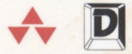


Рефакторинг

Улучшение проекта существующего кода

Мартин Фаулер

при участии Кента Бека, Джона Бранта, Уильяма Опдайка и Дона Робертса

С предисловием Эриха Гаммы, Object Technology International, Inc.

Рефакторинг

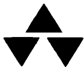
улучшение проекта существующего кода

REFACTORING

IMPROVING THE DESIGN OF EXISTING CODE

MARTIN FOWLER

With contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts



ADDISON-WESLEY

An imprint of Addison Wesley Longman, Inc.

Reading, Massachusetts • Harlow, England • Menlo Park, California
Berkeley, California • Don Mills, Ontario • Sydney
Bonn • Amsterdam • Tokyo • Mexico City

Рефакторинг

Улучшение проекта существующего кода

МАРТИН ФАУЛЕР

при участии **Кента Бека, Джона Бранта, Уильяма Опдайка и Дона Робертса**

Москва • Санкт-Петербург
2019

ББК 32.973.26-018.2.75
Р45
УДК 681.3.07

Компьютерное издательство "Диалектика"

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Фаулер, Мартин, **Бек**, Кент, **Брант**, Джон, **Опдайк**, Уильям, **Робертс**, Дон.

Р45 Рефакторинг: улучшение проекта существующего кода. : Пер. с англ. — СПб. : ООО "Диалектика", 2019. — 448 с. : ил. — Парал. тит. англ.

ISBN 978-5-9909445-1-0 (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.

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

Authorized translation from the English language edition published by Addison Wesley Longman, Inc., Copyright © 1999.

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

Научно-популярное издание

Мартин Фаулер, Кент Бек, Джон Брант, Уильям Опдайк, Дон Робертс

Рефакторинг: улучшение проекта существующего кода

Подписано в печать 18.12.2018. Формат 70x100/16
Гарнитура Times
Усл. печ. л. 36,1. Уч.-изд. л. 18,9
Доп. тираж 500 экз. Заказ № 16316

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

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30 Лит. А, пом. 848

ISBN 978-5-9909445-1-0 (рус.)
ISBN 978-0-201-48567-7 (англ.)

© Компьютерное издательство "Диалектика", 2019, перевод, оформление, макетирование

© Addison Wesley Longman, Inc., 1999

Оглавление

Предисловие 21

Введение 23

Глава 1. Первый пример рефакторинга 31

Глава 2. Принципы рефакторинга 69

Глава 3. Запах в коде 93

Глава 4. Создание тестов 109

Глава 5. На пути к каталогу рефакторингов 125

Глава 6. Составление методов 131

Глава 7. Перенос функциональности между объектами 161

Глава 8. Организация данных 189

Глава 9. Упрощение условных выражений 255

Глава 10. Упрощение вызовов методов 289

Глава 11. Работа с обобщенностью 337

Глава 12. Крупномасштабные рефакторинги 375

Глава 13. Рефакторинг, повторное использование и реальность 395

Глава 14. Инструментарий для выполнения рефакторинга 419

Глава 15. Заключение 427

Библиография 431

Примечания 433

Список рефакторингов 435

Список запахов 437

Предметный указатель 439

Содержание

Предисловие 21

Введение 23

Что такое рефакторинг 24

О чем эта книга 25

Рефакторинг и Java 26

На кого рассчитана эта книга 26

На плечах других 27

Благодарности 28

Глава 1. Первый пример рефакторинга 31

Начальная точка 32

Комментарии к программе 35

Первый шаг 37

Декомпозиция и перераспределение метода `statement` 37

Перенос метода расчета суммы оплаты 43

Выделение начисления бонусов 48

Устранение временных переменных 51

Замена логики условий полиморфизмом 57

А теперь — наследование 60

Заключительные замечания 67

Глава 2. Принципы рефакторинга 69

Определение рефакторинга 69

Меняя шляпы 70

Почему нужно заниматься рефакторингом 71

Рефакторинг совершенствует проектирование программного обеспечения 71

Рефакторинг упрощает понимание программ 72

Рефакторинг помогает находить ошибки 73

Рефакторинг ускоряет написание программ 73

Когда нужно выполнять рефакторинг 74

Правило трех 74

Выполняйте рефакторинг при добавлении функции 74

Выполняйте рефакторинг во время исправления ошибок 75
Выполняйте рефакторинг в ходе анализа кода 75
Что мне сказать руководству? 77
Проблемы при выполнении рефакторинга 79
Базы данных 80
Изменение интерфейсов 81
Изменения проекта, затрудняющие рефакторинг 83
Когда не следует прибегать к рефакторингу 83
Рефакторинг и проектирование 84
Рефакторинг и производительность 87
Истоки рефакторинга 89
Глава 3. Запах в коде 93
Дублируемый код 94
Длинный метод 94
Большой класс 96
Длинный список параметров 97
Расходящиеся изменения 98
Стрельба дробью 98
Завистливые функции 99
Группы данных 99
Одержимость примитивами 100
Инструкции switch 101
Параллельные иерархии наследования 102
Ленивый класс 102
Теоретическая общность 102
Временное поле 103
Цепочки сообщений 103
Посредник 104
Неуместная близость 104
Альтернативные классы с разными интерфейсами 105
Неполный библиотечный класс 105
Классы данных 106
Отказ от наследства 106
Комментарии 107

Глава 4. Создание тестов 109
Важность самотестируемого кода 109
Каркас тестирования JUnit 112
Модульные и функциональные тесты 117
Добавление новых тестов 118
Глава 5. На пути к каталогу рефакторингов 125
Формат описания рефакторинга 125
Поиск ссылок 127
Насколько зрелы предлагаемые методы рефакторинга 128
Глава 6. Составление методов 131
Извлечение метода (Extract Method) 132
Мотивация 132
Техника 133
Пример: локальных переменных нет 134
Пример: локальные переменные есть 135
Пример: присваивание локальной переменной 136
Встраивание метода (Inline Method) 139
Мотивация 139
Техника 140
Встраивание временной переменной (Inline Temp) 140
Мотивация 140
Техника 141
Замена временной переменной запросом (Replace Temp with Query) 141
Мотивация 142
Техника 142
Пример 143
Введение поясняющей переменной (Introduce Explaining Variable) 145
Мотивация 146
Техника 146
Пример 146
Пример с извлечением метода 147
Расщепление временной переменной (Split Temporary Variable) 149
Мотивация 149
Техника 150
Пример 150

Удаление присваиваний параметрам (Remove Assignments to Parameters) 152
Мотивация 152
Техника 153
Пример 153
Передача по значению в Java 154
Замена метода объектом методов (Replace Method with Method Object) 155
Мотивация 156
Техника 156
Пример 157
Замена алгоритма (Substitute Algorithm) 159
Мотивация 159
Техника 160
Глава 7. Перенос функциональности между объектами 161
Перенос метода (Move Method) 162
Мотивация 162
Техника 163
Пример 164
Перенос поля (Move Field) 166
Мотивация 166
Техника 167
Пример 168
Пример: использование самоинкапсуляции 168
Извлечение класса (Extract Class) 169
Мотивация 170
Техника 170
Пример 171
Встраивание класса (Inline Class) 174
Мотивация 174
Техника 174
Пример 175
Сокрытие делегирования (Hide Delegate) 176
Мотивация 177
Техника 177
Пример 178

Удаление посредника (Remove Middle Man) 179
Мотивация 179
Техника 180
Пример 180
Введение внешнего метода (Introduce Foreign Method) 181
Мотивация 181
Техника 182
Пример 182
Введение локального расширения (Introduce Local Extension) 183
Мотивация 183
Техника 184
Примеры 184
Пример: использование подкласса 185
Пример: использование оболочки 186
Глава 8. Организация данных 189
Самоинкапсуляция поля (Self Encapsulate Field) 190
Мотивация 191
Техника 192
Пример 192
Замена значения данных объектом (Replace Data Value with Object) 194
Мотивация 194
Техника 195
Пример 195
Замена значения ссылкой (Change Value to Reference) 198
Мотивация 198
Техника 199
Пример 199
Замена ссылки значением (Change Reference to Value) 202
Мотивация 202
Техника 203
Пример 203
Замена массива объектом (Replace Array with Object) 204
Мотивация 205
Техника 205
Пример 205

Дублирование видимых данных (Duplicate Observed Data) 207
Мотивация 208
Техника 209
Пример 210
Использование слушателей событий 215
Замена однонаправленной связи двунаправленной (Change Unidirectional Association to Bidirectional) 216
Мотивация 216
Техника 217
Пример 217
Замена двунаправленной связи однонаправленной (Change Bidirectional Association to Unidirectional) 219
Мотивация 220
Техника 220
Пример 221
Замена магического числа символической константой (Replace Magic Number with Symbolic Constant) 223
Мотивация 224
Техника 224
Инкапсуляция поля (Encapsulate Field) 224
Мотивация 225
Техника 225
Инкапсуляция коллекции (Encapsulate Collection) 226
Мотивация 226
Техника 227
Примеры 228
Пример: Java 2 228
Перемещение поведения в класс 231
Пример: Java 1.1 233
Пример: инкапсуляция массивов 234
Замена записи классом данных (Replace Record with Data Class) 235
Мотивация 235
Техника 236
Замена кода типа классом (Replace Type Code with Class) 236
Мотивация 236
Техника 237
Пример 238

Замена кода типа подклассами (Replace Type Code with Subclasses) 241
Мотивация 242
Техника 243
Пример 243
Замена кода типа состоянием/стратегией (Replace Type Code with State/Strategy) 245
Мотивация 245
Техника 245
Пример 246
Замена подкласса полями (Replace Subclass with Fields) 250
Мотивация 250
Техника 250
Пример 251
Глава 9. Упрощение условных выражений 255
Декомпозиция условного оператора (Decompose Conditional) 256
Мотивация 256
Техника 256
Пример 257
Консолидация условного выражения (Consolidate Conditional Expression) 258
Мотивация 258
Техника 259
Пример: логическое “ИЛИ” 259
Пример: логическое И 260
Консолидация дублирующихся условных фрагментов (Consolidate Duplicate Conditional Fragments) 260
Мотивация 261
Техника 261
Пример 261
Удаление управляющего флага (Remove Control Flag) 262
Мотивация 262
Техника 263
Пример: замена простого управляющего флага оператором `break` 263
Пример: использование оператора `return`, возвращающего значение управляющего флага 265
Замена вложенных условных операторов граничным оператором (Replace Nested Conditional with Guard Clauses) 267

Мотивация 267
Техника 268
Пример 268
Пример: обращение условий 270
Замена условной инструкции полиморфизмом (Replace Conditional with Polymorphism) 271
Мотивация 272
Техника 273
Пример 274
Введение нулевого объекта (Introduce Null Object) 276
Мотивация 277
Техника 278
Пример 279
Пример: тестирующий интерфейс 283
Другие особые случаи 284
Введение утверждения (Introduce Assertion) 284
Мотивация 285
Техника 285
Пример 286
Глава 10. Упрощение вызовов методов 289
Переименование метода (Rename Method) 290
Мотивация 291
Техника 291
Пример 292
Добавление параметра (Add Parameter) 292
Мотивация 293
Техника 293
Удаление параметра (Remove Parameter) 294
Мотивация 294
Техника 295
Разделение запроса и модификатора (Separate Query from Modifier) 296
Мотивация 296
Техника 296
Пример 297
Вопросы многопоточности 299

Параметризация метода (Parameterize Method) 300
Мотивация 300
Техника 300
Пример 301
Замена параметра явными методами (Replace Parameter with Explicit Methods) 302
Мотивация 302
Техника 303
Пример 303
Сохранение всего объекта (Preserve Whole Object) 305
Мотивация 305
Техника 306
Пример 307
Замена параметра вызовом метода (Replace Parameter with Method) 308
Мотивация 309
Техника 310
Пример 310
Введение объекта параметра (Introduce Parameter Object) 312
Мотивация 312
Техника 312
Пример 313
Удаление метода установки значения (Remove Setting Method) 317
Мотивация 317
Техника 317
Пример 318
Сокрытие метода (Hide Method) 320
Мотивация 320
Техника 320
Замена конструктора фабричным методом (Replace Constructor with Factory Method) 321
Мотивация 321
Техника 321
Пример 322
Пример: создание подклассов с помощью строки 322
Пример: создание подклассов явными методами 324
Инкапсуляция нисходящего приведения типа (Encapsulate Downcast) 325

Мотивация 325
Техника 326
Пример 326
Замена кода ошибки исключением (Replace Error Code with Exception) 327
Мотивация 327
Техника 328
Пример 329
Пример: непроверяемое исключение 329
Пример: проверяемое исключение 330
Замена исключения проверкой (Replace Exception with Test) 332
Мотивация 332
Техника 333
Пример 333
Глава 11. Работа с обобщенностью 337
Подъем поля (Pull Up Field) 338
Мотивация 338
Техника 338
Подъем метода (Pull Up Method) 339
Мотивация 339
Техника 340
Пример 341
Подъем тела конструктора (Pull Up Constructor Body) 342
Мотивация 342
Техника 343
Пример 343
Опускание метода (Push Down Method) 345
Мотивация 345
Техника 345
Опускание поля (Push Down Field) 346
Мотивация 346
Техника 346
Извлечение подкласса (Extract Subclass) 347
Мотивация 347
Техника 347
Пример 348

Извлечение суперкласса (Extract Superclass) 352
Мотивация 353
Техника 353
Пример 354
Извлечение интерфейса (Extract Interface) 357
Мотивация 357
Техника 358
Пример 359
Свертывание иерархии (Collapse Hierarchy) 360
Мотивация 360
Техника 360
Формирование шаблонного метода (Form Template Method) 361
Мотивация 361
Техника 362
Пример 362
Замена наследования делегированием (Replace Inheritance with Delegation) 369
Мотивация 369
Техника 370
Пример 370
Замена делегирования наследованием (Replace Delegation with Inheritance) 371
Мотивация 372
Техника 372
Пример 373
Глава 12. Крупномасштабные рефакторинги 375
Природа игры 375
Важность крупномасштабных рефакторингов 376
Четыре крупномасштабных рефакторинга 377
Разделение наследования (Tease Apart Inheritance) 378
Мотивация 378
Техника 379
Примеры 380
Преобразование процедурного проекта в объектный
(Convert Procedural Design to Objects) 383
Мотивация 384
Техника 384
Пример 385

Отделение предметной области от представления (Separate Domain from Presentation) 385
Мотивация 385
Техника 386
Пример 387
Выделение иерархии (Extract Hierarchy) 390
Мотивация 390
Техника 391
Пример 392
Глава 13. Рефакторинг, повторное использование и реальность 395
Проверка в реальных условиях 396
Почему разработчики не хотят применять рефакторинг к своим программам 398
Как и когда применять рефакторинг 399
Рефакторинг как средство получения скорейших выгод 405
Уменьшение стоимости рефакторинга 407
Безопасный рефакторинг 408
Проверка в реальных условиях 412
Ресурсы и ссылки, относящиеся к рефакторингу 413
Следствия повторного использования программного обеспечения и передачи технологий 414
Завершающее замечание 416
Библиография 416
Глава 14. Инструментарий для выполнения рефакторинга 419
Рефакторинг с помощью инструментов 419
Технические критерии инструментария для рефакторинга 421
База данных программы 422
Деревья синтаксического анализа 422
Точность 423
Практические критерии инструментария 424
Скорость 424
Отмена модификаций 425
Интеграция с другими инструментами 425
Краткое заключение 425

Глава 15. Заключение 427

Библиография 431

Примечания 433

Список рефакторингов 435

Список запахов 437

Предметный указатель 439

Посвящается Синди

Предисловие

Понятие рефакторинга возникло в кругах, близких к Smalltalk, но очень быстро проложило дорогу к другим языкам программирования. Поскольку оптимизация является неотъемлемой частью развития программного обеспечения, этот термин появляется, как только проектировщики начинают вести профессиональные беседы. Он приходит и когда речь идет об уточнении иерархии классов, и когда обсуждается, сколько строк кода можно удалить. Программисты знают, что с первого раза хорошо работающую программу не получить — она должна развиваться постепенно, по мере накопления опыта. Они также знают, что код будет куда чаще читаться и изменяться, чем писаться "с нуля". Рефакторинг является ключом к поддержке удобочитаемости и изменяемости кода — как для всего программного обеспечения в целом, так и для конкретных программ.

В чем же заключается проблема? Ответ прост: рефакторинг — это риск. Он требует изменения рабочего кода, в ходе которого могут возникнуть не только улучшения, но и новые ошибки. Небрежно выполненный рефакторинг может отбросить вас назад на дни и даже недели. Еще более рискованным оказывается рефакторинг, выполняемый неофициально или от случая к случаю. Вы начинаете копаться в коде. Вскоре вы обнаруживаете новые возможности для внесения изменений и закапываетесь глубже. Чем глубже вы копаете, тем больше возможностей для изменений обнаруживаете. В конце концов вы выкапываете яму, из которой уже не в состоянии выбраться. Чтобы эта яма не превратилась в могилу, рефакторинг необходимо выполнять систематически. В нашей книге "Design Patterns"[1] было упомянуто, что проектные шаблоны обеспечивают цели для рефакторинга. Однако определение цели является лишь частью проблемы; другой задачей является преобразование кода, позволяющее достичь поставленной цели.

Мартин Фаулер и его соавторы внесли неоценимый вклад в развитие объектно-ориентированного программного обеспечения, пролив свет на процесс рефакторинга. В этой книге описаны принципы и передовой опыт рефакторинга и указано, когда и где следует начинать работу с кодом для его улучшения. В основе книги находится подробный перечень рефакторингов. Каждый рефакторинг описывает мотивацию и технологию преобразования кода. Некоторые из разновидностей рефакторинга, такие как, например, "Извлечение метода" или

[1] Имеется русский перевод: Гамма Э., Хелм Р., Джонсон Р., Влиссидес Д. *Приемы объектно-ориентированного проектирования. Паттерны проектирования.* — С-Пб, "Питер", 2000.

"Перемещение поля", могут показаться очевидными. Но не обманывайтесь — понимание технологии таких рефакторингов является ключом к выполнению организованного рефакторинга. Описания рефакторингов в этой книге помогут вам изменять свой код небольшими порциями за один раз, снижая тем самым риск резкого изменения самих принципов вашего проекта. Названия этих рефакторингов очень быстро займут свое место в вашем словарном запасе.

Мой первый опыт "пошагового" рефакторинга я получил, работая на пару с Кентом Беком на высоте 9 км. Он сумел обеспечить пошаговое применение описанных в этой книге рефакторингов, поразив меня тем, насколько хорошо все это работало. Я не только получил более надежный код, в котором был более уверен, но и почувствовал себя менее напряженным. Я настоятельно рекомендую вам испытать эти рефакторинги: и ваш код, и вы сами почувствуете себя намного лучше.

— Эрих Гамма (Erich Gamma)
Object Technology International, Inc.

Введение

Как-то раз один консультант знакомился с проектом, связанным с разработкой программного обеспечения. Он посмотрел часть готового кода, который был основан на некоторой иерархии классов. Пытаясь пробраться сквозь нее, консультант нашел ее слишком запутанной. Классы в основе иерархии использовали ряд предположений о том, как будут действовать другие классы, и эти предположения были реализованы в коде производных классов. Однако этот код не подходил для всех подклассов, и в результате нередко перекрывался в подклассах. Сократить необходимость в перекрытии кода можно было бы путем внесения некоторых не слишком больших изменений в суперкласс. В некоторых местах выполнялось дублирование поведения суперкласса, связанное с тем, что назначение суперкласса было недостаточно очевидным. Возникали и ситуации, когда ряд подклассов реализовывал одно и то же поведение, которое можно было бы безболезненно переместить выше в иерархии классов.

Рекомендация консультанта заключалась в том, чтобы пересмотреть и подчистить код. Конечно же, это не вызвало радости у руководства. Ведь программа была вполне работоспособной, а сроки поджимали. Поэтому было принято обычное решение — заняться рефакторингом потом. Когда-нибудь. Может быть...

Консультант обратился также к программистам, указав им на найденные недостатки. Программисты отнеслись к замечаниям иначе. По сути, они даже не были виновны — зачастую, чтобы заметить проблему, нужен взгляд со стороны. Потратив пару дней на переделку иерархии, программисты сумели сократить код вдвое без снижения функциональности системы. Результат их вполне удовлетворил, тем более что в итоге облегчилось добавление новых классов в иерархию, а также использование уже имеющихся классов в других местах системы.

