  class ExtendedBehavior : public OriginalBehavior {
8  public:
9      void operation() {
10         // Предварительная работа
11         OriginalBehavior::operation();
12         // Завершающая работа
13     };
14 };

```

Родственные шаблоны

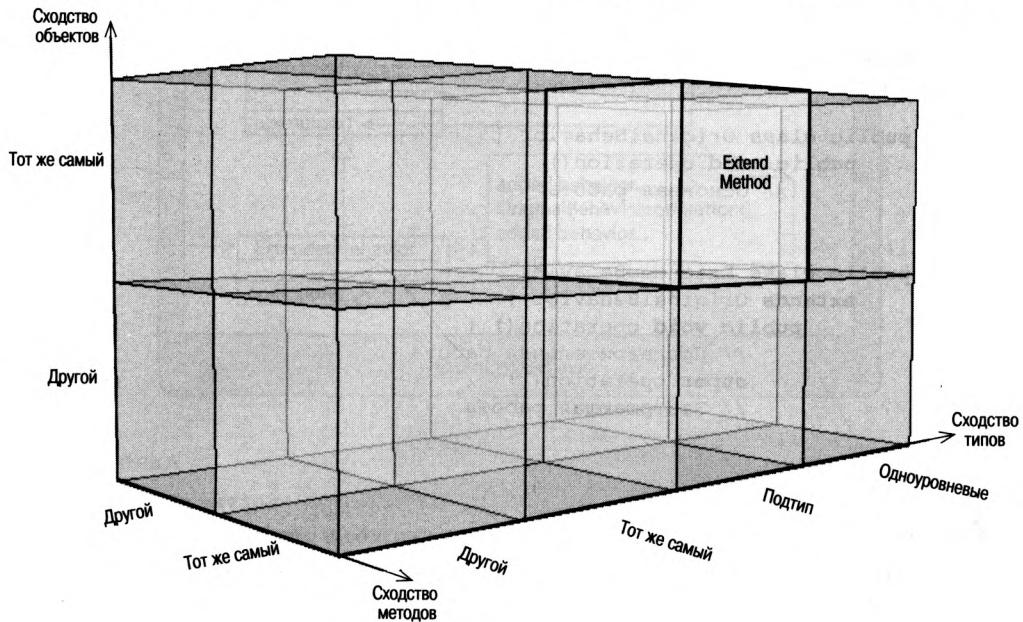
Шаблон *Extend Method*, как и шаблон *Recursion*, содержит один объект. Изменение координаты “одинаковые объекты” на “разные объекты” приводит к шаблону *Trusted Redirection*, использующему полиморфизм для обхода кластера родственных классов в поисках функций. Ослабление условия сходства методов на “разные методы суперкласса” приводит к шаблону *Revert Method*. Поскольку шаблон *Revert Method* представляет собой вариант шаблона *Conglomeration* с известным суперклассом, шаблон *Extend Method* является вариантом шаблона *Recursion* с известным суперклассом, даже несмотря на то что результирующая функция отличается от исходной. Шаблон *Recursion* можно получить, просто удалив доступ к суперклассу и вызвав текущую реализацию.

Классификация вызовов метода

Объекты: одинаковые

Тип объектов: суперкласс

Методы: одинаковые



Delegated Conglomeration

Поведенческий шаблон

Назначение

Используется в ситуациях, когда несколько однотипных объектов должны совместно решить одну задачу, при этом каждый должен выполнить отдельное задание.

Мотивация

Однотипные объекты часто согласованно решают одну и ту же задачу. Однородные среды данных с неоднородным поведением часто кодируются как коллекции объектов, обеспечивающие расширенные функциональные возможности.

В листинге 5.22 приведен пример сайта социальной сети. Учетная запись каждого пользователя содержит список друзей, и пользователи часто хотят иметь возможность приглашать своих друзей на разные мероприятия. Пользователи могут настроить свои предпочтения, включая адрес электронной почты для контактов.

Листинг 5.22. Наивное приглашение друзей в языке Java

```
1 import java.util.Vector;
2 import java.lang.String;
3
4 public class UserAccount {
5     Vector<UserAccount> allMyFriends;
6     public String notificationEmail;
7
8     public void inviteToEvent (
9         Vector<UserAccount> friends, String msg) {
10        for (UserAccount friend : friends) {
11            String email = friend.notificationEmail;
12            // Отослать e-mail
13        }
14    }
15};
```

Это решение вполне работоспособно, но требует открытой публикации электронного адреса. Большинство пользователей не желают раскрывать свой электронный адрес в социальных сетях. Для того чтобы скрыть эту информацию, организатор мероприятия может запросить у друзей электронные письма, которые те могут прислать, если захотят, как показано в листинге 5.23.

Листинг 5.23. Немного более удобный способ пригласить друзей

```
1 import java.util.Vector;
2 import java.lang.String;
3
4 public class UserAccount {
5     Vector<UserAccount> allMyFriends;
6     public String notificationEmail;
7 }
```

```

1   public void inviteToEvent (
2       Vector<UserAccount> friends, String msg) {
3           for (UserAccount friend : friends) {
4               String email = friend.getEmail(this);
5               // Отослать e-mail
6           }
7       }
8
9
10      public String getEmail (UserAccount requester) {
11          String returnVal = "";
12          if (allMyFriends.contains(requester)) {
13              returnVal = notificationEmail;
14          }
15          return returnVal;
16      };
17
18  };
19
20
21
22
23 };

```

При этом подходе пользователи также должны предоставлять свои адреса электронной почты любому члену из списка друзей. Большинство пользователей все же хотели бы получать приглашения без распространения своих адресов. Видимость адресов электронной почты можно отделить от возможности получать приглашения с помощью шаблона *Delegated Conglomeration*, как показано в листинге 5.24. Теперь, когда пользователи приглашают друзей, они не видят или не имеют прямого доступа к информации об адресах их электронной почты. Вместо этого они отсылают уведомление, содержащее запрос учетных записей друзей. Это гарантирует отправку приглашения без разглашения электронных адресов.

Листинг 5.24. Шаблон *Delegated Conglomeration* в языке Java

```

1  import java.util.Vector;
2  import java.lang.String;
3
4  public class UserAccount {
5      Vector<UserAccount> allMyFriends;
6      public String notificationEmail;
7
8      public void inviteToEvent (
9          Vector<UserAccount> friends, String msg) {
10         for (UserAccount friend : friends) {
11             friend.notify(this, msg);
12             // ^--- Delegated Conglomeration
13         }
14     }
15
16     public boolean notify (UserAccount inviter,
17                           String msg) {
18         // Отослать e-mail объекту notificationEmail
19         return true;
20     };
21 };

```

В листингах 5.23 и 5.24 формально описан шаблон *Delegated Conglomeration*, но второй вариант более надежен и понятен.

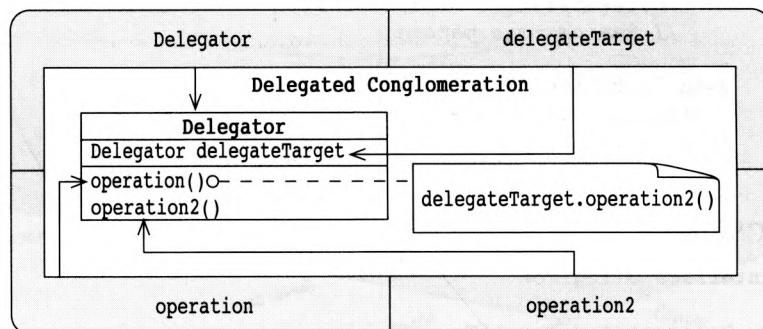
В листинге 5.23 избирательно показаны закрытые данные в зависимости от того, кто их запрашивает, а в листинге 5.24 демонстрируется, что шаблон *Delegated Conglomeration* можно использовать, чтобы полностью скрыть информацию, которая ни при каких условиях не должна предоставляться. Второй объект отвечает исключительно за решение задачи с помощью закрытых данных.

Применимость

Шаблон *Delegated Conglomeration* используется в следующих случаях.

- Задачу можно разбить на подзадачи, которые можно решить с помощью однотипных объектов.
- Для решения задачи необходима согласованная работа многих однотипных объектов.
- Отдельный объект не может полностью решить задачу.
- Задача требует данных, которые должны быть скрыты от остальных объектов.

Структура



Участники

Delegator

Тип объекта, содержащего ссылки на другие экземпляры того же типа.

delegateTarget

Вложенный экземпляр, вызываемый для выполнения задачи.

operation

Точка вызова в первом объекте.

operation2

Подзадача, выполняемая вторым объектом.

Отношения

В этом шаблоне используются только один тип объектов, *Delegator*, и два разных экземпляра этого типа. Один объект зависит от другого, потому что второй объект выполняет часть задания, как в шаблоне *Delegation*, и эта подзадача не связана прямо с текущим запросом, как в шаблоне *Conglomeration*.

Результаты

Как и шаблон *Redirected Recursion*, этот шаблон предоставляет возможность распределить задачи между многими объектами. Однако в отличие от своего аналога шаблон *Delegated Conglomeration* содержит четкую точку принятия решения, отделяя логику поведения от реализации.

Реализация

В языке C++

```

1  class Delegator {
2      Delegator* delegateTarget;
3  public:
4      void operation() {
5          // Предварительная работа
6          delegateTarget->operation2();
7          // Завершающая работа
8      };
9      void operation2() {
10         // Полезная работа
11     };
12 }
```

В языке Objective-C

```

1  @interface Delegator
2  {
3      Delegator* delegateTarget;
4  }
5  - (void) operation;
6  - (void) operation2;
7  @end
8
9  @implementation Delegator
10 - (void) operation
11 {
12     // Предварительная работа
13     [delegateTarget operation2];
14     // Завершающая работа
15 }
16 - (void) operation2
17 {
18     // Полезная работа
19 }
20 @end
```

Родственные шаблоны

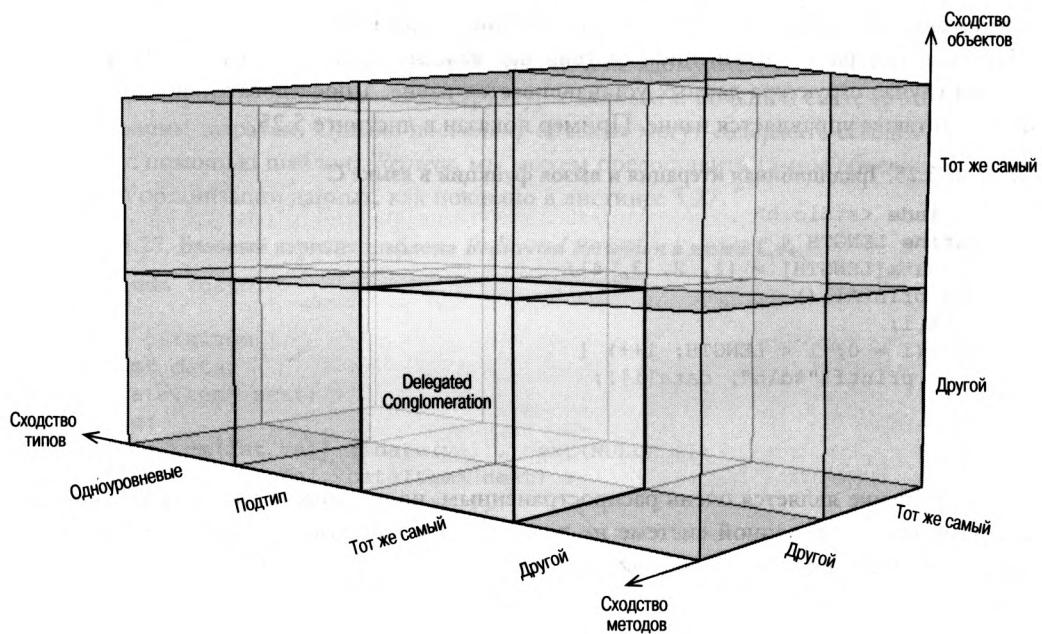
Очевидно, что шаблон *Delegated Conglomeration* можно преобразовать в шаблон *Delegation*, ослабив типовые отношения между объектами, или в шаблон *Conglomeration*, позволив объекту вызывать самого себя. Если отношение типов изменить на “одноуровневые”, то объект будет вызываться из семейства родственных классов, что приводит к шаблону *Trusted Delegation*. Сохранив сходство типов и оставив разные объекты, поместив при этом вызов метода в другой объект *с тем же самым поведением*, мы получим шаблон *Redirected Recursion*.

Классификация вызовов метода

Объекты: разные

Типы объектов: одинаковые

Методы: разные



Redirected Recursion

Поведенческий шаблон

Назначение

Предназначен для выполнения отдельного действия в нескольких объектах, несущих ответственность за распределение и активизацию этого действия.

Мотивация

Часто возникает необходимость выполнять одно и то же действие над большим объемом данных. Иногда эти данные лучше организовать во внешнем контейнере, а данные рассматривать как “простые”, не обращая внимания на их организацию; они обрабатываются внешней сущностью, обходящей контейнер. Примером такого процесса является обход контейнера и выполнение действия над каждым хранящимся в нем элементом. В этом случае структура данных устанавливается извне, а последовательность вызовов функции также управляет извне. Пример показан в листинге 5.25.

Листинг 5.25. Традиционная итерация и вызов функции в языке С

```

1 #include <stdio.h>
2 #define LENGTH 4
3 int data[LENGTH] = {1, 2, 3, 4};
4 void printAll() {
5     int i;
6     for(i = 0; i < LENGTH; i++) {
7         printf("%d\n", data[i]);
8     }
9 }
```

Это решение является очень распространенным, но его можно улучшить. Данные в объектно-ориентированной системе не должны быть “простыми”. При получении запроса они могут выполнить собственные действия. Возможный вариант решения приведен в листинге 5.26.

Листинг 5.26. Объектно-ориентированная итерация и вызов функции в языке C++

```

1 #include <cstdio>
2
3 class DataItem {
4     int data;
5 public:
6     DataItem(int val) : data(val) {};
7     void print() {
8         printf("%d\n", data);
9     }
10 };
11
12 #define LENGTH 4
13 DataItem data[LENGTH] =
```

```

15     {DataItem(1), DataItem(2),
16      DataItem(3), DataItem(4)};
```

- 17 **void** printAll() {
- 18 **int** i;
- 19 **for**(i = 0; i < LENGTH; i++) {
- 20 data[i].print();
- 21 }
- 22 };

Теперь данные несут ответственность за печать. Функция `printAll()` ничего не знает о данных, способе их кодировки или печати. Она просто знает, как выполнить обход контейнера, и вызывает из каждого объекта метод `print()`.

Это решение лучше, но можно сделать еще один шаг. Данные можно хранить не в виде массива, а в виде графа или сбалансированного дерева. В текущем состоянии любой фрагмент кода, желающий применить операцию к каждому элементу массива, пришлось бы модифицировать с учетом новой структуры данных. В большой системе это привело бы к большим затратам. Однако поскольку объекты могут содержать ссылки на другие объекты с помощью шаблона *Retrieve*, мы можем предоставить самим объектам позаботиться об организации данных, как показано в листинге 5.27.

Листинг 5.27. Базовый вариант шаблона *Redirected Recursion* в языке C++

```

#include <cstdio>
1
2   class DataItem {
3       int data;
4       DataItem* next;
5   public:
6       DataItem(int val) : data(val), next(NULL) {};
7       DataItem(int val, DataItem* next) :
8           data(val), next(next) {};
9   void print() {
10       printf("%d\n", data);
11       if (next) {
12           next->print();
13           // ^--- Redirected Recursion
14       }
15   };
16
17   DataItem data =
18     {DataItem(1,
19       new DataItem(2,
20         new DataItem(3,
21           new DataItem(4))))};
22
23   void printAll() {
24       data.print();
25   };

```

Теперь выполняется не только вывод на печать, но и поиск следующих порций данных. Поскольку данные сами отвечают за свою организацию, они контролируют и последовательность вызовов. Если разработчик класса `DataItem` решит изменить его реализацию и будет хранить объекты в красно-черном дереве, хешированном массиве или в другой структуре данных, он сможет это сделать, и код функции `printAll()` изменять не придется. Это не всегда целесообразно, но в некоторых ситуациях, когда объекты в системе сами отвечают за свою организацию и для инициации сложной последовательности вызовов достаточно простой команды, этот механизм является очень мощным.

Представьте себе, например, очередь парашютистов, готовых к прыжку. Места мало, поэтому командир не может пройти вдоль очереди и приказать прыгать каждому парашютисту индивидуально. Вместо этого он становится в конце очереди и когда наступает момент прыжка, хлопает по плечу последнего парашютиста. Этот парашютист знает, что должен, в свою очередь, хлопнуть по плечу стоящего перед ним парашютиста и прыгнуть после него. Этот процесс можно выполнять в очередях любой длины — от 2 до 200 парашютистов. Единственная задача каждого парашютиста — хлопнуть по плечу стоящего перед ним парашютиста, когда его самого хлопнули по плечу стоящий за ним парашютист, подождать, переместиться вперед на освободившееся место и, если предыдущий парашютист прыгнул, прыгнуть следующим. Итак, вместо того чтобы отдавать приказ каждому парашютисту, командир отдает только один приказ. Программа, описывающая этот процесс, показана в листинге 5.28.

Листинг 5.28. Парашютисты, реализующие шаблон *Redirected Recursion*

```

1  class Paratrooper {
2      bool _hasJumped;
3      Paratrooper* nextTrooper;
4
5      Paratrooper() : _hasJumped(false) {}
6      void jump() {
7          if (nextTrooper) {
8              nextTrooper->jump();
9              // ^--- Redirected Recursion
10             while (! nextTrooper->hasJumped()) {
11                 shuffleForward();
12             }
13         }
14         leap();
15     };
16     void leap() {
17         _hasJumped = true;
18         // Прыгаем
19     };
20     void shuffleForward() {
21         // Шаг вперед
22     };
23     bool hasJumped() {
24         return _hasJumped;
25     };
26 };

```

```

27
28     class Commander {
29         Paratrooper* backOfLine;
30     public:
31         void greenLight() {
32             backOfLine->jump();
33         };
34     };

```

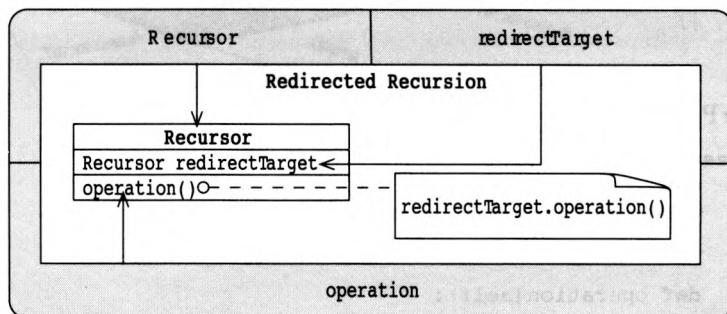
Парашютист не сможет прыгнуть до тех пор, пока предыдущий парашютист не выполнит свое задание, и т.д. В конце концов каждый парашютист выполнит свое задание, когда наступит его очередь, и для этого требуется очень мало инструкций. Приказ передается по цепочке вперед, но действие возвращается назад. Первый парашютист, которого хлопнули по плечу, прыгает последним.

Применимость

Шаблон *Redirected Recursion* используется в следующих ситуациях.

- Исходную задачу легко разбить на подзадачи с помощью рекурсии.
- Для выполнения задания необходимо взаимодействие между многими однотипными объектами.
- Объект сам отвечает за свое взаимодействие с другими объектами.

Структура



Участники

RecurSOR

Тип объекта, хранящий ссылку на следующий экземпляр того же типа.

redirectTarget

Вложенный экземпляр.

operation

Метод в классе **RecurSOR**, рекурсивно вызывающий себя с помощью объекта **redirectTarget**.

Отношения

В решении задачи участвует несколько экземпляров одного и того же типа. Каждый экземпляр знает, как передать сообщение следующему объекту, находящемуся в очереди.

Результаты

Это мощный метод реализации совместного рекурсивного поведения с использованием нескольких объектов, и применяется он во многих системах. Он позволяет разделить функциональные возможности среди разделенных наборов данных, которые допускают самоорганизацию, но ограничивает эти функциональные возможности отдельным конкретным действием. Это действие аналогично отдельной инструкции, которая применяется ко многим данным (SIMD) в графическом процессоре, за исключением того, что данные сами передают инструкцию по цепочке. Такой подход может оказаться очень гибким способом решения задачи, требующей применения алгоритма “разделяй и властвуй”, поскольку изменение реализации метода распространяется на все объекты.

Реализация

На языке Java

```

1  public class Recursor {
2      Recursor redirectTarget;
3      public void operation() {
4          // Предварительная работа...
5          redirectTarget.operation();
6          // Завершающая работа...
7      };
8  };

```

На языке Python

```

1  class Recursor:
2      def __init__(self):
3          __redirectTarget = Recursor()
4
5      def operation(self):
6          # Предварительная работа...
7          redirectTarget.operation();
8          # Завершающая работа...

```

Родственные шаблоны

Как следует из названия, шаблон *Redirected Recursion* может быть преобразован в шаблон *Redirection*, если исключить требование совпадения методов. Аналогично отмена требования различия между объектами и его замена противоположным условием приводит к шаблону *Recursion*. Условие различия между объектами, обеспечивающее взаимодействие между многочисленными объектами, и ослабление условия сходства методов порождает шаблон *Delegated Conglomeration*. В заключение шаблон *Trusted Redirection*

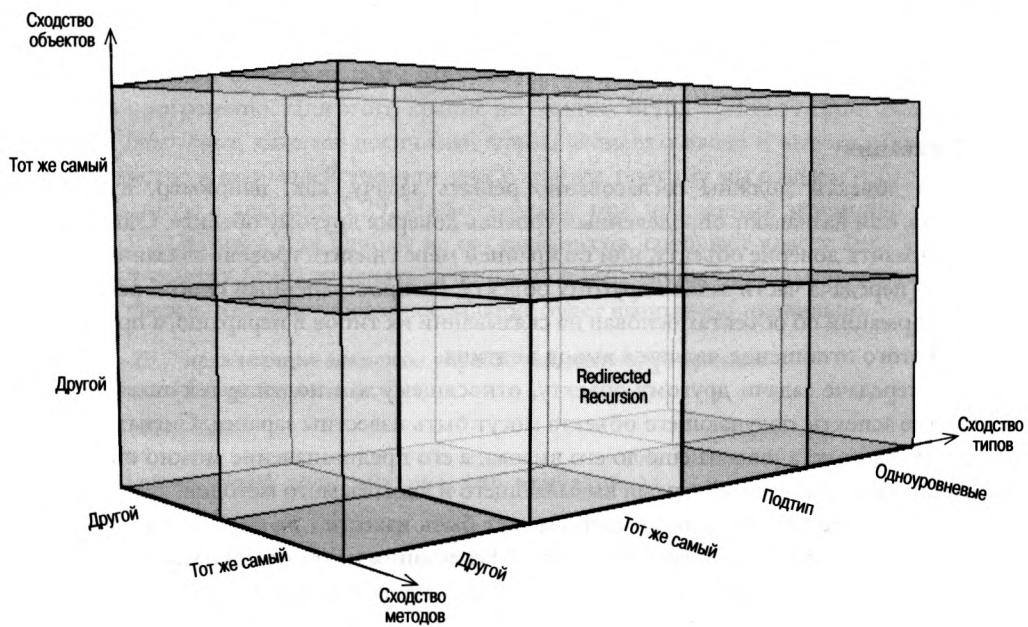
получается в результате замены отношения между типами разновидностью подтипа, обеспечивающего полиморфный вызов метода через доверенную коллекцию классов, связанных с классом, инициировавшим вызов.

Классификация вызовов метода

Объекты: разные

Типы объектов: одинаковые

Методы: одинаковые



Redirected Recursion

Поведенческий шаблон

Назначение

Родственные классы часто определяются так, чтобы выполнять задания коллективно. В этих ситуациях многочисленные объекты родственных типов могут взаимодействовать между собой, чтобы делегировать задания друг другу. Если объекты относятся к родственным типам, они могут выдвигать точные предположения об уровне доверия другому объекту.

Мотивация

Когда объекты должны согласованно решать задачу, как, например, в шаблоне *Delegation*, они назначают определенный уровень доверия другому объекту. Один из способов выразить доверие объекту, или по крайней мере снизить уровень связанного с ним риска, — передача части задачи другому объекту. Распространенный способ распределения информации об объектах основан на связывании их типов в иерархию, а простейшей формой этого отношения является вывод подтипа.

При передаче задачи другому объекту, относящемуся к подтипу текущего объекта, некоторые аспекты получающего объекта могут быть известны заранее. Сигнатура метода должна быть установлена еще до его вызова, а его предназначение можно определить по отношению между сигнатурой вызывающего и вызываемого методов, как в шаблоне *Delegation*. Кроме того, предназначение может быть известно *точно*, потому что вызывающий объект имеет *точно такой же метод*. Реализации могут отличаться, но вызывающий метод наследует предназначение и сигнатуру метода из базового класса. Эти методы могут даже иметь одну и ту же реализацию.

Однако самыми важными в этом случае являются предназначение и концепция. Вызывающий объект знает, что объект-делегат имеет *по крайней мере* семантику суперкласса. С помощью полиморфизма (если он возможен), объект-делегат может ограничить эту семантику, чтобы создать специализацию, но в целом он не имеет права ее ослабить. Вызывающий объект знает, что объект-делегат соответствует заранее установленному набору требований.

С формальной точки зрения это относится к любому языку со строгим контролем типов, поскольку вызывающий объект должен знать о типе объекта-делегата. Тот факт, что этот тип совпадает с типом вызывающего объекта, создает совершенно новый уровень знания. Это дополнительное знание позволяет вызывающему объекту назначить объекту-делегату более высокий уровень доверия.

Шаблон *Trusted Delegation* и родственные шаблоны, такие как *Deputized Delegation*, *Trusted Redirection* и *Deputized Redirection*, часто встречаются в пользовательских интерфейсах. Шаблон *Trusted Delegation* позволяет распределить задачи в семействе классов, которое часто называется кластером классов. В этом случае интерфейс и имя метода известны, но точный тип объекта, а значит, и тело метода могут оставаться неизвестными.

Этот шаблон создает полиморфное делегирование, в котором вызывающий объект имеет один из полиморфных типов.

Рассмотрим оконную систему, включающую в себя ползунки и дисковые номеронабиратели в качестве управляющих элементов ввода и поля редактирования и диаграммы в качестве управляющих элементов вывода данных. Управляющие элементы ввода связаны с конкретным управляющим элементом вывода и посылают ему значение, обновленное пользователем. Управляющие элементы ввода не обязаны знать вид элемента управления выводом, с которым они связаны; они просто должны знать, что обязаны вызвать метод `updateValue` с соответствующим значением в качестве параметра. Поскольку управляющие элементы вывода должны отображать значения на экране, их настройки можно изменять программно. Для этого крайне необходим метод `updateValue`. Вспоминая о шаблоне *Inheritance*, кажется логичным, чтобы элементы ввода и вывода образовывали одно семейство и взаимодействовали друг с другом, поэтому мы создаем иерархию классов, показанную в листинге 5.29. Информация о том, что данный экземпляр принадлежит классу `UIWidget` или одному из его подклассов, сообщает классу `InputControl`, что объект-делегат будет изменяться пользователем. Таким образом, программист может быть уверен, что делегированное задание будет выполнено правильно.

Листинг 5.29. Управляющие элементы управления, демонстрирующие шаблон *Trusted Delegation* в языке C++

```
1 class UIWidget {
2 public:
3     virtual void updateValue( int newValue );
4 };
5
6 class InputControl : UIWidget{
7     UIWidget* target;
8 public:
9     void userHasSetNewValue(int myNewValue) {
10         target->updateValue(myNewValue);
11         // ^--- Trusted Delegation
12     }
13 };
14
15 class SliderBar : public InputControl {
16 public:
17     // Перемещаем ползунок
18     void updateValue( int newValue );
19     void acceptUserClick() {
20         // Определяем новое значение, задаем его
21         int newVal;
22         // Обновляем ползунок
23         this->userHasSetNewValue(newVal);
24     }
25 };
26
27 class RotaryKnob : InputControl {
28 public:
29     // Вращение номеронабирателя
```

```

30     void updateValue( int newValue );
31     void acceptUserClick() {
32         // Определяем новое значение, задаем его
33         int newVal;
34         // Обновляем кнопку
35         this->userHasSetValue(newVal);
36     };
37 }
38
39 class GraphicsContext {
40 public:
41     virtual void render(int);
42 };
43
44 class DisplayWidget : public UIWidget {
45 protected:
46     // Каждый подкласс должен присвоить
47     // параметру GraphicsContext осмысленное
48     // значение
49     GraphicsContext* gc;
50 public:
51     void updateValue( int newValue ) {
52         gc->render( newValue );
53     };
54 }
55
56 class TextWidget : public DisplayWidget {};
57
58 class BarGraph : public DisplayWidget {};

```

Как показано на рис. 5.3, ключевые роли в шаблоне *Trusted Delegation* играют классы `InputControl` и `UIWidget`. Класс `InputControl` имеет больше информации о последствиях вызова метода `updateValue`, потому что он ему известен и тесно связан с классом `UIWidget` отношением подтипа. Если бы целевой объект имел тот же самый тип, то класс `InputControl` имел бы меньше информации о последствиях вызова метода `updateValue()`.

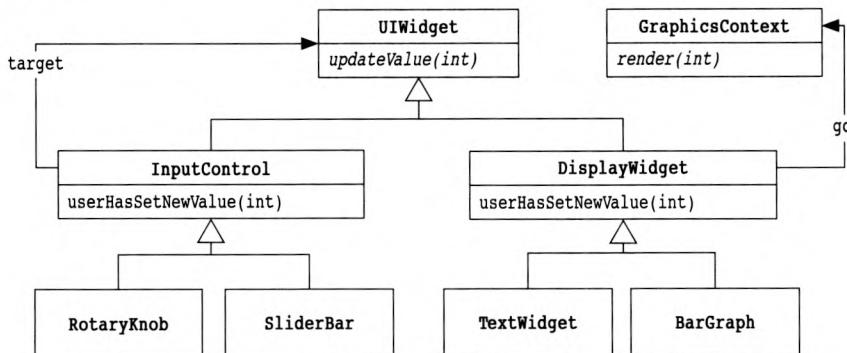


Рис. 5.3. Кластер классов пользовательского интерфейса как экземпляр шаблона Trusted Delegation

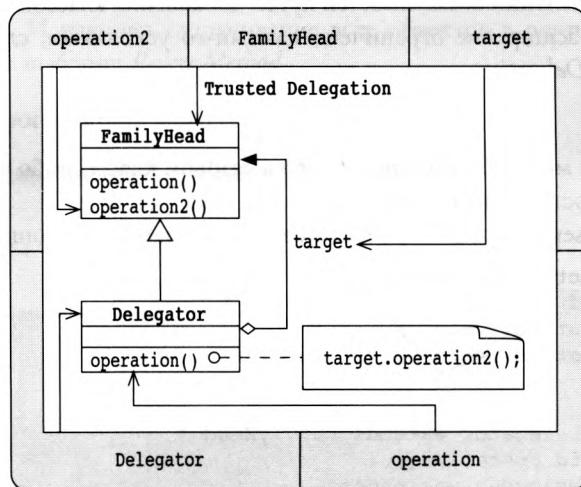
Объекты классов `SliderBar` и `RotaryKnob` не должны ничего знать о значениях и могут быть связаны один с другим, чтобы синхронно отображать данные. Можно даже связать один с другим два объекта подкласса `InputControl`, как и два объекта класса `SliderBar`. Мы отделили отправителя от получателя данных. Наша единственная задача — полиморфно отослать данные соответствующему клиенту и сделать вызывающий объект экземпляром класса из полиморфного семейства. Это позволяет создать единственный интерфейс для многих классов, которые согласованно решают много задач.

Применимость

Шаблон *Trusted Delegation* используется в следующих ситуациях.

- При решении связанных и/или не связанных одна с другой задач допускается делегирование.
- При выполнении задания предполагается определенный уровень доверия.
- Реальную реализацию заранее выбрать невозможно, поэтому для обработки сообщения может потребоваться полиморфизм.
- Вызывающий объект относится к классу, принадлежащему полиморфной иерархии.

Структура



Участники

`FamilyHead`

Базовый класс полиморфного кластера.

`Delegator`

Подкласс класса `FamilyHead`.

`target`

Полиморфный экземпляр класса `FamilyHead`, содержащийся в подклассе `Delegator`.

operation

Вызывающий метод, требующий доверенного делегирования.

operation2

Вызываемый метод, которому доверяется выполнение делегированной задачи.

Отношения

Класс FamilyHead является базовым для класса Delegator. Кроме того, экземпляр класса FamilyHead также хранится в классе Delegator для обработки сообщений. Этот экземпляр должен быть полиморфным и иметь возможность по-разному обрабатывать запрос. Класс *Trusted Delegation* отличается от класса *Trusted Redirection* тем, что имеет более общую форму и разделяет на подзадачи (но не уточняет) задачу, которую должен решать исходный метод.

Результаты

Как и при любом наследовании, класс Delegator связан с интерфейсом класса FamilyHead. Реализация целевого метода поручается каждому подклассу полиморфного кластера. По этой причине использование шаблона *Trusted Delegation* может привести к непредвиденным последствиям, если один из классов, входящих в иерархию, реализует свои методы неожиданным образом. С другой стороны, это мощный механизм для расширения функциональных возможностей путем добавления классов в семейство классов. Если возможное расширение ограничено какими-то условиями, следует использовать шаблон *Deputized Delegation*.

Реализация

Объект target может быть определен либо в базовом классе, либо в подклассе; он просто должен быть доступен из подкласса.

В языке Java объект target определяется в суперклассе следующим образом.

```

public abstract class FamilyHead {
    FamilyHead target;
    public abstract void operation();
    public abstract void operation2();
}

public class Delegator extends FamilyHead {
    public void operation() {
        // Предварительная работа
        target.operation2();
        // ^--- Trusted Delegation
        // Завершающая работа
    };
    public void operation2() {};
}

```

В языке Python объект target определяется в подклассе следующим образом.