Такое поведение программистов вызвало неприятие у руководства. График требовал ускорения работ, а эти два дня, затраченные программистами, не добавили ни одной из функциональностей, которых все еще не хватало в системе. Уже имеющийся код работал — просто он был немного менее понятным и "чистым". Руководство можно понять. Коммерческий результат приносит работающий код, а не код, который нравится "академической крысе". Консультант же предложил модифицировать и другие части кода, что могло приостановить работы на пару недель, и лишь ради более красивого кода, а не для расширения его возможностей.

Что вы скажете об этой истории? Кто, по-вашему, был прав? Может, действительно права старая программистская мудрость "Не трогай работающую систему"?

Думаю, вы сразу догадались, что этим консультантом был я. Шестью месяцами позже проект бесславно закрылся, рухнув под собственной тяжестью, — код стал слишком громоздким для отладки или получения приемлемой производительности.

Чтобы возобновить проект, был привлечен другой консультант — Кент Бек, и практически все пришлось писать заново. Кент спроектировал многое совершенно иначе, чем ранее, при этом настояв на постоянном совершенствовании кода с применением рефакторинга. Успех описанного проекта и роль рефакторинга в этом успехе подвигли меня на написание данной книги. Книга позволила мне поделиться опытом и знаниями, приобретенными Кентом и другими специалистами при использовании рефакторинга для повышения качества программного обеспечения.

Что такое рефакторинг

Рефакторинг — это процесс изменения программного проекта, в ходе которого внешнее поведение кода остается неизменным при усовершенствовании его внутренней структуры. Это систематизированный способ очистки кода, минимизирующий возможность появления новых ошибок. По сути, рефакторинг кода представляет собой улучшение проекта уже после того, как этот код написан.

“Улучшение проекта после написания кода” звучит непривычно. При нынешнем понимании процесса разработки программного обеспечения мы сначала создаем проект, а потом пишем код. Первым шагом идет проектирование, а уже затем выполняется кодирование. Со временем код будет изменяться, а целостность системы и ее соответствие первоначальному проекту будет постепенно размываться. Кодирование понемногу перестает быть инженерным искусством и превращается в хакерство.

Рефакторинг же представляет собой нечто совершенно иное. Он позволяет, взяв плохой и беспорядочный проект, превратить его в ясно спроектированный код. Каждый шаг этого преобразования очень прост. Это может быть перемещение поля из одного класса в другой, выделение части исходного текста из метода и ее перемещение в отдельный метод, перемещение некоторых фрагментов кода в том или ином направлении иерархии классов. Кумулятивный эффект таких малозаметных изменений может привести к существенному улучшению программы. Этот процесс оказывается прямой противоположностью описанной выше тенденции постепенной деградации программного проекта.

При рефакторинге требуется баланс между разными этапами работ над изменяемым проектом. Проектирование становится не отдельным начальным этапом разработки, а непрерывным процессом. В ходе работы над проектом вы

постоянно рассматриваете возможность его улучшения. В результате получается программный проект, качество которого не снижается в процессе разработки.

О чем эта книга

Эта книга представляет собой руководство по рефакторингу; она написана для программистов-профессионалов. Моя цель при написании книги — показать, как выполнять рефакторинг управляемо и эффективно. Вы научитесь выполнять рефакторинг так, чтобы не вносить при этом в код новые ошибки, а постоянно улучшать его структуру.

Обычно книги начинаются с введения. Хотя я согласен с этой традицией, мне кажется слишком сложным начинать знакомство с рефакторингом с общего обсуждения или определений. Поэтому я начну с примера. В главе 1, "Первый пример рефакторинга", приводится небольшая программа, в которой имеются обычные недостатки и которая с помощью рефакторинга превращается в более приемлемую объектно-ориентированную программу. Попутно мы рассмотрим как процесс рефакторинга в целом, так и применение некоторых полезных рефакторингов. Эта глава — ключевая для понимания того, чем в действительности является рефакторинг.

В главе 2, "Принципы рефакторинга", я рассказываю об общих принципах рефакторинга более детально; там же я даю некоторые определения и описываю некоторые причины проведения рефакторинга. Здесь же рассматриваются и некоторые проблемы, связанные с рефакторингом. В главе 3, "Запах в коде", посвященной тому, как обнаружить запахи в коде и избавиться от него, мне помогает Кент Бек. Очень большую роль при выполнении рефакторинга играет тестирование, поэтому глава 4, "Создание тестов", посвящена созданию тестов с использованием простого каркаса тестирования Java с открытым исходным кодом.

Основной материал книги, который представляет собой каталог методов рефакторинга, занимает главы с 5 по 12. Этот каталог ни в ком случае нельзя считать исчерпывающим; он представляет собой лишь начало такового каталога. В него включены те рефакторинги, с которыми я сталкивался во время работы в этой области. Когда я хочу выполнить какой-то из рефакторингов, например "Замена условной инструкции полиморфизмом" (с. 271), я обращаюсь к каталогу, чтобы посмотреть, как это сделать наиболее безопасно, шаг за шагом. Надеюсь, вы будете часто возвращаться к этой части книги.

В этой книге описаны не только мои результаты, но и многих других исследователей. Более того, последние главы книги написаны некоторыми из них. Глава 13, "Рефакторинг, повторное использование и реальность", принадлежит перу Билла Опдайка, который пишет о сложностях рефакторинга в коммерческих проектах.

Глава 14, “Инструментарий для выполнения рефакторинга”, написанная Доном Робертсом и Джоном Брантом, посвящена автоматизированному инструментарию — будущему рефакторинга. Завершает книгу глава 15, “Заключение”, автором которой является выдающийся мастер рефакторинга Кент Бек.

Рефакторинг и Java

В этой книге я везде использую примеры на языке программирования Java. Конечно, рефакторинг может выполняться и для других языков, и, я надеюсь, эта книга будет полезна и тем, кто работает на других языках программирования. Однако я предпочел Java, потому что этот язык я знаю лучше других. Я добавил несколько примечаний о рефакторинге на других языках и надеюсь, что на основе моей книги другие авторы напишут свои книги для других языков программирования.

Для лучшего изложения идей я не вдавался в особо сложные области языка Java. Поэтому в книге нет использования внутренних классов, рефлексии, потоков и многих других мощных возможностей Java. Я хотел как можно доступнее изложить базовые методы рефакторинга, а не блеснуть знаниями языка программирования.

Должен отметить, что приведенные методы рефакторинга не учитывают возможности параллельного или распределенного программирования. В этих областях имеются свои сложности, которые выходят за рамки данной книги.

На кого рассчитана эта книга

Данная книга предназначена для профессиональных программистов. В примерах и обсуждении имеется множество кода, который нужно прочесть и понять. Все примеры написаны на языке программирования Java. Этот язык выбран в силу роста его распространенности и простоты для понимания теми, кто знаком с языком программирования C. Это объектно-ориентированный язык, а объектно-ориентированные механизмы очень способствуют выполнению рефакторинга.

Хотя рефакторинг ориентирован на исходные тексты, он существенно влияет и на проектирование. Руководителям-проектировщикам и архитекторам жизненно необходимо понимать принципы рефакторинга и использовать их в своих проектах. Пожалуй, будет лучше, если рефакторингом будет руководить опытный и уважаемый человек. Он сможет лучше понять принципы, на которых основан рефакторинг, и адаптировать их для конкретного проекта. Это особенно важно в случае языков программирования, отличных от Java, так как приведенные в книге примеры требуют определенной адаптации.

Можно получить пользу от книги, даже не читая ее полностью.

- **Если вы хотите понять, что такое рефакторинг**, достаточно прочесть главу 1, "Первый пример рефакторинга"; приведенный пример должен облегчить ваше понимание.
- **Если вы хотите понять, для чего нужен рефакторинг**, прочтите главы 1 и 2. В них рассказывается о том, что такое рефакторинг и почему он необходим.
- **Если вы хотите узнать, когда следует применять рефакторинг**, прочтите главу 3, "Запах в коде". В ней описаны признаки, говорящие о необходимости применения рефакторинга.
- **Если вы хотите выполнить рефакторинг**, прочтите главы 1–4 полностью. Затем бегло ознакомьтесь с каталогом в главе 5, "На пути к каталогу рефакторингов", чтобы примерно представлять себе его содержание. Вам не нужно вдаваться во все детали. Когда вам понадобится какой-либо из методов рефакторинга, вы сможете прочесть детальную информацию о нем и использовать ее в своей работе. Каталог представляет собой справочный раздел, так что читать все подряд необязательно. Вашего внимания достойны также главы, написанные приглашенными соавторами, в особенности глава 15, "Заключение".

На плечах других

Хочу сразу же сообщить, что эта книга обязана своим появлением тем, чья работа в последнее десятилетие была посвящена развитию рефакторинга. В идеале эту книгу должен был написать кто-то из них, но время и энергия для этой работы нашлись у меня.

Двумя из ведущих сторонников рефакторинга являются Уорд Каннингем (Ward Cunningham) и Кент Бек. Для них рефакторинг давно стал центральной частью процесса разработки, принятой для ежедневного применения с целью получения всех его преимуществ. Именно сотрудничество с Кентом показало мне всю важность рефакторинга и подвигло меня на написание этой книги.

Ральф Джонсон (Ralph Johnson) возглавляет группу в Университете штата Иллинойс в Урбана-Шампань, известную своим вкладом в объектную технологию. Ральф давно является сторонником рефакторинга, как и множество его студентов. Автором первой работы, посвященной рефакторингу, стал Билл Опдайк со своей докторской диссертацией. Джон Брант и Дон Робертс не ограничились словами и создали инструментарий — Refactoring Browser, — предназначенный для выполнения рефакторинга исходных текстов на языке Smalltalk.

Благодарности

Несмотря на использование результатов упомянутых выше исследований, для написания книги мне понадобилась большая помощь. В первую очередь, она была оказана мне Кентом Беком. Первые семена книги были посеяны в баре в Детройте, где Кент рассказал мне о статье, которую он написал для *Smalltalk Report* [5]. Я не только взял из нее многие идеи для главы 1, "Первый пример рефакторинга", но и под ее влиянием начал собирать материал по рефакторингу. Кент помог мне концепцией запаха кода (code smells), а еще поддерживал меня в трудные минуты и делал все от него зависящее, чтобы эта книга увидела свет. Думаю, что сам он смог бы написать эту книгу лучше меня, но, как я уже говорил, время для этой работы нашлось у меня, и мне остается только надеяться, что мой труд не напрасен.

После написания своей части книги мне захотелось поделиться с вами знаниями непосредственно от специалистов в области рефакторинга, и я очень благодарен многим из них за то, что они не пожалели времени и написали для книги свою часть материала. Кент Бек, Джон Брант, Уильям Опдайк и Дон Робертс написали некоторые из глав (самостоятельно или в соавторстве). Кроме того, Рич Гарзанити (Rich Garzaniti) и Рон Джеффрис (Ron Jeffries) добавили полезные врезки.

Любой автор согласится с тем, что для технических книг, таких как эта, очень полезно участие рецензентов. Как обычно, Картер Шанклин (Carter Shanklin) и его команда из Addison-Wesley собрали группу квалифицированных рецензентов, в которую вошли следующие специалисты.

- Кен Ауэр (Ken Auer) из Rolemodel Software, Inc.
- Джошуа Блох (Joshua Bloch) из Sun Microsystems, Java Software
- Джон Брант из Иллинойского университета в Урбана-Шампань
- Скотт Корли (Scott Corley) из High Voltage Software, Inc.
- Уорд Каннингем (Ward Cunningham) из Cunningham & Cunningham, Inc.
- Стефен Дьюкасс (Stephane Ducasse)
- Эрих Гамма (Erich Gamma) из Object Technology International, Inc.
- Рон Джеффрис (Ron Jeffries)
- Ральф Джонсон (Ralph Johnson) из Иллинойского университета
- Джошуа Кериевски (Joshua Kerievsky) из Industrial Logic, Inc.
- Дуг Ли (Doug Lea) из SUNY Oswego
- Сандер Тишлар (Sander Tichelaar)

Все они внесли большой вклад в стиль и точность книги и выявили по крайней мере бо́льшую часть ошибок, которые всегда есть в любой рукописи. Хочу упомянуть два наиболее важных предложения, повлиявших на вид книги. Уорд и Рон заставили меня изменить стиль изложения главы 1, "Первый пример рефакторинга", а Джошуа Кериевски (Joshua Kerievsky) внес предложение дать в каталоге наброски кода.

Кроме официальных рецензентов, книгу читали и комментировали другие люди, сделавшие немало ценных замечаний. В их числе Лейф Беннет (Leif Bennett), Майкл Фезерс (Michael Feathers), Майкл Финни (Michael Finney), Нейл Галарнье (Neil Galarneau), Хишем Газули (Hisham Ghazouli), Тони Гоулд (Tony Gould), Джон Айзнер (John Isner), Брайен Марик (Brian Marick), Ральф Рейссинг (Ralf Reissing), Джон Солт (John Salt), Марк Свонсон (Mark Swanson), Дейв Томас (Dave Thomas) и Дон Уэллс (Don Wells). Есть и другие, которых я здесь не упомянул; приношу им свои извинения и благодарности.

Особенно забавными рецензентами оказались члены печально известной группы читателей из Университета штата Иллинойс в Урбана-Шампань. Поскольку эта книга в значительной мере отражает их работу, я особенно благодарен им за их труд, переданный мне в аудиоформате. Членами этой группы являются Фредрико "Фред" Балагер (Fredrico "Fred" Balaguer), Джон Брант, Ян Чай (Ian Chai), Брайен Фут (Brian Foote), Алехандра Гарридо (Alejandra Garrido), Жийанг "Джон" Хан (Zhijiang "John" Han), Питер Хэтч (Peter Hatch), Ральф Джонсон (Ralph Johnson), Сонгай "Раймонд" Лу (Songyu "Raymond" Lu), Драгос-Антон Манолеску (Dragos-Anton Manolescu), Хироки Накамура (Hiroaki Nakamura), Джеймс Овертерф (James Overturf), Дон Робертс, Чико Шираи (Chieko Shirai), Лес Тайрел (Les Tyrell) и Джо Йодер (Joe Yoder).

Любая хорошая идея требует проверки в серьезной промышленной системе. Я видел, какой эффект оказывал рефакторинг на систему Chrysler Comprehensive Compensation (C3). Хочу поблагодарить всех членов этой команды: Энн Андерсон (Ann Anderson), Эда Андери (Ed Anderi), Ральфа Битти (Ralph Beattie), Кента Бека, Дэвида Брайанта (David Bryant), Боба Коу (Bob Coe), Мари Д'Армен (Marie DeArment), Маргарет Фрончак (Margaret Fronczak), Рича Гарзанити, Денниса Гора (Dennis Gore), Брайена Хакера (Brian Hacker), Чета Хендриксона (Chet Hendrickson), Рона Джеффриса (Ron Jeffries), Дуга Джоппи (Doug Joppie), Дэвида Кима (David Kim), Пола Ковальски (Paul Kowalsky), Дебби Мюллер (Debbie Mueller), Тома Мураски (Tom Murasky), Ричарда Наттера (Richard Nutter), Адриана Панти (Adrian Pantea), Мэтта Сайджена (Matt Saigeon), Дона Томаса (Don Thomas) и Дона Уэллса (Don Wells). Работа с ними утвердила мое понимание принципов и преимуществ рефакторинга. Наблюдения за их работой показали мне, чего позволяет добиться рефакторинг, применяемый в большом проекте на протяжении ряда лет.

Мне также помогали Дж. Картер Шанклин (J. Carter Shanklin) из Addison-Wesley и его команда: Крыся Бебик (Krysia Bebick), Сьюзен Кестон (Susan Cestone), Чак Даттон (Chuck Dutton), Кристин Эриксон (Kristin Erickson), Джон Фуллер (John Fuller), Кристофер Гузиковски (Christopher Guzikowski), Симон Пэймент (Simone Payment) и Женевьев Раевски (Genevieve Rajewski). Работать с хорошим издателем — одно удовольствие! Они оказали мне большую поддержку и помощь.

Говоря о поддержке, хочу заметить, что больше всего неприятностей книга приносит тем, кто находится рядом с ее автором. Я имею в виду мою жену Синди. Спасибо за любовь ко мне, проявлявшуюся даже тогда, когда я прятался в своем кабинете. Но даже там на протяжении всего времени работы над книгой я помнил о тебе!

Мартин Фаулер
Мерлоуз, Массачусеттс
`fowler@acm.org`
`http://www.martinfowler.com`
`http://www.refactoring.com`

Глава 1

Первый пример рефакторинга

С чего же начать описание рефакторинга? Традиционно изложение новой концепции начинается с рассказа об исторических корнях, пояснения базовых принципов и т.п. Но когда на очередной конференции я слышу такое вступление, то сразу засыпаю. Мой мозг начинает работать в фоновом режиме с низким приоритетом, периодически опрашивая окружающую обстановку — не начались ли приводиться конкретные примеры. Тут я просыпаюсь, поскольку примеры помогают понять происходящее. Принципы позволяют легко делать обобщения, но, опираясь на них, очень тяжело понять, как применять их на практике. А пример все проясняет.

Поэтому начинаю свою книгу с примера рефакторинга. Пока я буду о нем рассказывать, вы узнаете, как действует рефакторинг, и получите представление о его выполнении. После этого я смогу дать обычное снотворное введение, знакомящее с историей и основными идеями и принципами.

Однако с упомянутым примером у меня возникли серьезные проблемы. Если взять в качестве примера большую программу, читателю будет трудно вникнуть в ход рефакторинга. (Я пытался поступить таким образом и выяснил, что не слишком сложный пример занял более сотни страниц текста.) Если же остановиться на небольшой программе, простой для понимания, то сколь-нибудь полно показать преимущества рефакторинга не получится.

Так я столкнулся с классической проблемой всех, кто пытается описывать технологии, полезные для программ из реальной жизни. Честно говоря, рефакторинг, который я покажу, не стоит затраченных на него усилий — такая маленькая программа будет использована в качестве примера. Но если приведенный код является частью куда большей системы, то рефакторинг сразу оказывается очень важным. Так что при изучении примера представьте его себе в контексте гораздо большей системы.

Начальная точка

Программа примера очень проста. Она рассчитывает и выводит отчет о покупках клиента в видеопрокате. Программа получает информацию о том, какие фильмы брал клиент и на какой срок. После этого она рассчитывает стоимость проката исходя из длительности и типа фильма. Фильмы могут быть трех типов: обычные, детские и новинки. Кроме расчета стоимости, начисляются бонусы, зависящие от того, является ли фильм новинкой.

Различные элементы системы представлены рядом классов, показанных на диаграмме (рис. 1.1).

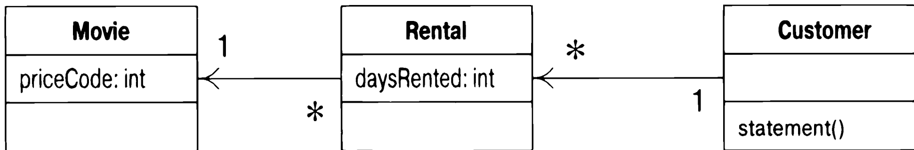


Рис. 1.1. *Диаграмма классов программы. Показаны только важнейшие свойства классов. Использован унифицированный язык моделирования UML [8]*

Я поочередно приведу код каждого из этих классов.

Movie

Класс `Movie` содержит информацию о фильме.

```
public class Movie {
    public static final int CHILDRENS = 2;
    public static final int REGULAR = 0;
    public static final int NEW_RELEASE = 1;

    private String _title;
    private int _priceCode;

    public Movie(String title, int priceCode) {
        _title = title;
        _priceCode = priceCode;
    }

    public int getPriceCode() {
        return _priceCode;
    }

    public void setPriceCode(int arg) {
        _priceCode = arg;
    }
```

```
    public String getTitle() {
        return _title;
    }
}
```

Rental

Класс Rental содержит данные о прокате фильма.

```
class Rental {
    private Movie _movie;
    private int _daysRented;

    public Rental(Movie movie, int daysRented) {
        _movie = movie;
        _daysRented = daysRented;
    }

    public int getDaysRented() {
        return _daysRented;
    }

    public Movie getMovie() {
        return _movie;
    }
}
```

Customer

Класс Customer представляет клиента проката. Как и прочие классы, он содержит данные и методы доступа.

```
class Customer {
    private String _name;
    private Vector _rentals = new Vector();

    public Customer(String name) {
        _name = name;
    };

    public void addRental(Rental arg) {
        _rentals.addElement(arg);
    }

    public String getName() {
        return _name;
    }
```

Класс Customer имеет метод, создающий отчет. На рис. 1.2 показаны взаимодействия этого метода с другими. Код тела метода приведен ниже.

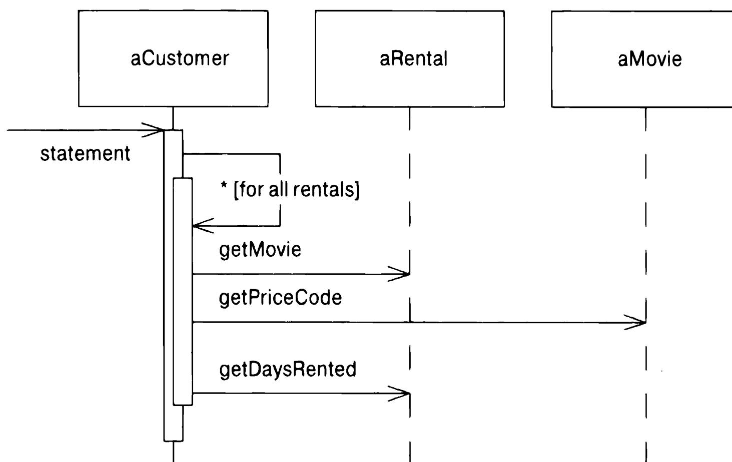


Рис. 1.2. *Взаимодействия метода* statement

```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Прокат " + getName() + "\n";

    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();

        // Определение суммы для каждой строки
        switch (each.getMovie().getPriceCode()) {
            case Movie.REGULAR:
                thisAmount += 2;

                if (each.getDaysRented() > 2)
                    thisAmount += (each.getDaysRented() - 2) * 1.5;

                break;

            case Movie.NEW_RELEASE:
                thisAmount += each.getDaysRented() * 3;
                break;

            case Movie.CHILDRENS:
                thisAmount += 1.5;
```

```
                if (each.getDaysRented() > 3)
                    thisAmount += (each.getDaysRented() - 3) * 1.5;

                break;
        }

        // Начисление бонусных очков
        frequentRenterPoints ++;

        // Бонус за двухдневный прокат новинки
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
                each.getDaysRented() > 1) frequentRenterPoints ++;

        // Вывод результатов для каждого проката
        result += "\t" + each.getMovie().getTitle() + "\t" +
                  String.valueOf(thisAmount) + "\n";
        totalAmount += thisAmount;
    }

    // Добавление колонтитула
    result += "Сумма задолженности: " +
              String.valueOf(totalAmount) + "\n";
    result += "Вы заработали "+String.valueOf(frequentRenterPoints)+
              " бонусных очков";
    return result;
}
```

Комментарии к программе

Каково ваше впечатление от этой программы? Я бы описал ее как не очень хорошо спроектированную и, определенно, не объектно-ориентированную. Для простой программы, подобной рассматриваемой, это может не иметь особого значения. Ничего страшного, если *простая* программа набросана на скорую руку. Но если это представительный фрагмент более сложной программы, то у нас действительно проблемы. Слишком длинный метод `statement` класса `Customer` выполняет слишком много действий. Многое из того, что он делает, на самом деле должны выполнять методы других классов.

Однако программа работает и в таком виде. Может быть, это просто эстетствование, вызванное не слишком красивым кодом? В принципе, это так — до попытки внести в код изменения. Компилятору все равно, красив код или уродлив. Но процессе внесения в него изменений осуществляется людьми, которым красота кода не безразлична. Вносить изменения в плохо спроектированную систему трудно, потому что нелегко понять, где нужны изменения. Если трудно понять, что следует изменять, то велика вероятность ошибки.

В нашем случае есть изменения, которые требуют пользователи. Во-первых, им нужен соответствующий моде вывод отчета в HTML, готовый для публикации в вебе. Давайте посмотрим, на что повлияет такое изменение. Из кода видно, что повторно использовать имеющийся метод `statement` для создания HTML-отчета невозможно. Остается только переписать метод заново, при этом во многом дублируя поведение старого метода. Для нашего небольшого примера это не слишком накладно — достаточно просто скопировать метод `statement` и внести в него необходимые изменения.

А что если изменятся правила оплаты? Придется вносить изменения как в метод `statement`, так и в метод `htmlStatement`, причем эти изменения должны быть согласованы. Проблема копирования и вставки возникает позже, когда имеющийся код нужно модифицировать. Если однажды написанную программу в будущем не придется изменять, то в ней можно обойтись копированием и вставкой. Но если программа пишется надолго и в нее во время ее жизненного цикла планируется внесение изменений, применение копирования и вставки является угрозой для проекта.

Это приводит меня ко второму изменению. Пользователи хотят внести изменения в классификацию фильмов, но еще не решились на это окончательно. В планах несколько изменений, которые затронут как плату за прокат, так и порядок начисления бонусов активным клиентам. Как опытный разработчик вы знаете, что какая бы схема ни была выбрана сейчас, за полгода ее гарантированно поменяют.

Изменения, вызванные новыми правилами классификации фильмов и оплаты проката, требуется внести в метод `statement`. Но при выдаче отчета в формате HTML требуется гарантировать полную согласованность изменений в обоих методах. Кроме того, при увеличении сложности правил все труднее определить, куда следует вносить изменения, и все труднее вносить их без ошибок.

Соблазнительно ограничиться минимальными изменениями, ведь, в конце концов, программа вполне работоспособна. Как любят говорить инженеры, "Не чините то, что не поломано". Программа может нормально работать, но при этом причинять неприятности, осложняющие жизнь, когда нужно внести в нее необходимые пользователям изменения. Вот тут в игру и вступает рефакторинг.

Когда выясняется, что в программу необходимо добавить новую функциональность, но при этом код программы не структурирован способом, удобным для добавления этой функциональности, сначала выполните рефакторинг, упрощающий внесение изменений, и только после этого приступайте к самим изменениям.

Первый шаг

Начиная рефакторинг, первым делом я всегда создаю надежный набор тестов для той части кода, с которой буду работать. Тесты очень важны, поскольку даже при структурированном рефакторинге, который уменьшает возможность внесения ошибок, я остаюсь человеком, а человеку свойственно ошибаться. Поэтому мне нужны надежные тесты.

Поскольку метод `statement` возвращает строку, я создаю несколько клиентов, даю им напрокат несколько типов фильмов и генерирую строки отчетов. После этого я сравниваю получаемые строки с контрольными, которые проверены мной вручную. Я организую тесты таким образом, чтобы их можно было выполнить одной командой из командной строки. При этом выполнение тестов занимает мало времени, что важно, так как я запускаю их, как вы увидите, очень часто.

Важной частью тестов является способ вывода результатов. Они выводят либо "OK", что означает совпадение с контрольными строками всех вычисляемых строк, либо список ошибок, т.е. строк, отличающихся от контрольных. Таким образом, тесты сами проверяют свои результаты. Это, опять же, очень важно, потому что иначе нужно тратить много времени на то, чтобы вручную сравнивать получаемые результаты с записанными на бумаге.

Выполняя рефакторинг, мы должны опираться на тесты. Если в программе будет допущена ошибка, тесты тут же об этом сообщат. Для рефакторинга крайне важно иметь хорошие, надежные тесты. Затраченное на создание тестов время окупается сторицей, поскольку тесты обеспечивают вас гарантией отсутствия ошибок для продолжения модификации программы. Это настолько важная часть рефакторинга, что я подробнее расскажу о ней в главе 4, "Создание тестов".

Перед тем как приступать к рефакторингу, убедитесь, что у вас есть комплект надежных самопроверяющихся тестов.

Декомпозиция и перераспределение метода `statement`

Очевидно, что моей первой целью является слишком длинный метод `statement`. Когда я вижу такие длинные методы, я прикидываю, как разложить их на меньшие части. Жизнь программиста существенно упрощается при небольших фрагментах кода. Такими фрагментами проще управлять; перемещать их также проще, чем большие куски.

Первый этап рефакторинга в этой главе показывает, как я разбиваю длинный метод на части и перемещаю эти части в более подходящие для них классы. Моя цель заключается в том, чтобы облегчить написание метода вывода отчета в формате HTML без дублирования кода (или с минимальным дублированием).

Мой первый шаг состоит в логической группировке кода и использовании рефакторинга "Извлечение метода" (с. 132). Очевидным кандидатом для этого является инструкция `switch`, которая так и просится в отдельный метод.

При выделении метода, как и при любом ином рефакторинге, мне необходимо знать, что может пойти не так. Если выполнить выделение плохо, можно внести ошибку в программу. Поэтому перед тем, как приступить к рефакторингу, следует разобраться, как безопасно его выполнить. Я уже не раз проводил такой рефакторинг и записал в каталог безопасную последовательность шагов.

Сначала нужно проверить, нет ли в данном фрагменте переменных, локальных в области видимости данного метода, т.е. локальных переменных и параметров. В рассматриваемом фрагменте кода таких переменных две: `each` и `thisAmount`. Переменная `each` этим фрагментом кода не изменяется, в отличие от `thisAmount`. Все неизменяемые переменные можно передавать как параметры. Что касается модифицируемых переменных, то тут ситуация сложнее. Если такая переменная единственная, метод может ее вернуть. Временная переменная инициализируется нулем на каждой итерации цикла и не меняется, пока не попадает в инструкцию `switch`. Поэтому значение этой переменной метод может просто вернуть.

Далее показан код до и после выполнения рефакторинга. Сначала показан первоначальный код, а затем получившийся в результате. Код, вынесенный мной из оригинала, и изменения в новом коде, которые не кажутся мне самоочевидными, выделены полужирным шрифтом. Такого соглашения о выводе кода до и после рефакторинга я буду придерживаться и далее в этой главе.

```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Прокат " + getName() + "\n";

    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();

        // Определение суммы для каждой строки
        switch (each.getMovie().getPriceCode()) {
            case Movie.REGULAR:
                thisAmount += 2;

                if (each.getDaysRented() > 2)
                    thisAmount += (each.getDaysRented() - 2) * 1.5;
```

```
                break;

            case Movie.NEW_RELEASE:
                thisAmount += each.getDaysRented() * 3;
                break;

            case Movie.CHILDRENS:
                thisAmount += 1.5;

                if (each.getDaysRented() > 3)
                    thisAmount += (each.getDaysRented() - 3) * 1.5;

                break;
        }

        // Начисление бонусных очков
        frequentRenterPoints ++;

        // Бонус за двухдневный прокат новинки
        if ((each.getMovie().getPriceCode()==Movie.NEW_RELEASE)&&
                each.getDaysRented() > 1)
            frequentRenterPoints ++;

        // Вывод результатов для каждого проката
        result += "\t" + each.getMovie().getTitle() + "\t" +
                  String.valueOf(thisAmount) + "\n";
        totalAmount += thisAmount;
    }

    // Добавление колонтитула
    result += "Сумма задолженности: " +
              String.valueOf(totalAmount) + "\n";
    result += "Вы заработали "+String.valueOf(frequentRenterPoints)+
              " бонусных очков";
    return result;
}

========== После рефакторинга ======================================

public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Прокат " + getName() + "\n";

    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();
```

```
        thisAmount = amountFor(each);

        // Начисление бонусных очков
        frequentRenterPoints ++;

        // Бонус за двухдневный прокат новинки
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
                each.getDaysRented() > 1) frequentRenterPoints ++;

        // Вывод результатов для каждого проката
        result += "\t" + each.getMovie().getTitle() + "\t" +
                  String.valueOf(thisAmount) + "\n";
        totalAmount += thisAmount;
    }

    // Добавление колонтитула
    result += "Сумма задолженности: " +
              String.valueOf(totalAmount) + "\n";
    result += "Вы заработали "+String.valueOf(frequentRenterPoints)+
              " бонусных очков";
    return result;
}

private int amountFor(Rental each) {

    int thisAmount = 0;

    switch (each.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            thisAmount += 2;

            if (each.getDaysRented() > 2)
                thisAmount += (each.getDaysRented() - 2) * 1.5;

            break;

        case Movie.NEW_RELEASE:
            thisAmount += each.getDaysRented() * 3;
            break;

        case Movie.CHILDRENS:
            thisAmount += 1.5;

            if (each.getDaysRented() > 3)
                thisAmount += (each.getDaysRented() - 3) * 1.5;

            break;
    }

    return thisAmount;
}
```

Преобразованный код я скомпилировал и протестировал. Увы, тест завершился неудачно: несколько тестов дали неверный результат. После недолгого размышления я понял, что задал тип возвращаемого значения amountFor как int, а не double:

```
private double amountFor(Rental each) {
    double thisAmount = 0;

    switch (each.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            thisAmount += 2;

            if (each.getDaysRented() > 2)
                thisAmount += (each.getDaysRented() - 2) * 1.5;

            break;

        case Movie.NEW_RELEASE:
            thisAmount += each.getDaysRented() * 3;
            break;

        case Movie.CHILDRENS:
            thisAmount += 1.5;

            if (each.getDaysRented() > 3)
                thisAmount += (each.getDaysRented() - 3) * 1.5;

            break;
    }

    return thisAmount;
}
```

Такие глупые ошибки — не редкость, и их зачастую довольно сложно отследить. В этом случае Java молча преобразует double в int без предупреждений с помощью простого округления [10]. К счастью, модификация была незначительной, а набор тестов хорошим, так что ошибка была легко выявлена. В этом и состоит сущность рефакторинга. Все изменения — небольшие, поэтому ошибки легко выявляются. Даже такому неаккуратному программисту, как я, не приходится тратить на отладку много времени.

Рефакторинг подразумевает изменение программы небольшими порциями, что облегчает обнаружение ошибок.

Работая на языке программирования Java, необходимо анализировать код и решать, что делать с локальными переменными. Такой анализ облегчается при

применении специального инструментария. Инструментарий для Smalltalk называется "Refactoring Browser". В нем достаточно выделить код, выбрать в меню пункт **Extract Method** и ввести имя метода. Все остальное будет сделано без вашего участия. Этот инструментарий достаточно умен, чтобы не делать глупых ошибок наподобие показанной выше. Жду не дождусь его версии для Java!

Теперь, когда исходный метод разделен на части, можно работать с ними по отдельности. Мне не нравятся некоторые имена переменных в `amountFor`, и сейчас самое время их изменить.

Вот исходный код:

```
private double amountFor(Rental each) {
    double thisAmount = 0;

    switch (each.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            thisAmount += 2;

            if (each.getDaysRented() > 2)
                thisAmount += (each.getDaysRented() - 2) * 1.5;

            break;

        case Movie.NEW_RELEASE:
            thisAmount += each.getDaysRented() * 3;
            break;

        case Movie.CHILDRENS:
            thisAmount += 1.5;

            if (each.getDaysRented() > 3)
                thisAmount += (each.getDaysRented() - 3) * 1.5;

            break;
    }

    return thisAmount;
}
```

А вот измененный:

```
private double amountFor(Rental aRental) {
    double result = 0;

    switch (aRental.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            result += 2;
            if (aRental.getDaysRented() > 2)
                result += (aRental.getDaysRented() - 2) * 1.5;
```

```
            break;
        case Movie.NEW_RELEASE:
            result += aRental.getDaysRented() * 3;
            break;
        case Movie.CHILDRENS:
            result += 1.5;
            if (aRental.getDaysRented() > 3)
                result += (aRental.getDaysRented() - 3) * 1.5;
            break;
    }
    return result;
}
```

Выполнив переименование, я компилирую и тестирую код, чтобы убедиться, что при этом я ничего не испортил.

Насколько нужно было это переименование? Конечно, оно нужно — хороший код сам по себе поясняет, что он делает, и правильные имена переменных играют в этом большую роль. Не бойтесь менять имена, если это приведет к более понятному коду. С помощью программных средств поиска и замены это обычно несложная операция. Возможные ошибки помогут обнаружить строгий контроль типов и тестирование.

Написать код, понятный компилятору, сможет любой дурак. Но лишь немногие смогут написать код, понятный людям.

Самодокументирование кода очень важно. Я часто выполняю рефакторинг во время простого чтения некоторого кода. Таким образом, свое понимание работы программы я тут же отражаю в коде, чтобы позже не забыть то, что я понял.

Перенос метода расчета суммы оплаты

Взглянув на amountFor, можно увидеть, что в этом методе используется информация из класса, представляющего прокат, но не информация из класса, представляющего клиент.

```
class Customer...
    private double amountFor(Rental aRental) {
        double result = 0;
        switch (aRental.getMovie().getPriceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (aRental.getDaysRented() > 2)
                    result += (aRental.getDaysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
```

```
                result += aRental.getDaysRented() * 3;
                break;
            case Movie.CHILDRENS:
                result += 1.5;
                if (aRental.getDaysRented() > 3)
                    result += (aRental.getDaysRented() - 3) * 1.5;
                break;
        }
        return result;
    }
```

Это приводит к мысли о том, что метод расположен в неправильном объекте. В большинстве случаев метод должен находиться в классе, данные которого он использует, поэтому данный метод нужно переместить в класс, представляющий прокат. Для этого я использую рефакторинг “Перенос метода” (с. 162). При этом весь код сначала копируется в класс проката и соответствующим образом изменяется для корректной работы на новом месте, а затем компилируется.

```
class Rental...
    double getCharge() {
        double result = 0;
        switch (getMovie().getPriceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (getDaysRented() > 2)
                    result += (getDaysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += getDaysRented() * 3;
                break;
            case Movie.CHILDRENS:
                result += 1.5;
                if (getDaysRented() > 3)
                    result += (getDaysRented() - 3) * 1.5;
                break;
        }
        return result;
    }
```

В этом случае изменение для работы в новом местоположении означает удаление параметра. Кроме того, я переименовал этот метод.

Теперь я могу его протестировать. Для этого я модифицирую метод `Customer.amountFor` таким образом, чтобы выполнялось делегирование новому методу.

```
class Customer...
    private double amountFor(Rental aRental) {
        return aRental.getCharge();
    }
```

Теперь я могу выполнить компиляцию и тестирование, чтобы убедиться, что все в порядке.

Следующий шаг состоит в поиске всех обращений к старому методу и их замене обращениями к новому методу.

```
class Customer...
    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Прокат " + getName() + "\n";

        while (rentals.hasMoreElements()) {
            double thisAmount = 0;
            Rental each = (Rental) rentals.nextElement();

            thisAmount = amountFor(each);

            // Начисление бонусных очков
            frequentRenterPoints ++;
            // Бонус за двухдневный прокат новинки
            if ((each.getMovie().getPriceCode()==Movie.NEW_RELEASE)&&
                    each.getDaysRented() > 1) frequentRenterPoints++;

            // Вывод результатов для каждого проката
            result += "\t" + each.getMovie().getTitle() + "\t" +
                      String.valueOf(thisAmount) + "\n";
            totalAmount += thisAmount;
        }

        // Добавление колонтитула
        result += "Сумма задолженности: " +
                  String.valueOf(totalAmount) + "\n";
        result += "Вы заработали "+String.
valueOf(frequentRenterPoints)+
                  " бонусных очков";
        return result;
    }
```

В нашем случае этот шаг выполнить очень просто, потому что мы только что создали метод и он вызывается только в одном месте. Однако в общем случае следует выполнить поиск во всех классах, которые могут использовать этот метод.

```
class Customer
        public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Прокат " + getName() + "\n";
        while (rentals.hasMoreElements()) {
```

```
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();

        thisAmount = each.getCharge();

        // Начисление бонусных очков
        frequentRenterPoints ++;
        // Бонус за двухдневный прокат новинки
        if ((each.getMovie().getPriceCode()==Movie.NEW_RELEASE)&&
                each.getDaysRented() > 1) frequentRenterPoints++;

        // Вывод результатов для каждого проката
        result += "\t" + each.getMovie().getTitle() + "\t" +
                  String.valueOf(thisAmount) + "\n";
        totalAmount += thisAmount;
    }

    // Добавление колонтитула
    result += "Сумма задолженности: " +
              String.valueOf(totalAmount) + "\n";
    result += "Вы заработали "+
              String.valueOf(frequentRenterPoints)+
              " бонусных очков";
    return result;
}
```

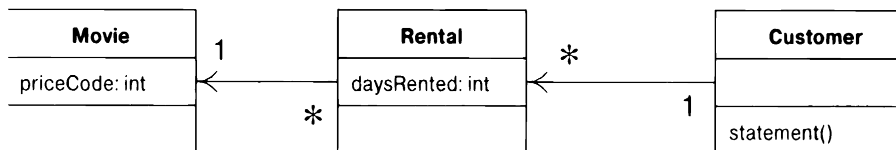


Рис. 1.3. *Диаграмма классов после переноса метода вычисления суммы оплаты*

После внесения изменений (рис. 1.3) следует удалить старый метод. Компилятор должен сообщить, все ли я сделал и не пропустил ли чего-то. Затем я выполняю тестирование для проверки, не нарушилась ли работа программы.

Иногда я сохраняю старый метод для делегирования задания новому методу. Это может быть полезно, если метод объявлен как открытый, а я не хочу изменять интерфейс другого класса.