```

1 class FamilyHead:
    def operation(self):
        pass

```

```

5     def operation2(self):
6         pass
7
8     class Delegator(FamilyHead):
9         target = FamilyHead()
10        def operation(self):
11            # Предварительная работа
12            target.operation2()
13            # ^--- Trusted Delegation
14            # Завершающая работа

```

Родственные шаблоны

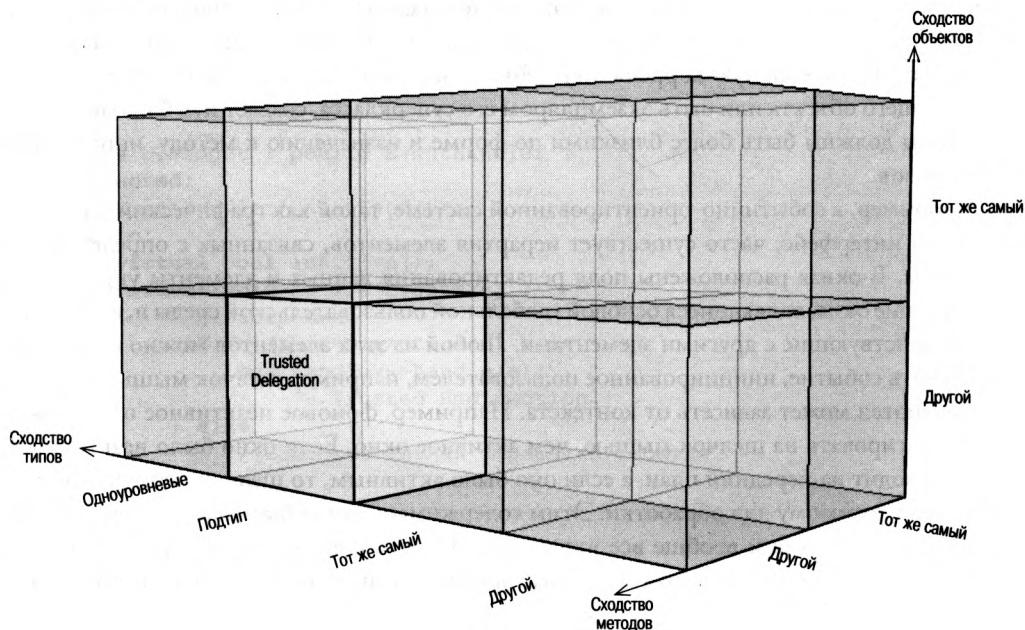
Как следует из названия, шаблон *Trusted Delegation* является специализированной версией шаблона *Delegation*, но делегирование выполняется по отношению к объекту доверенного типа, потому что тип объекта-делегата должен быть подклассом класса вызывающего объекта. Отмена этого условия приводит к обычному шаблону *Delegation*. С другой стороны, усиление ограничений, наложенных на отношения между типами, и возврат к отношению “одноуровневый тип” порождает шаблон *Deputized Delegation*. Усиление ограничений, наложенных на тип, и требование, чтобы он совпадал с типом вызывающего объекта, приводит к шаблону *Delegated Conglomeration*. Сохранение условия одинаковости типов и вызов такого же метода из объекта-делегата создает шаблон *Trusted Redirection*. И наконец сворачивание всех объектов в один экземпляр базового класса приводит к шаблону *Revert Method*.

Классификация вызовов метода

Объекты: одинаковые

Типы объектов: подтип

Методы: разные



Trusted Redirection

Поведенческий шаблон

Назначение

Предназначен для переадресации части реализации метода возможному кластеру классов, членом которого является текущий класс. Надежность классов определяется отношениями между типами.

Мотивация

Часто иерархическая структура родственных объектов создается во время выполнения программы, и возникает необходимость распределить действие между несколькими объектами, имеющими разные реализации. Это эффективный способ передачи ответственности при обработке запроса с помощью нескольких объектов, образующих основу шаблонов *Chain of Responsibility* (Цепочка обязанностей) и *Composite* (Компоновщик) [21]. Шаблон *Trusted Redirection* можно считать отдельным звеном в этих цепочках.

Несмотря на то что в этих ситуациях можно применять шаблон *Redirection*, цепочки родственных классов предполагают более сильные ограничения, чем простое сходство методов. Методы должны быть *identical*, а не просто сходными. Часто даже предполагается, что реализации имеют общее ядро.

Как и в шаблоне *Trusted Delegation*, между вызывающим и вызываемым объектами устанавливается определенный уровень доверия, основанный на знании того, что вызываемый объект имеет очень близкий тип. Однако в шаблоне *Trusted Redirection* требуется более высокий уровень доверия.

Использование шаблона *Redirection* подразумевает, что назначения вызывающего и целевого методов сходные и что между этими методами существует сильная корреляция. Гарантируя, что тип объекта-получателя является супертипом текущего объекта, можно установить более сильную корреляцию. Объявление, что целевой объект должен иметь тип текущего объекта или быть экземпляром его суперкласса, создает пул близких типов. Эти типы должны быть более близкими по форме и назначению к методу, иницииирующему вызов.

Например, в событийно-ориентированной системе, такой как графический пользовательский интерфейс, часто существует иерархия элементов, связанных с определенным событием. В окнах расположены поля редактирования данных и элементы управления, образующие окна, являющиеся основой глобальной пользовательской среды и, возможно, взаимодействующие с другими элементами. Любой из этих элементов можно попросить обработать событие, инициированное пользователем, например щелчок мышью, причем эта обработка может зависеть от контекста. Например, фоновое неактивное окно может иначе реагировать на щелчок мышью, чем активное окно. Если окно было неактивным, оно переходит на передний план, а если оно было активным, то щелчок переадресовывается ее содержимому для обработки. Этим содержимым могут быть панель с вкладками, поле редактирования и вообще все что угодно. Панель с вкладками также может быть активной или неактивной и по-своему реагировать на щелчок. Поле редактирования, в

свою очередь, также реагирует на щелчок, передавая его своему одержимому с учетом текущего состояния. Содержимое поля редактирования может реагировать на щелчок, перетаскивание и другие события, изменяя визуальные атрибуты. Не все обработчики событий имеют визуальные атрибуты. Все визуальные элементы *должны* правильно обрабатывать события, но не все они могут обработать эти события *в любой конкретный момент времени*. Этот временной динамизм является желательным, но не необходимым ограничением шаблона *Trusted Redirection*.

Пример кода обработчика события в языке C++ приведен в листинге 5.30 и на рис. 5.4.

Листинг 5.30. Обработчик событий в языке C++, демонстрирующий шаблон *Trusted Delegation*

```
1 #include <vector>
2 using std::vector;
3
4 class Event {};
5 class Position {};
6
7 class MouseEvent : public Event {
8     vector<bool> modifierKeys;
9     Position position;
10    bool mouseDown;
11 };
12
13 class EventHandler {
14 public:
15     virtual void handle(Event* e) {
16         // Делаем то, что нужно
17     };
18 };
19
20 class TextData : public EventHandler {
21     // Класс для хранения текста
22 };
23
24 class UIWidget : public EventHandler {
25 protected:
26     EventHandler* nextHandler;
27 public:
28     virtual bool isActive();
29     virtual void handle(Event* e) {
30         // Настройки
31         if ( !(this->isActive()) ) {
32             nextHandler->handle(e);
33             // ^--- Trusted Redirection
34         } else {
35             // Самостоятельная обработка события
36         }
37     };
38 };
39
40 class Button : public UIWidget {
```

```

    // Базовый класс для любой кнопки
42 };

44 class TabPane : public UIWidget {
    UIWidget* contents;
46 public:
    TabPane() {
48         // Настройка содержимого
        nextHandler = contents;
50     };
52 };
52
53 class TextField : public UIWidget {
54     TextData* text;
55 public:
56     TextField() {
57         // Настройка текста
58         nextHandler = text;
59     };
60 };

```

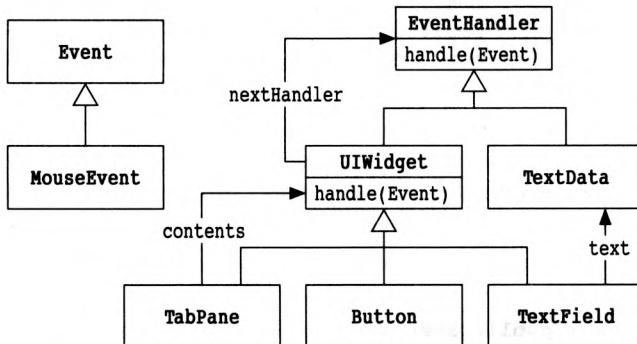


Рис. 5.4. Кластер классов пользовательского интерфейса, демонстрирующий шаблон Trusted Delegation

Каждый подкласс пользовательского интерфейса наследует поведение по умолчанию от класса UIWidget, хотя его можно заменить в любое время. В этой структуре каждый подкласс использует экземпляр класса EventHandler в качестве следующего претендента на обработку события, если текущий объект этого сделать не может. В качестве вариантов такого поведения можно предусмотреть проверку типа события перед его обработкой.

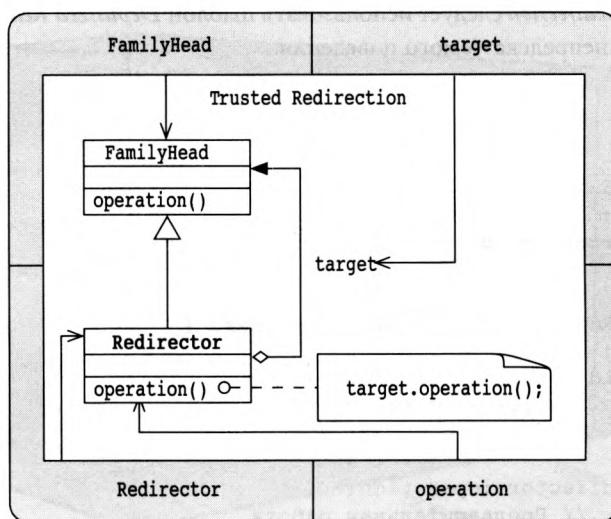
Применимость

Шаблон *Trusted Redirection* используется в следующих ситуациях.

- Во время компиляции или выполнения программы должны быть создана агрегированная структура близких объектов.
- Поведение необходимо разделить между разными объектами, входящими в семейство.

- Структура агрегированных объектов заранее неизвестна.
- Полиморфное поведение желательно, но не обязательно.

Структура



Участники

FamilyHead

Определяет интерфейс и содержит метод, который может быть замещен.

Redirector

Использует интерфейс класса **FamilyHead** и переадресует внутреннее действие назад в экземпляр класса **FamilyHead** для обеспечения полиморфизма в аморфной структуре.

target

Объект, которому поручается переадресованное задание.

operation

Указывает задание, которое должно быть выполнено в вызывающем и вызываемом объектах.

Отношения

Класс **Redirector** зависит от интерфейса класса **FamilyHead** и от своего экземпляра при рекурсивной реализации. Отношение переадресации является главной частью шаблона *Redirection*. Оно описывает принцип “делай, как я говорю”. Аспект близости возникает от переадресации работы текущего объекта супертипу для полиморфной обработки.

Результаты

Класс *Redirector* зависит от интерфейса класса *FamilyHead*, но его задание распределяется по всей иерархии классов для обеспечения разной реализации. По этой причине разные классы в иерархии могут иметь непредсказуемое поведение. В этом случае вместо шаблона *Redirection* следует использовать шаблон *Deputized Redirection*, чтобы снизить вероятность непредсказуемого поведения.

Реализация

В языке C++

```

1  class FamilyHead {
2  public:
3      virtual void operation();
4  };
5
6  class Redirector : public FamilyHead {
7  public:
8      void operation();
9      FamilyHead* target;
10     };
11 void
12 Redirector::operation() {
13     // Предварительная работа
14     target->operation();
15     // Завершающая работа
16 }
17

```

Родственные шаблоны

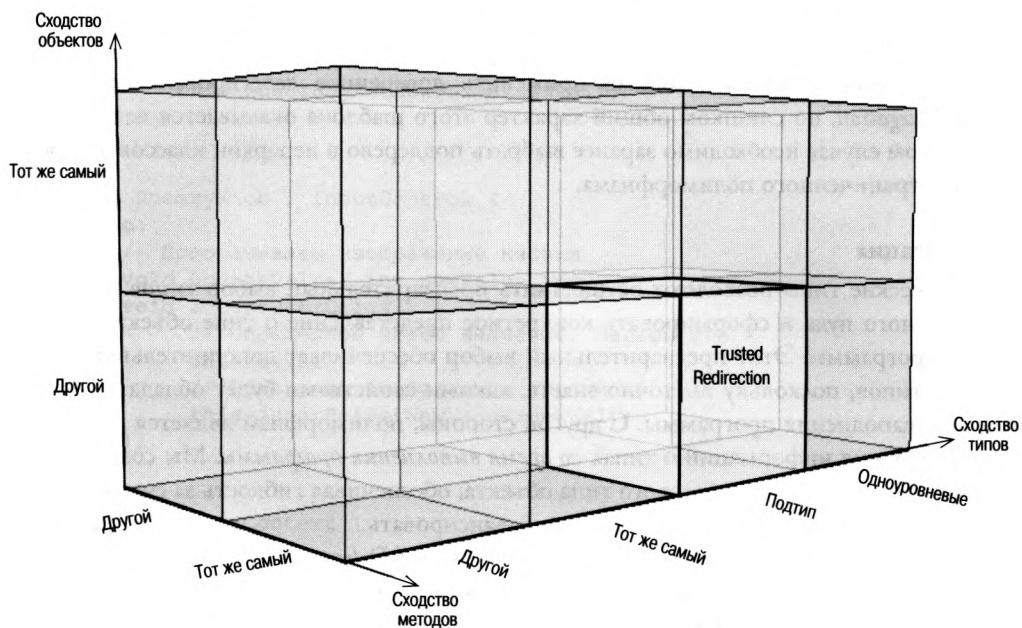
Шаблон *Trusted Redirection* представляет собой специализацию шаблона *Redirection*, в которой целями переадресованного задания являются подклассы, являющиеся членами кластера классов. Снижение уровня доверия приводит к более общим элементарным шаблонам проектирования. Дальнейшее ограничение доверия подмножеством подклассов классов одного и того же уровня приводит к шаблону *Deputized Redirection*. Если необходимо ограничиться только типом целевого объекта, следует использовать шаблон *Redirected Recursion*. Сохранение отношения подтипа и ограничение количества целевых объектов одним объектом приводит к шаблону *Extend Method*. Нарушение условия сходства методов порождает шаблон *Trusted Delegation*.

Классификация вызовов методов

Объекты: сходные

Типы объектов: подтип

Методы: сходные



Deputized Delegation

Поведенческий шаблон

Назначение

Используется, когда необходимо применить доверенное делегирование шаблона *Trusted Delegation*, но слишком общий характер этого шаблона оказывается неприемлемым. В этом случае необходимо заранее выбрать поддерево в иерархии классов для реализации ограниченного полиморфизма.

Мотивация

Статические типы позволяют осуществить предварительный выбор типов из точно определенного пула и сформировать конкретное представление о типе объекта еще *до* запуска программы. Этот предварительный выбор обеспечивает дополнительную безопасность типов, поскольку вы точно знаете, какими свойствами будет обладать объект во время выполнения программы. С другой стороны, полиморфизм является способом абстрагирования информации о типах *во время выполнения программы*. Мы сознательно игнорируем многие детали базового типа объекта, обеспечивая гибкость за счет безопасности типов. Иногда эти факторы следует сбалансировать.

Рассмотрим пример использования шаблона *Trusted Delegation*. В нем не описывается элемент управления для ввода текста. Главной является настройка: класс `TextInput`, являющийся подклассом класса `InputControl`, может изменить значение элемента `BarGraph`, демонстрирующего числа, и это может не соответствовать желаниям проектировщика.⁸ Проект, основанный на шаблоне *Trusted Delegation*, можно точно настроить с помощью дополнительных классов и модификаций, показанных в листинге 5.31.

Листинг 5.31. Элементы графического пользовательского интерфейса, демонстрирующие шаблон *Deputized Delegation* в языке C++

```
1 class UIWidget {
2 public:
3     virtual void updateValue( int newValue );
4 };
5
6 class InputControl : UIWidget{
7     UIWidget* target;
8 public:
9     void userHasSetnewValue(int myNewValue) {
10         target->updateValue(myNewValue);
11     }
12 };
13 class SliderBar : InputControl {
```

⁸ Существование отдельного элемента управления для отображения чисел является вполне разумным, если ввод ограничен лишь числами, но графический пользовательский интерфейс обычно устроен иначе.

```
15 public:
    // Перемещаем ползунок
17     void updateValue( int newValue );
    void acceptUserClick() {
19         // Определяем новое значение, задаем его
        int newVal;
21         // Обновляем изображение ползунка
        this->userHasSetNewValue(newVal);
23     };
25 }
27 class RotaryKnob : InputControl {
28 public:
    // Поворачиваем изображение кнопки
29     void updateValue( int newValue );
    void acceptUserClick() {
31         // Определяем новое значение, задаем его
        int newVal;
33         // Обновляем изображение кнопки
        this->userHasSetNewValue(newVal);
35     };
36 };
37
38 class GraphicsContext {
39 public:
    virtual void render(int);
40 };
41 }

43 class DisplayWidget : public UIWidget {
44 protected:
45     // Каждый подкласс должен задать объект
        // GraphicsContext в соответствии
46     // со своими требованиями
47     GraphicsContext* gc;
48 public:
    void updateValue( int newValue ) {
49         gc->render( newValue );
50     };
53 };

55 class TextWidget : public DisplayWidget {};

57 class BarGraph : public DisplayWidget {};

59     // Новый класс для шаблона Deputized Delegation

60     class TextInputControl : public InputControl {
61         TextWidget* textTarget;
62     public:
63         void userHasSetNewValue(int myNewValue) {
64             textTarget->updateValue(myNewValue);
65             // ^--- Deputized Delegation
66         };
67 };
```

Теперь класс `TextInputControl` может изменять экземпляры класса `TextField`, но не экземпляры класса `Bar-Graph` и другие нетекстовые элементы управления. Классы `SliderBar` и `RotaryKnob` можно просто ограничить иерархией классов, основанной на классе `NumericWidget`, например. Эта ситуация изображена на рис. 5.5.

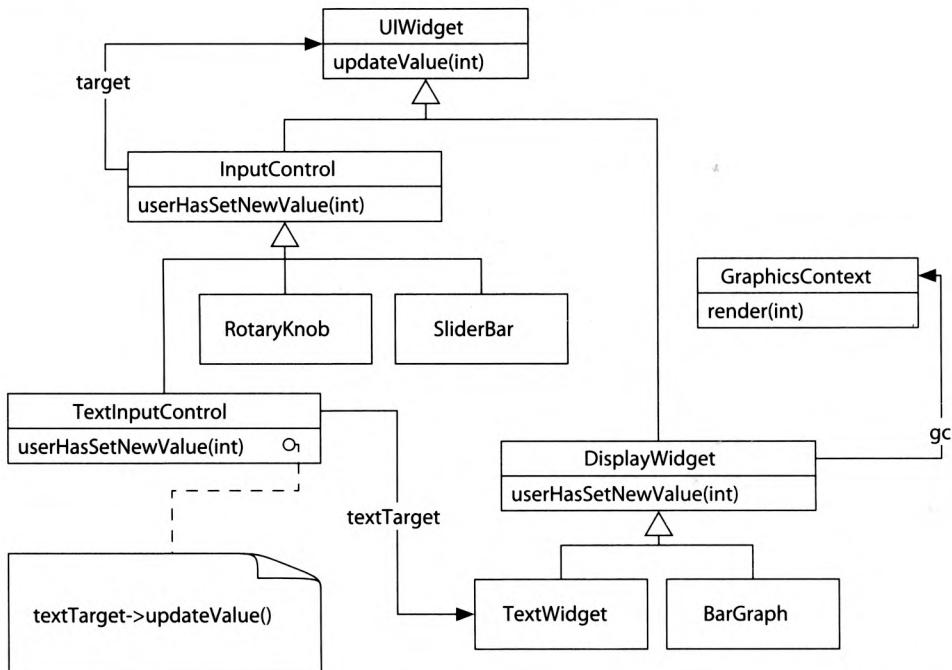


Рис. 5.5. Кластер классов пользовательского интерфейса, демонстрирующий шаблон Deputized Delegation

Применимость

Шаблон *Deputized Delegation* используется в следующих ситуациях.

- Шаблон *Trusted Delegation* является приемлемым.
- Требуется установить более строгий контроль за возможными типами.
- Делегируемые типы не включают в себя тип вызывающего объекта.

Участники

FamilyHead

Базовый класс полиморфного кластера, позволяющий задать обобщенные интерфейсы и семантику.

Delegator

Подкласс класса **FamilyHead**, который должен делегировать подзадачу доверенному делегату.

DelegateSibling

Другой подкласс класса FamilyHead, имеющий подходящую семантику для решения подзадачи.

target

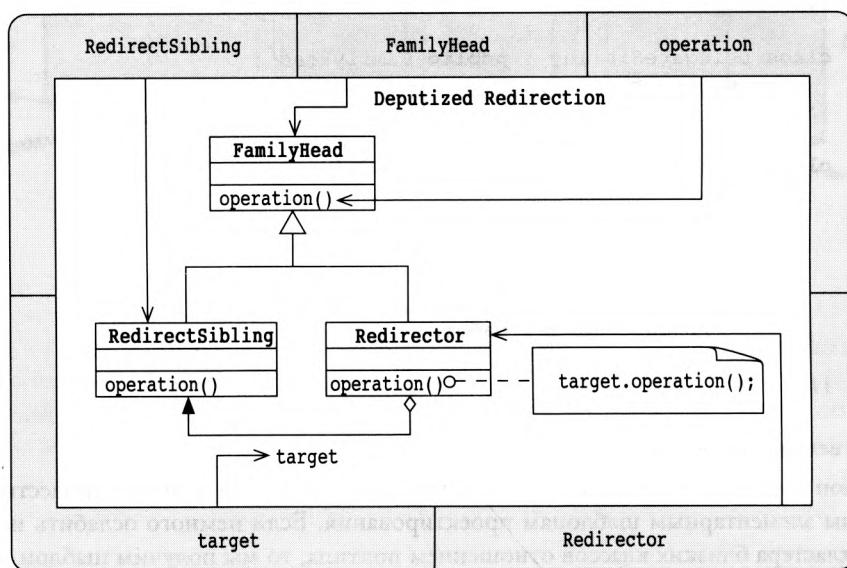
Экземпляр класса DelegateSibling, содержащийся в классе Delegator и выполняющий подзадачу.

operation

Вызывающий метод, отсылающий запрос на выполнение подзадачи.

operation2

Вызываемый метод, который должен решить подзадачу.

Структура**Отношения**

Помимо участников шаблона *Trusted Delegation*, этот шаблон содержит класс **DelegateSibling**, являющийся целью для реализации метода *operation*. Это ограничивает возможное поведение подкласса базового класса и обеспечивает выбор уточненной семантики.

Результаты

Этот шаблон отличается от шаблона *Trusted Delegation* тем, что проектное решение статически ограничивает полиморфизм поддеревом исходного семейства классов. Он может создать преимущества при решении сложных задач, но в то же время порождает проблемы, если, например, выбранный одноуровневый класс имеет *слишком сильные* ограничения. Несмотря на то что синтаксические изменения кода минимальны,

добавление широкого множества классов и функций может иметь далеко идущие последствия, которые следует внимательно анализировать. Кроме того, в отличие от шаблона *Trusted Delegation*, объект *target* не может быть членом общего суперкласса, потому что суперкласс имел бы априорную информацию о своих подклассах. Несмотря на то что некоторые языки программирования могут допускать такую возможность, это решение является ненадежным и его следует избегать.

Реализация

В языке C++

```

1 class FamilyHead {
2     protected:
3         void operation();
4         void operation2();
5     };
6
7 class DelegateSibling : public FamilyHead {
8     void operation2();
9 };
10
11 class Delegator : public FamilyHead {
12     DelegateSibling* target;
13     void operation() {
14         // Предварительная работа
15         target->operation2();
16         // ^--- Deputized Delegation
17         // Завершающая работа
18     };
19 };

```

Родственные шаблоны

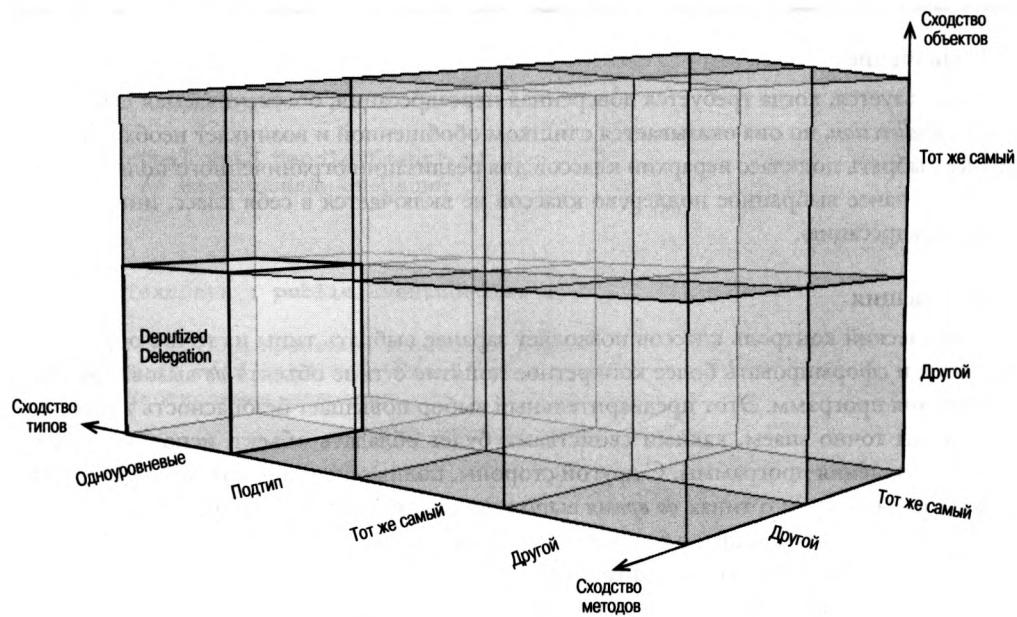
Шаблон *Deputized Delegation* является специализированным и может привести к более общим элементарным шаблонам проектирования. Если немного ослабить использование кластера близких классов отношением подтипа, то мы получим шаблон *Trusted Delegation*. Если вызываемый метод изменить на метод, аналогичный вызывающему, то получится шаблон *Deputized Redirection*. Это единственный случай, когда сворачивание вызывающего объекта и объекта-делегата в один объект нецелесообразно, поскольку при этом получается неправильная позиция в пространстве элементарных шаблонов.

Классификация вызовов метода

Объекты: разные

Типы объектов: одноуровневые

Методы: разные



Deputized Redirection

Поведенческий шаблон

Назначение

Используется, когда требуется доверенная переадресация, обеспечиваемая шаблоном *Trusted Redirection*, но она оказывается слишком обобщенной и возникает необходимость заранее выбрать подкласс иерархии классов для реализации ограниченного полиморфизма. Это заранее выбранное поддерево классов не включается в себя класс, инициирующий переадресацию.

Мотивация

Статический контроль классов позволяет заранее выбрать типы из точно определенного пула и сформировать более конкретное понятие о типе объекта *до* вызова системы выполнения программ. Этот предварительный выбор повышает безопасность типов, поскольку мы точно знаем, какими свойствами будет обладать объект, используемый во время выполнения программы. С другой стороны, полиморфизм — это метод абстрагирования информации о типах *во время* выполнения программы. Мы преднамеренно игнорируем множество деталей о базовом типе объекта, повышая гибкость за счет безопасности. Иногда эти два фактора необходимо балансировать.

В примере, иллюстрирующем использование шаблона *Trusted Redirection*, код описывал очень общую событийно-ориентированную систему. Однако такая открытая иерархия подходит не для всех ситуаций. Например, ползунок должен реагировать только на события, связанные с мышью, а не с клавиатурой (за редкими исключениями). Аналогично поле редактирования должно реагировать на события, связанные с мышью и клавиатурой, но ответ на нажатие клавиш лучше поручить соответствующему объекту текстовых данных. Следовательно, можно исключить из рассмотрения целую категорию событий. Возьмем за основу листинг 5.30, описывающий шаблон *Trusted Redirection*, и попробуем изменить его так, как показано в листинге 5.32.

Листинг 5.32. Элементы графического пользовательского интерфейса, демонстрирующие шаблон *Deputized Redirection* в языке C++

```
1 #include <vector>
2 using std::vector;
3
4 class Position {};
5 class Key {};
6 class Event {
7 public:
8     virtual int getID();
9 };
10
11 class MouseEvent : public Event {
12     vector<bool> modifierKeys;
13     Position position;
14     bool mouseDown;
15 };
```

```
17 class KeyEvent : public Event {
    vector<bool> modifierKeys;
19     Key key;
    bool keyDown;
21 };

23 class EventHandler {
public:
25     virtual void handle(Event* e) {
        // Необходимые операции
27 };
29 };
31 class TextData : public EventHandler {
public:
    // Класс для хранения текстовых данных
33     virtual void handle(KeyEvent* e) {
        // Необходимые операции
35 };
37 };
39 class UIWidget : public EventHandler {
protected:
    EventHandler* nextHandler;
41 public:
    bool isActive();
43     virtual void handle(Event* e) {
        // Настройки ...
45         if ( !(this->isActive()) ) {
            nextHandler->handle(e);
47         } else {
            // Самостоятельная обработка
49         }
    };
51 };

53 class Button : public UIWidget {
    // Базовый класс для любой кнопки
55 };

57 class TabPane : public UIWidget {
    UIWidget* contents;
59 public:
    TabPane() {
61         // Настройка содержания
        nextHandler = contents;
63     };
65 };
67 class TextField : public UIWidget {
68     TextData* text;
public:
69     TextField() {
        // Настройка текста
```

```

71         nextHandler = text;
72     };
73     virtual void handle(Event* e) {
74         // Является ли параметр e объектом KeyEvent или подклассом?
75         if (dynamic_cast<KeyEvent*>(e)) {
76             // Предварительная работа
77             text->handle(dynamic_cast<KeyEvent*>(e));
78             // ^--- Deputized Redirection
79             // Завершающая работа
80         } else {
81             // Обработка события MouseEvents
82         }
83     };
84 }
```

Новая структура показана на рис. 5.6. Мы добавили класс KeyEvent, а класс TextField теперь замещает метод handle(). Он самостоятельно обрабатывает экземпляры класса MouseEvent, но сначала просто устанавливает указатель next-Handler на объект text класса TextData. Раньше существовала опасность, что событие класса MouseEvent из-за ошибки останется необработанным и попадет в метод handle() класса TextData, но этот путь предназначен только для событий класса KeyEvent. Классы TextField и TextData имеют более высокий уровень доверия, чем класс TextField и другие подклассы класса Event-Handler.

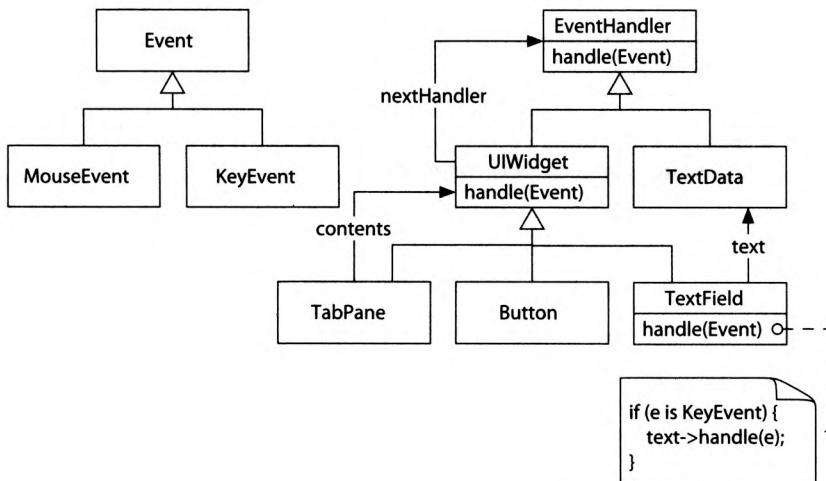


Рис. 5.6. Кластер классов пользовательского интерфейса, демонстрирующий шаблон Deputized Redirection

Теперь класс TextField гарантирует, что будет обрабатывать события класса MouseEvents, в то время как класс TextData будет обрабатывать только текстовые события. Классу TextData поручается обработка событий класса KeyEvent, поскольку класс TextField более точно знает, что с ними будет делать класс TextData. Кроме того, класс TextField не входит в список возможных получателей событий, как в

шаблоне *Trusted Redirection*. Класс `TextField` устроен так, чтобы эту работу выполнял другой класс.

Применимость

Шаблон *Deputized Redirection* используется в следующих ситуациях.

- Шаблон *Trusted Redirection* является приемлемым.
- Требуется более строгий контроль за возможными типами объектов.
- В список типов, которым перенаправляется задача, не входит тип вызывающего объекта.

Участники

FamilyHead

Определяет интерфейс, содержит метод, который может быть замещен и является базовым классом для классов `Redirector` и `RedirectSibling`.

Redirector

Использует интерфейс класса `FamilyHead`; перенаправляет внутреннее поведение обратно экземпляру класса `RedirectSibling`, чтобы обеспечить полиморфное поведение аморфной структуры объектов *с ограниченной областью видимости*.

RedirectSibling

Вершина нового дерева классов, обеспечивающего полиморфное поведение.

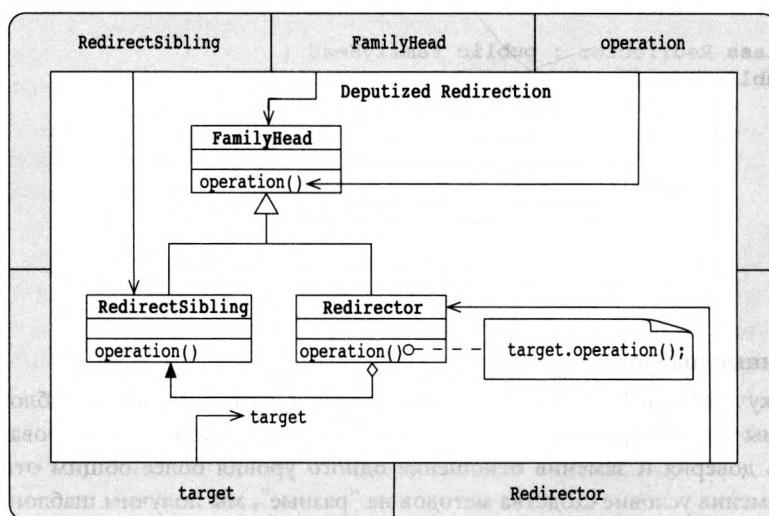
target

Полиморфный экземпляр класса `RedirectSibling`, содержащийся в классе `Redirector`.

operation

Вызывающий метод.

Структура



Отношения

Как и в шаблоне *Trusted Redirection*, классы *Redirector* и *FamilyHead* связаны интерфейсом и целью. Класс *RedirectSibling* представляет собой концептуальную отправную точку для конкретного типа реализации. В данном случае класс *Redirector* преднамеренно ограничивает множество возможных классов, которые могут быть полиморфными целями запроса, классом *Redirect-Sibling* или его подклассами. Тот факт, что классы *Redirector* и *RedirectSibling* имеют общий базовый класс *FamilyHead*, означает, что каждый из них может выдвигать более строгие предположения о другом.

Результаты

Этот шаблон отличается от шаблона *Trusted Redirection* тем, что проектное решение статически ограничивает полиморфизм поддеревом исходного семейства классов. Он может создать преимущества при решении сложных задач, но в то же время порождает проблемы, если, например, выбранный одноуровневый класс имеет *слишком* сильные ограничения. Несмотря на то что синтаксические изменения кода минимальны, добавление широкого множества классов и функций может иметь далеко идущие последствия, которые следует внимательно анализировать.

Реализация

В языке C++