Конечно, есть еще кое-что, что можно сделать с `Rental.getCharge`, но я пока оставлю его в таком состоянии и вернусь к `Customer.statement`.

```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Прокат " + getName() + "\n";
```

```
    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();

        thisAmount = each.getCharge();

        // Начисление бонусных очков
        frequentRenterPoints ++;
        // Бонус за двухдневный прокат новинки
        if ((each.getMovie().getPriceCode()==Movie.NEW_RELEASE)&&
                each.getDaysRented() > 1) frequentRenterPoints++;

        // Вывод результатов для каждого проката
        result += "\t" + each.getMovie().getTitle() + "\t" +
                  String.valueOf(thisAmount) + "\n";
        totalAmount += thisAmount;
    }

    // Добавление колонтитула
    result += "Сумма задолженности: " +
              String.valueOf(totalAmount) + "\n";
    result += "Вы заработали "+String.valueOf(frequentRenterPoints)+
              " бонусных очков";
    return result;
}
```

Теперь я обнаруживаю ненужность переменной `thisAmount`. Ей присваивается результат `each.charge`, который затем не изменяется. Поэтому можно устранить `thisAmount`, применяя рефакторинг “Замена временной переменной запросом” (с. 141).

```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Прокат " + getName() + "\n";

    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();

        // Начисление бонусных очков
        frequentRenterPoints ++;
        // Бонус за двухдневный прокат новинки
        if ((each.getMovie().getPriceCode()==Movie.NEW_RELEASE)&&
                each.getDaysRented() > 1) frequentRenterPoints++;

        // Вывод результатов для каждого проката
        result += "\t" + each.getMovie().getTitle() + "\t" +
```

```
                String.valueOf(each.getCharge()) + "\n";
        totalAmount += each.getCharge();
    }

    // Добавление колонтитула
    result += "Сумма задолженности: " +
              String.valueOf(totalAmount) + "\n";
    result += "Вы заработали "+String.valueOf(frequentRenterPoints)+
              " бонусных очков";
    return result;
}
```

Внеся это изменение в код, я выполняю компиляцию и тестирование, чтобы проверить работоспособность модифицированного кода.

Я предпочитаю избавляться от временных переменных. Они приводят к множеству проблем, потому что из-за них приходится передавать многие параметры там, где без этого можно обойтись. Можно легко забыть, для чего предназначена та или иная переменная. Особенно коварны они в длинных методах. Правда, за это приходится расплачиваться снижением производительности: сумма оплаты теперь вычисляется дважды. Но можно оптимизировать класс проката, и эта оптимизация оказывается существенно более эффективной, когда код правильно разделен на классы. Об этом будет сказано позже, в разделе "Рефакторинг и производительность" главы 2, "Принципы рефакторинга".

Выделение начисления бонусов

Следующий шаг состоит в выполнении аналогичных действий для подсчета бонусов. Правила могут зависеть от типа фильма, хотя вариантов здесь меньше, чем при оплате. Кажется разумным переложить эту ответственность на класс проката. Сначала надо использовать рефакторинг "Извлечение метода" (с. 132) к коду начисления бонусов (выделен полужирным шрифтом).

```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Прокат " + getName() + "\n";

    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();

        // Начисление бонусных очков
        frequentRenterPoints ++;
        // Бонус за двухдневный прокат новинки
        if ((each.getMovie().getPriceCode()==Movie.NEW_RELEASE)&&
                each.getDaysRented() > 1) frequentRenterPoints++;
```

```
        // Вывод результатов для каждого проката
        result += "\t" + each.getMovie().getTitle() + "\t" +
                  String.valueOf(each.getCharge()) + "\n";
        totalAmount += each.getCharge();
    }

    // Добавление колонтитула
    result += "Сумма задолженности: " +
              String.valueOf(totalAmount) + "\n";
    result += "Вы заработали "+String.valueOf(frequentRenterPoints)+
              " бонусных очков";
    return result;
}
```

Мы вновь ищем переменные с локальной областью видимости. У нас имеется переменная each, которую можно передать в качестве параметра. Есть и еще одна временная переменная frequentRenterPoints. Здесь frequent RenterPoints имеет значение, присвоенное ей ранее. Однако в теле выделяемого метода эта переменная не считывается, так что при использовании присваивающего сложения передавать ее в качестве параметра не следует.

Я выполнил выделение метода, компиляцию и тестирование кода, а затем перемещение и снова после него скомпилировал и протестировал код. Чтобы не столкнуться с неприятностями, рефакторинг лучше всего делать небольшими шагами.

```
class Customer...
    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Прокат " + getName() + "\n";

        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            frequentRenterPoints += each.getFrequentRenterPoints();

            // Вывод результатов для каждого проката
            result += "\t" + each.getMovie().getTitle() + "\t" +
                      String.valueOf(each.getCharge()) + "\n";
            totalAmount += each.getCharge();
        }

        // Добавление колонтитула
        result += "Сумма задолженности: " +
                  String.valueOf(totalAmount) + "\n";
        result += "Вы заработали "+
                  String.valueOf(frequentRenterPoints)+
                  " бонусных очков";
```

```
        return result;
    }

class Rental...
    int getFrequentRenterPoints() {
        if ((getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
                getDaysRented() > 1)
            return 2;
        else
            return 1;
    }
```

Подытожим внесенные изменения с помощью диаграмм унифицированного языка моделирования (UML), демонстрирующих состояния кода до и после выполнения рефакторинга (рис. 1.4–1.7). Сначала идут диаграммы, показывающие состояние до внесения изменений, а затем — после такового.

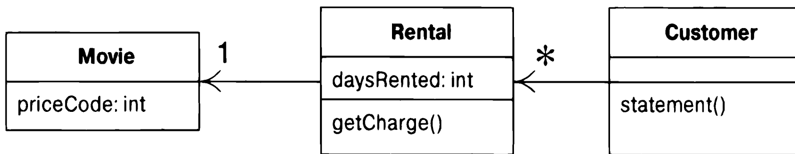


Рис. 1.4. *Диаграмма классов до выделения и переноса кода начисления бонусов*

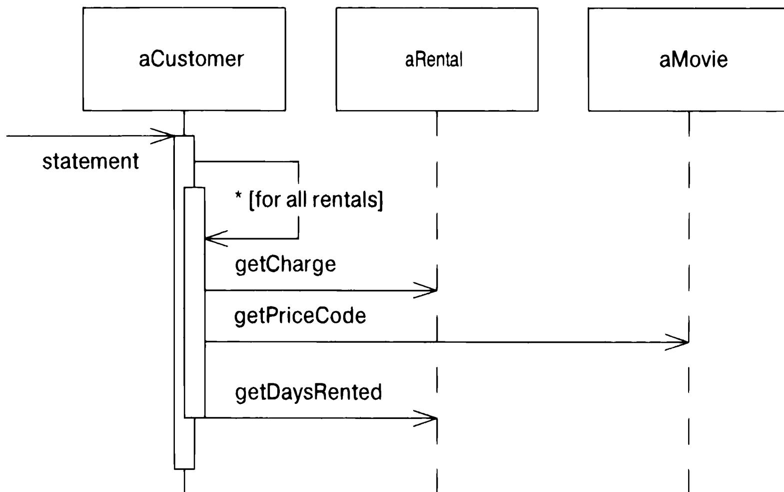


Рис. 1.5. *Диаграмма последовательности до выделения и переноса кода начисления бонусов*

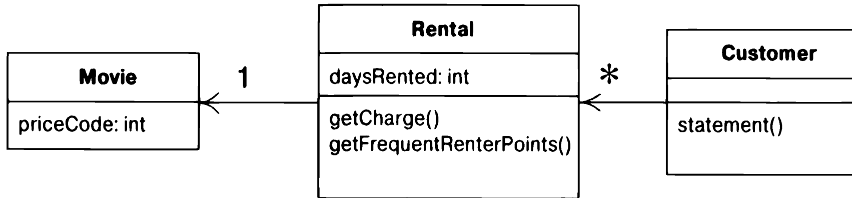


Рис. 1.6. *Диаграмма классов после выделения и переноса кода начисления бонусов*

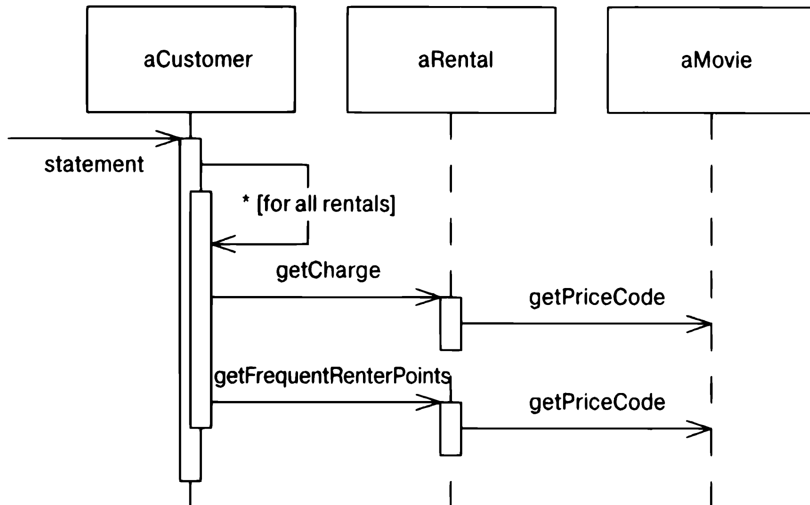


Рис. 1.7. *Диаграмма последовательности после выделения и переноса кода начисления бонусов*

Устранение временных переменных

Как я упоминал ранее, временные переменные могут приводить к проблемам. Эти переменные используются только в собственных методах и приводят к их удлинению и усложнению. В нашем случае имеются две временные переменные, использующиеся для получения итоговых сумм по прокатным операциям клиента. Эти итоговые суммы требуются для обеих версий отчета — и текстовой, и HTML. Я планирую выполнить рефакторинг "Замена временной переменной запросом" (с. 141) и заменить `totalAmount` и `frequentRentalPoints` вызовами методов запросов. Запросы доступны любому методу класса, а потому обеспечивают более ясный проект кода без длинных и сложных методов.

```
class Customer...
    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Прокат " + getName() + "\n";

        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            frequentRenterPoints += each.getFrequentRenterPoints();

            // Вывод результатов для каждого проката
            result += "\t" + each.getMovie().getTitle() + "\t" +
                      String.valueOf(each.getCharge()) + "\n";
            totalAmount += each.getCharge();
        }

        // Добавление колонтитула
        result += "Сумма задолженности: " +
                  String.valueOf(totalAmount) + "\n";
        result += "Вы заработали "+String.
valueOf(frequentRenterPoints)+
                  " бонусных очков";
        return result;
    }
```

Я начинаю с замены переменной `totalAmount` вызовом метода `getTotal Charge`.

```
class Customer...
    public String statement() {
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Прокат " + getName() + "\n";

        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            frequentRenterPoints += each.getFrequentRenterPoints();

            // Вывод результатов для каждого проката
            result += "\t" + each.getMovie().getTitle() + "\t" +
                      String.valueOf(each.getCharge()) + "\n";
        }

        // Добавление колонтитула
        result += "Сумма задолженности: " +
                  String.valueOf(getTotalCharge()) + "\n";
        result += "Вы заработали "+
                  String.valueOf(frequentRenterPoints)+
                  " бонусных очков";
        return result;
    }
```

```
    private double getTotalCharge() {
        double result = 0;
        Enumeration rentals = _rentals.elements();
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            result += each.getCharge();
        }
        return result;
    }
```

Это не простейший вариант рефакторинга “Замена временной переменной запросом” (с. 141), так как присваивание `totalAmount` выполнялось в цикле, который необходимо копировать в метод запроса.

После компиляции и тестирования рефакторинга я делаю то же самое и для переменной `frequentRenterPoints`. Ниже приведен код до и после выполнения рефакторинга.

```
class Customer...
    public String statement() {
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Прокат " + getName() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            frequentRenterPoints += each.getFrequentRenterPoints();

            // Вывод результатов для каждого проката
            result += "\t" + each.getMovie().getTitle() + "\t" +
                      String.valueOf(each.getCharge()) + "\n";
        }

        // Добавление колонтитула
        result += "Сумма задолженности: " +
                  String.valueOf(getTotalCharge()) + "\n";
        result += "Вы заработали "+
                  String.valueOf(frequentRenterPoints)+
                  " бонусных очков";
        return result;
    }

========== После рефакторинга ========================================

    public String statement() {
        Enumeration rentals = _rentals.elements();
        String result = "Прокат " + getName() + "\n";
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();

            // Вывод результатов для каждого проката
            result += "\t" + each.getMovie().getTitle() + "\t" +
                      String.valueOf(each.getCharge()) + "\n";
        }
```

```
        // Добавление колонтитула
        result += "Сумма задолженности: " +
                  String.valueOf(getTotalCharge()) + "\n";
        result += "Вы заработали " +
                  String.valueOf(getTotalFrequentRenterPoints()) +
                  " бонусных очков";
        return result;
    }

    private int getTotalFrequentRenterPoints() {
        int result = 0;
        Enumeration rentals = _rentals.elements();
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            result += each.getFrequentRenterPoints();
        }
        return result;
    }
```

На рис. 1.8–1.11 можно увидеть изменения, внесенные рефакторингом в диаграммы классов и взаимодействий для метода `statement`.

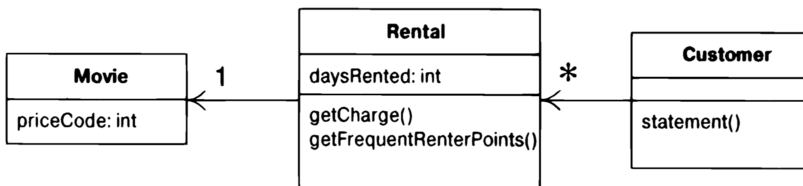


Рис. 1.8. *Диаграмма классов до выделения и переноса кода подсчета итоговых сумм*

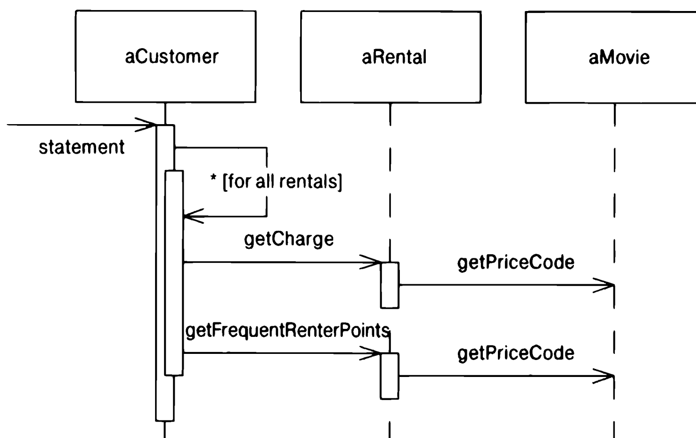


Рис. 1.9. *Диаграмма последовательности до выделения и переноса кода подсчета итоговых сумм*

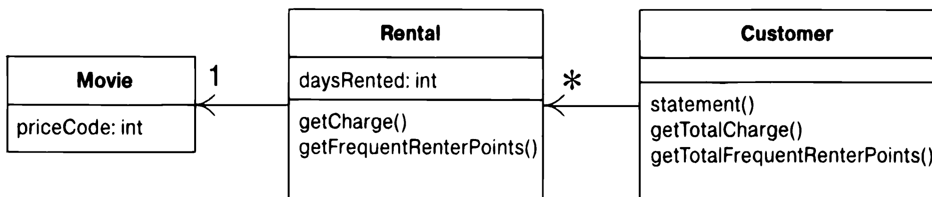


Рис. 1.10. *Диаграмма классов после выделения и переноса кода подсчета итоговых сумм*

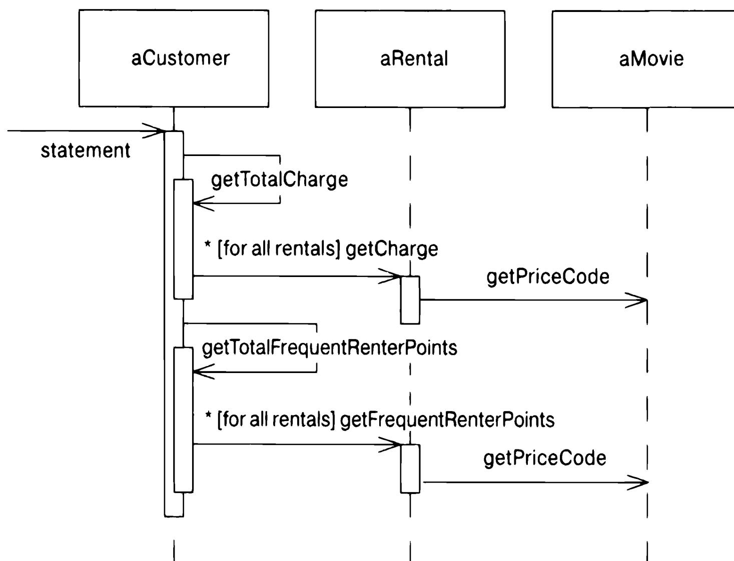


Рис. 1.11. *Диаграмма последовательности после выделения и переноса кода подсчета итоговых сумм*

Сейчас имеет смысл ненадолго остановиться и подумать о последнем рефакторинге. Большинство методов рефакторинга приводят к уменьшению объема кода, но не в данном случае. Дело в том, что Java 1.1 требует большого количества команд для цикла суммирования. Даже простой цикл суммирования с единственной строкой требует шесть строк поддержки. Это очевидная для любого программиста идиома, но требующая большого количества строк.

Еще одной проблемой, связанной с рефакторингом такого вида, является вопрос производительности. Прежний код выполнял цикл `while` один раз, новый код делает это три раза. Длительный цикл `while` может плохо сказаться на производительности. Для многих программистов этого достаточно, чтобы отказаться от рефакторинга. Однако обратите внимание на слово “может”.

Без профилирования невозможно уверенно сказать, сколько времени и как часто будет выполняться цикл и как это повлияет на итоговую производительность системы. Не беспокойтесь об этом при рефакторинге. Вы вернетесь к этому при оптимизации, но к тому времени ее проведение будет существенно упрощено благодаря выполненному улучшению кода. Вы получите бóльшие возможности для эффективной оптимизации (этот вопрос рассматривается в разделе "Рефакторинг и производительность" главы 2, "Принципы рефакторинга").

Созданные методы запроса доступны любому коду класса `Customer`. Если возвращаемая ими информация будет нужна в других частях нашей программы, эти методы легко добавить в интерфейс класса. При отсутствии данных методов запросов другим методам приходится иметь дело с внутренним устройством класса `Rental` и создавать циклы. В сложной программе это потребует написания большого объема исходного текста.

Вы почувствуете разницу, когда рассмотрите метод `htmlStatement`. Сейчас я займусь добавлением методов. Я могу написать метод `htmlStatement` так, как показано ниже (и, конечно же, добавить соответствующие тесты).

```
public String htmlStatement() {
    Enumeration rentals = _rentals.elements();
    String result = "<H1>Прокат <EM>" + getName() +
                    "</EM></H1><P>\n";

    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        // Вывод результатов для каждого проката
        result += each.getMovie().getTitle() + ": " +
                  String.valueOf(each.getCharge()) + "<BR>\n";
    }

    // Добавление колонтитула
    result += "<P>Сумма задолженности <EM>" +
              String.valueOf(getTotalCharge()) +
              "</EM><P>\n";
    result += "Вы заработали <EM>" +
              String.valueOf(getTotalFrequentRenterPoints()) +
              "</EM> бонусных очков<P>";
    return result;
}
```

Вынеся расчеты, я могу теперь написать метод `htmlStatement` с повторным использованием всего кода вычислений, который ранее был в исходной версии метода `statement`. Я не прибегал к копированию и вставке, так что, если правила расчетов изменятся, переписывать код придется в единственном месте. Можно также легко и просто добавить любой иной формат отчета. Выполнение рефакторинга не отняло много времени, причем бóльшую часть этого времени я выяснял, что именно делает данный код, а это я делал бы в любом случае.

Часть кода скопирована из текстовой версии, главным образом при создании цикла. Это также можно улучшить, выполнив дальнейший рефакторинг. Один из вариантов — выделение методов для заголовка, нижнего колонтитула и строки с подробной информацией. Как всего этого добиться, можно узнать из примера к рефакторингу “Формирование шаблонного метода” (с. 361). Но сейчас пользователи вновь ставят новые требования к программе. Они хотят изменить классификацию фильмов. Пока непонятно, как именно, но, скорее всего, будут введены новые категории фильмов (а старые будут изменены). Само собой разумеется, что для новых категорий будет установлен свой порядок оплаты и начисления бонусов. При нынешнем состоянии дел внесение изменений такого рода является сложной задачей. Внесение изменений в классификацию фильмов требует изменения в коде выбора с условными выражениями в методах вычисления оплаты и бонусных очков. Давайте вновь вернемся к рефакторингу.

Замена логики условий полиморфизмом

Первая сложность нашей задачи заключается в инструкции `switch`. Неудачна сама идея организации выбора в зависимости от атрибута некоторого другого объекта. Если даже необходимо применение инструкции `switch`, то она должна использовать ваши собственные данные, а не полученные откуда-то.

```
class Rental...
    double getCharge() {
        double result = 0;
        switch (getMovie().getPriceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (getDaysRented() > 2)
                    result += (getDaysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += getDaysRented() * 3;
                break;
            case Movie.CHILDRENS:
                result += 1.5;
                if (getDaysRented() > 3)
                    result += (getDaysRented() - 3) * 1.5;
                break;
        }
        return result;
    }
```

Это соображение приводит к тому, что метод `getCharge` должен быть перенесен в класс `Movie`.

```
class Movie...
    double getCharge(int daysRented) {
        double result = 0;
        switch (getPriceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (daysRented > 2)
                    result += (daysRented - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += daysRented * 3;
                break;
            case Movie.CHILDRENS:
                result += 1.5;
                if (daysRented > 3)
                    result += (daysRented - 3) * 1.5;
                break;
        }
        return result;
    }
```

Чтобы этот код был работоспособен, мне пришлось передать в него продолжительность проката, которая представляет собой информацию из класса `Rental`. Этот метод использует два элемента данных — продолжительность проката и тип фильма. Почему я предпочел передавать продолжительность проката классу `Movie`, а не тип фильма классу `Rental`? Это связано с тем, что предполагаемые изменения касаются только введения новых типов фильмов. Информация о типах в общем случае более подвержена изменениям. Я хочу, чтобы при изменениях типов фильмов влияние вносимых изменений распространялось как можно меньше, и поэтому решил вычислять сумму оплаты в классе `Movie`.

Я внес рассмотренный метод в `Movie` и изменил `getCharge` из `Rental` так, чтобы он использовал новый метод (рис. 1.12 и 1.13).

```
class Rental...
    double getCharge() {
        return _movie.getCharge(_daysRented);
    }
```

После переноса метода `getCharge` я выполнил то же самое с методом начисления бонусов. Таким образом, оба элемента, зависящие от типа фильма, оказываются вместе в классе, содержащем информацию об этом типе.

```
class Rental...
    int getFrequentRenterPoints() {
        if ((getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
                getDaysRented() > 1)
            return 2;
        else
            return 1;
    }
```

```
========= После рефакторинга ========================================

class Rental...
    int getFrequentRenterPoints() {
        return _movie.getFrequentRenterPoints(_daysRented);
    }

class Movie...
    int getFrequentRenterPoints(int daysRented) {
        if ((getPriceCode() == Movie.NEW_RELEASE) && daysRented > 1)
            return 2;
        else
            return 1;
    }
```

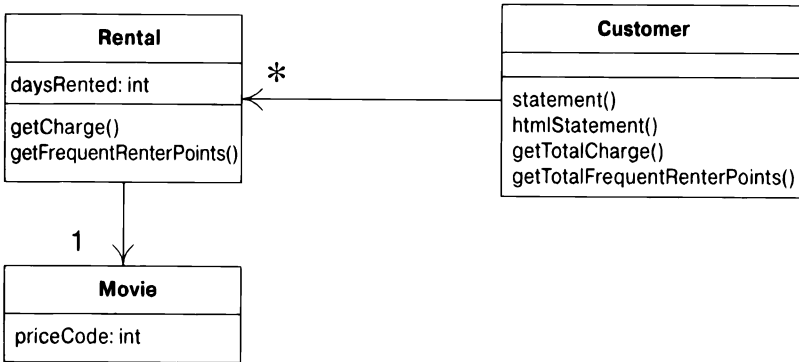


Рис. 1.12. *Диаграмма классов до вынесения методов в* `Movie`

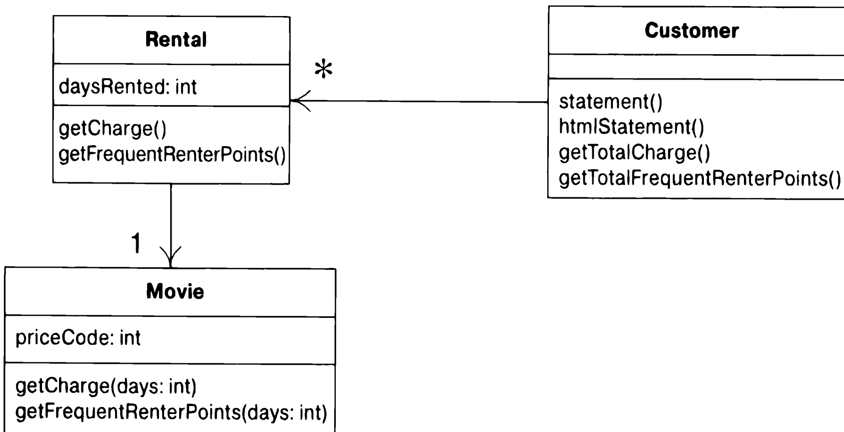


Рис. 1.13. *Диаграмма классов после вынесения методов в* `Movie`

А теперь — наследование

У нас есть несколько типов фильмов, которые по-разному отвечают на один и тот же вопрос. Так что здесь вполне могут пригодиться подклассы. Мы можем использовать три подкласса `Movie`, каждый из которых будет иметь собственную версию метода начисления платы (рис. 1.14).

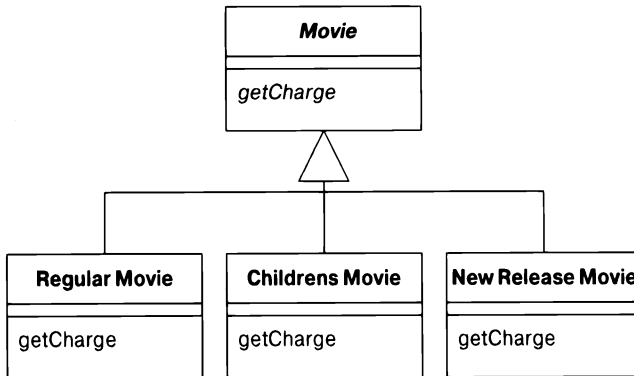


Рис. 1.14. *Использование наследования от класса* `Movie`

Это позволит нам заменить инструкцию `switch` полиморфизмом. К сожалению, у этого решения есть маленький недостаток: оно не работает. За время своего существования фильм может изменить тип, но объект во время своего существования не в состоянии изменить свой класс. Однако выход из этой ситуации есть — это проектный шаблон "Состояние" (State pattern) [9]. Классы при этом приобретают следующий вид (рис. 1.15).

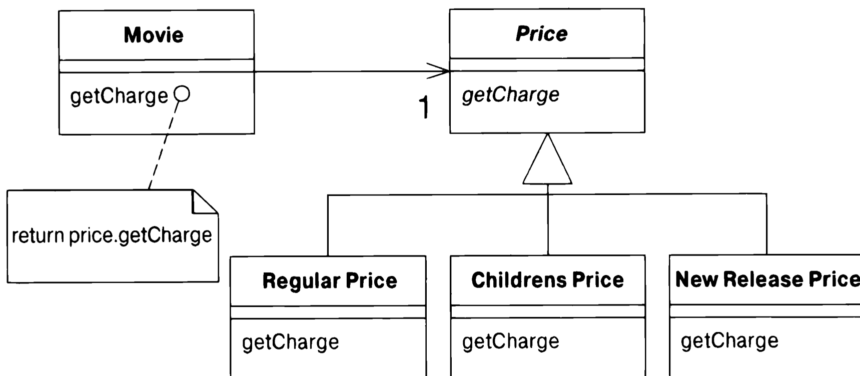


Рис. 1.15. *Применение проектного шаблона "Состояние" к классу* `Movie`

Добавив дополнительный уровень косвенности, можно создавать подклассы для класса Price и изменять цену в случае необходимости.

У тех, кто знаком с проектными шаблонами из указанной книги [9], может возникнуть вопрос: "Это состояние или стратегия?" Представляет ли класс Price алгоритм расчета цены (тогда я предпочел бы назвать его Pricer или PricingStrategy) или состояние фильма (*Star Trek X* — новинка проката)? На этом этапе выбор проектного шаблона (и имени) отражает способ желаемого представления структуры данных. Сейчас я представляю ее как состояние фильма. Если позднее я решу, что мои намерения лучше передает стратегия, я выполню рефакторинг и сменю имена.

Для того чтобы прибегнуть к проектному шаблону "Состояние", я использую три рефакторинга. Сначала я переношу код, зависящий от типа, в шаблон состояния с помощью рефакторинга "Замена кода типа состоянием/стратегией" (с. 245). Затем я могу использовать рефакторинг "Перенос метода" (с. 162), с помощью которого перенесу инструкцию switch в класс Price. И наконец с помощью рефакторинга "Замена условной инструкции полиморфизмом" (с. 271) я устраняю инструкцию switch вовсе.

Начнем с рефакторинга "Замена кода типа состоянием/стратегией" (с. 245). Этот первый этап заключается в применении рефакторинга "Самоинкапсуляция поля" (с. 190), чтобы гарантировать выполнение любых действий через методы получения и установки значений. Поскольку в основном код взят из других классов, в большинстве методов уже используются методы получения значений. Однако конструкторы устанавливают цену непосредственно:

```
class Movie...
    public Movie(String name, int priceCode) {
        _name = name;
        _priceCode = priceCode;
    }
```

Вместо этого я использую метод установки значения.

```
class Movie
    public Movie(String name, int priceCode) {
        _name = name;
        setPriceCode(priceCode);
    }
```

Я компилирую и тестирую получившийся код, чтобы гарантировать отсутствие внесенных ошибок. Теперь я добавляю новые классы. Код, зависящий от типа, предоставляется в объекте Price. Я достигаю этого с использованием абстрактного метода в классе Price и конкретных методов в подклассах.

```
abstract class Price {
    abstract int getPriceCode();
}
class ChildrensPrice extends Price {
    int getPriceCode() {
        return Movie.CHILDRENS;
    }
}
class NewReleasePrice extends Price {
    int getPriceCode() {
        return Movie.NEW_RELEASE;
    }
}
class RegularPrice extends Price {
    int getPriceCode() {
        return Movie.REGULAR;
    }
}
```

Теперь я могу скомпилировать новые классы.

Далее для использования нового класса я должен изменить методы доступа.

```
public int getPriceCode() {
    return _priceCode;
}
public setPriceCode(int arg) {
    _priceCode = arg;
}
private int _priceCode;
```

Это означает замену поля кода цены полем цены и изменение методов доступа.

```
class Movie...
    public int getPriceCode() {
        return _price.getPriceCode();
    }
    public void setPriceCode(int arg) {
        switch (arg) {
            case REGULAR:
                _price = new RegularPrice();
                break;

            case CHILDRENS:
                _price = new ChildrensPrice();
                break;

            case NEW_RELEASE:
                _price = new NewReleasePrice();
                break;
```

```
            default:
                throw new IllegalArgumentException(
                          "Incorrect Price Code");
        }
    }
    private Price _price;
```

Теперь я вновь могу выполнить компиляцию и тестирование, и при этом более сложные методы ничего не узнают о внесенных изменениях.

Применим к `getCharge` рефакторинг "Перенос метода" (с. 162).

```
class Movie...
    double getCharge(int daysRented) {
        double result = 0;
        switch (getPriceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (daysRented > 2)
                    result += (daysRented - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += daysRented * 3;
                break;
            case Movie.CHILDRENS:
                result += 1.5;
                if (daysRented > 3)
                    result += (daysRented - 3) * 1.5;
                break;
        }
        return result;
    }
```

Выполнить перенос очень легко.

```
class Movie...
    double getCharge(int daysRented) {
        return _price.getCharge(daysRented);
    }

class Price...
    double getCharge(int daysRented) {
        double result = 0;
        switch (getPriceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (daysRented > 2)
                    result += (daysRented - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += daysRented * 3;
```

```
                break;
            case Movie.CHILDRENS:
                result += 1.5;
                if (daysRented > 3)
                    result += (daysRented - 3) * 1.5;
                break;
        }
        return result;
    }
```

После переноса я могу приступить к рефакторингу “Замена условной инструкции полиморфизмом” (с. 271).

```
class Price...
    double getCharge(int daysRented) {
        double result = 0;
        switch (getPriceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (daysRented > 2)
                    result += (daysRented - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += daysRented * 3;
                break;
            case Movie.CHILDRENS:
                result += 1.5;
                if (daysRented > 3)
                    result += (daysRented - 3) * 1.5;
                break;
        }
        return result;
    }
```

Я выполняю его, беря по одной ветви из инструкции `switch` и создавая перекрывающий метод. Начну с `RegularPrice`.

```
class RegularPrice...
    double getCharge(int daysRented) {
        double result = 2;
        if (daysRented > 2)
            result += (daysRented - 2) * 1.5;
        return result;
    }
```

Так перекрывается родительская инструкция `case`, которую я оставляю в ее первоначальном виде. После компиляции и тестирования одной ветви я принимаюсь за следующую. (Чтобы убедиться, что действительно выполняется код подкласса, я умышленно допускаю ошибку и выполняю код, чтобы убедиться, что тесты неудачны.)

```
class ChildrensPrice
    double getCharge(int daysRented) {
        double result = 1.5;
        if (daysRented > 3)
            result += (daysRented - 3) * 1.5;
        return result;
    }

class NewReleasePrice...
    double getCharge(int daysRented) {
        return daysRented * 3;
    }
```

Выполнив рефакторинг для всех ветвей, я объявляю метод `Price.getCharge` абстрактным.

```
class Price...
    abstract double getCharge(int daysRented);
```

Теперь я могу проделать ту же процедуру с `getFrequentRenterPoints`.

```
class Movie...
    int getFrequentRenterPoints(int daysRented) {
        if ((getPriceCode() == Movie.NEW_RELEASE) && daysRented > 1)
            return 2;
        else
            return 1;
    }
```

Сначала я переношу метод в класс `Price`.

```
Class Movie...
    int getFrequentRenterPoints(int daysRented) {
        return _price.getFrequentRenterPoints(daysRented);
    }

Class Price...
    int getFrequentRenterPoints(int daysRented) {
        if ((getPriceCode() == Movie.NEW_RELEASE) && daysRented > 1)
            return 2;
        else
            return 1;
    }
```

Однако в этом случае я не делаю метод суперкласса абстрактным. Вместо этого я создаю перекрывающий метод в новых версиях и оставляю определение метода в качестве используемого по умолчанию в суперклассе.

```
Class NewReleasePrice
    int getFrequentRenterPoints(int daysRented) {
```

```
        return (daysRented > 1) ? 2 : 1;
    }

Class Price...
    int getFrequentRenterPoints(int daysRented) {
        return 1;
    }
```

Внедрение проектного шаблона "Состояние" потребовало существенных усилий. Стоили ли они того? Полученная выгода в том, что если необходимо изменить поведение Price, добавить новые цены или дополнительное поведение, зависящее от цены, реализовать такие модификации будет значительно легче. Остальные части приложения ничего не знают о применении проектного шаблона "Состояние". Для имеющегося в настоящий момент небольшого набора функций это не играет особой роли. Однако в более сложной системе, где поведений, зависящих от цены, больше десятка, разница будет заметна. Все внесенные изменения выполнялись малыми порциями. И хотя писать код таким образом достаточно утомительно, мне ни разу не пришлось прибегать к помощи отладчика; таким образом, процесс рефакторинга на деле прошел весьма быстро. На самом деле мне понадобилось куда больше времени на написание этого раздела книги, чем для внесения изменений в код.

Второй из основных рефакторингов выполнен. Теперь изменить структуру классификации фильмов или правила начисления оплаты и бонусов будет значительно проще. На рис. 1.16 и 1.17 показано, как примененный мною проектный шаблон "Состояние" работает с информацией о ценах.

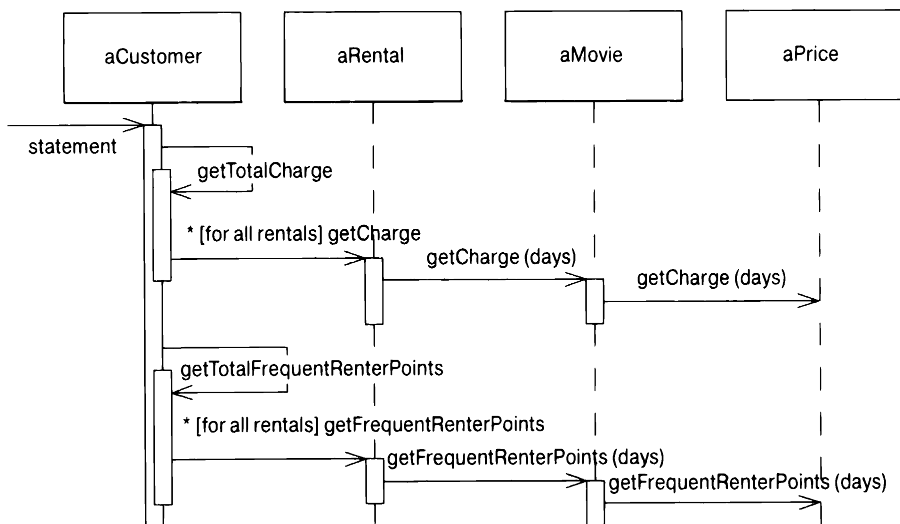


Рис. 1.16. *Взаимодействия с использованием проектного шаблона "Состояние"*

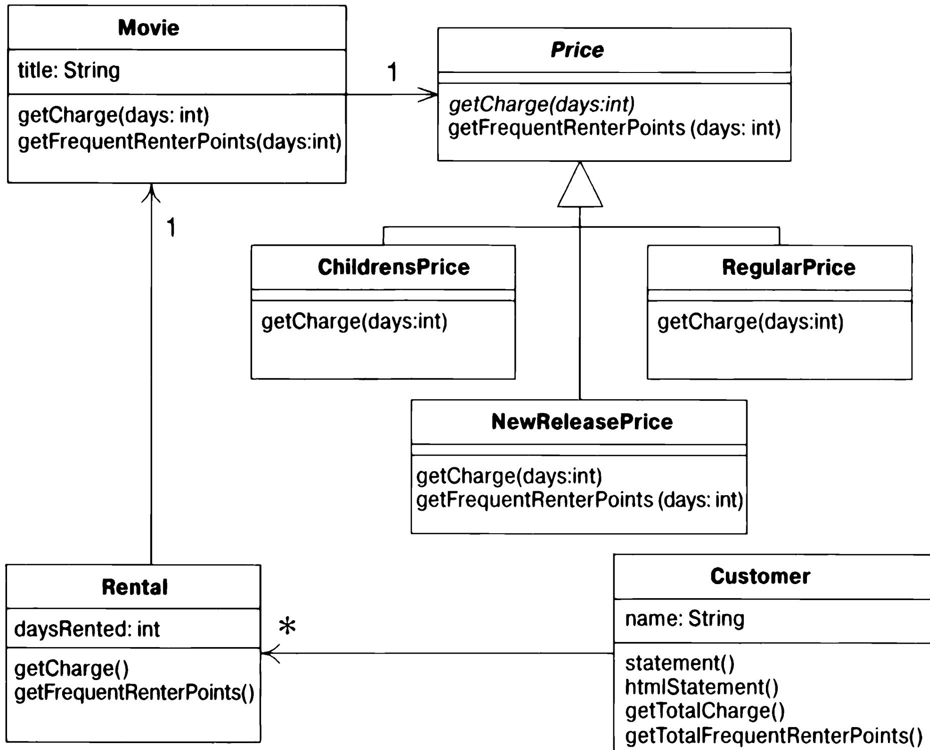


Рис. 1.17. *Диаграмма классов после добавления проектного шаблона "Состояние"*

Заключительные замечания

Надеюсь, этот простой пример дал вам возможность почувствовать, что же такое рефакторинг. Здесь я использовал несколько видов рефакторинга, в том числе рефакторинги "Извлечение метода" (с. 132), "Перенос метода" (с. 162) и "Замена условной инструкции полиморфизмом" (с. 271). Все они ведут к более правильному распределению ответственности и облегчают поддержку кода. Этот код существенно отличается от кода, выдержанного в процедурном стиле, и для свободного использования требует определенной привычки. Но, привыкнув использовать этот стиль, трудно возвращаться к процедурным программам.