```

1 class FamilyHead {
2 protected:
3     virtual void operation();
4 }
5
6 class RedirectSibling : public FamilyHead {
7     void operation();
8 }
9
10 class Redirector : public FamilyHead {
11 public:
12     RedirectSibling* target;
13     void operation() {
14         // Предварительная работа
15         target->operation();
16         // ^--- Deputized Redirection
17         // Завершающая работа
18     };
19 }
```

Родственные шаблоны

Поскольку шаблон *Deputized Redirection* является специализацией шаблона *Trusted Redirection*, мы можем вернуться к этому элементарному шаблону проектирования, ослабив уровень доверия и заменив отношение одного уровня более общим отношением подтипа. Изменив условие сходства методов на “разные”, мы получим шаблон *Deputized*

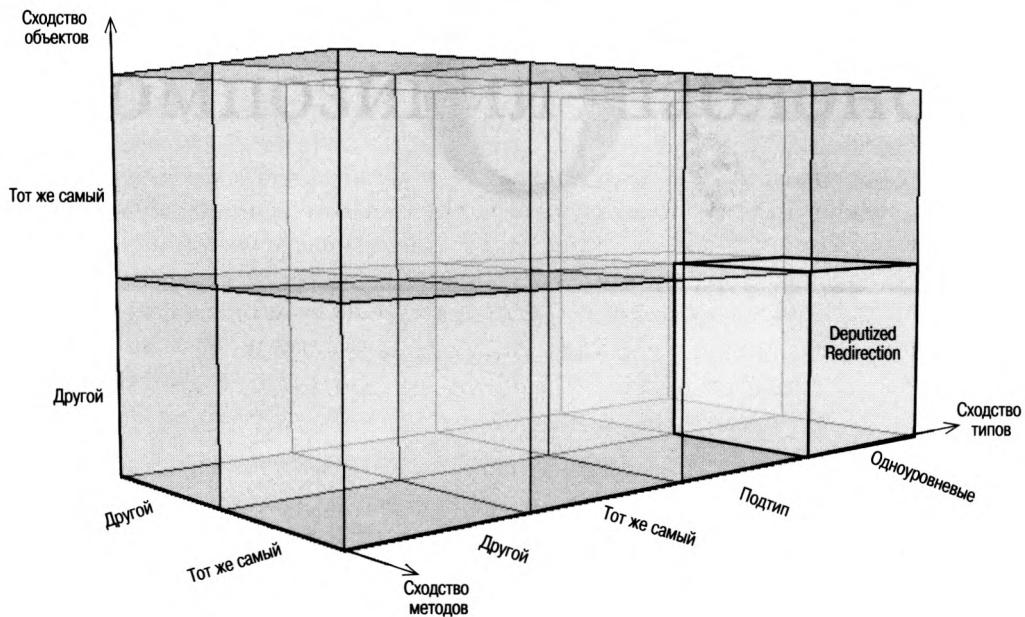
Delegation. И наконец следует подчеркнуть, что изменять условие сходства объектов на “одинаковые” бессмысленно, поскольку мы оказываемся в неопределенном месте пространства элементарных шаблонов.

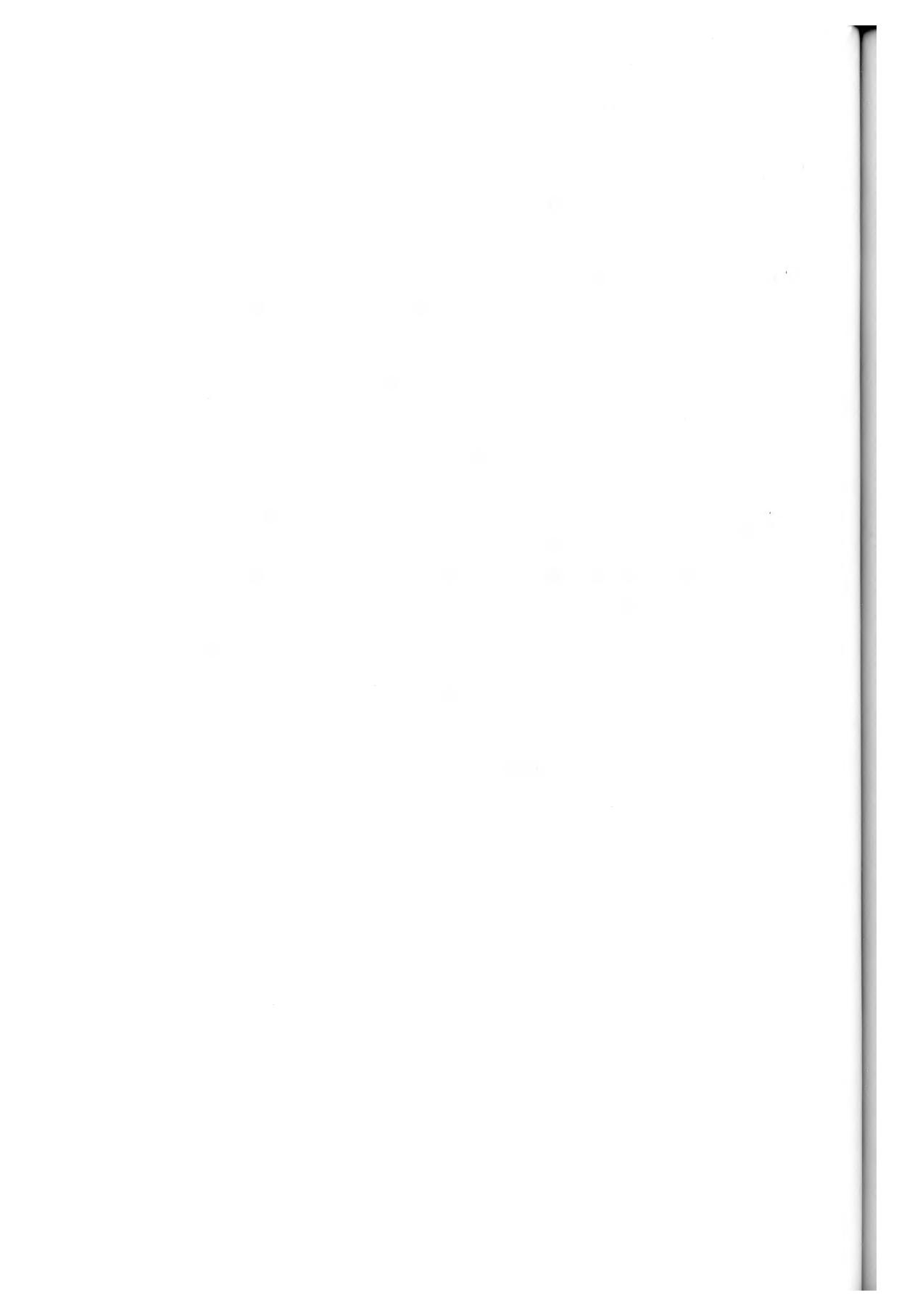
Классификация вызовов метода

Объекты: разные

Типы объектов: одноуровневые

Методы: одинаковые





6 Промежуточные композиции шаблонов

В этой главе формируется первый уровень шаблонов, составленных непосредственно из элементарных шаблонов проектирования. Мы уже описывали этот процесс в главе 4, где рассмотрели несколько общих простых, но не элементарных шаблонов проектирования. Для иллюстрирования составных частей этих шаблонов используется система обозначений Pattern Instance Notation (PIN), изложенная в главе 3. При описании элементарных шаблонов проектирования для демонстрации внутренних отношений наряду с системой обозначений PIN используется язык UML. В диаграммах PIN используются как язык UML, так и компоненты PINbox. Кроме того, для иллюстрации чисто концептуальных связей используется вариант диаграмм, использующий исключительно компоненты PINbox.

Промежуточные шаблоны получили чрезвычайно широкое распространение, и разработчики постоянно их используют, потому что они эффективны и позволяют по-новому сочетать небольшое количество простых концепций. Промежуточные шаблоны не являются элементарными, потому что их можно разложить на составные части и представить в виде комбинации более фундаментальных концепций. Однако часто при обсуждении этих шаблонов пересечение базовых концепций остается неявным. Студенты и разработчики должны сами догадываться о более глубокой семантике шаблонов. Это требует от читателей большого опыта в распознавании шаблонов.

Элементарные шаблоны проектирования, используемые в качестве базиса, позволяют более четко и ясно представить композицию и точки пересечения концепций, образующих шаблоны. В этой главе показано, как из маленьких концепций быстро создать сложные абстракции и проанализировать существующие шаблоны с новой точки зрения. Изучив внутреннее устройство промежуточных шаблонов, можно глубже понять, как из элементарных шаблонов проектирования и других шаблонов создать более крупные и содержательные шаблоны.

Спецификации шаблонов в этой главе не содержат раздела “Классификация вызовов метода”, потому что он характерен только для элементарных шаблонов вызова метода.

Fulfill Method

Структурный шаблон

Назначение

Этот шаблон предназначен для реализации абстрактного метода. Он выполняет контракт шаблона *Abstract Interface*, в соответствии с которым подкласс должен содержать реализацию абстрактного класса.

Мотивация

Шаблон *Abstract Interface* подразумевает, что некоторый подкласс будет содержать реализацию интерфейсного метода суперкласса. Такой подкласс можно определить с помощью наследования, и шаблон *Fulfill Method* описывает, как осуществить реализацию, сочетая экземпляры двух шаблонов.

Этот подход позволяет явно отделить абстракцию метода в суперклассе от любых возможных подклассов, которые могут содержать его реализацию.

Применимость

Шаблон *Fulfill Method* используется в следующих ситуациях.

- Реализация метода отложена с помощью шаблона *Abstract Interface*.
- Подкласс класса, указанного в шаблоне *Abstract Interface*, может осуществить требуемую реализацию метода.

Участники

Abstractor

Класс, в котором объявлен интерфейс метода *operation*.

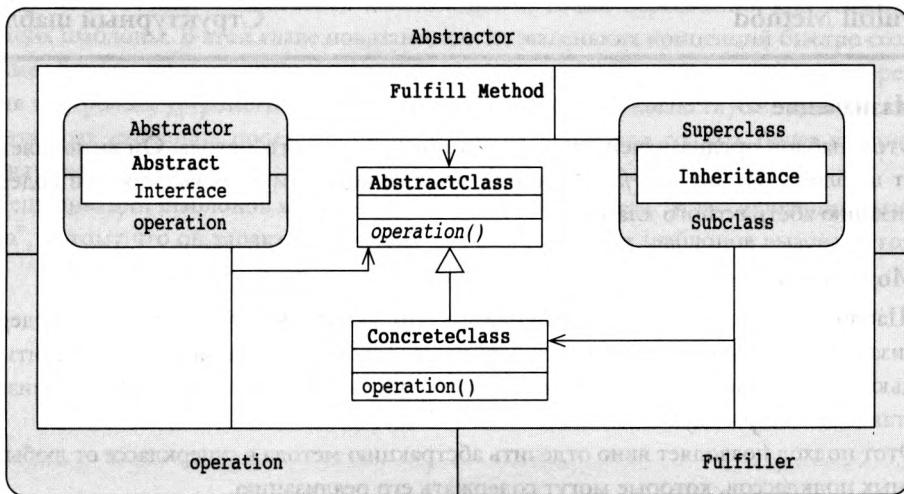
Fulfiller

Класс, в котором определено тело метода *operation*; наследник класса *Abstractor*.

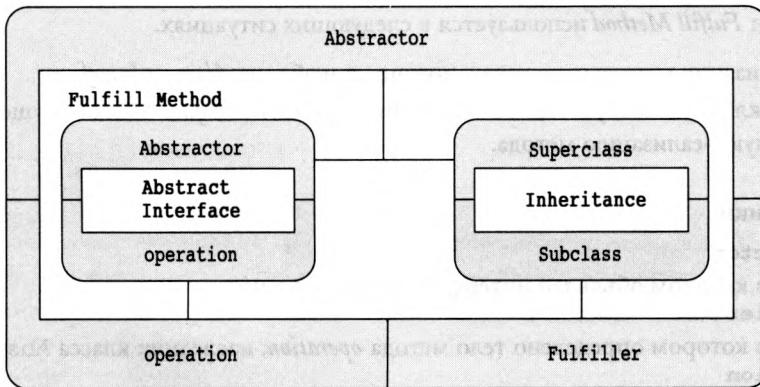
operation

Метод, объявленный в классе *Abstractor* и определенный в классе *Fulfiller*.

Структура



Версия, использующая только компоненты PINbox (см. рис. 4.4).



Отношения

Класс *Abstracter* определяет интерфейс метода, а класс *Fulfiller* обеспечивает его реализацию. Класс *Fulfiller* является подклассом класса *Abstracter*, наследуя его поля и интерфейс.

Класс *Abstracter* является базовым классом в шаблоне *Abstract Interface*, причем классы *Abstracter* и *Fulfiller* связаны один с другим экземпляром шаблона *Inheritance*.

Результаты

Класс *Fulfiller* зависит от класса *Abstracter*, наследуя интерфейс реализуемого метода. Это отношение должно быть выражено в синтаксисе, и существуют ситуации, в которых запутанный интерфейс на вершине иерархии классов может иметь отдаленное

влияние на подклассы. В таких случаях можно предпочесть шаблон *Delegation*. Решение, основанное на шаблоне *Delegation*, позволяет сделать внешний интерфейс независимым от объекта реализации. В качестве альтернативы, если используется только один класс или объект, можно выбрать шаблон *Conglomeration*, позволяющий выполнить задачу с помощью закрытых методов класса, сохраняя инкапсуляцию интерфейса и реализации.

Реализация

Каждый язык, допускающий наследование и абстрактные методы независимо от способа их выражения, поддерживает шаблон *Fulfill Method*. В следующих примерах использован шаблон *Abstract Interface*.

В языке C++

```

1  class Abstractor {
2      public:
3          virtual void operation() = 0;
4      };
5
6  class Fulfiller :
7  public Abstractor {
8      public:
9          void operation() {
10             // Выполняем соответствующие операции
11         };
12     };

```

В языке Java (метод включается в интерфейс или в абстрактный класс)

```

1  public interface Abstractor {
2      public void operation();
3  };
4
5  public abstract class AnotherAbstractor {
6      public abstract void operation2();
7      public void operation3();
8  };
9  public class Fulfiller
10 extends AnotherAbstractor
11 implements Abstractor {
12     public void operation() {
13         // Выполняем соответствующие операции
14     }
15     public void operation2() {
16         // Выполняем соответствующие операции
17     }
18 };
19

```

В языке Python 3.x

```

1  class Abstractor(metaclass=ABCMeta):
2      @abstractmethod
3      def operation(self, ...):
4          // Допускается реализация по умолчанию

```

```
5     return  
7 class Fulfiller(Abstractor):  
8     def operation(self, ...):  
9         // Выполняем соответствующие операции  
10        pass
```

Родственные шаблоны

Шаблон *Fulfill Method* состоит из двух элементарных шаблонов проектирования: *Abstract Interface* и *Inheritance*. Поскольку шаблон *Fulfill Method* выполняет предназначение шаблона *Abstract Interface*, он почти всегда обнаруживается вместе с шаблоном *Abstract Interface*. В тех случаях, когда шаблон *Fulfill Method* не сопровождается шаблоном *Abstract Interface*, возникающий абстрактный класс невозможно использовать в программе, пока подкласс не реализует соответствующий метод. Шаблон *Fulfill Method* еще встретится нам в этой главе при описании шаблона *Objectifier*. Шаблон *Fulfill Method* является неотъемлемой частью шаблонов Gang of Four, включая *Proxy* (Заместитель), *Command* (Команда), *Iterator* (Итератор), *Observer* (Наблюдатель), *State* (Состояние), *Decorator* (Декоратор), *Prototype* (Прототип), *Template Method* (Шаблонный метод) и многие другие [21].

Retrieve New

Управление объектами

Назначение

Используется в ситуациях, когда необходим новый экземпляр, но его создание является слишком сложной или затратной процедурой, которую необходимо инкапсулировать в другом объекте. Гарантируется, что возвращаемый объект не имеет никаких других ссылок на себя.

Мотивация

Часто вновь созданный объект должен быть совершенно независимым объектом с точно определенным отношением владения. Такой объект всегда можно создать локально, но это может быть нецелесообразно, если, например, создаваемый объект должен иметь сложное поведение. Например, решение о том, какой объект следует создать и на базе какого объекта это нужно сделать, можно принять на основе запроса к базе данных. Любой процесс создания объекта, являющийся частью сложной логики или трансакций, которые могут потребовать продолжительного времени, не заслуживает копирования или повторного использования. Мы хотим централизовать такую логику и инкапсулировать ее в собственный метод.

Разумеется, такая инкапсуляция логики порождает другие проблемы. Если на созданный объект существует много ссылок из других источников, а модель памяти является неуправляемой, как в языке C++, то любой фрагмент кода с такой ссылкой на объект может потребовать его уничтожения, делая остальные ссылки на уже несуществующий объект некорректными. Кроме того, ни один из ссылающихся объектов не может потребовать их уничтожения, вызывая утечку памяти, потому что такой объект становится недоступным.

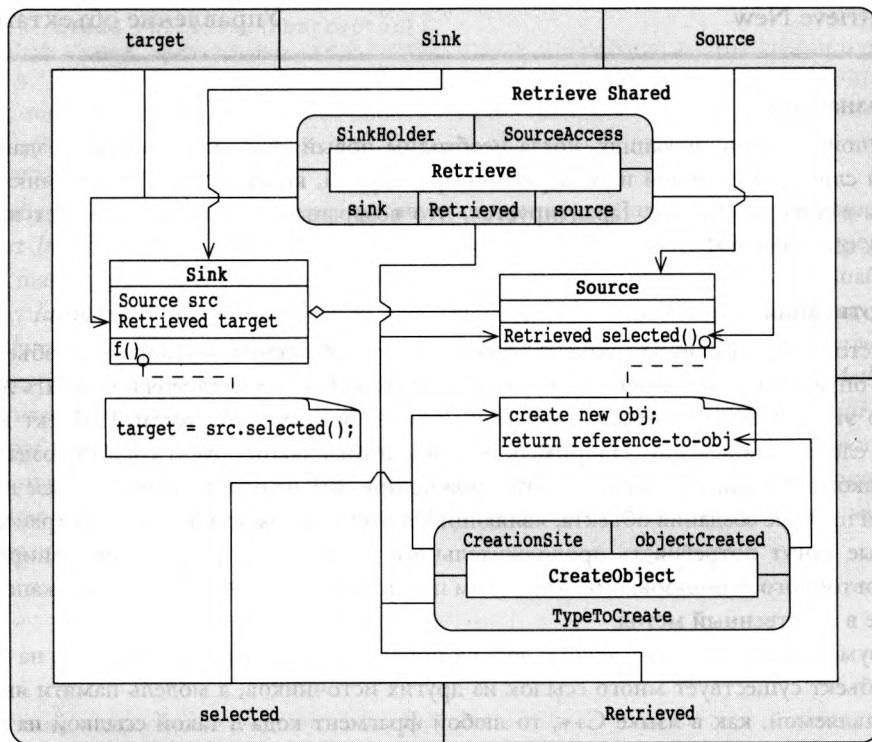
Решить эту проблему может механизм сбора мусора, но качественное управление памятью начинается с небольших решений. Одно из таких решений — гарантировать, что только получатель в шаблоне *Retrieve* имеет доступ к извлекаемому объекту, устанавливая ясное и точное отношение владения. Это можно сделать почти прозрачно, возвращая копию объекта, потому что копия создаст новый экземпляр. Другие схемы подразумевают возвращение локальной ссылки, которая уничтожается при выходе из метода, но этот способ уязвим для ошибок при дальнейшем сопровождении, если не установлено более строгое отношение владения. В любом случае это требует применения шаблона *Create Object*.

Применимость

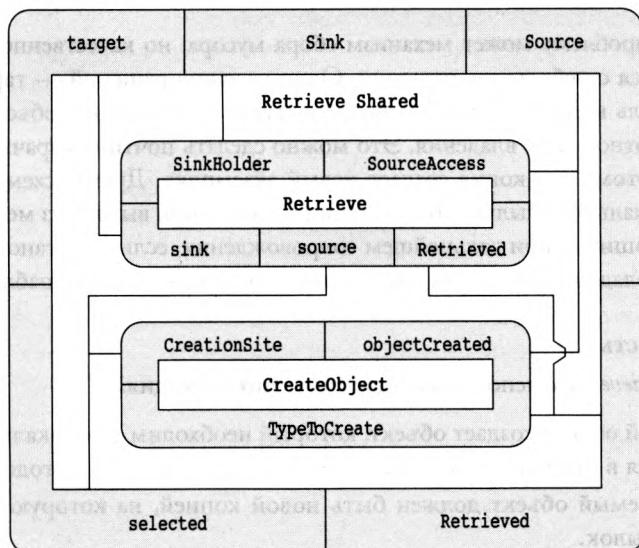
Шаблон *Retrieve New* используется в следующих ситуациях.

- Удаленный объект создает объект, который необходим для локальных вычислений и хранится в открытом поле или возвращается открытым методом.
- Возвращаемый объект должен быть новой копией, на которую не должно быть других ссылок.

Структура



Версия, использующая только компоненты PINbox (идентична шаблону *Retrieve Shared*).



Участники

Source

Тип объекта, который должен передавать управление извлекаемому объекту. Служит первичным источником, создающим объект.

Sink

Тип объекта (или класс), запрашивающего извлекаемый объект и включающий в себя поле `target`, принимающее новое значение.

Retrieved

Тип обновляемого и возвращаемого значений.

target

Поле, обновляемое извлеченным объектом для локального использования.

selected

Метод или поле, порождающее и возвращающее новое значение. В этом сценарии гарантируется, что на возвращаемый объект нет других ссылок.

Отношения

В этой схеме взаимодействуют только три объекта, которые являются источником запроса, исполнителем запроса и передаваемым объектом. Главные части шаблона *Retrieve New* проявляются в реализации метода извлечения `selected`. Этот метод должен гарантировать, что возвращаемый объект является новым и на него нет других ссылок.

Результаты

Разделение между владением и созданием объекта имеет несколько преимуществ. Например, тип созданного объекта можно определить гибко, возможно, с помощью полиморфизма, например шаблона *Abstract Factory*.

Однако это значит, что время жизни созданного объекта определяется исключительно получающим объектом. Целесообразно также передавать ссылки на созданный объект, если не используется механизм сбора мусора.

Реализация

В языке C++

```
1  class Retrieved {};
2
3  class Source {
4  public:
5      Retrieved giveMeAValue() {
6          Retrieved ret;
7          return ret;
8      };
9  };
10
11 class Sink {
12     Retrieved target;
13     Source srcobj;
14 public:
```

```
16     void operation() {
17         target = srcobj.giveMeAValue();
18     }
19 }
```

В языке Python

```
1  class Source:
2      def giveMeAValue(self):
3          ret = Retrieved()
4          return ret
5
6  class Sink:
7      def __init__(self):
8          srcobj = Source()
9          return self
10     def operation(self):
11         target = srcobj.giveMeAValue()
```

Родственные шаблоны

Шаблон *Retrieved New* тесно связан с шаблоном *Retrieve Shared*, и если посмотреть на диаграммы, использующие только компоненты PINbox, то они окажутся эквивалентными. Отличительной особенностью является поведенческий аспект, определяющий способ передачи запрашиваемого объекта вызывающему объекту: либо с помощью ясных отношений владения, как в шаблоне *Retrieved New*, либо с помощью ссылок, как в шаблоне *Retrieve Shared*.

Многие порождающие шаблоны из книги *Design Patterns* [21] тем или иным способом используют шаблон *Retrieve New*. Шаблон *Prototype* явно использует шаблон *Retrieve New*, гарантируя, что всегда будут возвращаться новые копии объекта, а шаблон *Builder* представляет собой яркий пример способности шаблона *Retrieve New* инкапсулировать сложную порождающую логику. Во многих случаях шаблон *Retrieve New* используется в шаблонах *Abstract Factory* и *Factory Method*.

Retrieve Shared

Управление объектами

Назначение

Предназначен для того, чтобы получать ссылку на совместно используемый объект без установки явного отношения владения этим объектом. В объектно-ориентированном программировании встречается повсюду.

Мотивация

Объекты — это концептуальные сущности. Любой конкретный экземпляр может инкапсулировать состояние или функциональные свойства, которые в то же время могут интересовать многие другие объекты. Например, рассмотрим очередь заданий на печать. Многие приложения хотят иметь доступ к этой очереди, и нет никаких причин для того, чтобы каждое приложение имело собственную очередь заданий на печать. Это неминуемо привело бы к борьбе за ресурсы между разными очередями. Вместо этого можно разрешить многим приложениям использовать очередь совместно. Запросы от разных объектов можно обрабатывать в порядке, который устанавливается самой очередью. Если очередь является общим ресурсом, она должна быть доступной для всех.

Шаблон *Retrieve Shared* использует шаблоны *Create Object* и *Retrieve*, как и шаблон *Retrieve New*, но отличается тем, что может выполнять кэширование или передавать ссылки на вновь созданные объекты для других целей. При этом не только нет никаких гарантий, что на вновь созданный объект больше нет никаких ссылок, но и, наоборот, явно предполагается, что такие ссылки существуют. Отношение владения между сущностями, содержащими такие ссылки, определены некорректно.

Шаблон *Retrieve Shared* является настолько широко распространенным, что многие языки прямо поддерживают его в той или иной степени. Например, в языке C++, есть конструкция `shared_ptr`, включенная в раздел 20.7.2.2 недавно принятого стандарта 2011 года [18].

В других языках и средах есть идиомы и библиотеки, поддерживающие полуавтоматическое управление объектами, на которые есть ссылки, например свойство `autorelease` в языке Objective-C. Во многих других языках для автоматического обнаружения и удаления объектов из системы во время выполнения программы используется механизм сбора мусора. К ним также относятся языки Java, C#, Python и большинство языков описания сценариев. Автоматическое управление памятью становится все более популярным и распространенным в языках программирования, но оно пока далеко не универсально, и все разработчики должны владеть основами управления памятью.

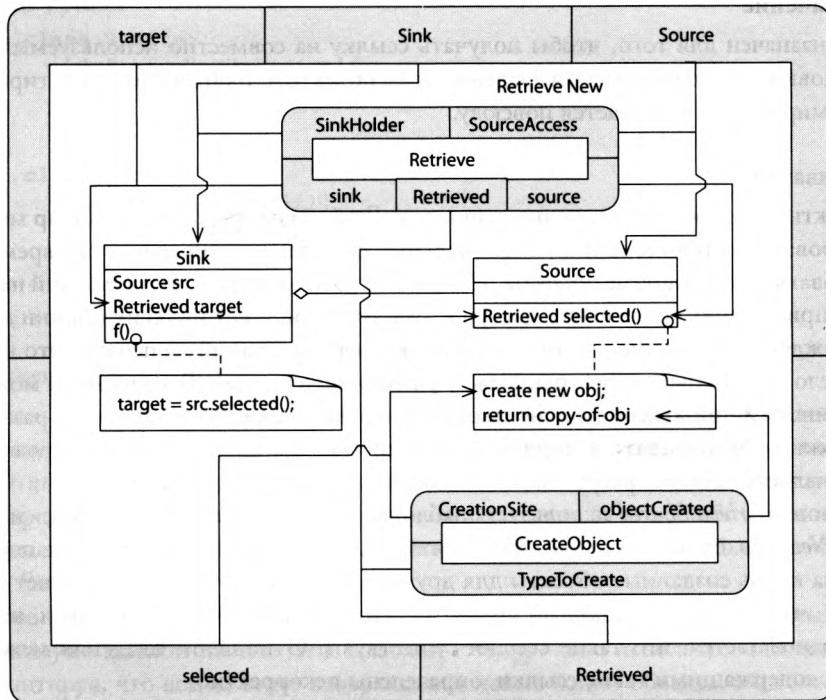
Применимость

Шаблон *Retrieve Shared* используется в следующих ситуациях.

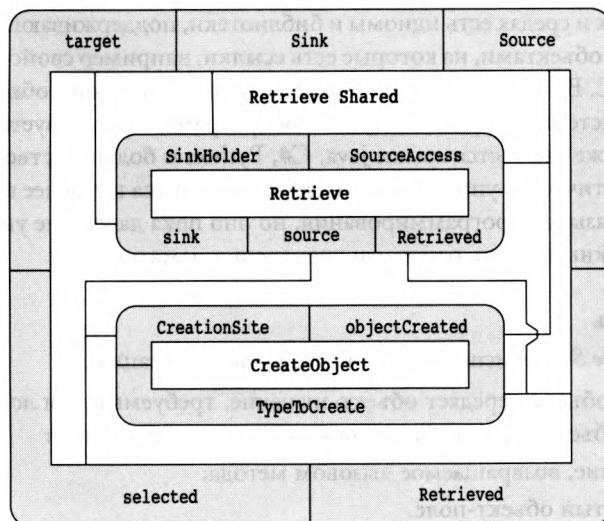
- Удаленный объект передает объект-значение, требуемый для локальных вычислений. Этот объект передается двумя возможными способами:
- через значение, возвращаемое вызовом метода;
- через открытый объект-поле.

- Объект должен использоваться другими объектами и не считаться закрытым ресурсом.

Структура



Версия, использующая только компоненты PINbox (идентична шаблону *Retrieve New*), такова.



Участники

Source

Объект, передающий требуемый объект. Служит первичным источником, создающим нужный объект.

Sink

Тип объекта (или класс), запрашивающий передаваемый объект и содержащий элемент *target*, который должен получить новое значение.

Retrieved

Тип обновляемого и возвращаемого значений.

target

Поле, обновляемое передаваемым объектом для локального использования.

selected

Метод или поле, создающее и возвращающее новое значение. В этом сценарии возвращаемый объект представляет собой ссылку на объект, причем нет гарантий, что она каким-либо образом ассоциирована с владением.

Отношения

В этой схеме участвуют только три объекта, играющие роль источника запроса, исполнителя запроса и передаваемого объекта. Сравните этот шаблон с шаблоном *Retrieve New*. Разница окажется очень незначительной, но последствия совершенно разные.

Результаты

В языках или средах, использующих механизм сбора мусора, передаваемый объект содержит счетчик ссылок, но не ограничивает создание новых ссылок. Генерирование новых ссылок может легко привести к утечке памяти, поэтому следует убедиться, что в проекте установлена строгая стратегия владения. Если передаваемый объект должен быть уникальным или быть главным владельцем вновь созданного общего ресурса, следует предпочесть шаблон *Retrieve New*.

Реализация

В языке Java

```
1  class Retrieved {  
2  };  
  
4  class Source {  
5      public Retrieved giveMeAValue() {  
6          Retrieved ret = new Retrieved();  
7              // Операции над объектом ret,  
8              // например кэширование  
9              return ret;  
10         };  
11     };  
12     class Sink {  
13         Retrieved target;
```

```
16     Source srcobj;
17     public void operation() {
18         target = srcobj.giveMeAValue();
19     }
20 }
```

Родственные шаблоны

Посмотрев на диаграммы, легко увидеть, что шаблоны *Retrieve Shared* и *Retrieve New* во многом похожи. Разница между ними заключается в уникальности владения возвращаемым объектом. Возможно, наиболее распространенным шаблоном, использующим шаблон *Retrieve Shared*, является *Singleton*.

Objectifier

Эта спецификация кратко описывает шаблон *Objectifier* (Объективатор), изобретенный Вальтером Циммером (Walter Zimmer). Она показывает, что этот шаблон представляет собой повторение одного шаблона, имеющего меньший размер. Полное описание и обсуждение этого шаблона можно найти в работе [43]. Пожалуйста, при необходимости обратитесь к этому базовому документу.

Назначение

Шаблон *Objectifier*, как следует из его названия, предназначен для “объективации одинакового поведения в дополнительных классах, чтобы клиенты могли изменять это поведение независимо от другого” [43]. Обратите внимание на фразу “одинакового поведения в дополнительных классах”. На первый взгляд, это похоже на шаблон *Redirection*, но выражение “в дополнительных классах” означает, что мы имеем дело с наследованием. Как известно из описания шаблона *Fill Method*, мы можем реализовать абстрактный метод в подклассе при условии, что абстрактный и конкретный методы имеют одинаковое предназначение. Многочисленные конкретные подклассы могут реализовывать поведение немного по-разному, но общее предназначение всех подклассов должно оставаться одинаковым.

Мотивация

Мы считаем, что раздел “Мотивация” в работе Циммера, был правильным. “При проектировании часто необходимо отделить абстракцию от реализации, и обеспечить взаимозаменяемость реализаций” [43].

Циммер описывает один суперкласс из шаблона *Abstract Interface* и несколько подклассов, обеспечивающих реализацию. Это значит, что в этой схеме есть несколько экземпляров шаблона *Fill Method*, но все они имеют общий суперкласс, играющий роль *Abstractor*. Такую схему можно изобразить в виде стека компонентов PINbox, как показано в разделе 3.2.4.

Циммер добавил один критически важный компонент в описанную выше коллекцию реализаций подкласса. Он указал, что “для объективации изменяющегося поведения... существуют независимые объекты, осуществляющие реализации, которые могут заменять одна другую во время выполнения программы” [43]. Это значит, что разработчик должен создать ссылку на объект класса *Abstractor*, а затем заменить его реальным экземпляром подкласса при необходимости. Все вызовы, обращенные к этому экземпляру, теперь неявно и автоматически пересыпаются правильной реализации с помощью полиморфизма. Способ вызова не имеет большого значения; важно, чтобы клиентский объект мог его осуществить.

Применимость

Поскольку добавить к описанию Циммера практически нечего, я приведу его раздел “Применимость” полностью.

Шаблон *Objectifier* используется в следующих ситуациях.

- Поведение необходимо разделить среди независимых объектов, которые можно заменять, сохранять, совместно использовать или вызывать.
- Конфигурация поведения задается во время выполнения программы.
- Существует несколько практически одинаковых классов, отличающихся один от другого небольшим количеством методов. Объективация разного поведения в дополнительных классах позволяет унифицировать классы в одном общем классе, который можно конфигурировать с помощью ссылки на дополнительные классы.
- Существует много условных операций, выбирающих поведение [43].

Решая, применять ли шаблон *Objectifier*, следует учитывать два обстоятельства. Во-первых, требуется конфигурация поведения во время выполнения программы; во-вторых, существует много условных операций, выбирающих поведение.

Каждый раз, когда вы попадаете в ситуацию, напоминающую листинг 6.1, следует подумать о шаблоне *Objectifier*. Если вы видите много условных конструкций с одним и тем же триггером, это явный признак того, что вы можете погрузить условные блоки в классы с общими суперклассом, предоставляемым интерфейсом. Затем условные блоки заменяются одним вызовом требуемого метода, как показано в листинге 6.2. Это нам уже знакомо: этот пример напоминает спецификацию шаблона *Abstract Interface* в главе 5.

Обратите внимание на то, что внешний интерфейс к методу `feedCritterInEnviron()` не изменился, но код стал намного проще. Для добавления нового вида животных требуется определить подкласс класса `Animal` и изменить только *одно* место в коде, в котором происходит выбор вида животного в методе `feedCritterInEnviron()`. Ранее добавление нового вида животных потребовало бы добавления новых условных блоков по всей программе, где происходит выбор вида.

Листинг 6.1. Условные конструкции для выбора поведения

```

1  typedef enum{FISH, HORSE, CHEETAH} CritterKind;
3  CritterKind critter;

5  void eatFood(CritterKind critter, FoodItem f) {
6      if (critter == FISH) {
7          swimToFood(critter, locationOfFood(f));
8          ingest(critter, f);
9          digest(critter, f);
10     } else if (critter == HORSE) {
11         checkHerdSafety(critter);
12         ingest(critter, f);
13         digest(critter, f);
14     } else if (critter == CHEETAH) {
15         turboMode(critter, locationOfFood(f));

```

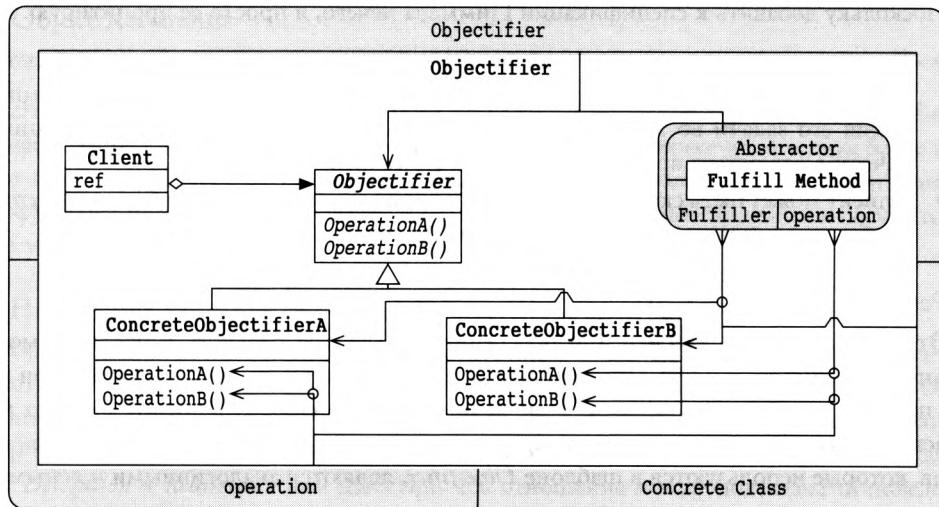
```
17         if (caughtPrey(f)) {
18             ingest(critter, f);
19             digest(critter, f);
20         }
21     } else {
22         ingest(critter, f);
23         digest(critter, f);
24     }
25 };
26
27 void ingest(CritterKind critter, FoodItem f) {
28     if (critter == FISH) {
29         ...
30     } else if (critter == HORSE) {
31         ...
32     } else if (critter == CHEETAH) {
33         ...
34     } else {
35         ...
36     };
37 }
38
39 typedef enum {OCEAN, PLAINS, SAVANNAH} Environment;
40
41 void feedCritterInEnvirons(Environment env) {
42     if (Environment == OCEAN) {
43         if (isHungry(FISH)) {
44             FoodItem f = findFood(FISH);
45             if (f) {
46                 eatFood(FISH, f);
47             }
48         }
49     } else if (Environment == PLAINS) {
50         if (isHungry(HORSE)) {
51             FoodItem f = findFood(HORSE);
52             if (f) {
53                 eatFood(HORSE, f);
54             }
55         }
56     } else if (Environment == SAVANNAH) {
57         if (isHungry(CHEETAH)) {
58             FoodItem f = findFood(CHEETAH);
59             if (f) {
60                 eatFood(CHEETAH, f);
61             }
62         }
63     }
64 }
```

Листинг 6.2. Использование шаблона *Objectifier* для выбора поведения

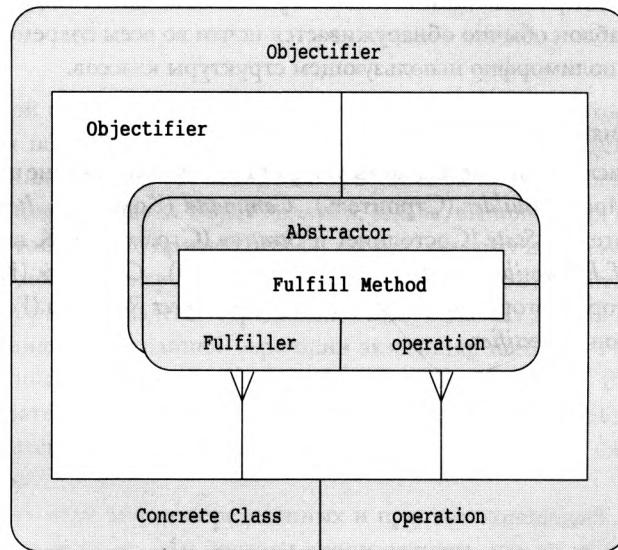
```
1 class Animal {
2     public:
3         void eatFood(FoodItem f) {
```