Наиболее важный урок из данного примера состоит в ритме рефакторинга: тестирование, небольшие изменения, тестирование, небольшие изменения, тестирование, небольшие изменения. Именно этот ритм делает рефакторинг быстрым и надежным.

Дойдя со мной до этого места, вы должны понимать необходимость рефакторинга. Теперь можно немного заняться основами, принципами и теорией рефакторинга (хотя и не слишком серьезно).

Глава 2

Принципы рефакторинга

Приведенный в предыдущей главе пример должен дать хорошее представление о том, что такое рефакторинг. Теперь можно сделать шаг назад и рассмотреть ключевые принципы рефакторинга и некоторые вопросы, о которых следует подумать при выполнении рефакторинга.

Определение рефакторинга

Я всегда настороженно отношусь к определениям, поскольку у каждого человека они свои. Но когда пишешь книгу, приходится выбирать свой вариант и не приходится выбирать, нужно ли давать определения. Я основываю мои определения на работе, выполненной группой Ральфа Джонсона (Ralph Johnson) с коллегами.

Первое, что следует отметить, — это наличие двух разных определений термина "рефакторинг" в зависимости от контекста. Этот факт может быть раздражающим (для меня — определенно), но он служит еще одним примером сложностей и проблем, возникающих при работе с естественными языками.

Вот как выглядит первое определение — для существительного.

Рефакторинг: *изменения внутренней структуры программного обеспечения, направленные на облегчение понимания его работы и упрощение модификации без изменения наблюдаемого поведения.*

Примеры рефакторинга можно найти в каталоге в данной книге, например рефакторинг "Извлечение метода" (с. 132) или "Подъем поля" (с. 338). Сам по себе рефакторинг обычно представляет собой небольшие изменения исходного текста, хотя один рефакторинг может включать другие. Например, рефакторинг "Извлечение класса" (с. 169) обычно подразумевает применение рефакторингов "Перенос метода" (с. 162) и "Перенос поля" (с. 166).

Еще одно определение термина “рефакторинг” для глагола.

Рефакторинг: *внесение изменений в структуру программного обеспечения, применяя ряд преобразований, не затрагивающих поведение модифицируемого программного обеспечения.*

Таким образом, можно в течение часов заниматься рефакторингом как глаголом и при этом выполнить пару дюжин рефакторингов-существительных.

Иногда меня спрашивают: “Можно ли считать, что рефакторинг — это просто приведение исходных текстов в порядок?” В определенном смысле это так, но мне кажется, что рефакторинг — нечто большее, поскольку он не просто приводит код в порядок, но и позволяет делать это более эффективно и управляемо. Мой опыт говорит, что, применяя рефакторинг, я стал работать значительно эффективнее, чем раньше. И это не удивительно, ведь теперь я знаю, какие методы рефакторинга использовать, умею применять их так, чтобы не вносить новые ошибки, и при каждой возможности провожу тестирование.

Хочу остановиться на паре моментов в приведенных определениях. Во-первых, цель рефакторинга заключается в упрощении понимания и модификации исходных текстов программ. Можно внести огромное количество изменений в программу, при которых ее видимое поведение не изменится или изменится крайне незначительно. Но рефакторингом являются только те изменения, цель которых — облегчение понимания и сопровождения исходных текстов. Хорошим противоположным примером может служить оптимизация производительности. Как и рефакторинг, оптимизация производительности обычно не изменяет функциональность программы (но изменяет ее производительность). Она изменяет только его внутреннее представление. Цели рефакторинга и оптимизации различны. Оптимизация производительности зачастую делает исходный текст более сложным для понимания, но это цена достижения желаемой производительности.

Во-вторых, я хочу заметить, что рефакторинг не меняет видимое поведение программы. Она продолжает функционировать, как и раньше. Ни пользователь, ни программист — никто не скажет по внешнему виду, что что-то в программе изменилось.

Меняя шляпы

Это второе замечание связано с метафорой Кента Бека о двух шляпах. Используя рефакторинг при разработке программного обеспечения, вы делите рабочее время между двумя разными видами деятельности: добавлением функциональности и рефакторингом. При добавлении новой функциональности вы не должны менять структуру существующего кода; вы просто добавляете новые

возможности. Достигаемый результат можно оценить, добавляя тесты и добиваясь их нормальной работы. При проведении рефакторинга вы стараетесь не добавлять функциональность, а только улучшать структуру исходных текстов. При этом новые тесты не добавляются (разве что в случае обнаружения пропущенного ранее тестового случая); изменяются тесты только тогда, когда это совершенно необходимо для проверки изменений в интерфейсе.

При разработке программного обеспечения иногда необходимо часто переключаться между указанными видами деятельности. Пытаясь добавить новую функциональность, можно выяснить, что если изменить саму структуру кода, то добиться этого будет куда легче. В таком случае нужно на время снять шляпу разработчика и одеть шляпу специалиста по рефакторингу. Изменив структуру кода, можно вновь одеть шляпу программиста и добавить новую функциональность. Добившись ее работоспособности, можно обратить внимание, что она написана так, что ее трудно понять, и тогда следует снова сменить шляпу и заняться рефакторингом. Эти роли могут меняться буквально каждые десять минут, но в каждый момент времени вы должны ясно отдавать себе отчет, какой именно деятельностью вы сейчас заняты.

Почему нужно заниматься рефакторингом

Я не стану утверждать, что рефакторинг — лекарство от всех болезней. Это не панацея. Это просто очень ценный инструмент — некие "серебряные плоскогубцы", позволяющие надежно вцепиться в код. Рефакторинг представляет собой инструмент, который можно и нужно использовать для нескольких целей.

Рефакторинг совершенствует проектирование программного обеспечения

Без рефакторинга проект программы постепенно ухудшается. Чем больше в код вносится изменений, связанных с достижением краткосрочных целей или вносимых без учета всех особенностей проекта в целом, тем больше программное обеспечение теряет структурированность. Становится все труднее понимать проект, просто читая код. В определенном смысле рефакторинг — это уборка кода. Из него устраняются части, находящиеся не на своих местах. Потеря структурности кода обычно накапливается с ускорением: чем труднее разобраться в проекте кода, тем сложнее его сохранить и тем быстрее происходит ухудшение. Регулярный рефакторинг поможет сохранить форму кода.

Плохо спроектированное программное обеспечение обычно требует большего кода, зачастую просто потому, что этот код выполняет в разных местах одни и

те же действия. Поэтому важной частью улучшения проекта является устранение дублирования кода. Важность этого связана с будущими изменениями кода. Уменьшение кода не делает программу более быстрой, так как размер программы в памяти редко играет большую роль. Однако размер кода играет роль при внесении в него изменений. Чем больше код, тем труднее корректно внести в него изменения, а чтобы понять, как он работает, приходится разбираться в большом количестве строк. При внесении изменений в код в одном месте программа может вести себя не так, как предполагалось, поскольку не внесены изменения в некоторый другой участок, который выполняет те же действия, но в ином контексте. Устраняя дублирование, мы гарантируем, что код содержит все, что нужно, причем в единственном месте, — а в этом и состоит суть хорошего проектирования.

Рефакторинг упрощает понимание программ

Во многом программирование напоминает общение с компьютером. Программист пишет код, который указывает компьютеру, что тот должен делать, и в ответ компьютер делает именно то, что ему сказано. Постепенно вы уменьшаете разрыв между тем, что должен делать компьютер, и тем, что вы ему говорите. Таким образом, суть программирования сводится к тому, чтобы абсолютно точно сказать компьютеру, что от него требуется. Но программа предназначена не только для компьютера. Пройдет некоторое время, и кому-нибудь может понадобиться ваш исходный текст, чтобы внести некоторые изменения. Об этом будущем программисте часто забывают, но он, по сути, и есть главный пользователь вашего исходного текста (не программы, а именно исходного текста). Вряд ли кого-то сильно обеспокоит несколько дополнительных минут для компиляции, но будет плохо, если программист потратит неделю на внесение изменений, которое заняло бы один час, если бы он мог сразу же разобраться в коде.

Проблема в том, что, пытаясь заставить программу работать, вы совсем не думаете о тех, кто будет заниматься ею в будущем. Внесение в код изменений, облегчающих его понимание, меняет ритм работы. Рефакторинг делает код более простым для чтения и понимания. При выполнении рефакторинга вы берете работоспособный код, который не отличается красотой структуры. Затратив немного времени на рефакторинг, можно добиться лучшей самодокументируемости кода. Теперь суть программирования сводится к тому, чтобы абсолютно точно сказать, что вы имеете в виду.

Такой подход связан не только с альтруизмом. Этим будущим разработчиком часто бываю я сам. И тогда рефакторинг приобретает особую важность. Я очень ленив. В частности, моя лень проявляется в том, что я не запоминаю деталей кода, который пишу. Более того, опасаясь перегрузить свою голову, я умышленно не запоминаю многие детали. И поэтому я стараюсь записывать в коде все, что

иначе мне пришлось бы держать в моей бедной голове. Это позволяет мне не беспокоиться о воздействии пары бокалов пива после работы на клетки моего головного мозга.

Рефакторинг помогает лучше понимать и чужой код. Когда я смотрю на чужой код, я пытаюсь сообразить, как он работает. Я смотрю на строки исходного текста и говорю сам себе: "Ага, так вот что делает этот фрагмент кода..." Применяя рефакторинг, я не только понимаю, как работает код, но и изменяю его так, чтобы он лучше отражал мое понимание, а затем тестирую свое понимание, выполняя модифицированный код и убеждаясь в его корректной работе.

С самого начала я выполняю рефакторинг небольших деталей. Когда код становится более ясным, я нахожу в нем то, что было незаметно раньше. Если бы я не вносил изменения, я, скорее всего, не увидел бы этих новых деталей — я не столь сообразителен, чтобы суметь представить все это визуально в своей голове. Ральф Джонсон (Ralph Johnson) сравнивает первые шаги рефакторинга с мытьем окон, которое позволяет видеть яснее и дальше. Мое мнение — рефакторинг при изучении кода обеспечивает более высокий уровень моего понимания. Без помощи рефакторинга я не мог бы его достичь.

Рефакторинг помогает находить ошибки

Чем яснее я понимаю код, тем проще мне найти в нем ошибки. Следует сказать, что я не мастак искать ошибки. Если другие ухитряются, просмотрев большой фрагмент кода, тут же увидеть в нем ошибки, то для меня это недостижимая мечта. Но я заметил, что, когда я выполняю рефакторинг, мне приходится глубоко вникать в код и разбираться, что и как он делает. Это понимание возвращается в код. После структуры программы я проясняю для себя некоторые высказанные мною предположения и в результате просто не могу пропустить ошибки.

Это напоминает мне высказывание Кента Бека, которое он часто повторяет: "Я не слишком хороший программист. Но я программист с хорошими привычками". Рефакторинг является такой хорошей привычкой, которая помогает мне писать надежный код.

Рефакторинг ускоряет написание программ

В конечном итоге все сказанное сводится к одному: рефакторинг ускоряет написание программ.

Создается впечатление внутреннего противоречия. Когда я рассказываю о рефакторинге, становится очевидно, что он повышает качество кода. Улучшение проекта, повышение удобочитаемости, уменьшение количества ошибок — все это способствует качеству кода. Но разве скорость разработки не снижается из-за всего этого?

Я считаю, что для быстрой разработки программного обеспечения крайне важен хороший проект. Главная цель хорошего проекта именно в том, чтобы насколько возможно ускорить разработку. Без этого некоторое время можно работать быстро, но очень скоро плохой проект начинает тормозить работу. Время будет затрачиваться на поиск и исправление ошибок, а не на добавление функциональности. Внесение изменений занимает гораздо больше времени, если сначала нужно разобраться в программе и выявить дублирующийся код. Если код состоит их одних заплаток в несколько слоев, добавление новой функциональности требует существенно большего объема кодирования.

Хороший проект важен для поддержания высокой скорости разработки программного обеспечения. Рефакторинг помогает разрабатывать его быстрее, так как не только предохраняет проект от ухудшения, но и способствует его улучшению.

Когда нужно выполнять рефакторинг

Когда речь идет о рефакторинге, мне часто задают вопрос о том, как он должен планироваться. Не нужно ли выделять для рефакторинга пару недель после каждой пары месяцев разработки?

Я почти всегда против откладывания рефакторинга. Как мне кажется, рефакторинг — это не тот вид деятельности, который можно отложить. Им следует заниматься понемногу и постоянно. Надо не принимать решение о проведении рефакторинга, а проводить его, потому что вы должны делать свою работу, а помочь вам в этом может именно рефакторинг.

Правило трех

Дон Робертс однажды сказал мне следующее. Когда вы делаете что-то в первый раз, вы просто это делаете. Во второй раз вы морщитесь от повторения тех же действий, но все же делаете их. Наконец, делая это же в третий раз, вы начинаете рефакторинг.

Начинайте рефакторинг после трех повторов.

Выполняйте рефакторинг при добавлении функции

Чаще всего я выполняю рефакторинг, когда добавляю новую функцию в некоторую программу. Часто причиной рефакторинга является то, что он помогает

лучше понять код, который мне нужно модифицировать. Этот код мог написать как другой программист, так и я сам. Каждый раз, когда приходится разбираться в том, что делает некий код, я задаю себе вопрос о возможности изменить его структуру так, чтобы код стал более понятным. После этого я выполняю рефакторинг кода. Частично это делается на тот случай, если мне придется с ним работать в будущем, но главным образом потому, что при выполнении рефакторинга код для меня становится куда понятнее.

Еще одна причина выполнения рефакторинга — это наличие проекта, который не позволяет легко добавлять новые функции. Глядя на проект кода, я говорю сам себе: "Если бы этот код был спроектирован иначе, то добавить нужную функцию было бы очень просто". Я не начинаю терзаться о прежних промахах, а исправляю их с помощью рефакторинга. Частично я делаю это для облегчения дальнейших усовершенствований кода, но в основном потому, что я считаю это самым быстрым способом. При правильном подходе рефакторинг — быстрый и спокойный процесс. После его выполнения добавление новой функции происходит значительно проще и требует меньше времени.

Выполняйте рефакторинг во время исправления ошибок

При исправлении ошибок главная польза рефакторинга в том, что код становится более понятным. Когда я рассматриваю код и пытаюсь понять его, применение рефакторинга упрощает мою задачу. Часто выясняется, что такая активная работа с кодом помогает найти в нем ошибки. Можно взглянуть на это и с другой стороны: если мы получаем сообщение об ошибке, значит, нужно подумать о рефакторинге, потому что код не был достаточно понятным и мы не смогли увидеть в нем ошибку.

Выполняйте рефакторинг в ходе анализа кода

В ряде организаций регулярно проводятся собрания по анализу кода. В других этим не занимаются, — и совершенно напрасно. Анализ, или обзор, кода способствует распространению знаний среди всей команды разработчиков. Более опытные разработчики таким образом делятся своими знаниями с менее опытными. Такие встречи для анализа кода помогают большему количеству программистов понять большее число аспектов крупной программной системы. Они очень важны для написания понятного кода. Код, написанный мною, может казаться понятным мне, но не моей команде. Это нормально, ведь очень трудно поставить себя на место того, кто не знает, как вы работаете. Анализ позволяет также высказать полезные мысли большему количеству людей. У одного человека — особенно такого, как я — столько разных хороших идей не появится и за неделю. Вклад всех облегчает жизнь каждого, поэтому лично я всегда стараюсь посещать такие собрания.

Оказалось, рефакторинг помогает мне разбираться в коде, написанном другими программистами. До того, как я стал активно использовать рефакторинг, я мог прочесть код, в определенной мере понять его и внести свои предложения по его улучшению. Теперь же, когда у меня возникают идеи, я сразу решаю, нельзя ли их немедленно реализовать с помощью рефакторинга. Если можно, то я выполняю рефакторинг. Проделав так несколько раз, я могу более ясно увидеть, как будет выглядеть код после внесения в него предлагаемых изменений. Мне не надо напрягать воображение, чтобы представить будущий код, я сразу *вижу* его. В результате я прихожу к идеям следующего уровня, которые без рефакторинга не появились бы.

Кроме того, рефакторинг помогает получать более конкретные результаты от анализа кода. При его выполнении новые предложения не только возникают, но в основном тут же и реализуются. В результате мероприятие оказывается куда более успешным, чем было бы без рефакторинга.

Чтобы описанный мною способ работал, группы, анализирующие код, должны быть невелики. Мой опыт говорит о том, что лучше всего, когда над кодом совместно работают рецензент и автор кода. Рецензент предлагает свои изменения, и вместе с автором они решают, насколько легок соответствующий рефакторинг. Если изменение небольшое, они тут же его вносят.

Для обзора больших проектов зачастую лучше собрать несколько разных мнений в большей по размеру группе. При этом простой показ кода может оказаться не лучшим решением. Я предпочитаю использовать UML-диаграммы и проходить сценарии с помощью карт ответственности классов (Class Responsibility Collaboration cards). В итоге обзор проекта я провожу в группах, а анализ самого кода — с отдельными рецензентами.

Эта идея активного обзора кода доведена до своего логического предела в практике программирования парами (Pair Programming) в экстремальном программировании [3]. При использовании этой технологии все важные разработки выполняются двумя разработчиками на одной машине. В результате процесс разработки включает непрерывный анализ кода и его рефакторинг.

Почему рефакторинг работает

Кент Бек

Программы имеют два вида ценности — то, что они могут делать для нас сегодня, и то, что они смогут делать завтра. Чаще всего при программировании мы сосредоточиваемся на том, что требуется сегодня. Исправляя ошибку или добавляя новую возможность, мы повышаем ценность сегодняшней программы, расширяя ее возможности.

Нельзя программировать долго и не понять, что то, что система делает сегодня, — лишь одна сторона медали. Если выполнять сегодня ту работу, которая нужна

сейчас, так, что это может не позволить сделать завтрашнюю работу, вы проиграли. Конечно, вы знаете, что нужно сегодня, и далеко не так уверены в том, что потребуется завтра. Это может быть нечто, что вы просто не в состоянии сегодня себе представить.

Я знаю достаточно, чтобы сделать сегодняшние дела. Я слишком мало знаю, чтобы выполнить завтрашнюю работу. Но если я буду работать только для сегодняшнего дня, завтра я не смогу работать вообще.

Один из путей решения указанной проблемы — рефакторинг. Выяснив, что вчерашнее решение потеряло смысл, мы изменяем его. Теперь можно приступить к сегодняшней задаче. Завтра наши сегодняшние взгляды на задачу устареют, и мы вновь будем изменять ее.

В чем причина трудностей работы с программами? Я вижу четыре основные причины.

- Трудно модифицировать трудные для чтения программы.
- Трудно модифицировать программы, в логике которых имеется дублирование.
- Трудно модифицировать программы, которым нужны новые функции, требующие изменений в работающем коде.
- Трудно модифицировать программы со сложной условной логикой.

Нам нужны программы, которые легко читать, логика которых сосредоточена в одном-единственном месте, внесение изменений в которые не меняет существующее поведение функции и которые выражают условную логику как можно более просто.

Рефакторинг представляет собой процесс повышения ценности работающей программы не путем изменения ее поведения, а с помощью придания ей свойств, обеспечивающих продолжение ее разработки с высокой скоростью.

Что мне сказать руководству?

Мне очень часто задают вопрос “Как пояснить важность рефакторинга руководству?” Если руководитель технически грамотен, то обычно нет проблем ознакомить его с рефакторингом. Если он *действительно* заботится о качестве, то следует подчеркивать преимущества рефакторинга для повышения качества. В таком случае неплохо показать преимущества рефакторинга, использовав его в процессе обзора кода. Многие исследования показывают, что технический обзор кода играет важную роль для уменьшения количества ошибок и ускорения разработки программного обеспечения. Подтверждения этого тезиса можно найти во множестве книг, посвященных анализу, экспертизе или технологиям разработки программного обеспечения. Всего этого должно быть достаточно, чтобы убедить руководство в важности проведения обзора кода. А отсюда только один шаг до применения рефакторинга как части обзора кода.

Конечно, часто руководители только говорят, что качество для них — самое главное, но на деле их беспокоит только график работ. В этом случае мой совет несколько спорный: не говорите им вообще ничего!