```
5         this->ingest(f);
6         this->digest(f);
7     };
8     void ingest(FoodItem f);
9 };
10
11 class Fish : public Animal {
12 public:
13     void eatFood(FoodItem f) {
14         this->swimToFood(locationOfFood(f));
15         this->ingest(f);
16         this->digest(f);
17     };
18     void ingest(FoodItem f);
19 };
20
21 class Horse : public Animal {
22 public:
23     void eatFood(FoodItem f) {
24         checkHerdSafety(critter);
25         ingest(critter, f);
26         digest(critter, f);
27     };
28     void ingest(FoodItem f);
29 };
30
31 class Cheetah : public Animal {
32 public:
33     void eatFood(FoodItem f) {
34         turboMode(critter, locationOfFood(f));
35         if (caughtPrey(f)) {
36             ingest(critter, f);
37             digest(critter, f);
38         }
39     };
40 };
41
42 typedef enum {OCEAN, PLAINS, SAVANNAH} Environment;
43
44 void feedCritterInEnvirons(Environment env) {
45     Animal* critter;
46     if (Environment == OCEAN) {
47         critter = new Fish();
48     } else if (Environment == PLAINS) {
49         critter = new Horse();
50     } else if (Environment == SAVANNAH) {
51         critter = new Cheetah();
52     }
53     if (isHungry(critter)) {
54         FoodItem f = findFood(critter);
55         if (f) {
56             eatFood(critter, f);
57         }
58     }
59 };
```

Структура



Версия, использующая исключительно компоненты PINbox.



Участники

Кроме участников, указанных Циммером, можно добавить следующих.

operation

Абстрагируемый метод, вызываемый клиентом.

Concrete Class

Один из подклассов, обеспечивающих реализацию.

Отношения

Поскольку добавить к спецификации Циммера нечего, я просто ее процитирую.

- Клиент может использовать шаблон *Objectifier* для делегирования частей своего поведения. Шаблон *Objectifier* получает информацию, необходимую для выполнения его задачи во время инициализации или клиент передает информацию в качестве параметра при вызове шаблона *Objectifier*.
- Клиент может быть сконфигурирован с помощью конкретного шаблона *Objectifier*, чтобы адаптировать поведение для выполнения задания [43].

Результаты

Этот шаблон очень полезен. Он обеспечивает высокую гибкость и расширяемость большинства систем, одновременно повышая понятность и ясность кода. Однако он может приводить к недоразумениям, если цель поведения сформулирована нечетко или замаскирована другим поведением в иерархии классов. Следует убедиться, что цели методов, которые используются в шаблоне *Objectifier*, являются аналогичными и ясными.

Реализация

Любой язык, поддерживающий шаблон *Fill Method*, может поддерживать шаблон *Objectifier*. Этот шаблон обычно обнаруживается почти во всем современном программном обеспечении, полиморфно использующем структуры классов.

Родственные шаблоны

Циммер перечисляет многие шаблоны Gang of Four, использующие шаблон *Objectifier*, включая *Bridge* (Мост), *Builder* (Строитель), *Command* (Команда), *Iterator* (Итератор), *Observer* (Наблюдатель), *State* (Состояние) и *Strategy* (Стратегия). К ним можно также добавить *Chain of Responsibility* (Цепочка обязанностей), *Composite* (Компоновщик) и *Decorator* (Декоратор), которые используют шаблон *Object Recursion* (Рекурсия объекта), содержащий шаблон *Objectifier*.

Object Recursion

Эта спецификация кратко описывает шаблон *Object Recursion*, изобретенный Бобби Вулфом (Bobby Woolf), и демонстрирует, что он является пересечением двух более простых объектов. Полное определение и обсуждение этого шаблона можно найти в работе Вулфа “The Object Recursion Pattern” [41]. Пожалуйста, при необходимости обратитесь к этому базовому документу.

Назначение

Бобби Вулф так формулирует предназначение этого шаблона: “Распределить обработку запроса по структуре, полиморфно используя делегирование. Шаблон *Object Recursion* позволяет разделить запрос на более мелкие части, которые легче обработать” [41]. В этом определении нет информации о более мелких частях. Из обсуждения шаблонов *Delegation* и *Redirection* нам известно, что отношение между подзадачами поведения могут существенно влиять на способ реализации системы. Прочитав всю спецификацию, мы можем добавить, что эти “более мелкие части” имеют близкое предназначение и поведение.

Мотивация

В оригинальной спецификации Вулф использовал пример, в котором сравниваются два объекта. Они получали команду сравнить соответствующие члены, и эта команда рекурсивно выполнялась в дереве объектов. Это удачный пример, иллюстрирующий необходимый компонент шаблона *Object Recursion*: задача, выполняемая на каждом уровне, является одной и той же. Это пример переадресации, поэтому при декомпозиции шаблона *Object Recurion* можно обнаружить какую-то форму шаблона *Redirection*.

Слово *полиморфизм* в разделе “Предназначение” означает две вещи. Во-первых, это значит, что мы увидим по крайней мере один экземпляр шаблона *Fulfill Method* и элементарные шаблоны проектирования *Inheritance* и *Abstract Method*. Во-вторых, в схеме практически обязательно будет больше двух экземпляров этого шаблона, т.е. по одному на каждый потенциальный подкласс. Следовательно, можно ожидать появления шаблона *Objectifier*.

Итак, у нас есть пара точно определенных и простых концепций, позволяющих попытать шаблон *Object Recursion*. Мы должны просто увидеть, как их можно уточнить и использовать для формирования более сложного поведения и концепции.

Применимость

В разделе “Применимость” оригинальной спецификации шаблона *Object Recursion* в работе Вулфа указано, что его следует использовать в следующих ситуациях.

- При передаче сообщения через связанную структуру при условии, что конечный пункт назначения неизвестен.

- При распространении сообщения по всем узлам в части связанной структуры.
- При распределении обязанностей по связанной структуре [41].

Следует подчеркнуть, что, ограничившись *лишь* этими условиями, мы можем решить задачу, не прибегая к шаблону Вулфа. В частности, здесь ничего не сказано ни о схожести поведения, распределяемого по структуре, ни об уровнях доверия, которые объект может присваивать другим объектам при реализации этого поведения. Описанные выше задачи можно решить с помощью цепочки разнородных объектов, используя шаблон *Redirection* или, при известной доле фантазии, шаблон *Delegation*.

Из разделов “Предназначение” и “Мотивация” можно вывести еще несколько ключевых характеристик шаблона *Object Recursion*. Например, известно, что в шаблоне существует полиморфизм и “связанная структура”, описанная выше, имеет несколько очень специальных ограничений, касающихся в основном того, что связанная структура состоит из объектов, типы которых имеют общий предок (это обусловлено полиморфизмом, упомянутым в разделе “Предназначение”). Это предположение подкрепляется остальной частью оригинальной спецификации.

Существование этого общего предка означает, что в шаблоне есть отношение подтипа из шаблона *Redirection*. Иначе говоря, вместо обобщенного шаблона *Redirection* мы получим шаблон *Trusted Redirection*.

Используя эту информацию, применимость шаблона *Object Recursion* можно уточнить следующим образом.

- В системе существует связанная структура, например дерево, и элементы этой структуры имеют общие предназначение, цель и поведение.
- Необходимо передать сообщение по связанной структуре, чтобы выполнить одно из общих действий; при этом содержание сообщения должно оставаться практически неизменным.
- Конечный пункт назначения передаваемого сообщения в связанной структуре неизвестен.
- Может возникнуть необходимость распространить сообщение по всем узлам отдельной части связанной структуры.
- Ответственность за реализацию поведения, требуемого в сообщении, распределяется по всей связанной структуре. Теперь становится ясно, что связанная структура состоит не из случайных элементов, а каждый узел структуры близок к остальным по своей природе и предназначению. Кроме того, теперь в спецификации открыто сказано, что в системе можно выполнить общее действие, что поведение можно распределить между объектами в структуре и/или что это поведение можно применить *ко всей* структуре. Все эти пункты важны для формирования окончательного вида системы.

Структура

Указанные выводы можно использовать при изучении оригинальной диаграммы UML, построенной Вулфом и дополненной компонентами PINbox. Как мы и предполагали, в диаграмме существуют экземпляры шаблонов *Objectifier* и *Trusted Redirection*.

Участники

В оригинальной спецификации Вулфа указаны четыре участника: *Initiator*, *Handler*, *RecurSOR* и *Terminator*. К ним можно добавить еще двух участников — *handleRequest* и *successor*, — показанных на диаграмме PIN. Эти участники выполняют следующие функции

Initiator

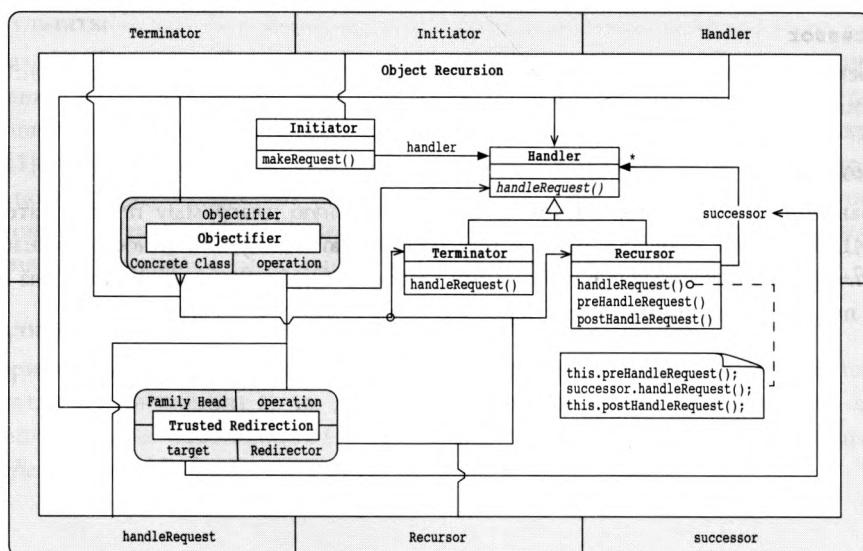
- Инициирует запрос.
- Обычно не является подтиром типа *Handler*; *makeRequest()* — это отдельное сообщение от метода *handleRequest()*.

Handler

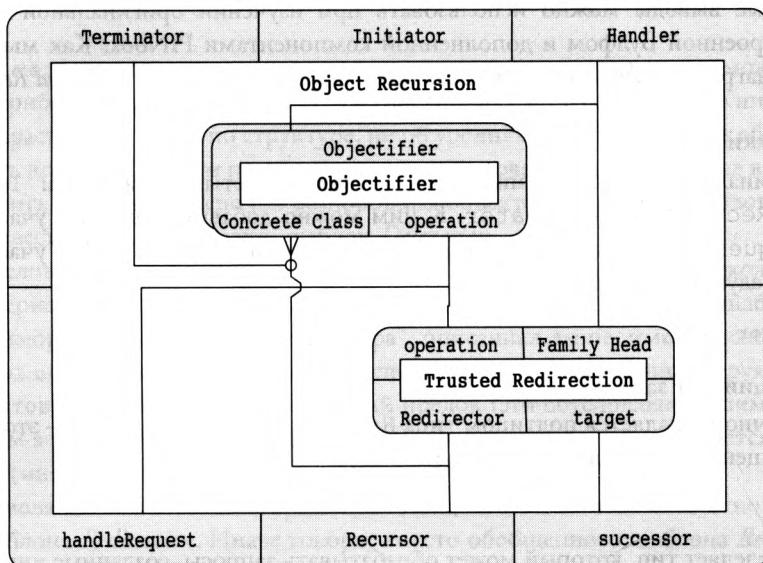
- Определяет тип, который может обрабатывать запросы, созданные инициатором.

RecurSOR

- Определяет связь *successor*.
- Обрабатывает запрос, делегируя его наследникам.
- Наследники, обрабатывающие запрос, могут зависеть от его содержания.
- Может выполнять дополнительные действия до или после делегирования запроса.
- Может играть роль терминатора для другого запроса.



Эту диаграмму можно изобразить исключительно с помощью компонентов PINbox.



Terminator

- Завершает запрос, полностью его выполняя и не передавая другим реализациям.
- Может играть роль рекурсора для другого запроса [41].

handleRequest

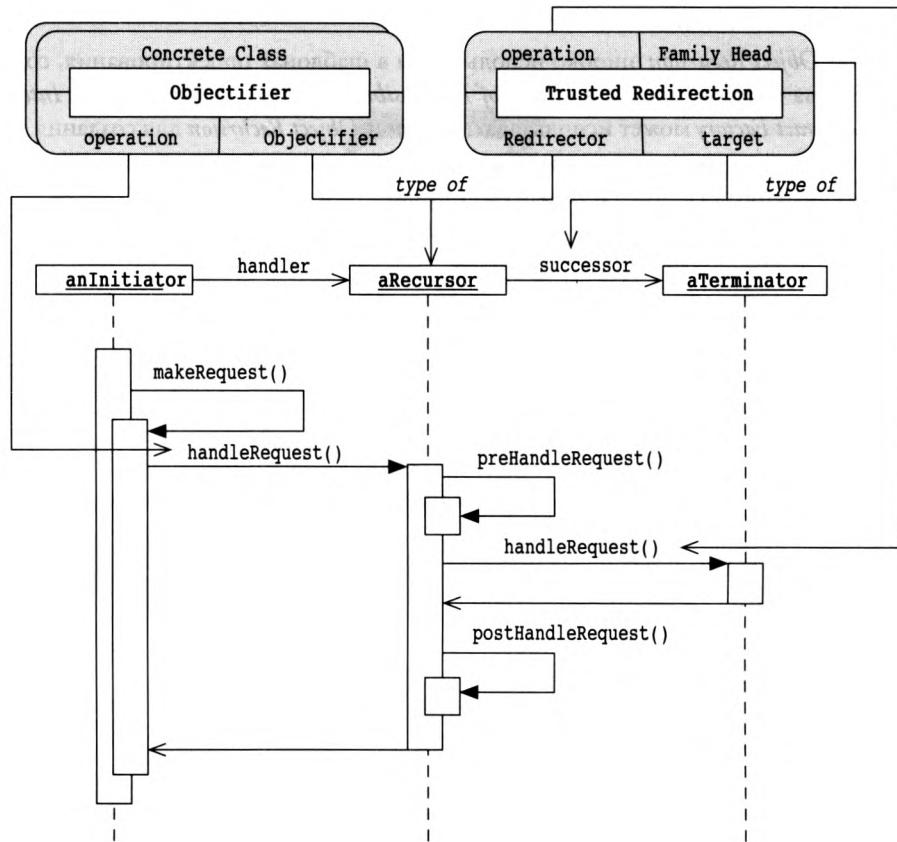
Метод, выполняющий требуемое действие на каждом этапе и при необходимости передающий его наследнику.

successor

Объект, получающий сообщение **handleRequest** от экземпляра класса **RecurSOR**; имеет тип **Handler**.

Отношения

Добавление компонентов PINboxes в оригинальную диаграмму последовательностей UML позволяет провести разделительную линию между обязанностями шаблонов *Objectifier* и *Trusted Redirection* в реализации шаблона *Object Recursion*. Каждый из них занимает половину диаграммы. Они пересекаются в объекте *aRecurSOR*.



Результаты

В разделе “Результаты” в работе Вулфа указан один недостаток: “Сложность программирования. Рекурсия, процедурная или объектно-ориентированная, является сложной концепцией. Излишнее ее использование может усложнить анализ и поддержку системы” [41]. Эту ситуацию можно разъяснить, разделив главные элементы рекурсии две части: полиморфную структуру (шаблон *Objectifier*) и рекурсивный вызов в этой полиморфной структуре (шаблон *Trusted Redirection*). Теперь эти два аспекта распределенного рекурсивного метода показаны явно, что делает систему более понятной.

Реализация

В оригинальной специализации шаблона *Object Recursion* утверждается: “Сообщение *Initiator.makeRequest()* не должно быть полиморфным относительно сообщения *Recursor.handleRequest()*” [41]. Выше, при описании экземпляров шаблонов *Objectifier* и *Trusted Redirection*, это различие было показано явно.

Родственные шаблоны

Шаблон *Object Recursion* широко используется в шаблонах проектирования, содержащих семейства классов, включая *Chain of Responsibility*, *Decorator*, *Composite* и *Interpreter*. Шаблон *Abstract Factory* может использовать шаблон *Object Recursion* для создания многослойных объектов, в которых каждый компонент содержит обратный вызов, передаваемый при необходимости через интерфейс фабрики.

Композиции шаблонов Gang of Four

И так, мы увидели весь каталог элементарных шаблонов вызова методов и несколько примеров шаблонов, которые представляют собой их комбинацию. В этой главе мы перейдем на следующий уровень и покажем, как, комбинируя шаблоны из глав 2 и 6, можно создать некоторые из наиболее популярных шаблонов проектирования Gang of Four (GoF).

Здесь рассматриваются шесть шаблонов GoF — два порождающих, два структурных и два поведенческих. Они образуют основу для работы с остальными шаблонами. Почему только шесть? Несмотря на то что каждый шаблон проектирования можно описать в терминах элементарных шаблонов, не все шаблоны проектирования можно адекватно представить только с помощью шаблонов EDP, упомянутых *в этой книге*. Напомним, что здесь мы описываем лишь элементарные шаблоны вызова методов, а вызов метода — это лишь одна из четырех разновидностей отношений между сущностями в объектно-ориентированном программировании. Вернитесь к главе 2, просмотрите табл. 2.3 в разделе 2.2.2 и освежите свои знания. Эти шесть шаблонов выбраны потому, что они тесно связаны с остальными шаблонами в своей группе, и эта связь, на первый взгляд, не очевидна и проявляется лишь после разбора их базовой концептуальной структуры.

В этой главе мы не придерживаемся формата спецификаций шаблонов, как в предыдущих двух главах. Эти шаблоны проектирования подробно описаны в многочисленных

книгах, и нет никакой необходимости дублировать их спецификации еще раз. Разумеется, для иллюстрирования релевантных отношения и связей используются диаграммы PIN и UML.

7.1. Порождающие шаблоны

Порождающие шаблоны описывают процессы создания объектов и управления ими. Как и следовало ожидать, в их определениях часто используются элементарные шаблоны *Create Object* и *Retrieve*. Способы связей этих двух элементарных шаблонов один с другим и с другими шаблонами порождают разнообразные различия между порождающими шаблонами. Мы проведем анализ и сравнение шаблонов *Abstract Factory* и *Factory Method*. Они демонстрируют, как небольшие различия в композициях могут привести к большим различиям в реализациях и целях.

7.1.1. Шаблон *Abstract Factory*

Первый порождающий шаблон, *Abstract Factory*, — идеальный пример того, как простые концепции, объединенные сложным образом, могут создать новую интересную концепцию. Для понимания семантики шаблона *Abstract Factory* необходимы только три элементарных шаблона: *Fulfill Method*, *Retrieve* и *Create Object*. Способы их соединения порождают богатые функциональные возможности.

Шаблон *Abstract Factory* “обеспечивает интерфейс для создания семейства связанных или зависимых объектов без указания их конкретных классов” [21]. Из этого утверждения можно сделать вывод, что для создания такого интерфейса потребуется шаблон *Inheritance*, а для реализации релевантных методов — шаблон *Fulfill Method*.

На рис. 7.1 показана структурная диаграмма шаблона *Abstract Factory*, упакованная в раскрытий компонент PINbox. Каждая роль, указанная на полях компонента PINbox, имеет по крайней мере одно внутреннее соединение с элементами, образующими шаблон *Abstract Factory*: метод `createProductA()`, реализованный в классе `ConcreteFactory2`, используется для создания экземпляра конкретного класса `ProductA2`, являющегося подклассом класса `AbstractProductA`. Аналогичные соединения существуют между остальными тремя классами продукции.

Это довольно сложная диаграмма, и по ней трудно понять, как работают отношения между классами. Отдельное внутреннее соединение более четко показано на рис. 7.2, на котором пропущены элементы структурной диаграммы, не связанные непосредственно с отдельным экземпляром шаблона *Abstract Factory*. Рис. 7.3 еще больше упрощает шаблон, игнорируя внутренние связи в компоненте PINbox, и в итоге мы получаем рис. 7.4, иллюстрирующий простую концептуальную структуру шаблона *Abstract Factory*.

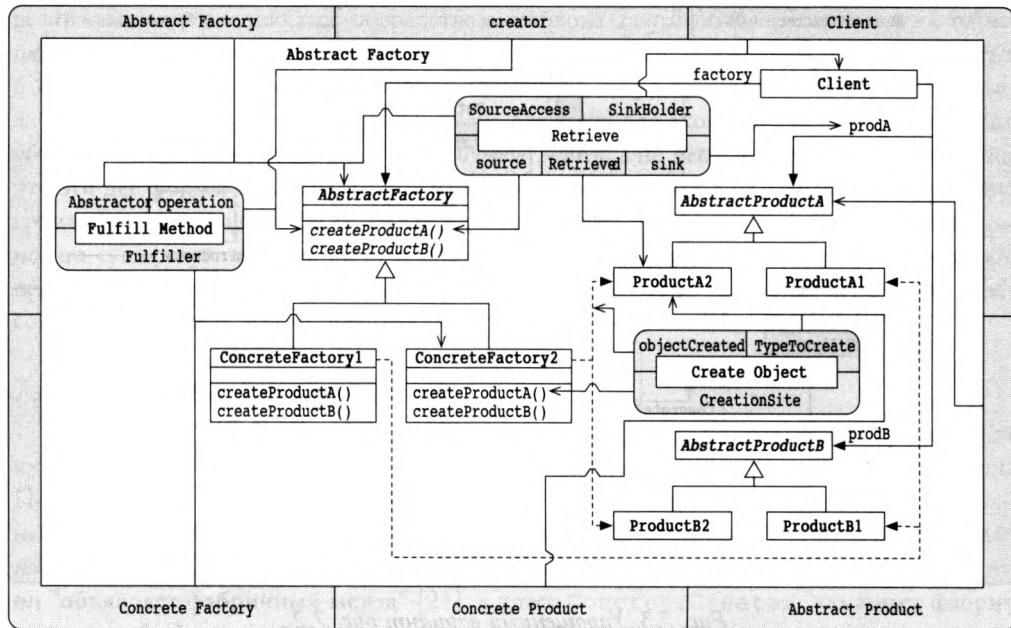


Рис. 7.1. Шаблон проектирования Abstract Factory в виде раскрытия компонента PINbox

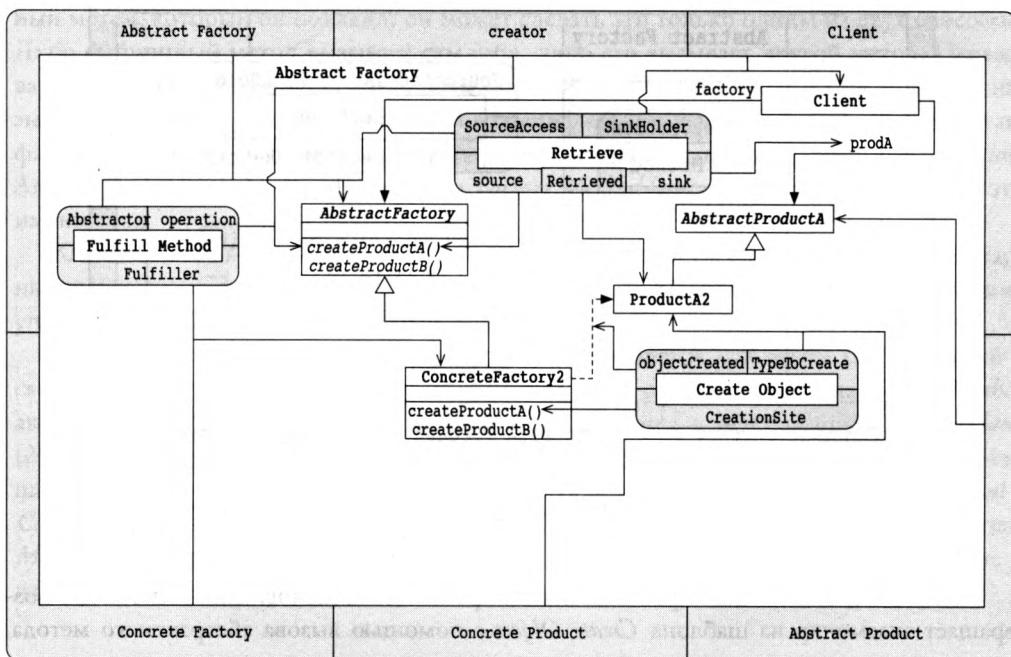


Рис. 7.2. Сокращение диаграммы до одного экземпляра шаблона Abstract Factory

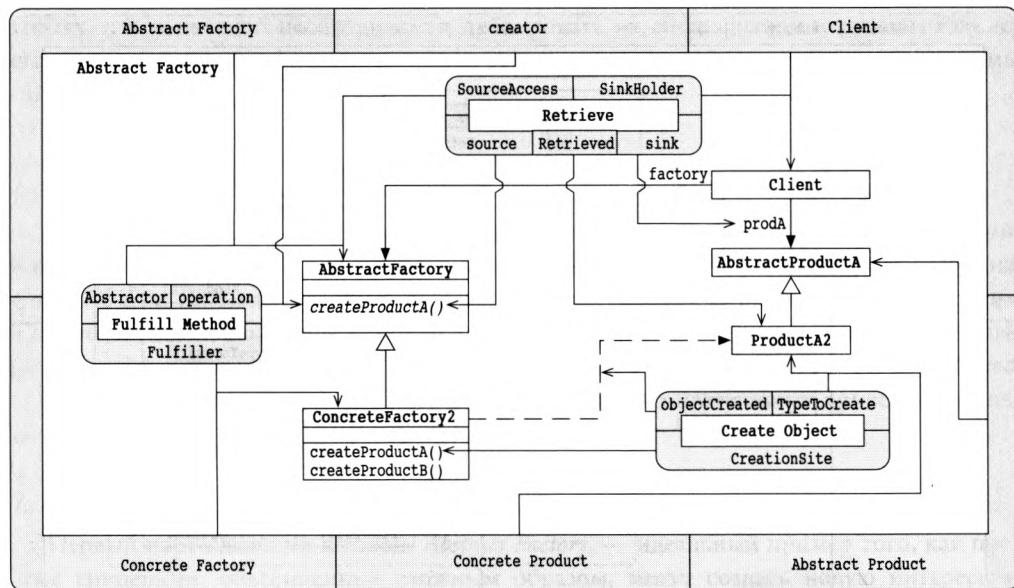


Рис. 7.3. Упрощенный вариант рис. 7.2

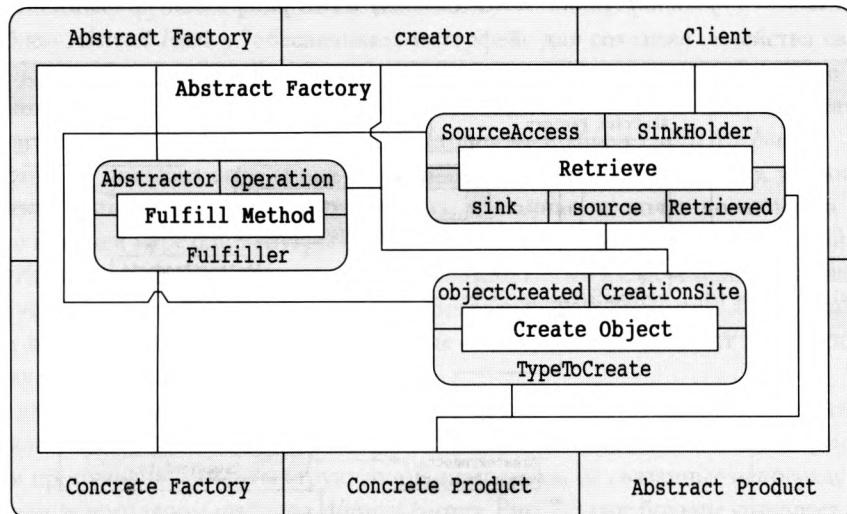


Рис. 7.4. Шаблон Abstract Factory, изображенный с помощью исключительно системы PIN

Как видим, в этой схеме работает очень простое соединение. Шаблон *Retrieve* возвращает экземпляр из шаблона *Create Object* с помощью вызова абстрактного метода,

реализованного в подклассе посредством шаблона *Fulfill Method*. Несмотря на то что диаграмма выглядит так, как будто мы просто свернули экземпляры шаблонов *Retrieve* и *Create Object* в шаблон *Retrieve New*, в некоторых ситуациях шаблон *Abstract Factory* может использовать другой шаблон (скажем, *Flyweight*) для экономии памяти при возврате объекта, тем самым реализуя шаблон *Retrieve Shared*, а не *Retrieve New*. Напомним также, что эти две формы шаблона *Retrieve* имеют идентичные PIN-диаграммы, так что по диаграммам их различить нельзя. Поскольку предвидеть такие ситуации заранее невозможно, но существует возможность переложить эту проблему на плечи разработчика или программиста, мы решили оставить эти два элементарных шаблона видимыми, чтобы сохранить гибкость и не усложнять проблему.

7.1.2. Шаблон *Factory Method*

Этот шаблон перепоручает создание реального объекта подклассу, обеспечивая четкий интерфейс и базовый метод для создания объекта и его внутреннего использования. Поскольку это порождающий объект, следует ожидать, что в нем используются шаблоны *Create Object* и *Retrieve*. Способ декомпозиции шаблона *Factory Method* подсказывает нам список его участников. Например, в описании класса *Creator* утверждается, что он “объявляет фабричный метод” [21], а класс *ConcreteCreator* “замещает фабричный метод”. Это яркий пример шаблона *Fulfill Method*. В описании также говорится, что класс *Creator* “может вызывать фабричный метод”. Здесь есть некий подтекст, но, если немного подумать, его можно понять. Если класс *Creator* вызывает фабричный метод, который он объявил, он может сделать это только одним из двух способов. Либо фабричный метод вызывает сам себя, либо его вызывает другой метод. Первый вариант соответствует шаблону *Recursion*, и довольно важно, что его можно прямо называть по имени в разделе отношений в спецификации шаблона *Factory Method*. Итак, фабричный метод должен вызываться другим методом, как в шаблоне *Conglomeration*. Анализ диаграммы UML для шаблона *Factory Method* (рис. 7.5) подтверждает, что это именно так.

Как показано на рис. 7.5, шаблон *Factory Method* сплетает в одну иерархию элементарные шаблоны *Create Object*, *Retrieve*, *Fulfill Method* и *Conglomeration*. Эту структуру можно упростить, представив в виде PIN-диаграммы (рис. 7.6).

Шаблон *Factory Method* — интересен тем, что является шаблоном GoF, который в своем определении использует другой шаблон GoF. В описании шаблона *Factory Method* авторы пишут: “Фабричные методы обычно вызываются в методах шаблона *Template*” [21]. Теперь мы точно понимаем, как они взаимодействуют. На рис. 7.19 и в обсуждении шаблона *Template Method* показано, как заменить экземпляры шаблонов *Fulfill Method* и *Conglomeration* эквивалентным экземпляром шаблона *Template Method*. Шаблон *Factory Method* — это расширение шаблона *Template Method*, в который добавлено создание и извлечение объекта.

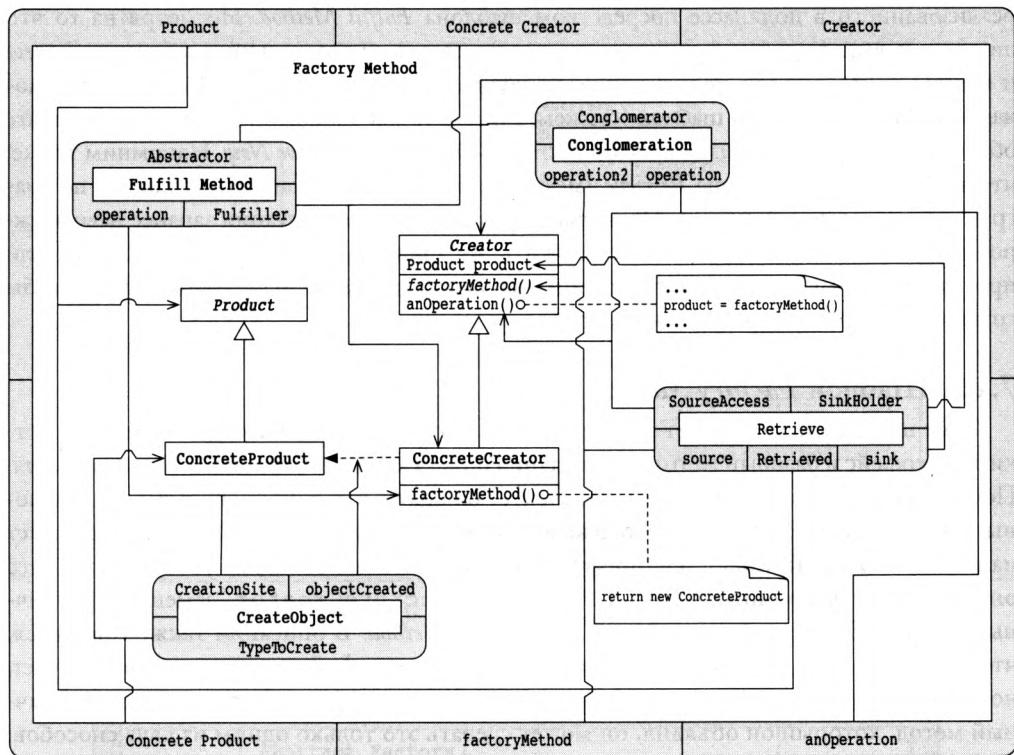


Рис. 7.5. Шаблон Factory Method в виде раскрытия компонента PINbox

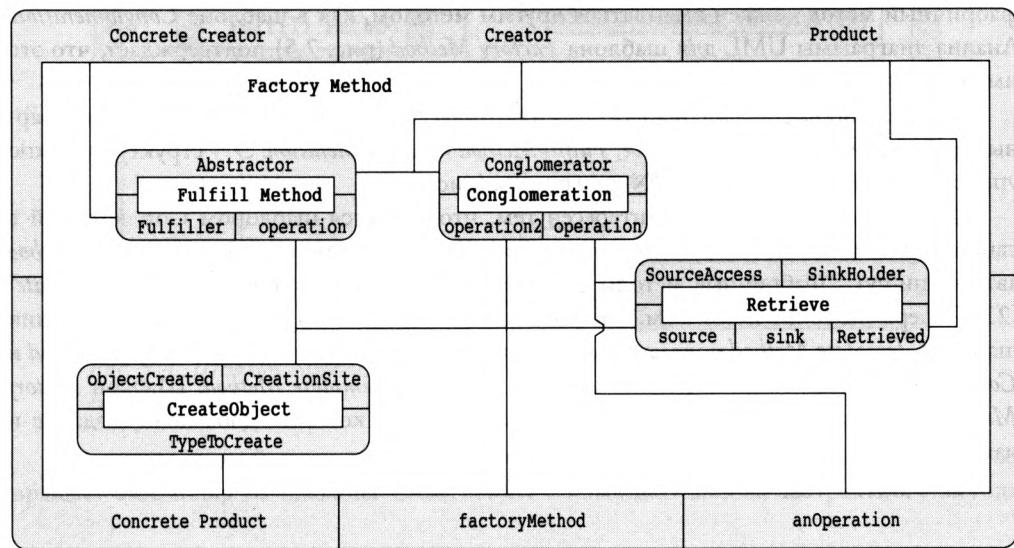


Рис. 7.6. Шаблон Factory Method, изображенный с помощью исключительно системы PIN

7.2. Структурные шаблоны

Из группы структурных шаблонов мы выбрали шаблоны *Decorator* и *Proxy*. Несмотря на различия между их целями и структурами, как ни странно, они имеют нечто общее.

7.2.1. Шаблон *Decorator*

Детально шаблон *Decorator* рассматривается в разделе 4.2. Здесь мы опишем контекст, в котором он используется. Рис. 7.7 содержит в себе рис. 4.17 из раздела 4.2 в виде компонента PINbox.

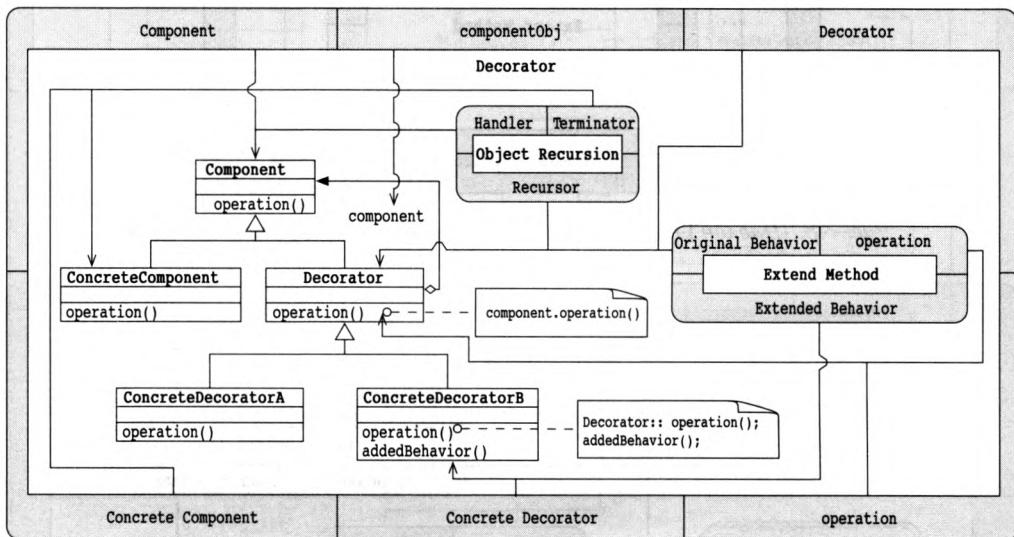


Рис. 7.7. Шаблон *Decorator*, изображенный в виде компонента PINbox

Упрощая рис. 7.7 до свернутой PIN-диаграммы, показанной на рис. 4.18, мы получим простую диаграмму, представленную на рис. 7.8. Теперь все базовые концепции сформулированы четко, а их связи друг с другом очевидны. На этом простом и ясном рисунке показано, что внутренняя структура шаблона *Decorator* состоит из шаблонов *Object Recursion* и *Extend Method*. Однако для целей нашей дискуссии мы продолжим их анализ. Мы можем представить информацию, показанную на рис. 4.22, в усеченном виде, как на рис. 7.9. Здесь мы отбросили компоненты PINbox, соответствующие шаблонам *Object Recursion* и *Objectifier*, и прямо показали экземпляры шаблонов *Fulfill Method* и *Trusted Redirection*.

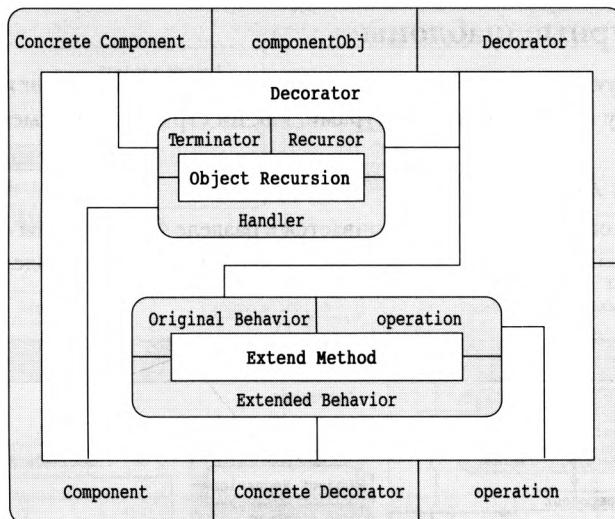


Рис. 7.8. Шаблон Decorator, изображенный с помощью системы PIN

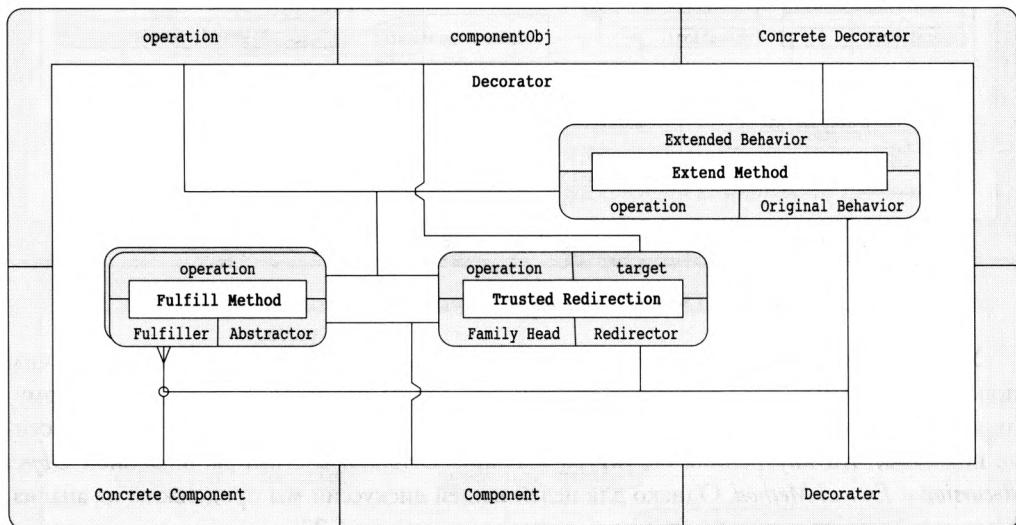


Рис. 7.9. Шаблон Decorator, раскрытый на три уровня в глубину

7.2.2. Шаблон *Proxy*

Шаблон *Proxy* ограничивает доступ к объекту или структуре, состоящей из объектов, скрывая детали от клиентов. Это настолько простой шаблон, что его анализ можно сделать, просто взглянув на структурную диаграмму (рис. 7.10). Легко видеть, что на рисунке изображен экземпляр шаблона *Fulfill Method*, а вызов метода *RealSubject::request()* из метода *Proxy::request()* представляет собой экземпляр шаблона *Deputized Redirection*.

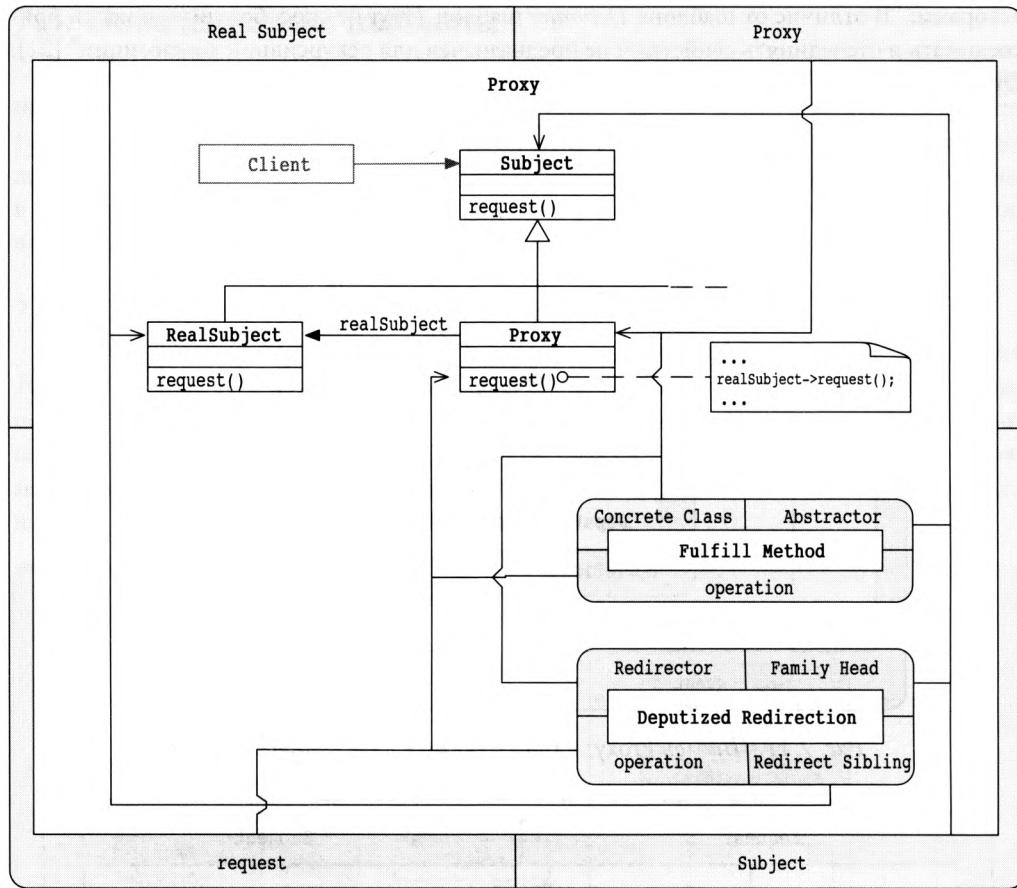


Рис. 7.10. Шаблон Proxy, изображенный в виде компонента PINbox

Важной характеристикой этого шаблона является использование шаблона *Deputized Redirection*. Это значит, что объекты, которым мы переадресовываем задание, относятся к типам с высоким уровнем доверия. Для простоты эту диаграмму можно сократить до диаграммы PIN (рис. 7.11), а затем немного перестроить (рис. 7.12).

Причина такой перестройки диаграмм заключается в том, что после нее определение шаблона *Proxy* становится более тесно связанным с разделом шаблона *Decorator*, содержащего экземпляр шаблона *Object Recursion* (см. рис. 7.9). Если присмотреться к ним повнимательнее, можно увидеть, что шаблон *Proxy* представляет собой, по существу, вариант шаблона *Object Recursion* с более высоким уровнем доверия и одним важным отличием. Когда экземпляр объекта-получателя находится в одном из подклассов, рекурсивная природа шаблона *Object Recursion* нарушается. Вместо нее мы получаем отдельную связь, одномоментную переадресацию от заместителя к целевому объекту. Кроме того, в шаблоне *Proxy* нет экземпляра шаблона *Extend Method*, который есть в шаблоне *Decorator*. Это свидетельствует о том, что шаблон *Decorator* может расширять функциональные возможности, а шаблон *Proxy* — нет. Эти два обстоятельства подтверждаются самими

авторами: “В отличие от шаблона *Decorator* шаблон *Proxy* не способен динамически присоединять и отсоединять свойства и не предназначен для рекурсивной композиции” [21]. Эта разница четко прослеживается на диаграммах PIN и композициях этих шаблонов.

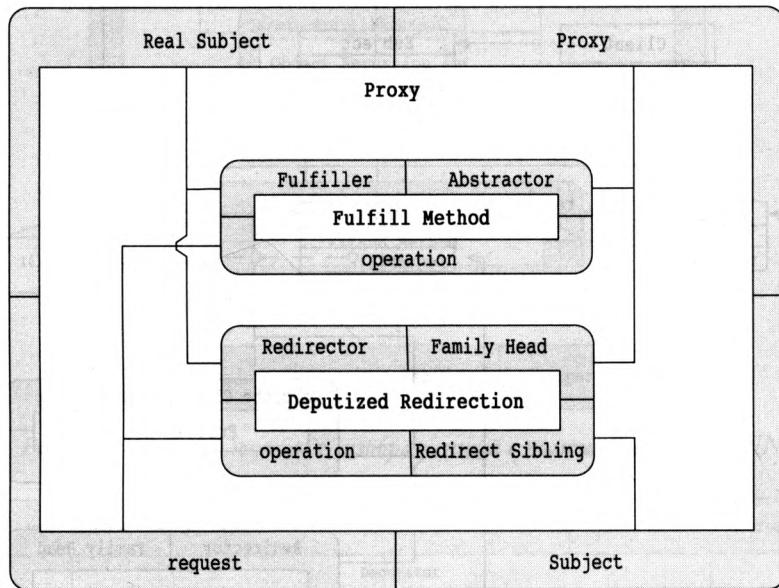


Рис. 7.11. Шаблон Proxy, изображенный в виде диаграммы PIN

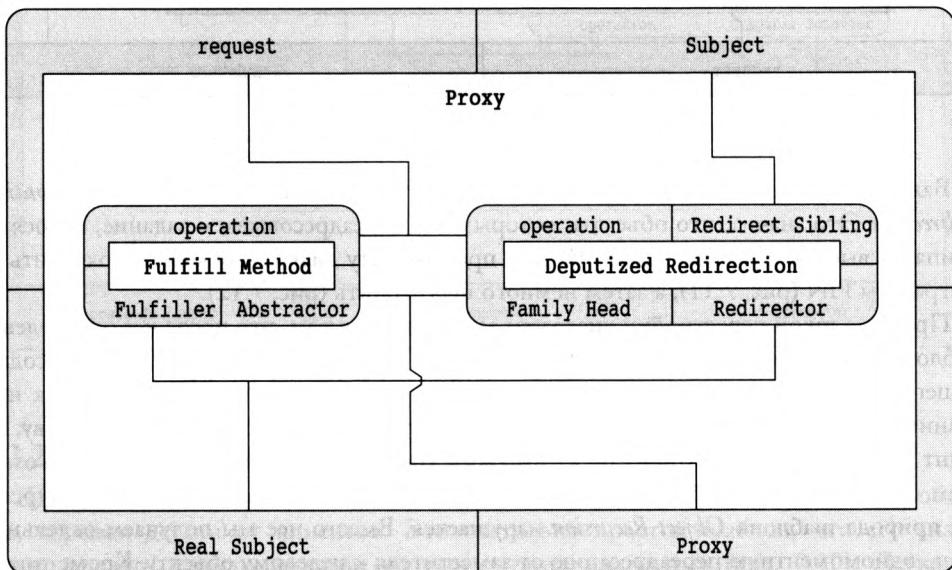


Рис. 7.12. Перестроенная диаграмма PIN для шаблона Proxy

7.3. Поведенческие шаблоны

Поведенческие шаблоны GoF представляют собой интересную проблему, если при их описании использовать только отношение “вызов метода” и соответствующие элементарные шаблоны. Однако трехмерное пространство проектирования, описанное в разделе 2.2.2, обеспечивает достаточно контекстной информации о многих из этих шаблонов и позволяет сформулировать удивительно надежные определения. В частности, заслуживают внимания шаблоны *Chain of Responsibility* и *Template Method*.

7.3.1. Шаблон *Chain of Responsibility*

Шаблон *Chain of Responsibility* — это шаблон, имеющий много общего с шаблонами *Proxy* и *Decorator*. Как обычно, мы начнем со структурной диаграммы, свернутой в компоненте PINbox (рис. 7.13). Общая конструкция шаблона вполне очевидна. Он состоит всего из двух элементарных шаблонов проектирования. На диаграмме есть несколько экземпляров шаблона *Extend Method*, но она достаточно проста, чтобы не останавливаться на ней и перейти непосредственно к PIN-диаграмме (рис. 7.14).

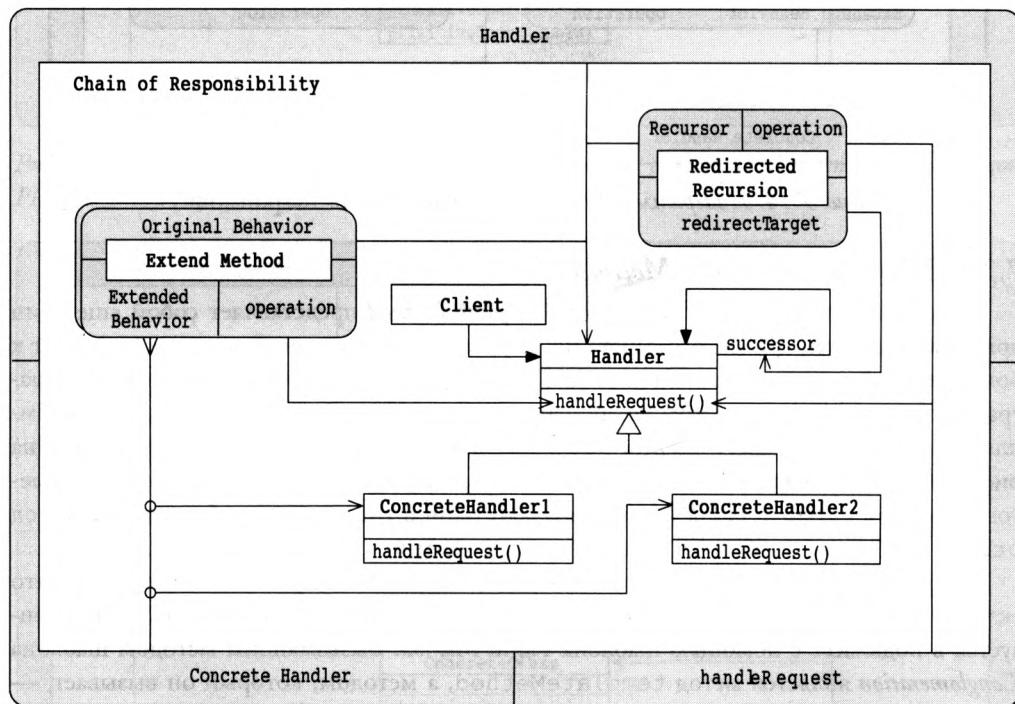


Рис. 7.13. Шаблон *Chain of Responsibility*, изображенный в виде развернутого компонента PINbox

Фактически, если сравнить определения шаблонов *Chain of Responsibility* и *Decorator*, основанные на элементарных шаблонах проектирования, становится очевидным различие

между ними: шаблон *Redirected Recursion* в шаблоне *Chain of Responsibility* заменяет шаблон *Object Recursion* в шаблоне *Decorator*. Если используется вариант шаблона *Chain of Responsibility*, в котором класс *ConcreteHandlers* определяет наследника явно [21], то мы встретим шаблон *Object Recursion*. Единственное отличие заключается в том, где именно встречается шаблон *Object Recursion*. Очень интересно наблюдать, как почти незаметные концептуальные изменения приводят к совершенно разным и значительным последствиям. Шаблоны *Chain of Responsibility* и *Decorator* очень близки концептуально, но при этом настолько сильно отличаются один от другого по структуре, что даже относятся к разным категориям.

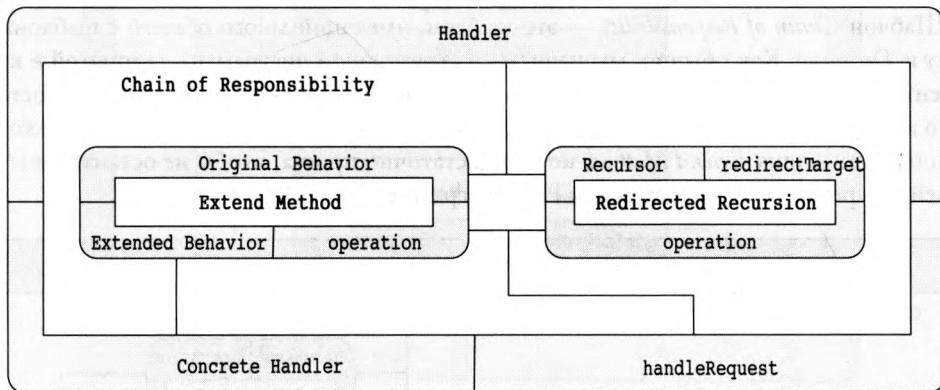


Рис. 7.14. Диаграмма PIN для шаблона Chain of Responsibility

7.3.2. Шаблон *Template Method*

В заключение отметим, что шаблон *Template Method* представляет собой еще один яркий образец специфического сочетания двух простых концепций, которое приводит к более мощной конструкции. На рис. 7.15 представлена оригинальная структурная диаграмма, свернутая в раскрытом компоненте PINbox. На ней показано несколько экземпляров шаблонов *Conglomeration* и *Fulfill Method*, поэтому ее можно упростить, как на рис. 7.16. Обратите внимание на то, что метод *operation2* в шаблоне *Conglomeration* и метод *operation* в шаблоне *Fulfill Method* означают один и тот же метод: абстрактный метод *primitiveOperation1()* из класса *AbstractClass*.

Теперь легко получить окончательную диаграмму PIN. На рис. 7.17 ясно видно, что экземпляр шаблона *Conglomeration* вызывается абстрактным методом, который реализуется в подклассе с помощью шаблона *Fulfill Method*. Вызывающим методом шаблона *Conglomeration* является метод *templateMethod*, а методом, который он вызывает, — абстрактный метод *primitiveOperation* в том же классе. Следовательно, этот класс является абстрактным, а соответствующую функциональную возможность обеспечивает абстрактный класс в шаблонах *Template Method* и *Fulfill Method*. Таким образом, этот подкласс выполняет роль конкретного класса в шаблонах *Fulfill Method* и *Template Method*. Требование, чтобы целевой метод в шаблоне *Conglomeration* был абстрактным в

экземпляре шаблона *Fulfill Method* вынуждает подкласс обрабатывать детали так, чтобы реализации фрагментов алгоритма настраивались динамически.

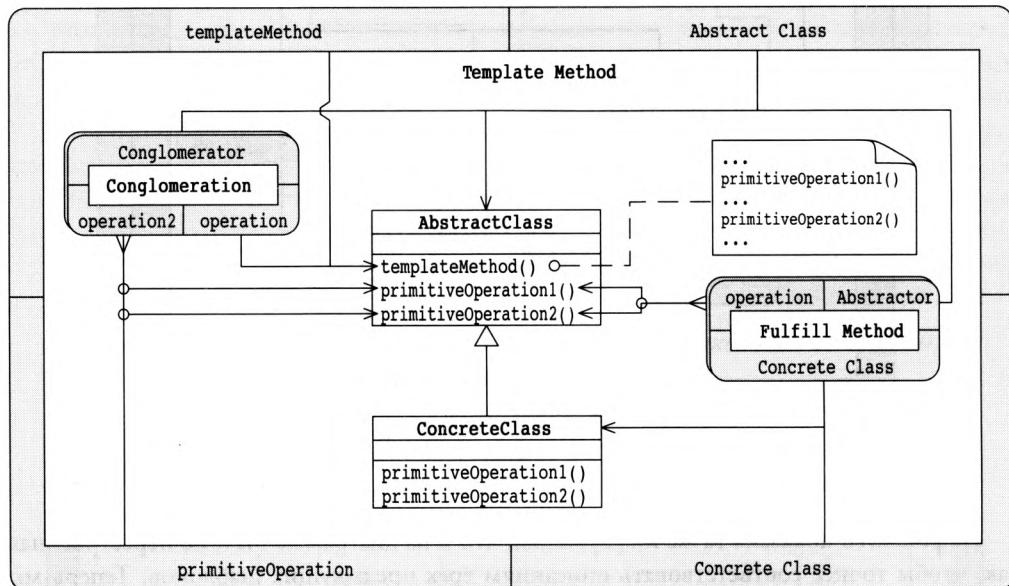


Рис. 7.15. Шаблон Template Method, изображенный в виде развернутого компонента PINbox

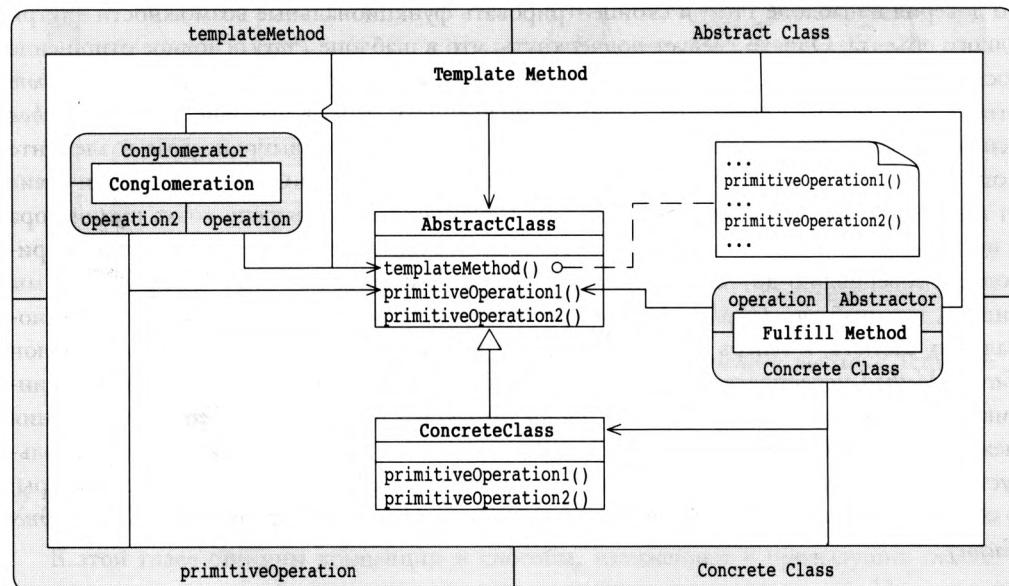


Рис. 7.16. Шаблон Template Method, сокращенный до одного экземпляра

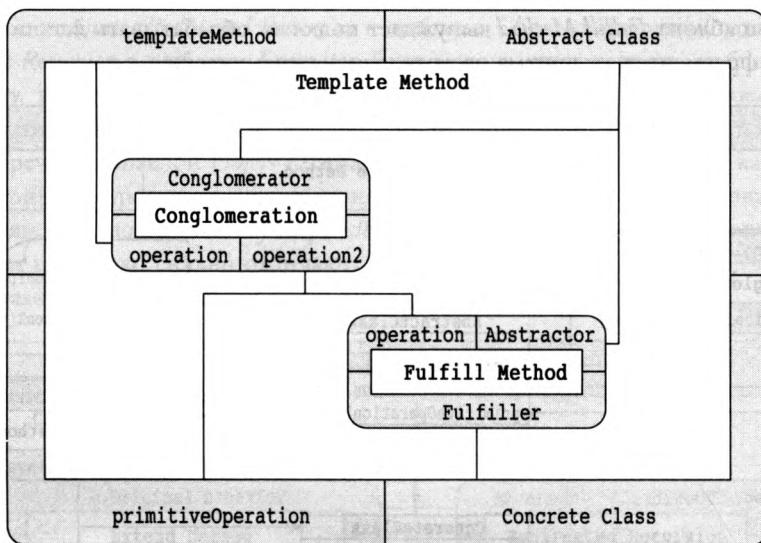


Рис. 7.17. Шаблон Template Method в виде PIN-диаграммы

На рис. 7.18 показана та же информация, что и на диаграмме PIN, но перестроенная так, чтобы точнее соответствовать описаниям трех предыдущих шаблонов. Теперь мы можем увидеть, где именно шаблон *Proxy* использует шаблоны *Fulfill Method* и *Deputized Redirection*, и что шаблон *Template Method* заменяет экземпляр шаблона *Deputized Redirection* экземпляром шаблона *Conglomeration*. Это позволяет исключить из рассмотрения аспекты доверия в шаблоне *Proxy* и сконцентрировать функциональные возможности внутри одного объекта. Однако следует подчеркнуть, что в шаблоне *Proxy* основное отношение состоит в том, что абстрактные методы, выполняемые в элементе *Proxy*, инициируют отношение *Deputized Redirection*. В противоположность этому в шаблоне *Template Method* основным свойством является то, что абстрактные методы, выполняемые в элементе *Concrete Class*, являются целями для экземпляров шаблона *Conglomeration*. Ни сдвиг от шаблона *Deputized Redirection* к шаблону *Conglomeration*, ни перестановка инициатора и цели вызова метода не являются чем-то сложным или необычным, но вместе они приводят к совершенно другим результатам. При обсуждении шаблона *Factory Method* мы видели, что шаблон *Template Method* предоставлял возможность реализации функциональных свойств, а теперь мы видим, как он это делает. На рис. 7.19 показан шаблон *Factory Method*, переопределенный с помощью шаблона *Template Method* путем замены индивидуальных экземпляров шаблона *Fulfill Method* и *Conglomeration*. Теперь отношения между этими шаблонами четко определены и понятны. Шаблон *Factory Method* использует шаблон *Template Method* для повышения гибкости обслуживания инфраструктуры, в которой используются шаблоны *Retrieve* и *Create Object*, аналогично шаблону *Abstract Factory*.

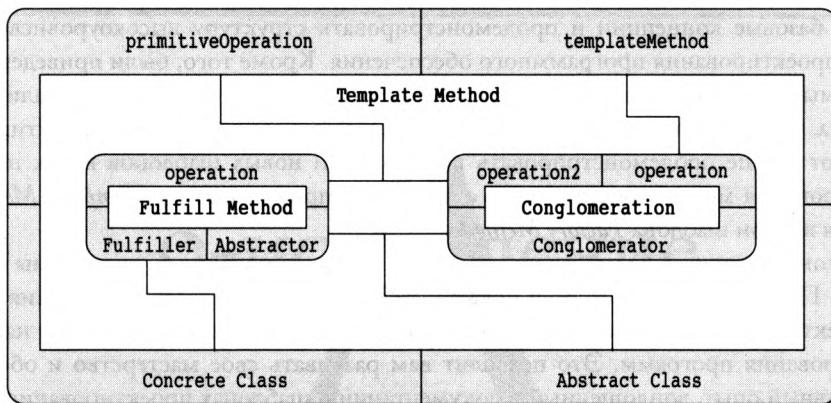


Рис. 7.18. Шаблон Template Method в виде перестроенной PIN-диаграммы, которая точнее соответствует шаблону Decorator

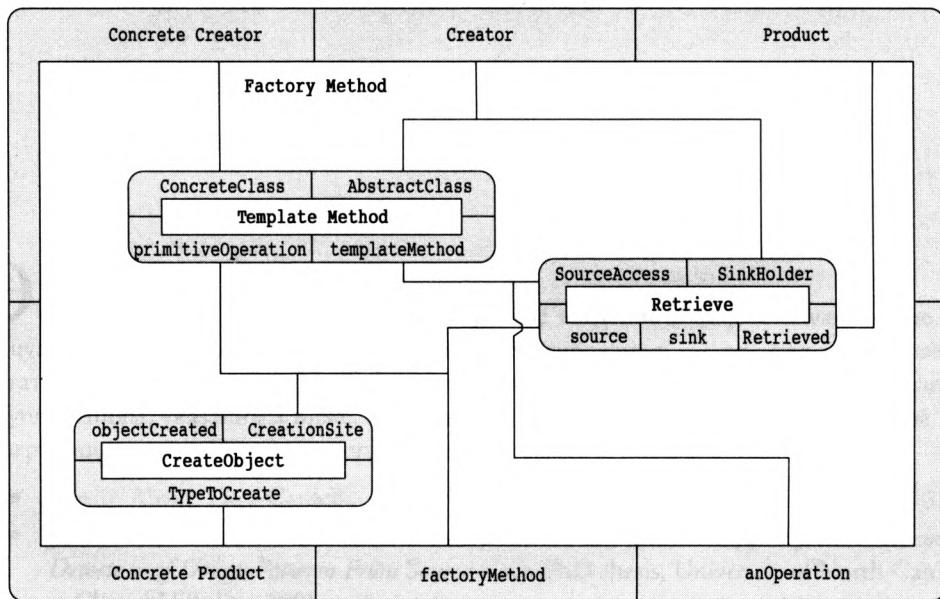


Рис. 7.19. Шаблон Factory Method, переопределенный с помощью шаблона Template Method

7.4. Заключение

В этой главе описаны концепции и способы, изложенные в предыдущих главах, и проанализированы наиболее распространенные шаблоны проектирования. При этом использовались элементарные и промежуточные шаблоны проектирования, что позволило

выявить базовые концепции и продемонстрировать структуру высокоуровневых концепций проектирования программного обеспечения. Кроме того, были приведены PIN-диаграммы этих шаблонов в разных вариантах в зависимости от уровня детализации и контекста. Было показано, как спецификации существующих шаблонов проектирования позволяют лучше продемонстрировать возможности новых шаблонов и как шаблоны проектирования можно сочетать один с другим, например шаблон *Template Method* содержитя внутри шаблона *Factory Method*.

На этом введение в элементарные шаблоны проектирования и диаграммы PIN закончено. Перед вами открываются новые интересные перспективы применения шаблонов проектирования в нашей индустрии. Теперь вы вооружены знаниями и навыками проектирования программ. Это позволит вам развивать свое мастерство и обогащать коллективный опыт, воплощенный в документации о шаблонах проектирования. Добро пожаловать в наше сообщество! Посмотрим, что будет дальше!

ρ-исчисление

Это приложение представляет собой краткое введение в формальную теорию, лежащую в основе элементарных шаблонов проектирования, — ρ- или ρ-исчисление. Читатели, которых интересует более полное описание этой теории, могут обратиться к публикациям, указанным ниже. Я надеюсь, что они помогут им войти в богатый мир программирования. Продолжая чтение, иногда полезно возвращаться к главе 2.

- Martin Abadi, Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.
- Jason McC. Smith. *SPQR: Formal Foundations and Practical Support for the Automated Detection of Design Patterns From Source Code*. PhD thesis, University of North Carolina at Chapel Hill, Dec 2005.

В разделе 2.2.2 была сформулирована идея о том, что все объектно-ориентированное программирование можно описать с помощью всего четырех сущностей: объектов, методов, полей и типов. Методы, поля и типы — вполне очевидные компоненты, потому что им соответствуют три основных элемента программирования: функции, хранение данных и описание данных. Никакие вычисления невозможно выполнить без всех этих трех элементов, какую бы парадигму программирования вы ни применяли. Настал момент объяснить, почему мы добавили к ним объекты.

Несмотря на самоочевидность, формальной причиной добавления объектов является то, что они являются ядром объектно-ориентированного программирования. Без объектов нет объектно-ориентированного программирования. Вас не устраивает такой ответ? Читайте работу Абади (Abadi) и Карделли (Cardelli) о ζ -исчислении (сигма-исчислении). [1] Сигма-исчисление — это денотационная семантика (denotational semantics), представляющая собой ядро ρ -исчисления. Используя ζ -исчисление, можно свести семантику любого объектно-ориентированного языка программирования¹ к четырем элементам: объектам, методам, полям и типам.² Получившийся набор элементов программирования настолько мал, что способы их перечисления и взаимодействия можно просто перечислить. Иначе говоря, мы можем полностью описать любые отношения в объектно-ориентированном программировании.

С другой стороны, объектно-ориентированное программирование можно моделировать с помощью методов (функций), полей и типов традиционного процедурного программирования. При этом происходит потеря некоторых фундаментальных понятий. Попытки расширить семантику лямбда-исчисления, лежащего в основе процедурного программирования, на объектно-ориентированное программирование предпринимаются уже тридцать лет, но все они закончились неудачей. В любом случае, независимо от того, насколько близкой к объектно-ориентированным системам оказалась расширенная модель процедурного программирования, математический аппарат, лежащий в их основе, настолько сложен и запутан, что не позволяет продемонстрировать концептуальную элегантность объектно-ориентированных языков программирования. В то же время ζ -исчисление, учитывающее сущности первого порядка, позволило создать успешную математическую модель, отражающую идеи объектно-ориентированного программирования и заложить прочную основу для формальных манипуляций объектами.

A.1. Операторы зависимости

Сигма-исчисление исключает из рассмотрения все элементы, кроме четырех необходимых сущностей — объектов, методов, полей и типов, — открывая возможности для простого и подробного анализа. Например, для четырех элементов существует только 16 способов их сочетания. Этот пункт проиллюстрирован в табл. 2.2 и воспроизведен в табл. A.1.

¹ За некоторым исключением, в частности за исключением языка Eiffel; подробное объяснение не входит в нашу задачу, но его можно найти в работе [35]. Если вы хотите глубже разобраться в этом вопросе, сравните в качестве отправной точки ковариантные и контравариантные типы значений, возвращаемых методами.

² Да, это кажется игрой в наперстки с типами. Но, прочитав книгу [1], посвященную сигма-исчислению, вы сразу поймете, что это не так. Основа теории является прочной.

Таблица A.1. Все взаимодействия между сущностями в объектно-ориентированных языках программирования

	Объект	Метод	Поле	Тип
Объект	Определяет	Определяет	Определяет	Определяет <i>или</i> имеет тип
Метод	Нет	Определяет <i>или</i> вызывает метод	Определяет <i>или</i> использует поле	Определяет <i>или</i> имеет тип возвращаемого значения
Поле	Нет	Изменяет состояние	Связывает	Имеет тип
Тип	Определяет	Определяет	Определяет	Определяет <i>или</i> является подтипом

Эта таблица достаточно мала, чтобы проанализировать ее шаг за шагом. Большую часть можно охватить, изучив лишь взаимодействие “Определяет”. Объект определяет внутри себя метод или поле, или тип. Такие объекты, как пространства имен или пакеты, также могут определять внутри себя другие объекты. В некоторых языках методы могут определять внутренние методы и типы, но при этом почти всегда определяют в себе объекты и поля. Поля по своей природе не предназначены для того, чтобы сворачивать в себе или определять внутри себя другие элементы, но *типы* поля, такие как класс, могут определять в себе любой из четырех элементов.

Такое определение внутренних сущностей также называется *заданием области видимости* (scoping). Класс, определяющий метод, является областью видимости для внутреннего метода. Для того чтобы обратиться к методу, необходимо войти внутрь класса либо непосредственно, либо через объект или поле, созданное на основе класса. Объект является областью видимости для своих полей и методов. Примерами такого обращения к полям и методам в языке C++ является статический метод в классе или поле в пространстве имен: `Some-Class::aMethod()` и `ANameSpace::aField` соответственно. В языке Java нет оператора разрешения области видимости (двойного двоеточия), а вместо него единообразно используется обычная точка, например `SomeClass.aMethod()` или `APackage.aField`. Такие же обозначения используются в ζ -исчислении. Во всех случаях область видимости указывается *оператором точки*. Большинству программистов такое обозначение хорошо знакомо, поскольку оно используется для доступа внутрь объектов и выбора сущностей из определения. На практике никакой разницы на самом деле нет. Глагол *определяет* просто является синонимом выражения “*задает область видимости*”.

Почему же мы использовали слово “*определяет*”, а не “*задает область видимости*”? Главное различие между традиционным денотационным исчислением процедурного программирования (λ -исчислением) и ζ -исчислением заключается в том, что выбор элемента из поля подразумевает выполнение оператора σ . Он связывает экземпляр с типом, позволяя находить выбранный элемент в соответствующей области видимости, заданной полем, даже при наличии полиморфизма.

Это связывание (binding) является отличительной чертой ζ -исчисления и в сочетании с обработкой объектов как сущностей первого класса обеспечивает чрезвычайно высокую эффективность и элегантность. Однако в контексте нашей книги мы можем его игнорировать.

Исключение взаимодействия “Определяет” из табл. 2.3 приводит нас к табл. А.2.

Таблица А.2. Взаимодействия между сущностями в объектно-ориентированном программировании, не связанные с определением области видимости

Объект	Метод	Поле	Тип
Объект			Имеет тип
Метод	Вызывает метод	Использует поле	Возвращает тип
Поле	Изменяет состояние	Связывает	Имеет тип
Тип			Определяет подтип

Объекты и поля “имеют тип”. Сигма-исчисление выражает этот факт следующим образом: $anObject : itsType$. Это обозначение также используется для того, чтобы указать тип значения, возвращаемого методом. Тип либо имеет подтипы, либо сам является подтипом какого-то типа. В ζ -исчислении это выражается так: $Subtype <: Supertype$.³ В таком случае табл. А.2 становится почти пустой, за исключением четырех *операторов зависимости* (*reliance operators*): вызов метода, использование поля, изменение состояния и связывание.

В ρ -исчислении существует четыре оператора зависимости, которые основаны на четырех отношениях, определенных выше. Оператор зависимости “метод–метод” (т.е. вызов метода) представляет собой мю-форму. Это единственный оператор зависимости, который мы рассмотрим в нашей книге. Оператор зависимости “метод–поле” — использование поля — является фи-формой. Оператор зависимости “поле–метод”, представляющий собой изменение состояния, т.е. состояние поля зависит от вызова метода, — известен как сигма-форма. Наконец зависимость “поле–поле” — это традиционное связывание, или каппа-форма. Эти формы в ρ -исчислении обозначаются как $<_{\mu}$, $<_{\phi}$, $<_{\sigma}$ и $<_{\kappa}$ соответственно. Эти обозначения происходят от оператора вывода подтипа $<:$, который можно интерпретировать так: “Существование и цель подтипа зависят от супертипа”. Поскольку $:$ — это оператор определения типа, мы можем интерпретировать обозначение $<:$ как “зависит от”. Это обозначение сопровождается соответствующим индексом, образующим один из четырех операторов зависимости. Они являются бинарными операторами, образующими зависимость левой части выражения от его правой части. Иначе говоря, выражение $someMethod <_{\mu} anotherMethod$ следует читать как “*someMethod* зависит от *anotherMethod* я”.

A.2. Транзитивность и изотопы

В разделе 4.1.1 были описаны изотопы и способы их образования с помощью косвенных зависимостей. Для того чтобы описать этот процесс формально с помощью ρ -исчисления, мы используем транзитивность. Следующий исходный код воспроизводит пример из раздела 4.1.1. Как и ранее, мы можем утверждать, что метод *f.foo* зависит

³ Определение подтипов, подклассов и наследование — это с формальной точки зрения не одно и то же, но в контексте нашей книги эти слова можно использовать как синонимы. Детали см. в [1].

от метода `b.bar`, но теперь мы можем записать это как $f.foo <_{\mu} b.bar$. Пока все хорошо. В нашем примере мы можем утверждать, что $f.foo <_{\mu} g.goo$ и $g.goo <_{\mu} b.bar$.