Саботаж? Вовсе нет. Мы, разработчики программного обеспечения, — профессионалы, и наша работа заключается в наиболее быстром создании эффективного программного обеспечения. Мой опыт показывает, что рефакторинг существенно повышает скорость создания приложений. Когда нужно добавить новую функцию, а проект при этом не позволяет выполнить эту модификацию быстро и просто, то быстрее будет изменить структуру проекта, а уж после добавлять новую функциональность. Если нужно исправить ошибку, то первое, что нужно, — это понять, как работает программа, а быстрее всего этого можно добиться с помощью рефакторинга. Руководитель, подгоняемый графиком работ, хочет от меня повышения скорости; ну, а как именно я буду этого добиваться — не его дело. Самый быстрый путь — рефакторинг, а потому я буду им заниматься.

Косвенность и рефакторинг

Кент Бек

Кибернетика — это дисциплина, верящая в то, что любые проблемы можно решить введением одного или нескольких уровней косвенности.

— Деннис Де Брюлер (Dennis DeBruler)

С учетом приверженности разработчиков программного обеспечения косвенности не следует удивляться тому, что обычно рефакторинг приводит к дополнительной косвенности в программе. Как правило, рефакторинг разделяет большие объекты (а также большие методы) на несколько меньших.

Рефакторинг является обоюдоострым мечом. При каждом разделении чего-либо надвое количество объектов, с которыми приходится управляться, растет. Это может также затруднить чтение исходного текста, потому что один объект делегирует работу другому объекту, который делегирует ее третьему. Косвенность в связи с изложенным желательно минимизировать.

Но спешка нужна лишь при ловле блох. Косвенность может иметь собственные преимущества, проявляемые разными способами.

- **Может позволить совместно использовать логику**. Например, подметод, вызываемый из разных мест, или метод суперкласса, доступный всем подклассам.
- **Может отдельно пояснить намерения и реализацию**. Выбор имени каждого класса и каждого метода дает возможность объяснить ваши намерения. Внутренний код класса или метода объясняет, как эти намерения реализуются. Если внутренний код написан в том же духе, с помощью разделения намерений на еще меньшие части, то вы можете написать код, который связывает воедино большую часть важной информации о ваших структурах.

- **Помогает изолировать изменения кода**. Допустим, я использую объект в двух разных местах, и нужно изменить его поведение в одном из них. Если изменить объект, это может повлиять на оба его использования. Поэтому я сначала создаю подкласс и использую его там, где нужно внести изменения. Теперь можно модифицировать суперкласс без риска непреднамеренного изменения во втором случае.
- **Помогает кодировать условную логику**. В объектах есть такой механизм, как полиморфные сообщения, гибко, но ясно выражающие условную логику. Преобразуя явные условные операторы в сообщения, зачастую можно одновременно уменьшить дублирование, улучшить ясность и повысить гибкость программы.

Вся деятельность рефакторинга сводится к следующему: как, сохранив текущее поведение системы, повысить ее ценность — путем повышения качества или снижения стоимости.

Наиболее распространенный вариант действий — просмотреть программу и найти места, где не хватает преимуществ косвенности. Затем нужно внести в них необходимую косвенность, не меняя существующее поведение. В результате этих действий ценность программы возрастает, потому теперь у нее больше качеств, которые будут ценными завтра.

Сравните это с тщательным предварительным проектированием. Теоретическое проектирование представляет собой попытку обеспечить систему всеми ценными возможностями еще до написания кода. После этого код просто наращивается на основательно продуманном каркасе. Главная проблема в том, что можно очень легко ошибиться в своих предположениях о будущем программы. Рефакторинг же исключает опасность испортить все неудачным проектированием. Программа после рефакторинга ведет себя так же, как и до него. Кроме того, мы получаем возможность добавить в код ценные свойства.

Есть и второй, более редкий вариант. Нужно найти косвенность, которая не окупает себя, и устранить ее. Часто это проявляется в форме промежуточных методов, служивших для решения каких-то ныне не актуальных задач. Это также может быть задуманный как совместно используемый или полиморфный компонент, который оказался нужным в одном-единственном месте. При обнаружении такой "паразитной" косвенности ее нужно удалить. Ценность программы при этом повысится не за счет наличия в ней упомянутых выше свойств, а из-за того, что тот же результат получается с использованием меньшей косвенности.

Проблемы при выполнении рефакторинга

При изучении новой технологии, существенно повышающей производительность труда, бывает трудно увидеть ситуации, когда она неприменима. Обычно она изучается в конкретном контексте, часто просто в отдельном проекте. Трудно представить, что эта технология может стать малоэффективной, а то и попросту

вредной. Десять лет назад[1] та же ситуация была с объектами. Если бы тогда меня спросили, в каких случаях не стоит пользоваться объектами, я бы затруднился с ответом. Не то чтобы я не считал, что применение объектов имеет свои границы — я достаточно циничен для этого. Я просто не знал эти ограничения, хотя преимущества объектов были мне хорошо известны.

Теперь то же самое происходит и с рефакторингом. Мы знаем его преимущества. Мы знаем, что он может существенно изменить нашу работу. Но наш опыт еще недостаточно богат, чтобы увидеть присущие ему ограничения.

Этот раздел короче, чем мне бы хотелось, и в определенной мере тестовый. Чем больше программистов узнает о рефакторинге, тем больше мы о нем знаем. Для вас это должно означать, что, несмотря на мою уверенность в необходимости применения рефакторинга для получения предоставляемых им преимуществ, надо внимательно следить за его процессом. Замечайте трудности, возникающие при выполнении рефакторинга, и сообщите нам о них. Чем больше мы узнаем о рефакторинге, тем больше решений возникающих проблем мы найдем и сможем определить, какие из них не поддаются решению.

Базы данных

Одной из проблемных областей являются базы данных. Большинство бизнес-приложений тесно связаны с базами данных. Это одна из причин трудностей внесения изменений в базы данных. Другой причиной является миграция данных. Даже если ваша система тщательно разделена на слои для уменьшения зависимостей между схемой базы данных и объектной моделью, изменение схемы базы данных заставляет выполнить миграцию данных, которая может оказаться длительной и рискованной операцией.

В случае необъектных баз данных с этим можно справиться с помощью отдельного программного слоя между объектной моделью и моделью базы данных. Так можно изолировать одни от других изменения двух разных моделей. При модификации одной модели не обязательно делать то же с другой — достаточно модифицировать промежуточный слой. Этот слой добавляет сложность в систему, но зато резко повышает гибкость. Даже без рефакторинга это может быть очень важным, когда имеется несколько баз данных или при сложной модели базы данных, которой вы не можете управлять.

Не обязательно начинать с отдельного слоя. Можно создать этот слой, если вы заметите, что части объектной модели становятся изменчивыми. Так достигаются наилучшие возможности для внесения изменений.

---

[1] Напомним, что книга издана в 1999 году. — *Примеч. пер.*

Объектные базы данных одновременно и облегчают задачу, и усложняют ее. Некоторые объектно-ориентированные базы данных обеспечивают автоматическую миграцию от одной версии объекта к другой. Это снижает необходимые усилия, но приводит к потере времени на выполнение миграции. Если миграция не автоматизирована, ее приходится осуществлять самостоятельно, что требует больших усилий. В этой ситуации следует больше заботиться об изменении структур данных классов. Можно свободно перемещать поведение, но нужно быть более осторожным при перемещении полей. Нужно применять методы доступа к данным, чтобы получать иллюзию их перемещения, в то время как в действительности оно не происходило. При достаточной уверенности в том, что вы знаете, где должны находиться данные, их можно переместить с помощью одной операции. Модифицировать следует только методы доступа, что снижает риск появления ошибок.

Изменение интерфейсов

Одним из важных вопросов применения объектов является то, что они позволяют изменять реализацию программного модуля отдельно от изменений интерфейса. Можно безопасно изменить внутреннее устройство объекта, не потревожив при этом ничего в программе, но интерфейс — дело другое, если его изменить, может случиться всякое.

Иногда вызывает беспокойство то, что многие рефакторинги действительно изменяют интерфейс. Такой простой рефакторинг, как "Удаление параметра" (с. 294), целиком посвящен изменению интерфейса. Как же это соотносится с самой идеей инкапсуляции?

Если доступен код, вызывающий метод, поменять его имя нетрудно. Даже если это открытый метод, но есть доступ ко всем местам, где он вызывается, метод можно переименовывать. Проблема возникает только тогда, когда интерфейс используется кодом, который нельзя найти и изменить. Я называю такой интерфейс *опубликованным интерфейсом* (что еще на шаг дальше, чем открытый интерфейс). Если интерфейс опубликован, небезопасно его изменять и редактировать места вызова. Процесс становится более сложным.

Это понятие несколько меняет постановку вопроса. Задача теперь формулируется иначе: как быть с рефакторингами, изменяющими опубликованные интерфейсы?

Коротко говоря, при изменении в рефакторинге опубликованного интерфейса необходимо сохранять как старый интерфейс, так и новый, хотя бы до того времени, когда пользователи отреагируют на это изменение. К счастью, это не так уж сложно. Обычно можно выполнить работу так, чтобы старый интерфейс оставался работающим. Попробуйте сделать так, чтобы старый интерфейс вызывал новый, и, изменив название метода, сохраните его. Не пытайтесь копировать

тело метода — это дорога в никуда. Работая на языке программирования Java, следует также воспользоваться возможностью пометки кода как устаревшего (необходимо добавить к методу, который не следует использовать, модификатор `deprecated`). Благодаря этому пользователи кода будут знать, что ситуация изменилась.

Хорошим примером этого процесса являются классы коллекций Java. Новые классы в Java 2 заменяют собой те, которые были предоставлены изначально. Когда были выпущены классы Java 2, JavaSoft приложила массу усилий для обеспечения пути миграции к ним.

Защита интерфейсов обычно осуществима, но неприятна. Требуется создать и (по крайней мере временно) сопровождать дополнительные методы. Данные методы усложняют интерфейс, усложняя его применение. Имеется и альтернативное решение: не публикуйте интерфейс. Речь не идет о полном запрете — очевидно, что опубликованные интерфейсы должны иметься. Если вы создаете API для внешнего использования, как это делает Sun, то без опубликованных интерфейсов не обойтись. Я говорю об этом потому, что часто сталкиваюсь со злоупотреблениями публикацией интерфейсов группами разработчиков. Я знал команду из трех программистов, каждый из которых публиковал интерфейсы для двух других. Это вело к лишней работе по поддержке этих интерфейсов, в то время как было бы куда проще выполнить необходимое редактирование в базе кода. Этот стиль работы присущ организациям, бережно и обстоятельно относящимся к вопросам собственности на код. Публикация интерфейсов полезна, но имеет свою цену, так что не прибегайте к ней без необходимости. Это может означать изменение правил в отношении владения кодом, позволяя программистам менять чужой код для поддержания изменений в интерфейсе. Часто неплохой идеей является поступать так при программировании в паре.

Не спешите с публикацией интерфейсов. Измените стратегию в отношении вопросов собственности на код так, чтобы она не мешала рефакторингу.

Имеется одна особая область проблем, связанных с изменениями интерфейсов в Java, — добавление исключений в конструкции `throw` объявлений методов. Это не изменение сигнатуры, поэтому делегирование проблему не решает. Однако компилятор не будет компилировать такой измененный код. Решать данную проблему достаточно тяжело. Можно выбрать для нового метода новое имя, вызывать его из старого метода и преобразовать проверяемое исключение в непроверяемое. Можно также генерировать непроверяемое исключение, хотя при этом вы теряете возможность проверки. Если вы выбираете этот способ, то можете предупредить всех, кто вызывает этот метод, что исключение в будущем станет

проверяемым. Это даст возможность программистам поместить обработчики в свой код. По этой причине я предпочитаю определять суперкласс исключения для всего пакета (например, `SQLException` для `java.sql`) и гарантировать, что в конструкции `throw` опубликованных методов объявляется только это исключение. Тогда при желании можно создавать подклассы исключений, но это не повлияет на вызывающий код, который осведомлен только об общем случае.

Изменения проекта, затрудняющие рефакторинг

Может ли рефакторинг исправить любые недостатки проектирования, или некоторые проектные решения столь фундаментальны, что их невозможно изменить с помощью более позднего рефакторинга? Имеющаяся на этот счет информация не может считаться достаточной. Конечно, рефакторинг часто оказывается эффективным, но иногда при его выполнении возникают трудности. Мне известен проект, в котором с помощью рефакторинга с большим трудом удалось привести архитектуру системы, созданной без учета требований безопасности, к хорошо защищенной архитектуре.

На этой стадии мой подход состоит в том, чтобы представить себе рефакторинг. Рассматривая различные альтернативы проекта, я спрашиваю себя, насколько трудным окажется рефакторинг одного проекта в другой. Если он кажется простым, то я не слишком волнуюсь о выборе и останавливаюсь на самом простом проекте, даже если он не отвечает всем потенциальным требованиям, которые могут возникнуть в дальнейшем. Однако если я не могу найти простых способов рефакторинга, то продолжаю работать над проектом. Мой опыт показывает, что это случается достаточно редко.

Когда не следует прибегать к рефакторингу

В некоторых случаях рефакторинг не нужен вовсе. Главный пример — когда вместо этого решено переписать программу "с нуля". Иногда имеющийся код настолько запутан, что его проще переписать заново, чем подвергнуть рефакторингу. Такое решение принять нелегко, и я признаюсь, что не могу дать какие-либо надежные рекомендации по этому поводу.

Явный признак необходимости переписывания кода заново — его неработоспособность. Она обнаруживается только при тестировании, если ошибок настолько много, что не удается получить стабильный код. Помните, что перед началом рефакторинга код должен работать (в основном) правильно.

Компромисс заключается в разделении путем рефакторинга больших частей кода на компоненты с сильной инкапсуляцией. Затем для каждого из компонентов по отдельности принимается решение об изменении его структуры с помощью рефакторинга или переписывания заново. Это многообещающий подход, но

имеющихся у меня данных недостаточно, чтобы четко сформулировать правила для его принятия. Такой подход становится особенно привлекательным при работе со старой унаследованной системой.

Еще одна ситуация, когда следует воздержаться от рефакторинга, — близость даты завершения проекта. Рост производительности за счет применения рефакторинга проявится слишком поздно, после срока завершения. Уорд Каннингем (Ward Cunningham) по этому поводу сравнил незавершенный рефакторинг с долгами. Большинству компаний для нормальной работы нужны кредиты. Но вместе с долгами появляются и проценты — дополнительная стоимость обслуживания и расширения, вызванная излишней сложностью кода. Выплату определенных процентов можно пережить, но если платить нужно слишком много, вы разоритесь. Важно управлять своими долгами, выплачивая их часть посредством рефакторинга.

Приближение срока окончания работ представляет собой единственный случай, когда можно отложить рефакторинг из-за недостатка времени. Опыт работы над несколькими проектами показывает, что выполнение рефакторинга ведет к росту производительности труда. Нехватка времени обычно сигнализирует как раз о необходимости рефакторинга.

Рефакторинг и проектирование

Рефакторинг играет особую роль как дополнение к проектированию. Когда я начинал изучать программирование, я просто писал программу и кое-как доводил ее до рабочего состояния. Постепенно я понял, что если заранее подумать об архитектуре программы, то позже можно избежать дорогостоящей переработки. Я все больше привыкал к этому стилю *предварительного проектирования*. Многие программисты полагают, что важнее всего проектирование, а кодирование представляет собой чисто механический процесс. Аналогией проекта служит технический чертеж, а аналогией кодирования — изготовление по нему детали. Но программа — не кусок железа. Она куда более податлива и требует усиленного обдумывания. Как говорит Алистер Кокберн (Alistair Cockburn), "при наличии проекта я могу думать очень быстро, но при этом с большим количеством мелких пробелов".

Бытует мнение, что рефакторинг может быть альтернативой предварительному проектированию. При таком сценарии проектирование попросту отсутствует. Первое же пришедшее в голову решение воплощается в коде, доводится до рабочего состояния, а потом обретает требуемый вид с помощью рефакторинга. Такой подход и в самом деле может работать. Мне встречались люди, которые работают именно таким образом и получают в итоге систему с неплохой архитектурой.

Пропагандистами такого подхода часто считают тех, кто поддерживает “экстремальное программирование” [3].

Хотя указанный подход и работает, он не является самым эффективным. Даже программисты-“экстремалы” сначала продумывают некоторую архитектуру будущей системы. Они испытывают разные идеи с помощью CRC-карт или чего-то похожего, пока не получат подходящего первого приближения. После этого они приступают к кодированию, а затем и к рефакторингу. Главное, что рефакторинг меняет роль предварительного проектирования. Если на рефакторинг не рассчитывать, то необходимо как можно лучше выполнить предварительное проектирование. Возникает чувство, что любые будущие изменения проекта будут слишком дорогостоящими, а потому в предварительное проектирование вкладывается больше времени и усилий — чтобы избежать таких изменений впоследствии.

При использовании рефакторинга акценты меняются. Предварительное проектирование сохраняется, но теперь оно не ставит целью найти *единственно верное* решение. Все, что от него требуется, — это найти приемлемое решение. По мере построения решения понимание задачи углубляется, и вы видите, что принятое первоначально решение отнюдь не наилучшее. В этом нет ничего страшного, если в процессе участвует рефакторинг, потому что изменения ст*о*ят не слишком дорого.

Важным результатом такого изменения акцентов является большее стремление к простоте проекта. До применения рефакторинга в своей работе я всегда искал гибкие решения. Для каждого требования к программе я искал возможности его изменения в течение срока жизни системы. Поскольку изменения в проекте дорогостоящи, я старался создать проект, способный выдержать любые изменения, которые мог предвидеть. Недостаток гибкости решения в том, что за нее приходится платить. Гибкие решения сложнее обычных. Создаваемые по таким проектам программы в общем случае труднее сопровождать, хотя и легче перенастраивать. Но даже такие решения не избавляют от необходимости модифицировать проект. Для пары функций сделать это нетрудно, но изменения происходят во всей системе. Если обеспечивать гибкость во всех возможных местах, то система становится значительно сложнее и дороже в сопровождении. Весьма грустно то, что по большому счету вся эта гибкость оказывается ненужной. Требуется лишь некоторая ее часть, но сказать заранее, какая, невозможно. Достижение гибкости требует ее гораздо больше, чем в действительности необходимо.

Рефакторинг представляет собой другой подход к рискам изменений. Возможные модификации все равно нужно пытаться предвидеть, а гибкие решения продолжать рассматривать как возможные варианты. Но вместо реализации этих решений следует задуматься, насколько сложным будет преобразование обычного решения в гибкое с помощью рефакторинга. Если, как чаще всего и бывает, ответом будет “совсем не сложно”, то лучше реализовать обычное решение.

Рефакторинг ведет к упрощению проектирования без принесения гибкости в жертву. Это делает процесс проектирования более легким и менее напряженным. Научившись чувствовать, что легко поддастся рефакторингу, а что нет, о гибкости решений перестаешь задумываться. Появляется уверенность в возможности рефакторинга, когда он понадобится. В результате строятся самые простые решения, которые могут работать; гибкие и сложные решения по большей части остаются невостребованными.

Нужно много времени, чтобы ничего не сделать

— Рон Джеффрис (Ron Jeffries)

Программная система Chrysler Comprehensive Compensation работала слишком медленно. И пусть разработка еще не была завершена, это стало серьезно беспокоить разработчиков, потому что замедлялось выполнение тестов.

Кент Бек, Мартин Фаулер и я решили исправить такое положение дел. В ожидании встречи с коллегами я, будучи хорошо знакомым с системой, пытался понять, что же может так ее тормозить. Я подумал о нескольких причинах и поговорил с парнями о том, какие изменения могли бы потребоваться. Во время обсуждения нам пришло в голову несколько неплохих мыслей о том, как можно ускорить систему.

После этого мы измерили производительность с помощью профайлера. Увы, ни одна из предполагаемых мною причин не имела отношения к проблеме. Выяснилось, что система тратила половину рабочего времени на создание экземпляров класса дат. Самое интересное, что все эти объекты содержали одни и те же данные.

Изучив логику создания дат, мы придумали возможность оптимизировать этот процесс. При создании везде использовалось преобразование строк, даже если никакие внешние данные не вводились. Это было сделано просто для удобства ввода исходного текста, и это вполне можно было оптимизировать.

Затем мы рассмотрели применение этих дат. Выяснилось, что по большей части они участвовали в создании диапазонов дат, т.е. объектов, содержащих дату "от" и дату "до". Поразбиравшись еще немного, мы обнаружили, что большинство этих диапазонов пусты!

При работе с диапазонами дат было условлено, что любой диапазон, конечная дата которого предшествует начальной, является пустым. Это удобное соглашение, хорошо согласующееся с тем, как работает класс. При использовании данного соглашения вскоре выяснилось, что код, создающий диапазоны дат, начинающиеся после своего окончания, запутан и непонятен, так что мы вынесли эту функциональность в фабричный метод, создающий пустые диапазоны дат.