```

1 class F {
2     B b;
3     void foo() {
4         b.bar();
5     };
6 }
7 F f;
f.foo();

```

`f.foo()` зависит от `b.bar()`

```

class F {
    G g;
    void foo() {
        g.goo();
    };
}
class G {
    B b;
    void goo() {
        b.bar();
    };
}
F f;
f.foo();

```

`f.foo()` по-прежнему зависит
от `b.bar()`

Эти два бита информации можно использовать в *правиле редукции* (reduction rule). Это простейшая форма вывода в ζ - и ρ -исчислениях. Если дано множество истинных высказываний, можно вывести новое свойство. В таком случае говорят, что выражения редуцируются к результату. Например, существует правило редукции для транзитивности $<_{\mu}$, как в формуле (A.1).

$$\begin{aligned}
 E \vdash x, y, z \\
 x \equiv [m = b_0] \\
 y \equiv [n = b_1] \\
 z \equiv [o = b_2] \\
 x.m <_{\mu} y.n \\
 y.n <_{\mu} z.o \\
 \underline{x.m <_{\mu} z.o}
 \end{aligned} \tag{A.1}$$

Это правило редукции начинается с утверждения, что у нас имеется среда E , в которой мы определили элементы x , y и z . Следующие три утверждения означают, что эти элементы имеют подэлементы m , n и o соответственно, которые являются методами. Следующие два утверждения задают операторы зависимости: $x.m$ зависит от $y.n$ и $y.n$ зависит от $z.o$. Из этих утверждений следует, что $x.m$ зависит от $z.o$. Это один из многих транзитивных переходов в ρ -исчислении. Транзитивность обеспечивает аналитическую силу этого исчисления. Очень большие системы можно сократить до небольших простых наборов операторов зависимости, а поиск высокоуровневых абстракций, таких как шаблоны проектирования, можно свести к анализу упрощенных представлений результатов.

Без этой технологии нам пришлось бы проверять все части системы. Если, например, мы ищем доказательство того, что метод $x.m$ вызывается методом $z.o$, то простая проверка этой системы ничего не даст. Используя транзитивные переходы ρ -исчисления и сведя

задачу к поиску ситуации, в которой метод $x.m$ зависит от метода $z.o$, можно быстро доказать существование этого отношения.

Точнее говоря, этот транзитивный переход позволяет создавать конкретные определения шаблонов проектирования, основанные на зависимостях, которые могут описывать огромное количество структурных реализаций. Это изотопы канонической, фундаментальной и прямой реализаций. Именно благодаря им этот подход является настолько мощным и гибким.

A.3. Сходство

Три координатные оси, характеризующие сходство в разделе 2.2.2, также можно формализовать с помощью ρ-исчисления. Напомним, что существует три способа, которыми отдельный оператор зависимости может установить сходство между своими операндами. Например, если задана зависимость $f.foo <_{\mu} b.bar$, можно говорить о сходстве между объектами f и b , о сходстве между типами объектов f и b , а также о сходстве между методами foo и bar . В ρ-исчислении символ \sim используется для обозначения сходства, а символ \neq — для обозначения различия. Например, выражение $A \sim B$ означает “ A похож на B ”, а выражение $B \neq C$ означает “ B отличается от C ”.

В ρ-исчислении сходство между типами проявляется в определении f и b . Для того чтобы использовать сущности f и b в каком-то определении в рамках ρ-исчисления, они должны быть определены и должны иметь типы. Следовательно, информация о типах сущностей f и b должна быть заранее определена. Однако об этом отношении между типами мы можем говорить следующим образом: если типы не связаны один с другим, будем называть их просто разными и обозначать символом \neq ; если сущности имеют один и тот же тип, будем называть их одинаковыми и обозначать символом \sim ; если один тип является подтипом другого, мы используем обозначение $<:$; а если типы относятся к одному и тому же уровню иерархии супертипа, используем обозначение $<:>$. Вся эта информация уже включена в ρ-исчисление.

Мы можем также говорить о сходстве или различиях между объектами и методами, используя те же самые обозначения \sim и \neq . Однако для краткости и вследствие уникальности этой информации относительно конкретного оператора зависимости мы будем добавлять ее к оператору зависимости в виде верхнего индекса.

Несмотря на то что обозначения \sim и \neq легко различить в обычном тексте, для текста, набранного маленьким шрифтом мы будем использовать символы $+/-$.⁴ Оператор точки имитирует обозначение сходства: сходство объекта указывается слева от точки, а сходство методов — справа. Например, оператор вызова метода при разных объектах, но одинаковых методах обозначается как $<_{\mu}^{+}$. Оператор зависимости при сходных объектах и разных методах обозначается как $<_{\mu}^{-+}$ и т.д. Это позволяет записать большой объем информации в лаконичной форме. В сочетании с транзитивными переходами это позволяет просто, точно и ясно описать отношения между двумя сущностями в системе. Вы

⁴ Извините за каламбур, но символы \sim и \neq в индексах, набранных мелким шрифтом, очень похожи. Символы $+$ и $-$ легче различить.

можете сказать, что нам не всегда известно, являются ли элементы системы сходными. Вы правы. В таком случае для обозначения неизвестного отношения используется символ \circ .

A.4. Формализм элементарных шаблонов проектирования

Теперь мы готовы использовать ρ -исчисление для создания основ элементарных шаблонов проектирования. Мы можем заново проанализировать диаграммы пространства проектирования EDP из раздела 4.4 и обозначить эту информацию так, как показано на рис. A.1 и A.2.

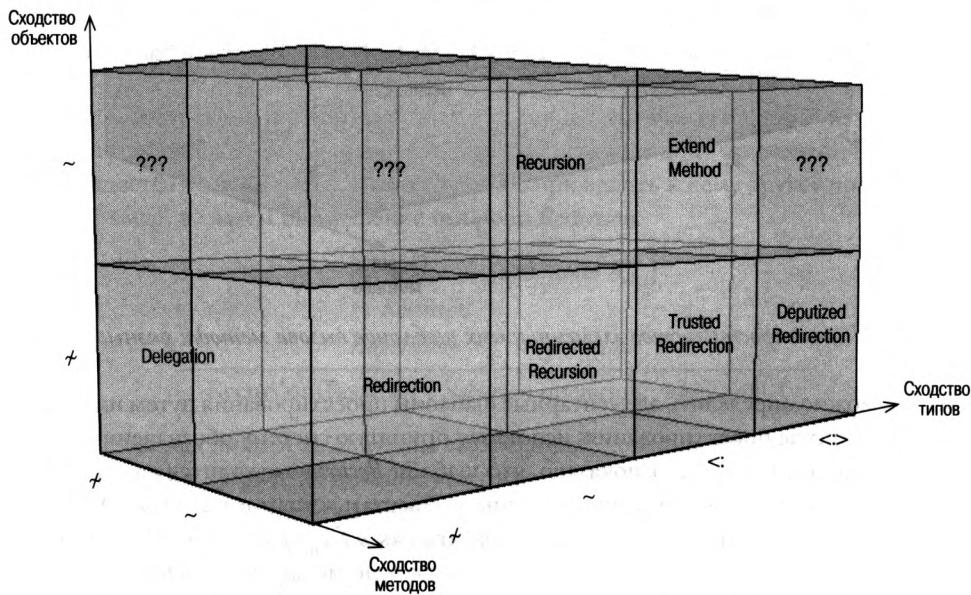


Рис. A.1. Полное пространство элементарных шаблонов вызова метода:
одинаковые методы

Сначала примем несколько предположений. Допустим, что у нас есть объект o типа A и объект p типа B . Запишем эти факты следующим образом: $o : A$ и $p : B$ соответственно. Тип A определяет метод f , а тип B определяет метод g . Это условие можно проиллюстрировать в виде листинга A.1.

Листинг A.1. Просто пример

```

1  class A {
2      void f();
3  };
4
5  class B {
6      void g();
7  };
8
9  A o;
10 B p;

```

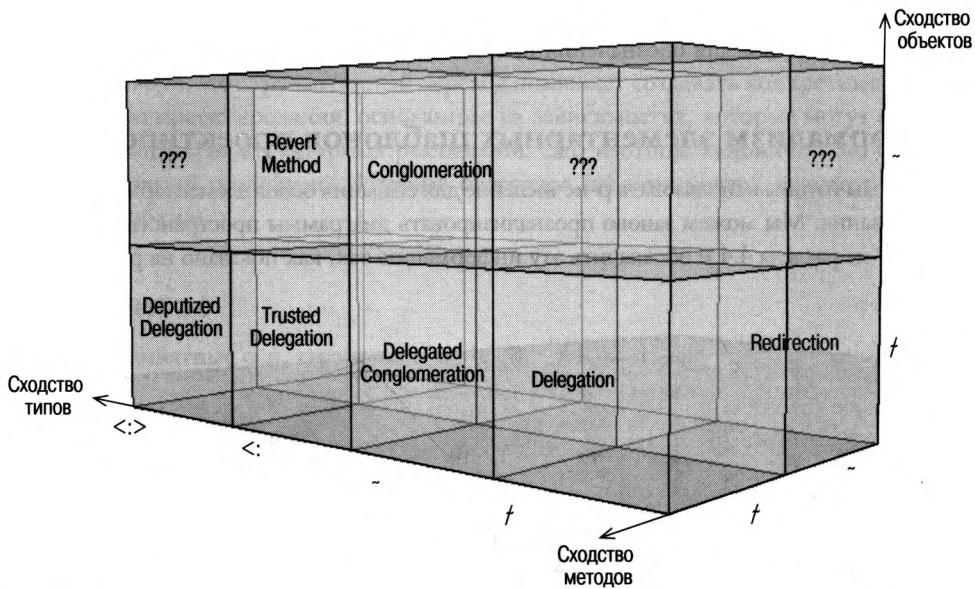


Рис. A.2. Полное пространство элементарных шаблонов вызова метода: разные методы

Теперь можно определить элементарные шаблоны проектирования путем их размещения в пространстве проектирования, используя принятую систему обозначений и операторы зависимости. На рис. A.1 показано, что шаблон *Recursion* находится на пересечении координатных осей, соответствующих сходным объектам, типам и методам. Это значит, что шаблон *Recursion* формально можно выразить как $o.f <_{\mu}^{++} p.g, A \sim B$. Напомним, что верхний индекс означает, что это отношение установлено между двумя элементами обеих сторон оператора точки, определяющего область видимости. В данном случае объект o схож с объектом p , а метод f схож с методом g .

Легко видеть, что шаблон *Delegation* можно записать как $o.f <_{\mu}^{--} p.g, A \not\sim B$, а шаблон *Redirection* — как $o.f <_{\mu}^{-+} p.g$. Следовательно, утверждение из раздела 2.2.4 о том, что шаблоны *Redirection* и *Delegation*, взятые вместе, образуют концепцию связности, можно записать следующим образом: $o.f <_{\mu}^{-\circ} p.g$. Наконец-то, нам хоть как-то пригодилось обозначение неизвестного отношения.

Я уверен, что вы уже догадались, как записывается шаблон *Conglomeration ()*, и теперь понятие связности, описанное в разделе 2.2.4, сводится к выражению $o.f <_{\mu}^{+\circ} p.g$, поскольку сходство в шаблоне *Recursion* и различия в шаблоне *Conglomeration* компенсируют друг друга.

Для шаблона *Redirected Recursion* зависимость между методами записывается так, как и для шаблона *Redirection*: $o.f <_{\mu}^{-+} p.g$. Кстати, этим объясняется выбор названия: шаблон *Redirected Recursion* уточняет шаблон *Redirection*, а не *Recursion*. В результате достаточно изменить только один тип отношений: $A = B$.

В два последних элементарных шаблона проектирования, расположенных на нижнем ряду на рис. A.1, добавлены отношения подтипа и одного уровня. Как и для шаблона

Redirection, зависимость между методами записывается в следующем виде: $o.f <_{\mu}^{-+} p.g$, и снова мы лишь изменили отношение между типами. На этот раз мы задали отношение $A < : B$, означающее, что тип A является подтипов типа B . Это приводит к шаблону *Trusted Redirection*.

Шаблон *Deputized Redirection* можно получить на основе шаблона *Redirection*, изменив отношение между типами на $A < :> B$. В заключение рассмотрим шаблон *Recursion*. Начнем с вызова метода и утверждений о типах: $, o : A, p : B$. Изменим отношение между типами на $A < : B$. Этого достаточно, чтобы из шаблона *Recursion* получить шаблон *Extend Method*.

Этот подход можно использовать для определения любого элементарного шаблона проектирования в описанном пространстве, элегантно и точно выразив их главные отношения с помощью ρ-статистики. В разделе A.7 приведены полные описания каждого элементарного шаблона проектирования с помощью правила редукции из раздела A.2. Из заданного набора выражений можно вывести, что результат применения правила редукции является очевидным, и, в свою очередь, применить к нему другое правило редукции. Например, возьмем определение шаблона *Recursion*.

$$\begin{aligned} \textit{Recursor} : [l; \textit{operation}: B_{m+1}], \\ r : \textit{Recursor}, \\ r.\textit{operation} <_{\mu}^{++} r.\textit{operation} \end{aligned}$$

Recursion(Recursor : Recursor, operation : operation)

Выражения над линией аналогичны выражениям, которые мы видели в определении шаблона *Recursion*. Первое выражение утверждает, что *Recursor* — это тип и что он имеет по крайней мере один метод с именем *operation* и, возможно, другие методы и/или поля. Второе выражение утверждает, что объект r имеет тип *Recursor*. Третье выражение утверждает, что метод $r.\textit{operation}$ вызывает сам себя и при этом объекты и методы совпадают. Последнее выражение утверждает, что элементы предыдущих выражений образуют экземпляр шаблона *Recursion*. Мы используем имена в формальном определении как названия ролей в шаблоне. Они используются при обсуждении шаблонов и при определении структуры их презентации в виде диаграммы PIN, как будет показано в разделе A.6. Элементы, связанные с этими ролями, в математическом смысле являются переменными. Как только сущность получает роль, говорят, что данная роль исполняется этой сущностью. В предыдущем случае любой класс, удовлетворяющий потребности шаблона *Recursion*, заданные в определении, исполняет роль *Recursor* в шаблоне *Recursion*.

A.5. Правила композиции и редукции

На основе этих определений можно создавать более сложные шаблоны. Рассмотрим, например, шаблон *Decorator*, с которым мы работали в разделе 4.2. Мы ввели шаблон *Fulfill Method* как сочетание шаблонов *Inheritance* и *Abstract Method*, а затем использовали его для определения шаблона *Objectifier*. Объединив шаблоны *Objectifier* и *Trusted Redirection*, мы определили шаблон *Object Recursion*, который, в свою очередь, в сочетании с шаблоном *Extend Method*, позволил создать шаблон *Decorator*.

Определим шаблон *Fulfill Method* следующим образом.

$$\begin{aligned} & \text{AbstractInterface}(\text{Abstractor} : \text{Abstractor}, \text{operation} : \text{operation}_n) \\ & \text{Inheritance}(\text{Superclass} : \text{Abstractor}, \text{Subclass} : \text{operation}_n) \\ & \text{ConcreteClass} \equiv [\text{operation}_i \leftarrow b_i^{i \in 1 \dots n-1, n+1 \dots m}, \text{operation}_n \leftarrow b_n] \\ \hline & \text{FulfillMethod}(\text{Abstractor} : \text{Abstractor}, \text{Fulfiller} : \text{ConcreteClass}, \\ & \quad \text{operation} : \text{operation}_n) \end{aligned}$$

Первое выражение объявляет экземпляр шаблона *Abstract Interface*, имеющий два аргумента: тип сущности и метод сущности. Он устанавливает отношения между ними в соответствии с определением шаблона *Abstract Interface*, но допускает замену полного формального определения его сокращенной версией. Второе высказывание утверждает, что *ConcreteClass* — это подкласс класса *Abstractor*, созданный с помощью элементарного шаблона *Inheritance*. Третья строка представляет собой новый элемент головоломки. Она утверждает, что метод, объявленный в шаблоне *Abstract Interface*, определен в классе *ConcreteClass* в соответствии с контрактом класса *Abstract Interface*. Этот момент является очень важным, поскольку для определения абстрактного метода на самом деле подкласс не нужен — его вполне может заменить другой абстрактный класс в иерархии классов. Имена, указанные в этих выражениях, выбраны не случайно. Если имя появляется в правиле редукции более одного раза, то соответствующие роли в системе исполняет одна и та же сущность. Именно это позволяет связывать небольшие шаблоны в крупные композиции.

Можно продолжить процесс применения правил редукции и определить шаблон *Objectifier*, добавив класс *Client* с абстрактными операциями.

$$\begin{aligned} & \text{ObjectifierBase}[l_i : B_i^{i \in 1 \dots n}] \\ & \text{Client} : [\text{ref. Objectifier}] \\ & \text{Client.someMethod} <_{\mu} \text{Client.ref.} l_i \\ & \text{FulfillMethod}(\text{Abstractor} : \text{Abstractor}, \text{Fulfiller} : \text{ConcreteClass}, \\ & \quad \text{operation} : l_j^{j \in 1 \dots n} \\ & \text{ConcreteClass} \equiv [\text{operation}_i \leftarrow b_i^{i \in 1 \dots n-1, n+1 \dots m}, \text{operation}_n \leftarrow b_n] \\ \hline & \text{Objectifier}(\text{Objectifier} : \text{ObjectifierClass}, \text{ConcreteClass} : \text{ConcreteObjectifier}, \\ & \quad \text{operation} : l_k) \end{aligned}$$

Определение шаблона *Trusted Redirection* похоже на определение шаблона *Recursion*.

$$\begin{aligned} & \text{Redirector} < : \text{FamilyHead}, \\ & r : \text{Redirector}, \\ & fhr : \text{FamilyHead}, \\ & r.\text{operation} <_{\mu}^{-+} fhr.\text{operation} \end{aligned}$$

$$\text{TrustedRedirection}(\text{Redirector} : \text{Redirector}, \text{FamilyHead} : \text{FamilyHead}, \\ \text{target} : fhr, \text{operation} : \text{operation})$$

Формальное определение шаблона *Object Recursion* выглядит следующим образом.

```

Objectifier(Objectifier : Handler, ConcreteClass : Recursori ∈ I...m,
            operation : handleRequest)
Objectifier(Objectifier : Handler, ConcreteClass : Terminatori ∈ I...m,
            operation : handleRequest),
init.someMethod <μ obj.handleRequest,
init : Initiator,
obj: Handler,
TrustedRedirection(Redirector : Recursor, FamilyHead : Handler,
                    target : obj, operation: handleRequest)
!TrustedRedirection(Redirector : Terminator, FamilyHead : Handler,
                     target: obj, operation: handleRequest)


---


ObjectRecursion(Handler : Handler, Recursor: Recursori ∈ I...m,
                 Terminator : Terminatori ∈ I...m, Initiator : Initiator,
                 target : obj, operation : handleRequest, successor : obj)

```

Здесь указано, что некоторые классы *ConcreteClasses* шаблона *Objectifier* задействованы в шаблоне *Trusted Redirection*, а некоторые — нет. Если бы в переадресации участвовали все такие классы, то вызов в этой архитектуре никогда бы не завершился.

Далее мы будем использовать стандартное обозначение для метода класса *aMethod ∈ meth(aClass)* и определим шаблон *Extend Method* следующим образом.

```

operation ∈ meth(OriginalBehavior)
ExtendedBehaviour <: OriginalBehavior,
eb : ExtendedBehavior,
eb.operation <μ++ eb ^ operation

```

```

ExtendMethod(OriginalBehavior : OriginalBehavior,
             ExtendedBehavior : ExtendedBehavior,
             operation: operation)

```

В заключение определим шаблон *Decorator*.

```

ObjectRecursion(Handler : Component, Recursor : Decoratori ∈ I...m,
                 Terminator : ConcreteComponentj ∈ I...n, Initiator : any,
                 handleRequest : operatork ∈ I...o, successor :),
ExtendMethod(OriginalBehavior : Decorator,
             ExtendedBehavior : ConcreteDecoratork ∈ I...o,
             operation : operatork ∈ I...o),
!ExtendMethod(OriginalBehavior : Decorator,
              ExtendedBehavior : ConcreteDecoratorl ∈ I...p,
              operation : operatorl ∈ I...p)

```

```

Decorator(Component : Component, Decorator : Decoratori ∈ I...m
          ConcreteComponent : ConcreteDecoratorj ∈ I...n,
          ConcreteDecorreator : ConcreteDecoratorBk ∈ I...o,
          Terminator : ConcreteDecoratorAl ∈ I...p,
          operation : operatork ∈ I...o+p)

```

Обратите внимание на то, что эти выражения тесно связаны с PIN-диаграммами из раздела 4.2. Фактически, как будет показано ниже, система обозначений PIN и шаблоны проектирования в правилах редукции сильно коррелируют друг с другом.

A.6. Обозначения и роли в экземпляре шаблона

Система обозначений PIN предназначена для удобного изображения составных шаблонов с помощью компонентов PINbox. Правила построения диаграмм PIN отображают правила редукции, описанные выше. Например, рис. A.3 соответствует рис. 3.9, а экземпляры этих шаблонов в ρ-исчислении выглядят следующим образом: **PatternA(Role1 : a, Role2 : b)** и **PatternB(Role1 : a, Role2 : b, Role3 : c, Role4 : d, Role5 : e, Role6 : f)**, где *a*— обозначают свободные роли.

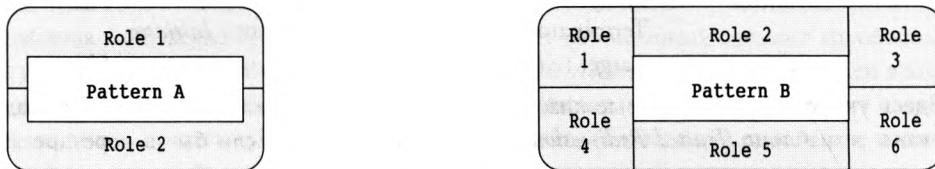


Рис. A.3. Стандартный компонент PINbox

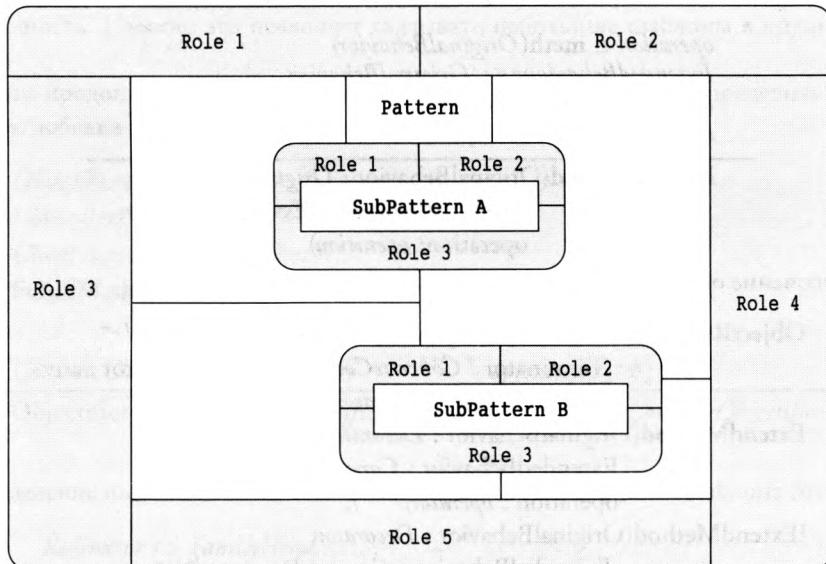


Рис. A.4. Развёрнутый экземпляр PIN

Аналогично на рис. A.4 показан экземпляр PINbox из рис. 3.14, который можно определить формально.

$$\begin{array}{c}
 \text{SubPatternA(Role1 : } x, \text{Role2 : } y, \text{Role3 : } z) \\
 \text{SubPatternB(Role1 : } z, \text{Role2 : } a, \text{Role3 : } b) \\
 \hline
 \text{Pattern(Role1 : } x, \text{Role2 : } y, \text{Role3 : } z, \text{Role4 : } a, \text{Role5 : } b)
 \end{array}$$

Существует однозначное соответствие между ролями, окружающими диаграмму PINbox, и ролями в формальном определении. Кроме того, каждая связь в диаграмме PINbox иллюстрирует точку зрения, в соответствии с которой одна и та же связанная переменная используется для исполнения нескольких ролей в определении.

A.7. Определения элементарных шаблонов проектирования

В этом разделе содержатся формальные определения всех элементарных шаблонов проектирования с помощью ρ-исчисления. Не все аспекты этих определений объясняются полностью, хотя большинство из них вы уже видели. Если вы хотите ознакомиться с ними глубже, обратитесь к первоисточникам, в частности к *A Theory of Objects* [1], где изложены основы, а также к *SPQR: Formal Foundations and Practical Support for the Automated Detection of Design Patterns from Source Code* [35], где приведены оригинальные определения и намного более подробные описания. По сравнению с первоисточниками мы внесли небольшие изменения, например явные имена ролей в экземплярах шаблонов. Это оказалось очень полезным, но выяснилось уже после первых публикаций. Каждое из следующих определений иллюстрируется эквивалентной PIN-диаграммой, чтобы читатели могли сразу увидеть соответствие между этими формализмами.

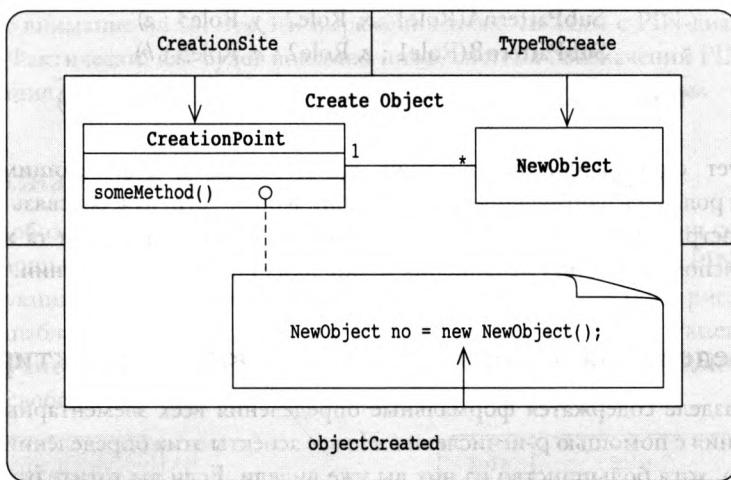
A.7.1. Шаблон *Create Object*

Это определение является ближайшим к основам ζ-исчисления. Если вы действительно хотите глубже понять это определение, мы рекомендуем изучить и основы. Впрочем, можно просто принять к сведению, что это очень специфическое определение очень простой концепции.

$$\begin{aligned}
 A &\equiv \text{Object}(X)[l_i v_i; B_i^{j \in 1..n+m}] \\
 \bar{A}: \text{Class}(A) &\triangleq \text{subclass of } \bar{A}': \text{Class}(A') \\
 &\quad \text{with (self : [A])} \\
 &\quad l_i = b_i^{j \in n+1..n+m} \\
 &\quad \text{override} \\
 &\quad l_i = b_i^{j \in Ovr} \\
 &\quad \text{end} \\
 a &= \text{new } \bar{A}
 \end{aligned}$$

$$\text{CreateObject}(\text{CreationSite} : l_i, \text{TypeToCreate} : A, \text{ObjectCreated} : a)$$

Здесь $A' = \text{Object}(X)[l_i v_i; B_i^{j \in 1..n}]$ (и может быть *Root*), $\lfloor A \rfloor = A$ для O–1, $X < : A$ для O–2 и $X < \#A$ для O–3–языков по классификации, принятой в [1].



Иногда намного проще осуществить аксиоматический вывод шаблона с помощью модели языка, чем использовать ζ -исчисление. В частности, если необходимо проанализировать исходный код, следует использовать языковую семантику, а не примитивы ζ -исчисления.

A.7.2. Шаблон *Retrieve*

Большинство объектно-ориентированных языков не позволяют модифицировать методы новыми телами. Вместо этого в них можно обновлять только поля. С теоретической точки зрения между этими сценариями нет заметной разницы. Мы допускаем оба варианта. Их выбор зависит от семантики языка. Откладывая определение до завершения изложения фактов, касающихся ρ -исчисления, мы избегаем массы сложностей. Существует две основные формы шаблона *Retrieve*.

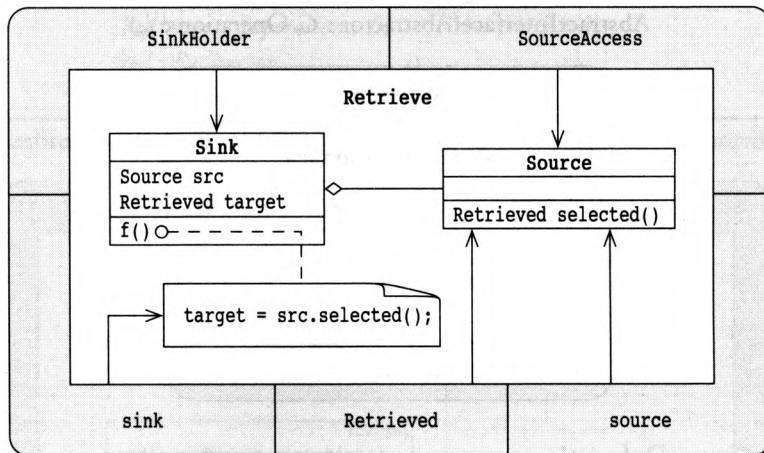
$$\begin{aligned}
 o &: \text{SinkHolder}, \\
 o' &: \text{SourceAccess}, \\
 o.s &\Leftarrow o'.s' \\
 x.ReturnType \\
 o'.s' &\xrightarrow{\nu} x
 \end{aligned}$$

Retrieve(SinkHolder : SinkHolder, sink : s , SourceAccess : o' ,
source : s' , Retrieved : ReturnType)

$$\begin{aligned}
 o &: \text{SinkHolder}, \\
 o' &: \text{SourceAccess}, \\
 o.s &\Leftarrow o'.s' \\
 x.ReturnType \\
 o'.s' &\xrightarrow{n} x
 \end{aligned}$$

Retrieve(SinkHolder : SinkHolder, sink : s , SourceAccess : o' ,
source : s' , Retrieved : ReturnType)

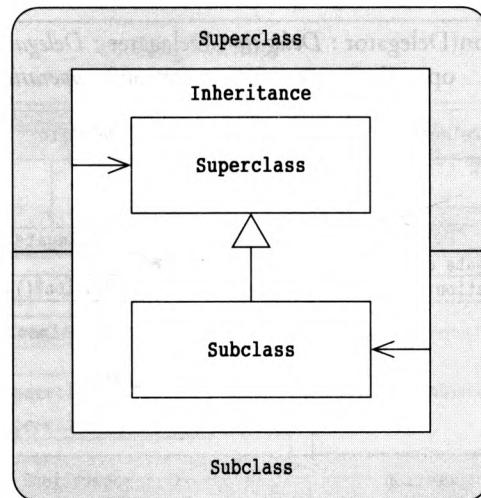
Обе формы подразумевают обновление в первом утверждении и возвращение значения во втором утверждении, либо по имени, что означает общую ссылку, либо по значению, что означает создание новой копии.



A.7.3. Шаблон *Inheritance*

$$E \vdash \text{Subclass}, \text{Superclass}, \text{Subclass} <: \text{Superclass}$$

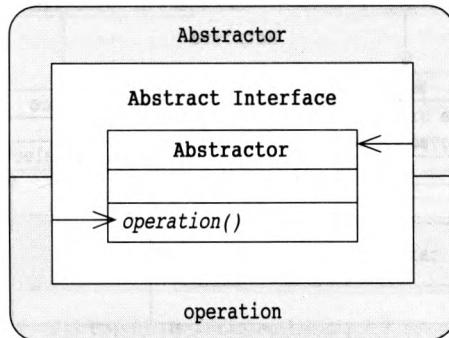
Inheritance(Superclass : Superclass, Subclass : Subclass)



A.7.4. Шаблон *Abstract Interface*

$$\begin{aligned} E \vdash C, C: [l_i : B_i^{i=1..n}] \\ C \equiv [l_i : B_i^{i=1..m-1, m+1..n}, l_m = []] \end{aligned}$$

AbstractInterface(Abstractor : C, Operation : l_m)



A.7.5. Шаблон *Delegation*

Delegator : [target : *Delegator*, operation : B_i]

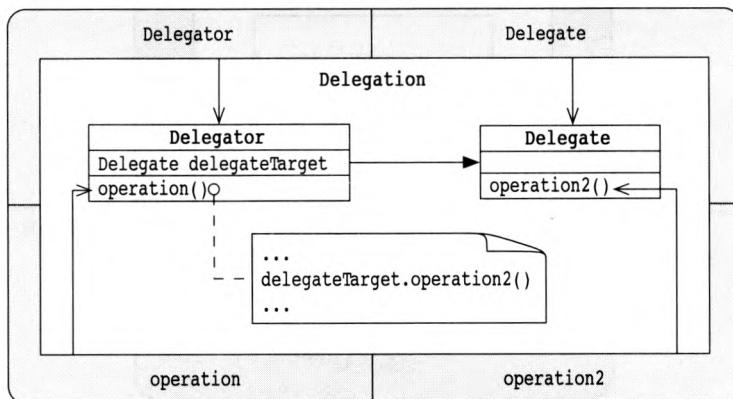
Delegator \equiv [operation $\Leftarrow \dots, target.operation2, \dots$]

Delegatee : [operation2 : B_i]

del : *Delegator*

del.operation $<_{\mu}^- target.operation2$

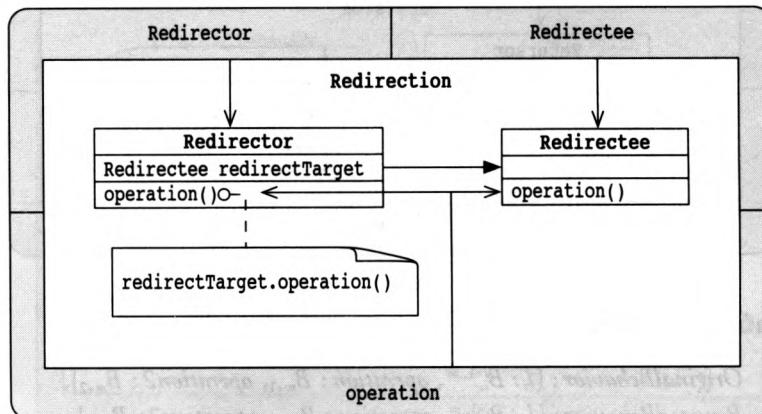
Delegation(Delegator : *Delegator*, Delegatee : *Delegatee*, operation : operation, operation2 : operation2)



A.7.6. Шаблон *Redirection*

$\text{Redirector} : [\text{target} : \text{Redirectee}, \text{operation} : B]$
 $\text{Redirector} \equiv [\text{operation} \Leftarrow \dots, \text{target}.\text{operation}, \dots]$
 $\text{Redirectee} : [\text{operation} : B]$
 $\text{red} : \text{Redirector}$
 $\text{redirector}.\text{operation} <_{\mu}^{\sim+} \text{target}.\text{operation}$

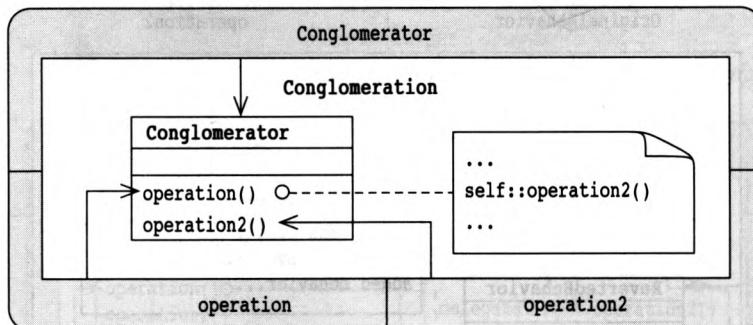
Redirection(Redirector : Redirector, target : target, operation : operation)



A.7.7. Шаблон *Conglomeration*

$c : \text{Conglomerator}$
 $c.\text{operation} <_{\mu}^{+-} c.\text{operation2}$

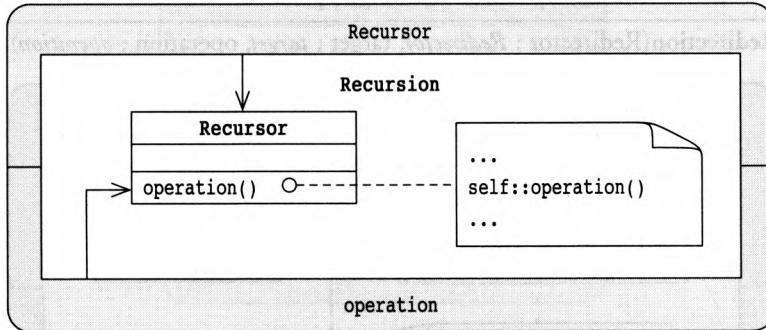
Conglomeration(Conglomerator : Conglomerator, operation : operation, operation2 : operation2)



A.7.8. Шаблон *Recursion*

Recurser: [$[l_i : B_i^{i=1..m}]$, *operation* : B_{m+1}],
r : *Recurser*,
r.operation $<_{\mu}^{++}$ *r.operation*

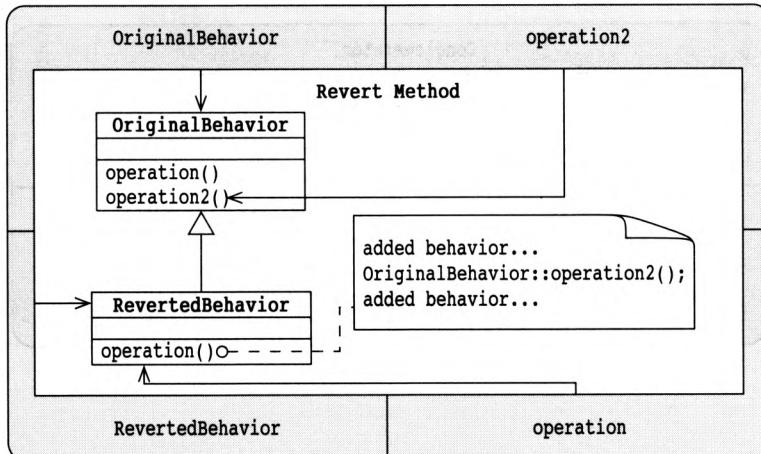
Recursion(Recurser : Recursor, operation : operation)



A.7.9. Шаблон *Revert Method*

OriginalBehavior: [$[l_i : B_i^{i=1..m}]$, *operation* : B_{m+1} , *operation2* : B_{m+2}],
RevertedBehavior: [$[l_i : B_i^{i=1..m}]$, *operation* : B_{m+1} , *operation2* : B_{m+2}],
RevertedBehavior <: *OriginalBehavior*,
rb : *RevertedBehavior*,
rb.operation $<_{\mu}^{+-}$ *rb^operation2*

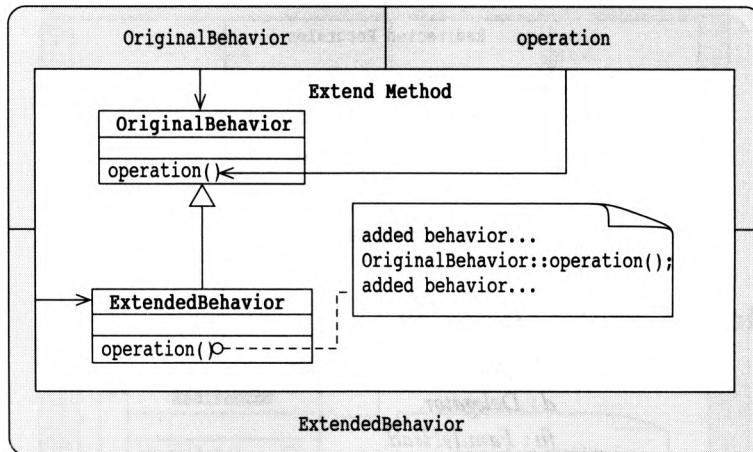
RevertMethod(*OriginalBehavior* : *OriginalBehavior*,
***RevertedBehavior* : *RevertedBehavior*,**
***operation* : *operation*, *operation2* : *operation2*)**



A.7.10. Шаблон *Extend Method*

$operation \in \mathbf{meth}(\mathit{OriginalBehavior})$,
 $\mathit{ExtendedBehavior} <: \mathit{OriginalBehavior}$,
 $eb : \mathit{ExtendedBehavior}$,
 $eb.\mathit{operation} <_{\mu}^{+ +} eb^{\wedge} operation$

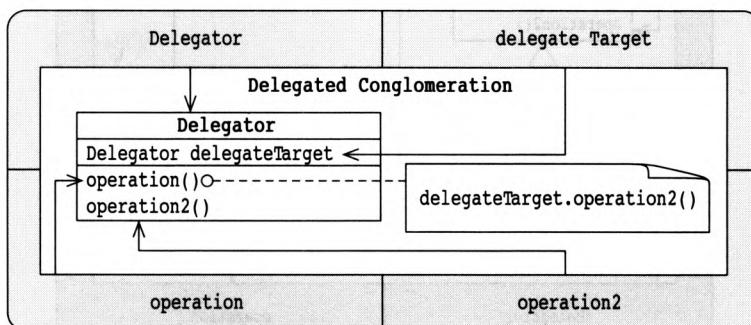
ExtendMethod($\mathit{OriginalBehavior} : \mathit{OriginalBehavior}$,
 $\mathit{ExtendedBehavior} : \mathit{ExtendedBehavior}$, $operation : operation$)



A.7.11. Шаблон *Delegated Conglomeration*

$del : DelConglomerator$
 $\mathit{delegateTarget} : DelConglomerator$
 $del.\mathit{operation} <_{\mu}^{--} \mathit{delegateTarget}.operation2$

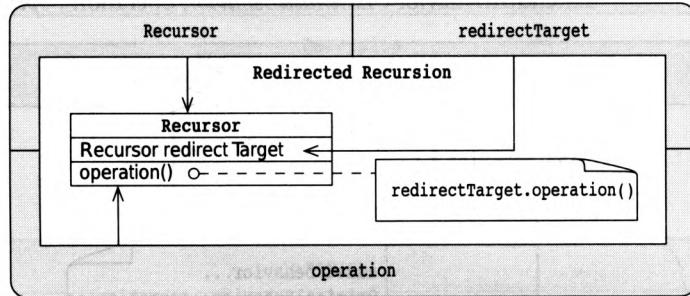
DelegatedConglomeration($\mathit{Delegator} : DelConglomerator$,
 $\mathit{delegateTarget} : \mathit{delegateTarget}$,
 $operation : operation$, $operation2 : operation2$)



A.7.12. Шаблон *Redirected Recursion*

rec : Recursor
redirectTarget : Recursor
 $rec.operation <_{\mu}^{+} redirectTarget.operation$

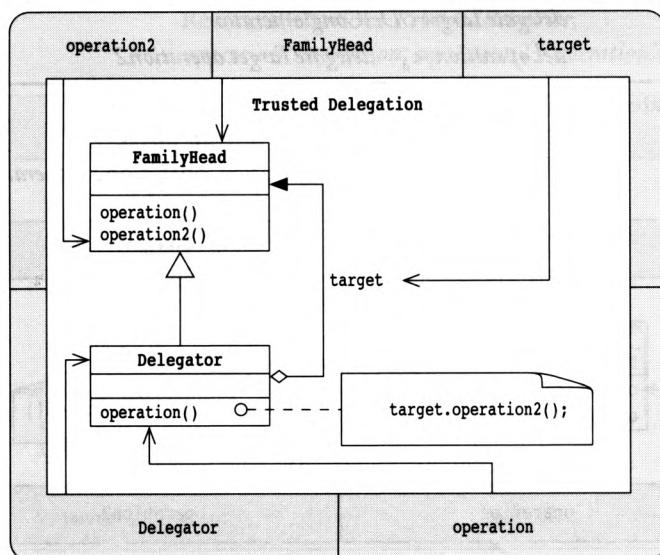
RedirectedRecursion(*Recursor* : *Recursor*,
redirectTarget : *redirectTarget*, *operation* : *operation*)



A.7.13. Шаблон *Trusted Delegation*

Delegator < : FamilyHead,
d : Delegator,
fh : FamilyHead,
 $d.operation <_{\mu}^{+} fh.operation2$,

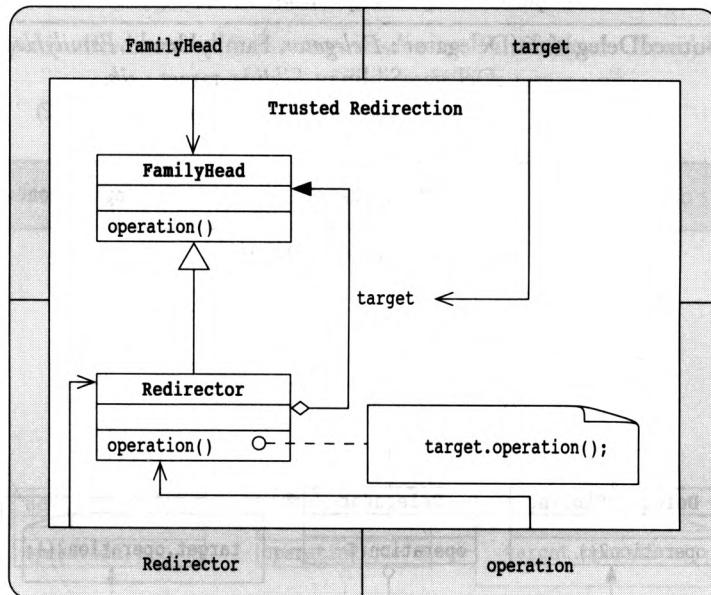
TrustedDelegation(*Delegator* : *Delegator*, *FamilyHead* : *FamilyHead*,
target : *fh*, *operation* : *operation* *operation2* : *operation2*)



A.7.14. Шаблон *Trusted Redirection*

$\text{Redirector} <: \text{FamilyHead}$,
 $r : \text{Redirector}$,
 $\text{fb} : \text{FamilyHead}$,
 $r.\text{operation} <_{\mu}^{-+} \text{fb}.\text{operation}$,

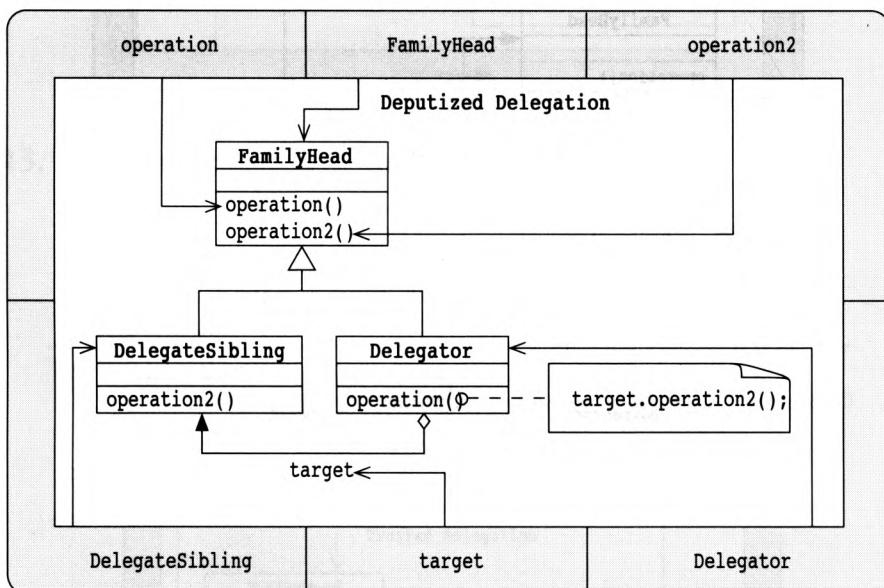
TrustedRedirection($\text{Redirector} : \text{Redirector}$, $\text{FamilyHead} : \text{FamilyHead}$,
 $\text{target} : \text{fb}$, $\text{operation} : \text{operation}$)



A.7.15. Шаблон *Deputized Delegation*

$Delegator <: FamilyHead,$
 $Sibling <: FamilyHead,$
 $Delegator \neq Sibling,$
 $Delegator \leftarrow Sibling,$
 $d : Delegator,$
 $sib : Sibling,$
 $d.operation <_{\mu}^{--} sib.operation2,$

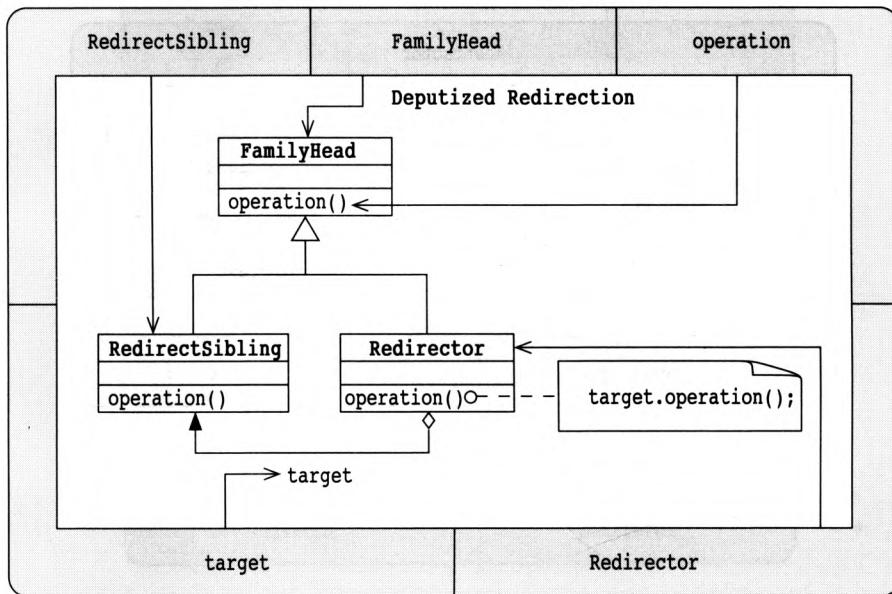
DeputizedDelegation(*Delegator* : *Delegator*, *FamilyHead* : *FamilyHead*,
DelegateSibling : *Sibling*, *target* : *sib*,
operation : *operation*, *operation2* : *operation2*)



A.7.16. Шаблон *Deputized Redirection*

$\text{Redirector} <: \text{FamilyHead}$,
 $\text{Sibling} <: \text{FamilyHead}$,
 $\text{Redirector} \neq \text{Sibling}$,
 $\text{Redirector} \not<: \text{Sibling}$,
 $r: \text{Redirector}$,
 $sib: \text{Sibling}$,
 $r.\text{operation} <_{\mu}^{\sim} sib.\text{operation}$,

DeputizedRedirection($\text{Redirector} : \text{Redirector}$, $\text{FamilyHead} : \text{FamilyHead}$,
 $\text{RedirectSibling} : \text{Sibling}$, $\text{target} : sib$,
 $\text{operation} : operation$)



A.8. Определения промежуточных шаблонов

В этом разделе приведены определения шаблонов, описанных в главе 6. Это первичные шаблоны, демонстрирующие правила редукции композиций, состоящих из элементарных шаблонов. Каждое определение, как и ранее, сопровождается PIN-диаграммой.

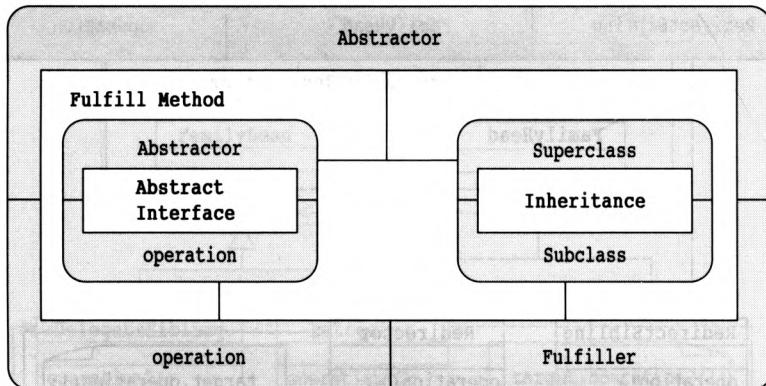
A.8.1. Шаблон *Fulfill Method*

AbstractInterface(*Abstractor* : *Abstractor*, *operation* : *operationn*),

Inheritance(*Superclass* : *Abstractor*, *Subclass* : *ConcreteClass*),

ConcreteClass \equiv [*operation_i* \Leftarrow $b_i^{i \in 1..n-1, n+1..m}$, *operation_n* \Leftarrow b^n]

FulfillMethod(*Abstractor* : *Abstractor*, *Fulfiller* : *ConcreteClass*,
operation : *operationn*)



A.8.2. Шаблон *Retrieve New*

$o : SinkHolder$

$o' : SourceAccess$

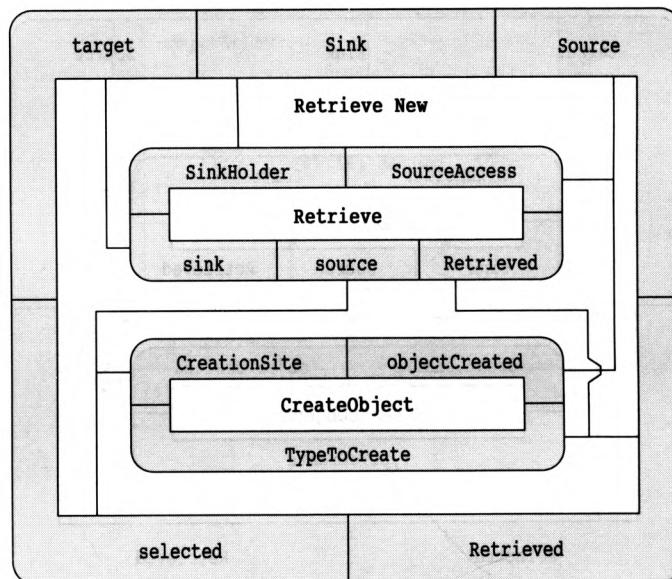
$x : TypeToCreate$

$o'.s' \not\rightarrow x$

$\text{CreateObject}(\text{CreationSite} : o'.s', \text{TypeToCreate} : TypeToCreate,$
 $\quad \text{ObjectCreated} : o'.s'.x)$

$\text{Retrieve}(\text{SinkHolder} : SinkHolder, \text{sink} : s, \text{SourceAccess} : SourceAccess,$
 $\quad \text{source} : s', \text{Retrieved} : TypeToCreate)$

$\text{RetrieveNew}(\text{Sink} : SinkHolder, \text{target} : s, \text{Retrieved} : TypeToCreate,$
 $\quad \text{Source} : o, \text{selected} : s')$



A.8.3. Шаблон *Retrieve Shared*

$o : SinkHolder$

$o' : SourceAccess$

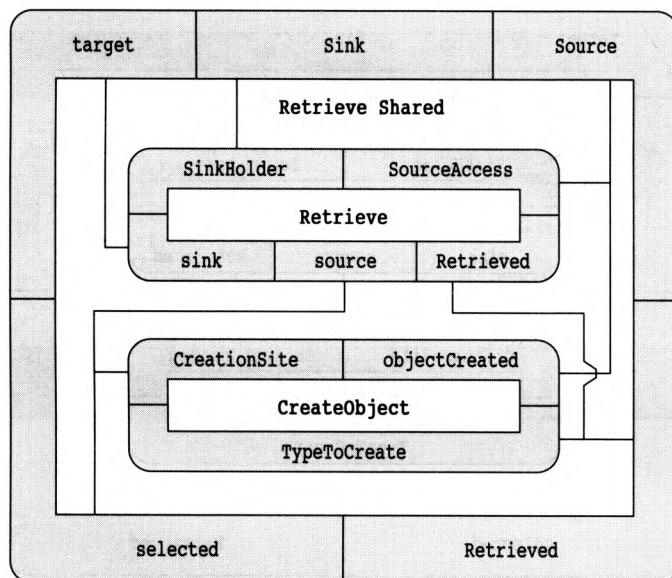
$x : TypeToCreate$

$o'.s' \xrightarrow{v} x$

CreateObject(*CreationSite* : $o'.s'$, *TypeToCreate* : *TypeToCreate*,
ObjectCreated : $o'.s'.x$)

Retrieve(*SinkHolder* : *SinkHolder*, *sink* : *s*, *SourceAccess* : *SourceAccess*,
source : s' , *Retrieved* : *TypeToCreate*)

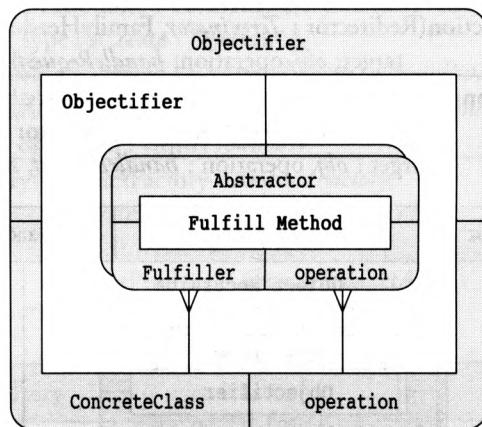
RetrieveShared(*Sink* : *SinkHolder*, *target* : *s*, *Retrieved* : *TypeToCreate*,
Source : o' , *selected* : s')



A.8.4. Шаблон *Objectifier*

ObjectifierBase : [$l_i : B_i^{i=1..n}$]
Client : [ref: *Objectifier*],
Client.someMethod <_μ *Client.ref.l_i*,
FulfillMethod(*Abstractor* : *ObjectifierBase*,
 Fulfiller : *ConcreteObjectifier*,
 operation : $l_j^{j=1..n}$,

Objectifier(*Objectifier* : *ObjectifierBase*,
 ConcreteClass : *ConcreteObjectifier*, *operation* : l_i)



A.8.5. Шаблон *Object Recursion*

Objectifier(Objectifier : Handler, ConcreteClass : Recursor_i^{i=1..m},
operation : handleRequest)

Objectifier(Objectifier : Handler, ConcreteClass : Terminator_i^{i=1..m},
operation : handleRequest),

init.someMethod <_μ obj.handleRequest,

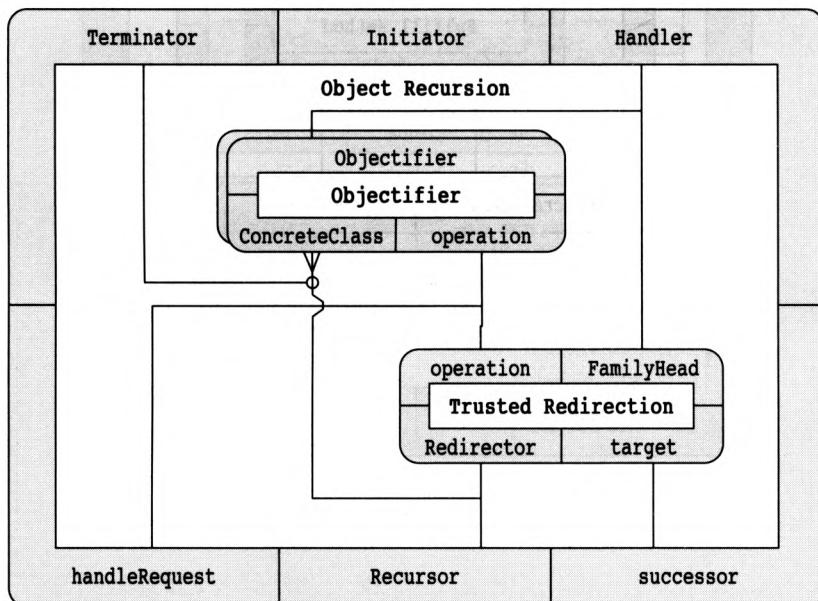
init : Initiator,

obj: Handler,

TrustedRedirection(Redirector : Recursor, FamilyHead : Handler,
target : obj, operation: handleRequest)

!TrustedRedirection(Redirector : Terminator, FamilyHead : Handler,
target: obj, operation: handleRequest)

ObjectRecursion(Handler : Handler, Recursor: Recursor_i^{i=1..m},
Terminator : Terminator_i^{i=1..m}, Initiator : Initiator,
target : obj, operation : handleRequest, successor : obj)



A.9. Шаблоны *Gang of Four*

В заключение приведены определения и PIN-диаграммы шаблонов *Gang of Four*, описанных в главе 7.

A.9.1. Шаблон *Abstract Factory*

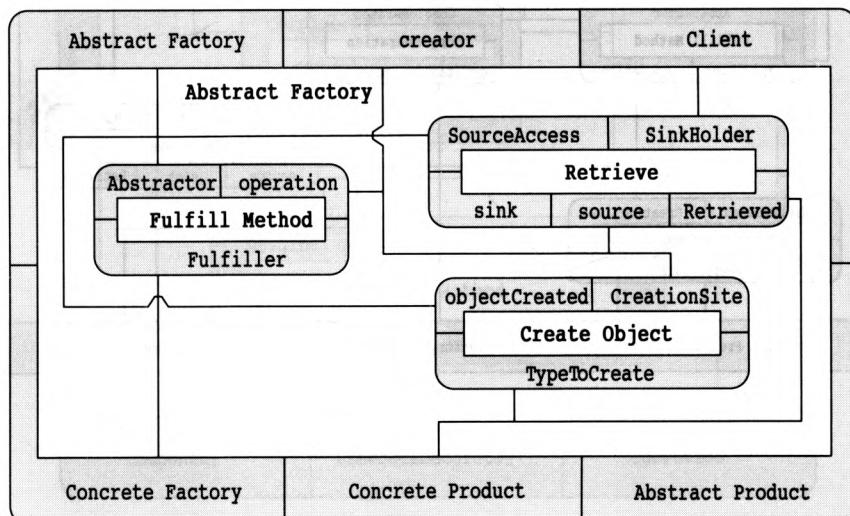
```

Client.prodA : AbstractProductA
ProductA2 < : AbstractProductA
Retrieve(SinkHolder : Client, sink : someMethod, SourceAccess : Client,
         source : prodA, Retrieved : ProductA2)
CreateObject(CreationSite : ConcreteFactory2.createProductA,
             TypeToCreate : ProductA2,
             objectCreated : ConcreteFactory2.CreateProductA.rtnVal)
FulfillMethod(Abstractor : AbstractFactory, Fulfiller : ConcreteFactory2,
              operation : createProductA)


---


AbstractFactory(AbstractFactory : AbstractFactory,
                ConcreteFactory : ConcreteFactory2,
                creator : createProductA,
                AbstractProduct : AbstractProductA,
                ConcreteProduct : ProductA2)

```



A.9.2. Шаблон *Factory Method*

Conglomeration(Conglomerator : *Creator*, operation : *anOperation*,
operation2 : factoryMethod)

FulfillMethod(Abstractor : *Creator*, Fulfiller : *ConcreteCreator*,
operation : factoryMethod)

Creator.product : *Product*

ConcreteCreator.factoryMethod ~~~ *RetVal*

Retrieve(SinkHolder : *Creator*, sink : *operation.any*,

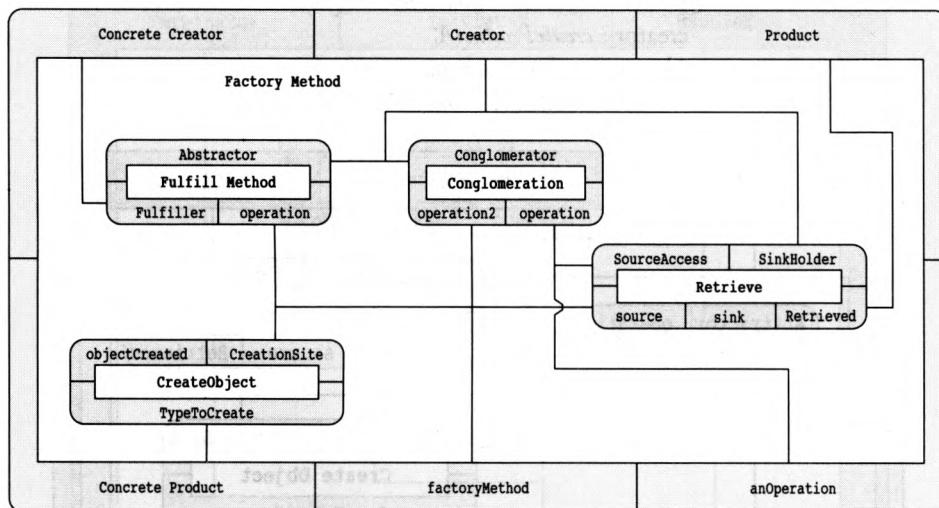
SourceAccess : *Creator*, source : *operation2*, Retrieved : *Product*)

ConcreteProduct < : *Product*

CreateObject(CreationSite : *ConcreteCreator.factoryMethod*,

TypeToCreate : *ConcreteProduct*, objectCreated : *RetVal*)

FactoryMethod(Creator : *Creator*, ConcreteCreator : *ConcreteCreator*,
anOperation : anOperation, factoryMethod : *factoryMethod*,
Product : Product, ConcreteProduct : *ConcreteProduct*)



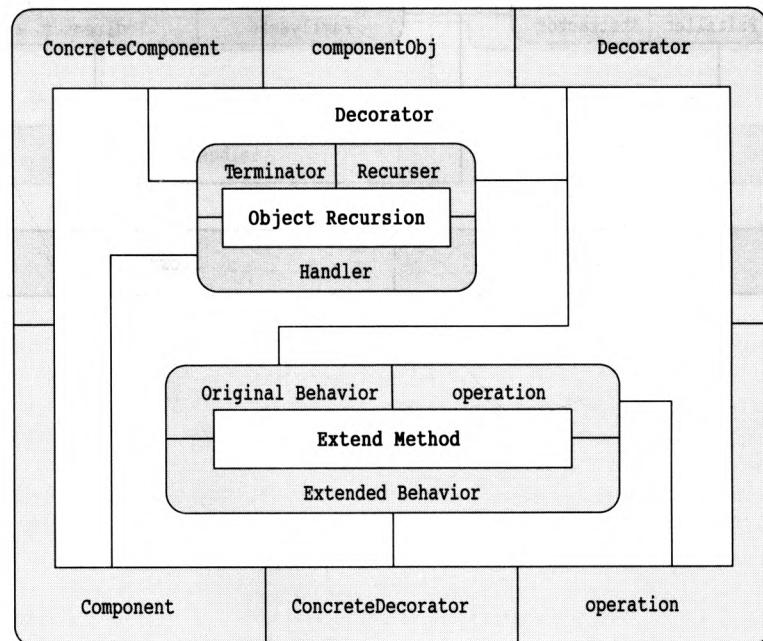
A.9.3. Шаблон *Decorator*

ObjectRecursion(Handler : *Component*, Recursor : *Decorator*_i^{i ∈ 1..m}, Terminator : *ConcreteComponent*_j^{j ∈ 1..n}, Initiator : *any*, handleRequest : *operator*_k^{k ∈ 1..o}, successor :),

ExtendMethod(OriginalBehavior : *Decorator*, ExtendedBehavior : *ConcreteDecorator*_k^{k ∈ 1..o}, operation : *operator*_k^{k ∈ 1..o}),

!ExtendMethod(OriginalBehavior : *Decorator*, ExtendedBehavior : *ConcreteDecorator*_k^{k ∈ 1..p}, operation : *operator*_k^{k ∈ 1..p})

Decorator(Component : *Component*, Decorator : *Decorator*_i^{i ∈ 1..m}, ConcreteComponent : *ConcreteDecorator*_j^{j ∈ 1..n}, ConcreteDecororeator : *ConcreteDecoratorB*_k^{k ∈ 1..o}, Terminator : *ConcreteDecoratorA*_l^{l ∈ 1..p}, operation : *operator*_k^{k ∈ 1..o+p})



A.9.4. Шаблон *Proxy*

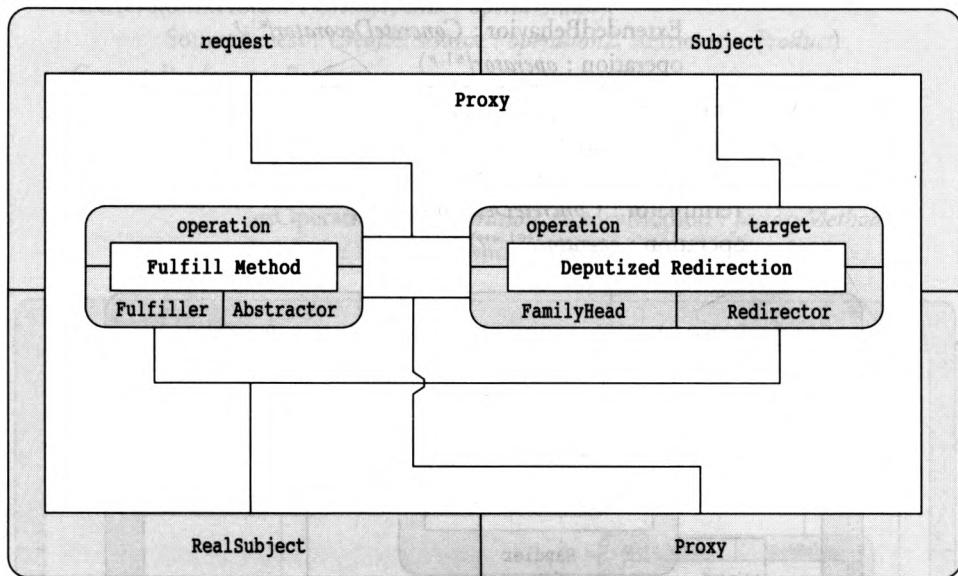
FulfillMethod(Abstractor : *Subject*, Fulfiller : *Proxy*, operation : *request*)

FulfillMethod(Abstractor : *Subject*, Fulfiller : *RealSubject*, operation : *request*)

DeputizedRedirection(Redirector : *Proxy*, RedirectSibling : *RealSubject*,

FamilyHead : *Subject*, target : *any*, operation : *request*)

Proxy(Subject : *Subject*, Proxy : *Proxy*, RealSubject : *RealSubject*,
request : *request*)

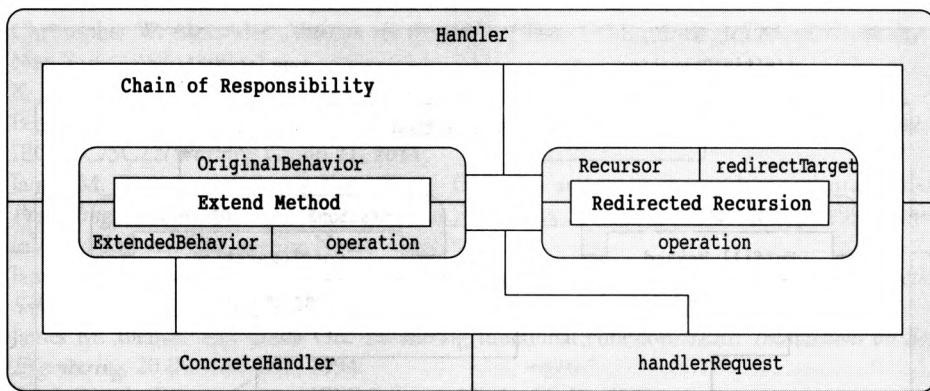


A.9.5. Шаблон *Chain of Responsibility*

RedirectedRecursion(*RecurSOR* : *Handler*, *redirectTarget* : *successor*,
operation : *handleRequest*)

ExtendMethod(*OriginalBehavior* : *Handler*,
 ExtendedBehavior : *ConcreteHandler*,
operation : *handleRequest*)

ChainOfResponsibility(*Handler* : *Handler*,
 ConcreteHandler : *ConcreteHandler*,
handleRequest : *handleRequest*)

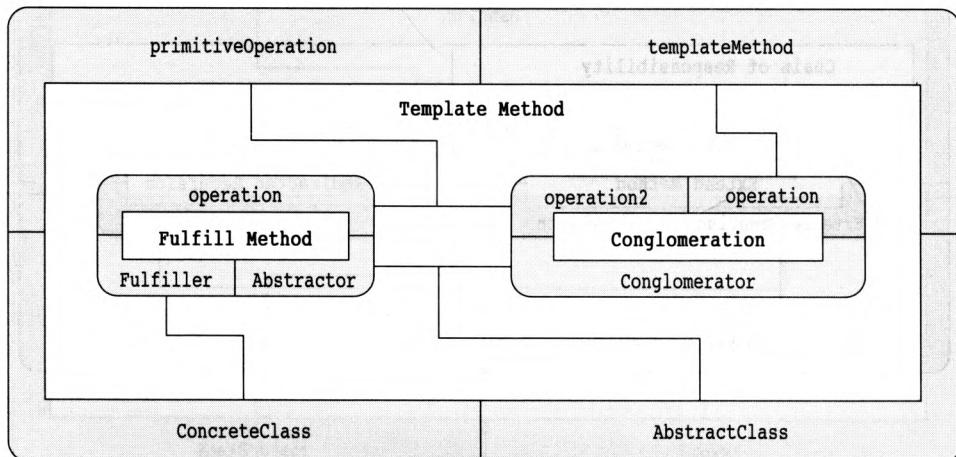


A.9.6. Шаблон *Template Method*

FulfillMethod(Abstractor : *AbstractClass*, Fulfiller : *ConcreteClass*,
 operation : *primitiveOperation*)

Conglomeration(Conglomerator : *AbstractClass*, operation : *templateMethod*,
 operation2 : *primitiveOperation*)

TemplateMethod(AbstractClass : *AbstractClass*,
 ConcreteClass : *ConcreteClass*,
 templateMethod : *templateMethod*,
 primitiveOperation : *primitiveOperation*)



Библиография

1. Marten Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.
2. Lee Ackerman and Celso Gonzalez. *Patterns-Based Engineering: Successfully Delivering Solutions via Patterns*. Addison-Wesley, Boston, 2010.
3. Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Building, Construction*. Oxford University Press, New York, 1977.
4. Christopher W. Alexander. *Notes on the Synthesis of Form* (15th printing). Oxford University Press, New York, 1964, 1999.
5. Kent Beck. *Smalltalk Best Practice Patterns*. Prentice Hall, Upper Saddle River, NJ, 1997.
6. Pete Becker. Working draft, standard for programming language c++. Technical Report N3242, ISO/IEC JTC/SC22/Working Group 21, 2011.
7. James M. Bieman and Byung-Kyoo Kang. Cohesion and reuse in an object-oriented system. In *Proceedings of the ACM Symposium on Software Reusability, SSR'95*, pp. 259–262, Apr 1995. Reprinted in ACM Software Engineering Notes, Aug 1995.
8. James M. Bieman and Byung-Kyoo Kang. Measuring design-level cohesion. *IEEE Transactions on Software Engineering*, 24 (2): 111–124, 1998.
9. James M. Bieman and Linda Ott. Measuring functional cohesion. *IEEE Transactions on Software Engineering*, 20 (8): 644–657, 1994.
10. Grady Booch. Tribal memory. *IEEE Software*, 25 (2): 16–17, 2008.
11. Grady Booch and Celso Gonzalez. Handbook of software architecture. www.handbookofsoftware-architecture.com/, Oct 2010.
12. Jan Bosch. Design patterns as language constructs. *Journal of Object Oriented Programming*, 1 (2): 18–52, May 1998.
13. L. C. Briand and J. W. Daly. A unified framework for cohesion measurement in object-oriented systems. In *Proceedings of the Fourth Conference on METRICS'97*, pp. 43–53, Nov 1997.
14. William J. Brown, Raphael C. Malveau, William H. Brown, W. McCormick Hays III, and Thomas J. Mowbray. *Anti-Patterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley, New York, 1998.
15. Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object-oriented design. In *Proceedings of OOPSLA '91*, pp. 197–211. ACM, 1991.
16. Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20 (6): 476–493, 1994.
17. James Coplien. C++ idioms. In *Proceedings of the Third European Conference on Pattern Languages of Programming and Computing*, 1998.
18. C++ Standards Committee. Working Draft, Standard for Programming Language C++, Section 20.7.2.2 [util.smartptr.shared]. ISO/IEC Document N3242=11-0012, Feb 2011.
19. Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, Boston, 1999.
20. Martin Fowler and Kendall Scott. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3rd ed. Addison-Wesley, Boston, 2003.
21. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, Boston, 1995.

22. Byung-Kyoo Kang and James M. Bieman. Design-level cohesion measures: Derivation, comparison, and applications. In *Proceedings of the 20th International Computer Software and Applications Conference (COMPSAC'96)*, pp. 92–97, Aug 1996.
23. Byung-Kyoo Kang and James M. Bieman. Using design cohesion to visualize, quantify and restructure software. In *Eighth International Conference of Software Engineering and Knowledge Engineering, SEKE '96*, Jun 1996.
24. Joshua Kerievsky. *Refactoring to Patterns*. Addison-Wesley, Boston, 2005.
25. Howard C. Lovatt, Anthony M. Sloane, and Dominic R. Verity. A pattern enforcing compiler (PEC) for Java: Using the compiler. In Sven Hartmann and Markus Stumptner, eds., *Conferences in Research and Practice in Information Technology*, vol. 43. Appeared at the Second Asia-Pacific Conference on Conceptual Modeling (APCCM2005), 2005.
26. O. L. Madsen, B. Miller-Pederson, and K. Nygaard. *Object-Oriented Programming in the BETA Language*. Addison-Wesley, Boston, 1993.
27. Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, Upper Saddle River, NJ, 2008.
28. Scott Meyers. *Effective C++*. Addison-Wesley, Boston, 1992.
29. Scott Meyers. *More Effective C++*. Addison-Wesley, Boston, 1996.
30. Jörg Niere, Lothar Wendehals, and Albert Zündorf. An interactive and scalable approach to design pattern recovery. Technical Report tr-ri-03-236, University of Paderborn, Paderborn, Germany, Jan 2003.
31. S. Patel, W. Chu, and R. Baxter. A measure for composite module cohesion. In *International Conference on Software Engineering*, pp. 38–48, May 1992.
32. Dirk Riehle. Composite design patterns. In *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 218–228. ACM Press, 1997.
33. James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual, 2nd ed.*, Addison-Wesley, Boston, 2004.
34. M. H. Samadzadeh and S. J. Khan. Stability, coupling and cohesion of object-oriented software systems. In *Proceedings of the 22nd Annual ACM Computer Science Conference on Scaling Up*, pp. 312–319, Mar 8–10, 1994.
35. Jason McC. Smith. *SPQR: Formal Foundations and Practical Support for the Automated Detection of Design Patterns from Source Code*. Ph.D. thesis, University of North Carolina at Chapel Hill, Dec 2005.
36. Jason McC. Smith. The Pattern Instance Notation: A simple hierarchical visual notation for the dynamic visualization and comprehension of software patterns. *Journal of Visual Languages and Computing*, 22 (5): 355–374, 2011.
37. Jason McC. Smith and David Stotts. SPQR: Flexible automated design pattern extraction from source code. In *18th IEEE International Conference on Automated Software Engineering*, pp. 215–224, Oct 2003.
38. Jason McC. Smith and David Stotts. *Intent-Oriented Design Pattern Formalization Using SPQR*, chapter 7. IDEA Group, 2007.
39. Mark Twain. *The Adventures of Tom Sawyer*. Gutenberg Press, www.gutenberg.org/ebooks/74, 2005.
40. Bobby Woolf. The abstract class pattern. In Neil Harrison, Brian Foote, and Hans Rohnert, eds., *Pattern Languages of Program Design 4*. Addison-Wesley, Boston, 1998.
41. Bobby Woolf. The object recursion pattern. In Neil Harrison, Brian Foote, and Hans Rohnert, eds., *Pattern Languages of Program Design 4*. Addison-Wesley, Boston, 1998, 41–52.
42. E. Yourdon and L. Constantine. *Structured Design*. Prentice Hall, Englewood Cliffs, NJ, 1979.
43. Walter Zimmer. Relationships between design patterns. In James O. Coplien and Douglas C. Schmidt, eds., *Pattern Languages of Program Design*. Addison-Wesley, Boston, 1995, 345–364.

Предметный указатель

A

Анализ

процедурный, 114

Антишаблон, 97

З

Зависимость

“метод–метод”, 45

“метод–поле”, 45

“поле–метод”, 45

“поле–поле”, 45

Замещение, 132

И

Изотоп, 260

Изотоп шаблона, 84

Инкапсуляция, 121

К

Каталог

EDP, 33

Класс, 38

Компонент PINbox, 65

раскрытый, 70

свернутый, 65

стандартный, 67

стек, 72

Л

Лямбда-исчисление, 259

М

Метод, 38

Метрика

относительная плотность абстракций, 114

плотность абстракций, 113

ясность выражения, 112

О

Область видимости, 38; 259

Объект, 43

Оператор

зависимости, 258

\, 260

определения типа, 260

точки, 259

Отношение

вызов метода, 44

задание поля методом, 44

использование поля методом, 44

Ошибка

завышения на единицу, 133

занизжения на единицу, 133

поста охранения, 133

П

Пилинг, 77

Подкласс, 131

Поле, 43

Правило

композиции, 265

редукции, 261

Проект, 37

контекст, 37

отношения, 38

проблема, 37

решение, 37

сотрудник, 37

участник, 37

Проектирование

неосознанное, 18

осознанное, 18

Пространство проекта, 48

Р

Рекурсия, 164

взаимная, 165

Рефакторинг, 96

Extract Method, 98

Move Method, 98

Refactor, 98

Rename Method, 98

ρ-исчисление, 257

С

Связанность, 49

Связность, 49

Связывание, 259

Сигма-исчисление, 258

Система

PIN, 61

SPQR, 30

Суперкласс, 131
 Сходство, 262
 между методами, 47
 между объектами, 47
 между типами, 47

Т

Тип, 43
 Транзитивность, 260

У

Указатель
 d-указатель, 121
 на реализацию, 121
 непрозрачный, 121

Ф

Форма
 каппа, 260
 мю, 260
 сигма, 260
 фи, 260
 Функция, 38

III

Шаблон, 24
 Шаблон проектирования, 17
 Calls, 144
 Decomposing Message, 155
 Defer Implementation, 139
 Extending Super, 174
 GoF
 Abstract Factory, 72; 242; 285
 Chain of Responsibility, 251; 289
 Decorator, 72; 247; 287
 Factory Method, 245
 Proxy, 248; 288
 Template Method, 252; 290
 Helper Methods, 155
 Intention Revealing Selector, 47
 Instantiation, 119
 IsA, 131
 Messaging, 144
 Method Invocation, 144
 Object Recursion, 36; 92
 Objectifier, 35
 Polymorphism, 139
 Shop Foreman, 149
 The Executive, 144
 Tom Sawyer, 149

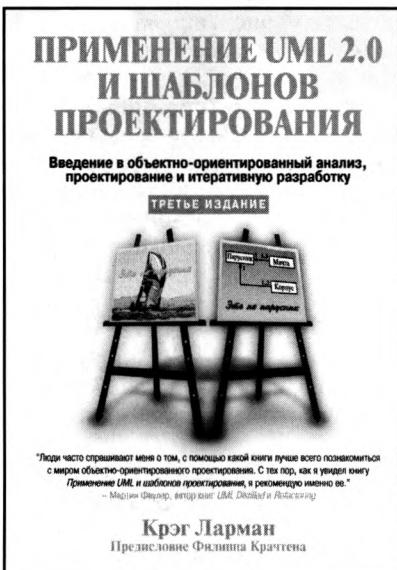
Type Reuse, 131
 Virtual Method, 139
 злокачественный, 97
 промежуточный
 Fulfill Method, 217; 280
 Objectifier, 229; 283
 Object Recursion, 284
 Retrieve New, 221; 281
 Retrieve Shared, 225; 282
 раздел спецификации, 117
 Мотивация, 117
 Назначение, 117
 Отношения, 117
 Реализация, 117
 Результаты, 117
 Участники, 117
 роль, 62
 участник, 62
 частичный, 97
 элементарный
 Abstract Interface, 58; 80; 139; 272
 Conglomeration, 52; 155; 273
 Create Object, 57; 119; 127; 269
 Delegated Conglomeration, 102; 179; 275
 Delegation, 51; 272
 Delegation Behavioral, 144
 Deputized Delegation, 104; 202; 278
 Deputized Redirection, 208; 279
 Extend Method, 56; 92; 174; 275
 Factory Method, 286
 Fulfill Method, 81
 Inheritance, 58; 80; 271
 Inheritance Relation, 131
 категория Method Invocation, 118
 категория Object Elements, 118
 категория Type Relation, 118
 Recursion, 50; 160; 274
 Redirected Recursion, 184; 190; 276
 Redirection, 51; 149; 273
 Retrieve, 58; 127; 270
 Revert Method, 103; 166; 274
 Trusted Delegation, 103; 276
 Trusted Redirection, 90; 196; 277
 ятрогенный, 97

Я

Язык UML
 дескриптор "шаблон-роль", 63
 элемент взаимодействия, 62

ПРИМЕНЕНИЕ UML 2.0 И ШАБЛОНОВ ПРОЕКТИРОВАНИЯ, третье издание

Крэг Ларман



www.williamspublishing.com

В книге вы найдете новые сведения об итеративном и гибком моделировании, шаблонах проектирования GRASP и GoF, прецедентах, архитектурном анализе и многоуровневой архитектуре, а также рефакторинге, разработке на основе обязанностей и многих других вопросах. Весь материал рассматривается в контексте использования унифицированного процесса (UP) как легкого и гибкого подхода к разработке совместно с приемами из других итеративных методов, таких как XP и Scrum.

Данная книга будет прекрасным руководством для как для новичков, так и для специалистов, кто интересуется вопросами ООА/П, языком моделирования UML 2 и самыми современными эволюционными подходами к разработке программного обеспечения.

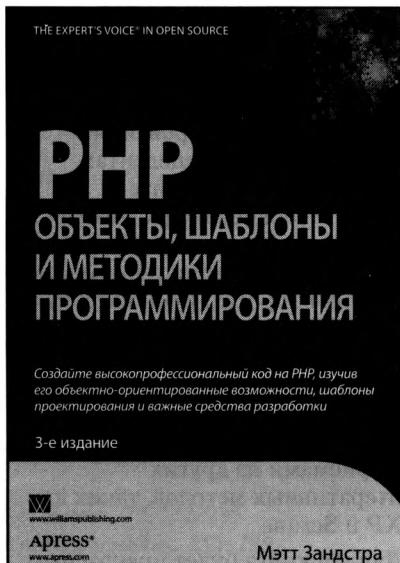
ISBN 978-5-8459-1185-8 в продаже

PHP

ОБЪЕКТЫ, ШАБЛОНЫ И МЕТОДИКИ ПРОГРАММИРОВАНИЯ

3-Е ИЗДАНИЕ

Мэтт Зандстра



www.williamspublishing.com

ISBN 978-5-8459-1689-1 в продаже

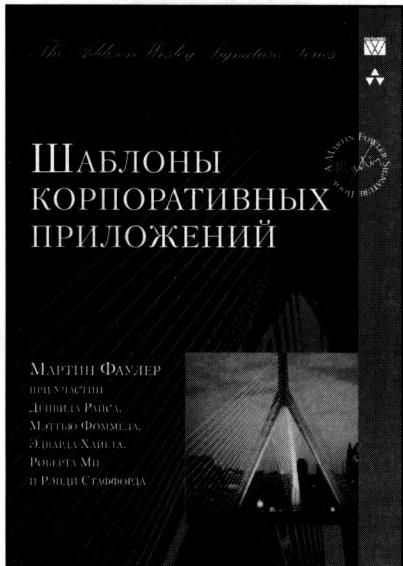
За последние десять лет PHP буквально охватила объектно-ориентированная революция, причем это относится как к самим средствам языка, так и к разработчикам, использующим эти средства, и к приложениям, которые они создают. Теперь основной акцент делается на объектах и объектно-ориентированном подходе к проектированию.

Книга начинается с обзора объектно-ориентированных возможностей PHP, в который включены важные темы, такие как определение класса, создание объектов, наследование, инкапсуляция методов и свойств. Вы изучите также и дополнительные темы: статические методы и свойства, абстрактные классы, обработка исключений, клонирование объектов, пространства имён, механизм замыканий и многое другое.

Следующая часть книги посвящена шаблонам проектирования, которые органически дополняют тему ООП и являются описанием элегантных решений распространенных проблем, возникающих при проектировании программного обеспечения. В ней описываются концепции шаблонов проектирования и показаны способы реализации нескольких важных шаблонов в приложениях на PHP.

ШАБЛОНЫ КОРПОРАТИВНЫХ ПРИЛОЖЕНИЙ

Мартин Фаулер



www.williamspublishing.com

ISBN 978-5-8459-1611-2

Книга дает ответы на трудные вопросы, с которыми приходится сталкиваться всем разработчикам корпоративных систем. Автор, известный специалист в области объектно-ориентированного программирования, заметил, что с развитием технологий базовые принципы проектирования и решения общих проблем остаются неизменными, и выделил более 40 наиболее употребительных подходов, оформив их в виде типовых решений. Результат перед вами — незаменимое руководство по архитектуре программных систем для любой корпоративной платформы. Это своеобразное учебное пособие поможет вам не только усвоить информацию, но и передать полученные знания окружающим значительно быстрее и эффективнее, чем это удавалось автору. Книга предназначена для программистов, проектировщиков и архитекторов, которые занимаются созданием корпоративных приложений и стремятся повысить качество принимаемых стратегических решений.

в продаже

ШАБЛОНЫ РЕАЛИЗАЦИИ КОРПОРАТИВНЫХ ПРИЛОЖЕНИЙ

Кент Бек



www.dialektika.com

Один из самых креативных и признанных лидеров в индустрии программного обеспечения Кент Бек собрал 77 шаблонов, предназначенных для выполнения ежедневных программистских задач и написания более читаемого кода. Эта новая коллекция шаблонов предназначена для реализации многих аспектов разработки, включая классы, состояние, поведение, методы, коллекции, инфраструктуры и т.д. Автор использует диаграммы, истории, примеры и эссе для того, чтобы увлечь читателя по ходу освещения шаблонов. Вы обнаружите проверенные решения для управления всем, от именования переменных до проверки исключений. Эта книга предназначена для программистов всех уровней подготовки, особенно для тех, кто применяет в своей практике шаблоны проектирования и методы быстрой разработки. Книга также окажется неоценимым ресурсом для команд разработчиков, ищущих более эффективные методы совместной работы и построения более управляемого ПО.

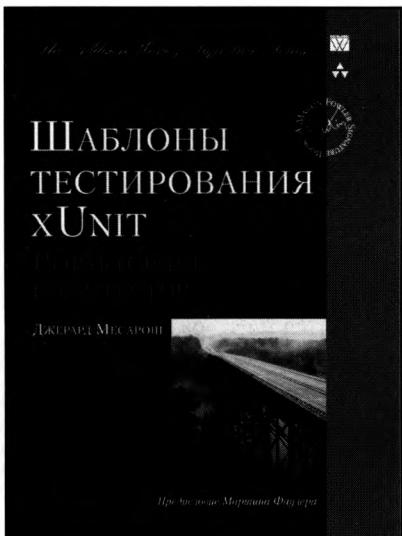
ISBN 978-5-8459-1406-4 в продаже

ШАБЛОНЫ ТЕСТИРОВАНИЯ

XUNIT

РЕФАКТОРИНГ КОДА ТЕСТОВ

Джерард Месарош



www.williamspublishing.com

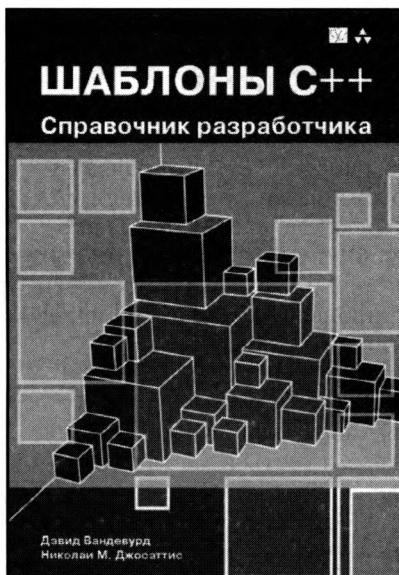
ISBN 978-5-8459-1448-4

В данной книге показано, как применять принципы разработки программного обеспечения, в частности шаблоны проектирования, инкапсуляцию, исключение повторений и описательные имена, к написанию кода тестов. Книга состоит из трех частей. В первой части приводятся теоретические основы методов разработки тестов, описываются концепции шаблонов и "запахов" тестов (признаков существующей проблемы). Во второй и третьей частях книги приводится каталог шаблонов проектирования тестов, "запахов" и других средств обеспечения большей прозрачности кода тестов. Кроме этого, в третьей части книги сделана попытка обобщить и привести к единому знаменателю терминологию тестовых двойников и подставных объектов, а также рассмотрены некоторые принципы их применения при проектировании как тестов, так и самого программного обеспечения. Книга ориентирована на разработчиков программного обеспечения, практикующих гибкие процессы разработки.

в продаже

ШАБЛОНЫ С++: СПРАВОЧНИК РАЗРАБОТЧИКА

**Дэвид Вандевурд,
Николай М. Джосаттис**



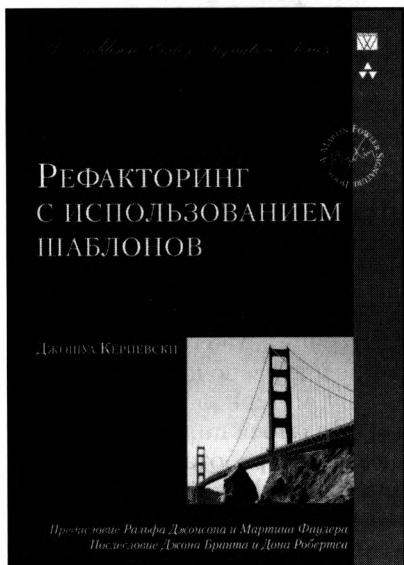
www.williamspublishing.com

в продаже

Несмотря на то, что шаблоны вошли в язык программирования C++ более десяти лет назад, они представляют собой активно развивающуюся часть C++, предоставляющую программисту новые возможности быстрой разработки эффективных и надежных программ и повторного использования кода. Будучи одной из наиболее мощных возможностей языка, шаблоны одновременно являются одной из самых запутанных, трудно понимаемых и зачастую неверно используемых частей C++. Данная книга, написанная в соавторстве теоретиком C++ и программистом-практиком с большим опытом, удачно сочетает строгость изложения и полноту освещения темы с вопросами практического использования шаблонов. В книге содержится масса разнообразного материала, относящегося к программированию с использованием шаблонов, включая такие вопросы, как новые идиомы программирования, связанные с шаблонами, метaprogramмирование или возможные направления развития шаблонов в будущем. Кроме того, книга снабдит читателя материалом, который даст опытным программистам возможность преодолеть современные ограничения в этой области. Книга предполагает наличие у читателя достаточно глубоких знаний языка C++, поскольку в книге дается детальное описание конкретного средства языка программирования, но не основ самого языка. Тем не менее стиль изложения обеспечивает доступность материала как для квалифицированных специалистов, так и для программистов среднего уровня.

РЕФАКТОРИНГ С ИСПОЛЬЗОВАНИЕМ ШАБЛОНОВ

Джошуа Кериевски



www.williamspublishing.com

Данная книга представляет собой результат многолетнего опыта профессионального программиста по применению шаблонов проектирования. Авторский подход к проектированию состоит в том, что следует избегать как недостаточного, так и избыточного проектирования, постоянно анализируя готовый работоспособный код и реорганизуя его только в том случае, когда это приведет к повышению его эффективности, упрощению его понимания и сопровождения. Автор на основании как собственного, так и чужого опыта детально рассматривает различные признаки кода, требующего рефакторинга, описывает, какой именно рефакторинг наилучшим образом подходит для той или иной ситуации, и описывает его механику, подробно разбирая ее на конкретных примерах из реальных задач. Книга может рассматриваться и как учебник по рефакторингу для программиста среднего уровня, и как справочное пособие для профессионала.

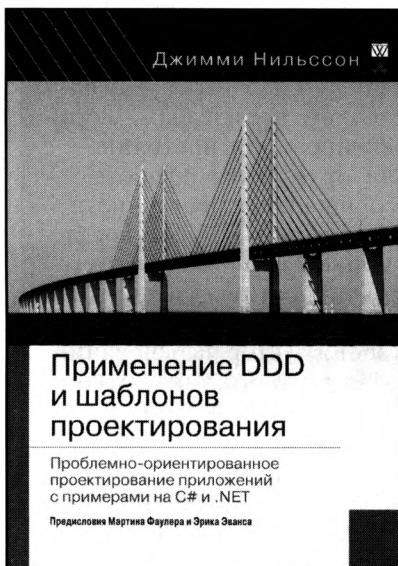
ISBN 5-8459-1087-0

в продаже

ПРИМЕНЕНИЕ DDD И ШАБЛОНОВ ПРОЕКТИРОВАНИЯ

ПРОБЛЕМНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЙ С ПРИМЕРАМИ НА C# И .NET

Джимми Нильссон



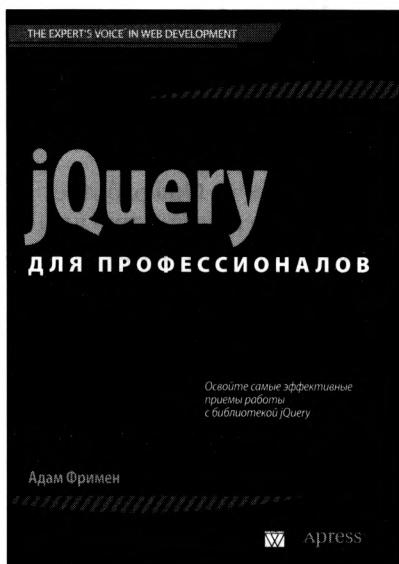
www.williamspublishing.com

Эта книга о разработке корпоративных программных приложений в среде .NET с применением шаблонов проектирования. В ней описаны: проблемно-ориентированные методы проектирования (DDD, или Domain Driven Design), разработка посредством тестирования (TDD, или Test-Driven Development), объектно-реляционное преобразование, т.е. методы, которые многие относят к ключевым технологиям разработки программного обеспечения. Хотя большинство примеров кода представлено на языке C#, материал книги может оказаться полезным и для тех, кто работает на платформе Java. Книга адресована опытным разработчикам архитектуры и прикладного программного обеспечения уровня предприятий, в том числе и в среде .NET.

ISBN 978-5-8459-1296-1 в продаже

jQUERY ДЛЯ ПРОФЕССИОНАЛОВ

Адам Фримен



www.williamspublishing.com

В книге показано, как создавать профессиональные веб-приложения с меньшими усилиями и при меньшем размере кода. Вы изучите методы работы со встроенными и дистанционными данными, научитесь создавать функционально насыщенные интерфейсы для веб-приложений, а также познакомитесь с возможностями сенсорно-ориентированного фреймворка jQuery Mobile.

Основные темы книги:

- возможности и особенности библиотеки jQuery;
- применение базовых инструментов jQuery для улучшения HTML-документов, включения в них таблиц, форм и средств отображения данных;
- применение библиотеки jQuery UI для создания гибких и удобных в использовании веб-приложений;
- программирование различных элементов взаимодействия, как перетаскивание и вставка объектов, сортировка данных и сенсорная чувствительность;
- применение библиотеки jQuery Mobile при разработке сенсорно-ориентированных интерфейсов для мобильных устройств и планшетных компьютеров;
- расширение библиотеки jQuery путем создания собственных подключаемых модулей и виджетов.

ISBN 978-5-8459-1799-7 В продаже

SPRING 3 ДЛЯ ПРОФЕССИОНАЛОВ

**Кларенс Хо,
Роб Харроп**



www.williamspublishing.com

ISBN 978-5-8459-1803-1

Гибкая, облегченная, с открытым кодом платформа Spring Framework продолжает занимать место лидирующей инфраструктуры для разработки приложений на Java для современных программистов и разработчиков. Она работает в тесной интеграции с другими гибкими и облегченными Java-технологиями с открытым кодом, такими как Hibernate, Groovy, MyBatis и т.д. В настоящее время Spring также может взаимодействовать с Java EE и JPA 2. Благодаря настоящей книге, вы изучите основы Spring, освоите ключевые темы, а также ознакомитесь с реальным опытом реализации в приложениях удаленной обработки, Hibernate и EJB. Помимо основ, вы узнаете, как использовать Spring Framework для построения различных уровней или частей корпоративного Java-приложения, в том числе транзакций, веб-уровня и уровня презентаций, развертывания и многого другого.

в продаже

Элементарные шаблоны проектирования

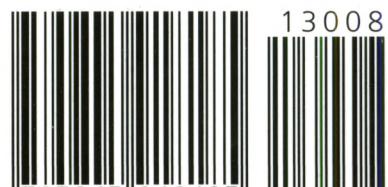
Даже опытным программистам не всегда удается применять шаблоны проектирования настолько эффективно, чтобы они приносили ощутимую пользу. В данной книге Джейсон Мак-Колм Смит рассматривает эту проблему во всей полноте, тем самым помогая разработчикам реализовать всю мощь шаблонов, более точно и ясно воплотить их в программном обеспечении и достичь наилучших результатов. Книга носит частично учебный, частично практический характер. Она поможет программистам, проектировщикам, архитекторам и аналитикам успешно использовать шаблоны проектирования в сочетании с широким спектром языков программирования, интегрированных сред разработки и проблемных областей. Каждая часть этой книги важна, поскольку дает читателям глубокое понимание выполняемой ими работы.

Автор описывает то, чего практики, использующие шаблоны проектирования, давно ожидали: базовую коллекцию простых шаблонов проектирования, которые, в свою очередь, раскладываются на составляющие их элементы. Практически все разработчики программного обеспечения используют эти элементарные шаблоны проектирования в своей ежедневной работе. Стремясь к полноте и точности, автор называет их имена, описывает и объясняет их важность, помогает читателям сравнить их между собой и сделать правильный выбор, а также предлагает каркас, в рамках которого их можно использовать совместно. Кроме того, он представляет новаторскую систему диаграмм Pattern Instance Notation, облегчающую работу с шаблонами на разных уровнях детализации независимо от ваших целей и роли.

Тем, кто еще ничего не знает о шаблонах проектирования, эта наполненная примерами книга поможет постепенно овладеть ими как интуитивно, так и логически. Опытным практикам автор, придерживаясь общезвестного формата, предложенного "Бандой четырех", объясняет, как из элементарных шаблонов проектирования составить стандартные шаблоны, и предлагает новый и эффективный способ реализации уже известных идей. Каким бы ни был уровень вашей подготовки, эта чрезвычайно практичная книга поможет вам воплотить абстрактные шаблоны проектирования в весьма ценные решения.

Джейсон Мак-Колм Смит получил докторскую степень в области информатики в 2005 году в Университете Северной Каролины в Чапел-Хилле. (Элементарные шаблоны проектирования появились как часть проекта по созданию системы запросов и распознавания шаблонов.) Затем он четыре года работал в компании IBM Watson Research, применяя опыт работы над системой SPQR и каталогом EDP, а также композиционный подход к их использованию в программном обеспечении, как унаследованном, так и современном. В настоящее время доктор Смит является старшим научным сотрудником в компании Software Revolution, Inc., в Киркланде, шт. Вашингтон, где продолжает уточнять каталог EDP и искать способы повышения эффективности работы компании за счет автоматизированной модернизации и трансформации унаследованных систем.

ISBN 978-5-8459-1818-5



13 008
9 785845 918185



www.williamspublishing.com

Addison-Wesley

Pearson Education