Это изменение должно было просто сделать код более ясным, но мы получили неожиданный бонус. Мы создали пустой диапазон в виде константы, которую фабричный метод возвращал вместо создания объекта каждый раз заново.

После этой модификации скорость системы удвоилась, чего было вполне достаточно для выполнения тестов. Эта работа отняла у нас примерно пять минут.

Мы обсуждали вопрос, что может быть плохого в хорошо известном коде, со многими участниками проекта. Мы даже прикинули некоторые возможные усовершенствования кода, не проводя предварительных измерений.

Оказалось, что мы полностью ошибались. Если не считать пары рассказанных во время обсуждения анекдотов, толку от него не было никакого.

Вывод из этого рассказа следующий. Даже если вы точно знаете, как работает система, занимайтесь не гаданием, а замерами. Полученная информация в девяти случаях из десяти покажет, что ваши предположения были ошибочными!

Рефакторинг и производительность

Распространенный вопрос, связанный с рефакторингом, — его влияние на производительность программы. Для облегчения понимания работы программы часто осуществляется модификация, приводящая в конечном итоге к замедлению программы. Это важный вопрос. Я не считаю возможным пренебрегать производительностью программного обеспечения, предпочитая чистоту проекта или надеясь на увеличение вычислительной мощности техники. Рефакторинг может приводить к замедлению программы, но при этом он делает более легкой настройку ее производительности. Секрет создания быстрых программ (если только они не предназначены для работы в режиме реального времени) состоит в том, чтобы начать с написания программы, производительность которой можно настраивать, а затем с помощью настройки достичь приемлемой скорости ее работы.

Я знаком с тремя подходами к написанию быстрых программ. Наиболее трудный путь зачастую применяется в системах с жесткими требованиями к выполнению в режиме реального времени. В этом случае при разложении проекта на составные части каждому компоненту выделяются определенные ресурсы времени и памяти. Компонент не должен выйти за рамки выделенного ресурса, хотя возможно применение механизма обмена ресурсами. При этом подходе жестко соблюдается выполнение программы в заданных временн*ы*х рамках. Это очень важно в таких системах, как, например, кардиостимуляторы, в которых опоздание может стоить человеческой жизни. В других системах, например в корпоративных информационных системах, с которыми я обычно имею дело, такой подход является избыточным.

Второй подход предполагает постоянное внимание к проекту. Каждый программист в любой момент времени делает все от него зависящее, чтобы

обеспечивать высокую производительность программы. Этот подход широко распространен и кажется привлекательным, но на деле не так уж хорош. Вносимые изменения, повышающие производительность, обычно затрудняют работу с программой, что приводит к замедлению написания программы. Это могло бы быть разумной ценой, если бы в результате получалось более быстрое программное обеспечение, но, как правило, этого не происходит. Повышающие производительность изменения разбросаны по всей программе, и каждое из них относится только к некоторой узкой функциональности программы.

Интересный факт, связанный с вопросами производительности, — в ходе анализа большинства программ выясняется, что бо́льшую часть времени работы программы выполняется небольшой фрагмент кода. Если оптимизировать весь код в одинаковой степени, то 90% оптимизации окажется выполненной впустую, так как это была оптимизация редко выполняемого кода. Время, затраченное на ускорение программы, и время, потерянное из-за ее невыразительности, потеряно при этом напрасно.

Третий подход к повышению производительности программы основан на упомянутой статистике. Он предполагает создание программы с высокой степенью декомпозиции на компоненты без учета получаемой производительности вплоть до этапа оптимизации последней (который обычно наступает на достаточно поздней стадии разработки и при выполнении которого осуществляется отдельная процедура настройки производительности программы).

Все начинается с запуска профайлера, который исследует выполнение программы и сообщает, где в основном расходуются время и память. Это позволяет выявить небольшой фрагмент кода, который представляет собой узкое место в смысле производительности. Усилия программистов сосредоточиваются в этих местах, и над ними осуществляется та же оптимизация, которая применяется при подходе с постоянным вниманием. Благодаря тому, что при этом усилия сосредоточены на узких местах, удается достичь бо́льших результатов при меньших усилиях. Но даже в такой ситуации необходима бдительность. Как и при рефакторинге, изменения следует вносить небольшими порциями, каждый раз компилируя, тестируя и выполняя профилирования. Если производительность не увеличивается, выполняется откат. Поиск и избавление от узких мест продолжается до достижения производительности, удовлетворяющей пользователей. Мак-Коннелл (McConnell) подробно рассказывает об этом подходе в [12].

Разделение программы на компоненты способствует этому стилю оптимизации двумя путями. Во-первых, благодаря разделению появляется время, которое можно затратить на оптимизацию. В хорошо структурированный код быстрее добавляется новая функциональность и выигрывается время для работы над производительностью; профилирование же гарантирует, что это время не будет

потрачено впустую. Во-вторых, хорошо структурированная программа обеспечивает возможности более качественного анализа производительности. Профайлер указывает на узкие места в более мелких фрагментах кода, которые легче настроить. Благодаря более понятному коду легче выбрать возможные варианты настройки и понять, какая именно настройка может оказаться эффективной.

Я вижу, что рефакторинг позволяет мне быстрее писать программы. На какое-то время программы становятся более медленными, но зато их легче настраивать на этапе оптимизации. В результате в конечном счете мы получаем больший выигрыш, чем без рефакторинга.

Истоки рефакторинга

Я не смог точно выяснить, когда именно появился термин *рефакторинг*. Хорошие программисты всегда посвящают хотя бы некоторое время наведению порядка в своем коде. Они занимаются этим, так как понимают, что проще модифицировать аккуратный код, а не сложный и запутанный, и хорошо известно, что написать такой хороший код сразу удается очень редко.

Рефакторинг идет дальше. В данной книге рефакторинг рассматривается как ведущий элемент процесса разработки программного обеспечения в целом. Первыми, кто понял важность рефакторинга, были Уорд Каннигем (Ward Cunningham) и Кент Бек, с 1980-х годов работавшие со Smalltalk. Smalltalk уже тогда был открытой для рефакторинга средой. Она очень динамична и позволяет быстро писать высокофункциональные программы. Smalltalk имеет короткий цикл “компиляция-компоновка-выполнение”, обеспечивающий возможность быстро модифицировать программы. Этот язык программирования является объектно-ориентированным и предоставляет развитые средства для уменьшения влияния изменений, скрываемых за точно определенными интерфейсами. Уорд и Кент разработали методику создания программного обеспечения, приспособленную для применения в такой среде. (Сейчас Кент называет этот стиль *экстремальным программированием* (Extreme Programming) [3].) Они осознали значение рефакторинга для повышения производительности труда программистов и с тех пор применяют его в сложных программных проектах и совершенствуют данную технологию.

Идеи Уорда и Кента всегда оказывали сильное влияние на сообщество программистов на Smalltalk, так что понятие рефакторинга стало важным элементом культуры Smalltalk. Еще одна крупная фигура в сообществе программистов Smalltalk — Ральф Джонсон (Ralph Johnson), профессор Университета штата Иллинойс в Урбана-Шампань, известный как член “банды четырех” [9]. Среди областей знания, в которых сосредоточены интересы Ральфа, находится создание

программных каркасов (frameworks). Им также исследован вопрос о применении рефакторинга для создания эффективной и гибкой среды разработки.

Билл Опдайк был одним из аспирантов Ральфа и проявлял особый интерес к этим каркасам. Он обратил внимание на потенциальную ценность рефакторинга и заметил, что его применение не ограничивается рамками Smalltalk. Билл имел опыт разработки программного обеспечения для телефонных станций, характеризующегося нарастанием сложности со временем и трудностью модификации. Диссертация Билла рассматривала рефакторинг с точки зрения разработчика инструментальных средств. Билл изучил разновидности рефакторинга, которые могли оказаться полезными при разработке каркасов C++, и исследовал способы рефакторинга, сохраняющие семантику, а также способы доказательства сохранения семантики и возможности реализации своих идей в инструментальных средствах. Диссертация Билла [14] является на сегодняшний день одной из самых влиятельных работ на тему рефакторинга. Билл также написал главу 13, "Рефакторинг, повторное использование и реальность", данной книги.

Я встречался с Биллом на конференции OOPSLA в 1992 году. За чашкой кофе мы обсуждали некоторые стороны моей работы по созданию концептуальных каркасов в здравоохранении. Билл рассказал мне о своих исследованиях, и, помню, я тогда подумал, что это интересно, но не имеет практического применения. Как же я ошибался!

Джон Брант и Дон Робертс значительно продвинули идеи инструментальных средств рефакторинга, создав инструмент для выполнения рефакторинга в Smalltalk — Refactoring Browser. В этой книге их перу принадлежит глава 14, "Инструментарий для выполнения рефакторинга", посвященная инструментам рефакторинга.

Ну, а что же я? Мне всегда нравился хороший код, но я никогда не думал, что это так важно. Затем я участвовал в одном проекте с Кентом и увидел, как он применяет рефакторинг. Я понял, какое огромное влияние он оказывает на производительность труда и качество результатов. Этот опыт убедил меня в том, что рефакторинг — очень важная технология разработки программного обеспечения. Но я был разочарован отсутствием книг, которые мог бы прочесть практикующий программист, а никто из упомянутых выше экспертов по этому вопросу и не собирался писать такую книгу! Мне пришлось брать этот труд на себя — с их помощью.

Оптимизация зарплатной системы

— Рич Гарзанити

Мы уже длительное время разрабатывали платежную систему Chrysler Comprehensive Compensation System к моменту, когда ее стали переносить на GemStone. Естественно, мы обнаружили, что программа работает недостаточно быстро. Мы пригласили специалиста Джима Хонгса (Jim Haungs) помочь нам в оптимизации нашей системы.

Проведя некоторое время с командой и разобравшись, как работает наша система, Джим воспользовался ProfMonitor производства GemStone для написания профилирующего инструментария, который и добавил к нашим функциональным тестам. Этот инструмент показывал количество создаваемых объектов и место, где они создавались.

К нашему удивлению, самым узким местом оказалось создание строк. Рекорд поставило многократное создание строк из 12000 символов. При этом возникали проблемы особого рода, связанные с тем, что обычные средства сборки мусора GemStone с такими большими строками не справлялись. Из-за большого размера строк GemStone выгружал строки на диск всякий раз при их создании. Как выяснилось, эти строки создавала наша система ввода-вывода, причем сразу по три такие строки для каждой выводимой записи!

Для начала мы стали кешировать 12000-символьные строки, что в основном решило проблему. После этого мы изменили среду так, чтобы запись велась непосредственно в файловый поток, что устранило необходимость создания даже одной строки. После того как длинные строки перестали нам мешать, профайлер Джима нашел аналогичные проблемы с маленькими строками — по 800, 500 символов и т.п. Использование файловых потоков решило и эти проблемы.

Так мы постепенно, но уверенно повышали производительность системы. В начале разработки казалось, что для расчета выплат потребуется более 1000 часов. Когда же работа над системой завершилась, требовалось только 40 часов. Еще через месяц продолжительность расчетов была снижена до 18 часов, а после ввода системы в эксплуатацию — до 12. После года эксплуатации и усовершенствования системы это время было сокращено до 9 часов.

Самое большое усовершенствование было связано с работой программы на нескольких потоках в многопроцессорной машине. При начальной разработке системы многопоточность не предполагалась, но благодаря хорошей структурированности исходного текста мы смогли модифицировать систему для многопоточной среды за три дня. В настоящее время расчет выплат выполняется за пару часов.

До того как Джим обеспечил нас инструментарием профилирования работы системы, у нас было множество разных предположений о причинах неудовлетворительной работы системы. Однако реальные замеры указали другое направление работы и обеспечили более высокие результаты.

Глава 3

Запах в коде

Кент Бек и Мартин Фаулер

Если что-то стало пованивать, его лучше сменить.
— Бабушка Бек при обсуждении проблем подгузников

Сейчас вы уже должны хорошо представлять себе, как действует рефакторинг. Но если вы знаете "как", это еще не значит, что вы знаете "когда". Решение о том, когда приступать к рефакторингу и когда прекратить его выполнение, не менее важно, чем умение выполнять рефакторинг.

И здесь мы сталкиваемся с дилеммой. Легко объяснить, как удалить переменную экземпляра или создать иерархию, — это достаточно простые вопросы в отличие от объяснения, когда это следует делать. Вместо того чтобы взывать к расплывчатым представлениям об эстетике программирования (честно говоря, мы, консультанты, часто поступаем именно так), я попытался подвести под это более прочную основу.

Я как раз размышлял над этим сложным вопросом, когда посетил Бека в Цюрихе. Возможно, он тогда находился под впечатлением запахов от своей новорожденной дочки, потому что выразил представление о том, когда проводить рефакторинг, именно с использованием этой аналогии. Вы можете спросить "Чем, собственно, запах лучше туманных рассуждений об эстетике?" Но мне кажется, что использование понятия запаха лучше рассуждений об эстетичности. Мы просмотрели очень большое количество кода, написанного для разных проектов, степень успешности которых простиралась в очень широком диапазоне — от весьма удачных до полудохлых. При этом мы научились искать в коде определенные признаки, которые предполагают возможность рефакторинга (а иногда просто кричат о его необходимости). (Слово "мы" в этой главе отражает тот факт, что у главы два автора — я и Кент. Определить, кто и что писал, просто: авторство смешных шуток принадлежит мне, а всего остального — ему.)

Мы не будем даже пытаться дать точные критерии необходимости рефакторинга. Наш опыт показывает, что никакие системы показателей не смогут соперничать с человеческой интуицией, основанной на знаниях. Мы приведем лишь симптомы неприятностей, устраняемых путем рефакторинга. У вас должно развиться собственное чувство того, какое количество следует считать "чрезмерным" для атрибутов класса или строк кода для метода.

Эта глава и таблица в конце книги должны подсказать, когда вам неясно, какие именно методы рефакторинга применять. Прочтите эту главу (или просмотрите таблицу) и попробуйте выяснить, какой именно запах вы чувствуете. Затем обратитесь к предлагаемым методам рефакторинга и проверьте, подойдут ли они вам. Вполне возможно, что запах не совпадет тютелька в тютельку, но общее направление вполне может быть выбрано правильно.

Дублируемый код

Дурнопахнущий парад открывает дублируемый код. Увидев одинаковые структуры кода в нескольких местах, можно быть уверенным, что если их удастся объединить, то программа от этого только выиграет.

Простейшая задача с дублированием кода возникает, когда одно и то же выражение присутствует в двух методах одного и того же класса. В этом случае достаточно применить рефакторинг “Извлечение метода” (с. 132) и вызывать код вновь созданного метода из обеих точек.

Другая распространенная проблема заключается в наличии одного и того же кода в двух подклассах, находящихся на одном уровне иерархии. Устранить это дублирование кода можно с помощью рефакторинга “Извлечение метода” (с. 132) для обоих классов с последующим рефакторингом “Подъем метода” (с. 339). Если код похож, но полностью не совпадает, можно применить рефакторинг “Извлечение метода” (с. 132) для отделения совпадающих фрагментов от отличающихся. После этого может оказаться возможным применить рефакторинг “Формирование шаблонного метода” (с. 361). Если оба метода делают одно и то же с помощью разных алгоритмов, можно выбрать из этих алгоритмов более эффективный и применить рефакторинг “Замена алгоритма” (с. 159).

Если дублируемый код находится в двух разных классах, попробуйте применить рефакторинг “Извлечение класса” (с. 169) в одном классе, а затем использовать новый компонент в другом. Еще одной возможностью является ситуация, когда метод должен принадлежать только одному из классов и вызываться из другого класса, либо принадлежать третьему классу, к которому будут обращаться оба первоначальных класса. Вам нужно решить, где должен находиться этот метод, и гарантировать, что он находится только там и нигде более.

Длинный метод

Программы, использующие объекты, живут долго и счастливо, если методы этих объектов короткие. Программистам с малым опытом работы с объектами

часто кажется, что на самом деле никаких вычислений не происходит, а программы состоят только из нескончаемой цепочки делегирований действий от одного объекта к другому. Однако, работая с такой программой на протяжении нескольких лет, вы понимаете, какую ценность представляют собой маленькие методы. Все выгоды, которые дает косвенность — понятность, совместное использование и выбор, — поддерживаются именно маленькими методами (см. врезку “Косвенность и рефакторинг” главы 2, “Принципы рефакторинга”).

Еще на заре программирования было ясно, что чем длиннее процедура, тем труднее понять, как она работает. В старых языках программирования вызов процедур был связан с большими накладными расходами, которые удерживали программистов от применения маленьких методов. Современные объектно-ориентированные языки в значительной мере устранили накладные расходы вызовов. Но издержки сохраняются для того, кто читает код, так как ему приходится переключаться между контекстами, чтобы понять, что делает та или иная процедура. Среда разработки, позволяющая видеть одновременно два метода, помогает облегчить это переключение; но главное, что способствует пониманию маленьких методов, — это присвоение им разумных имен. Правильно выбранное имя метода зачастую позволяет не изучать его тело.

В итоге мы приходим к необходимости активнее применять декомпозицию методов. Эвристическое правило, которому мы следуем, гласит, что если возникает необходимость что-то прокомментировать, то пора писать метод. В этом методе содержится код, который требовал комментариев, но его название отражает его назначение, а не метод решения им задачи. Такая процедура может применяться к группе строк или даже к единственной строке кода. К ней можно прибегнуть даже тогда, когда вызов метода оказывается длиннее, чем замененный им код, — при условии, что имя метода разъясняет его предназначение. Главным является не длина метода, а семантическое расстояние между тем, что метод делает, и тем, как он это делает.

В 99% случаев, чтобы укоротить метод, требуется использовать рефакторинг “Извлечение метода” (с. 132). Найдите части метода, которые кажутся связанными между собой, и образуйте новый метод.

Если у вас метод со множеством параметров и временных переменных, это мешает выделению нового метода. Применяя рефакторинг “Извлечение метода”, приходится передавать в качестве параметров такое количество параметров и временных переменных, что результат оказывается ничуть не проще, чем оригинал. Устранить временные переменные можно с помощью рефакторинга “Замена временной переменной запросом” (с. 141), а длинные списки параметров можно сократить с помощью рефакторингов “Введение объекта параметра”(с. 312) и “Сохранение всего объекта” (с. 305).

Если после этого все равно остается слишком много временных переменных и параметров, приходится вызывать тяжелую артиллерию, а именно — рефакторинг "Замена метода объектом методов" (с. 155).

Как же выявить фрагменты кода, которые должны быть выделены в отдельные методы? Хороший способ — поискать комментарии: они часто указывают на такое семантическое расстояние. Блок кода с комментариями говорит о том, что его можно заменить методом, имя которого основано на этом комментарии. Даже одна строка может быть выделена в метод, если она нуждается в пояснениях.

Условные инструкции и циклы также служат признаками возможного выделения в метод. Для работы с условными выражениями подходит рефакторинг "Декомпозиция условного оператора" (с. 256). В случае цикла выделите его и содержащийся в нем код в отдельный метод.

Большой класс

Когда класс пытается взять на себя слишком многое, это часто проявляется в чрезмерном количестве членов-данных. А отсюда недалеко и до дублирования кода.

Рефакторинг "Извлечение класса" (с. 169) позволяет связать некоторое количество членов-данных. Выбирайте члены, которые должны оказаться вместе в компоненте, так, чтобы это имело смысл для каждого из них. Например, `depositAmount` (сумма вклада) и `depositCurrency` (валюта вклада) вполне могут принадлежать одному компоненту. Обычно на мысль о создании компонента наводят одинаковые префиксы или суффиксы у некоторого подмножества членов класса. Если имеет смысл создание компонента как подкласса, можно воспользоваться рефакторингом "Извлечение подкласса" (с. 347).

Иногда класс не использует все свои переменные экземпляра все время. В таком случае оказывается возможным неоднократно применить рефакторинги "Извлечение класса" (с. 169) и "Извлечение подкласса" (с. 347).

Как и в случае класса с большим количеством членов-данных, класс со слишком большим количеством кода создает предпосылки для дублирования кода, хаоса и гибели. Простейшее решение (мы уже не раз говорили, что предпочитаем простейшие решения) — это устранение избыточности в самом классе. Если у вас есть пять методов по сотне строк и с большим количеством дублируемого кода, возможно, их можно заменить пятью методами по десять и еще десятком двухстрочных методов, выделенных из исходных.

Как и в случае класса с большим количеством членов-данных, обычное решение для класса со слишком большим количеством кода состоит в том, чтобы применить рефакторинг "Извлечение класса" (с. 169) или "Извлечение подкласса"

(с. 347). Полезным приемом является определение того, как клиенты используют класс, и применение рефакторинга "Извлечение интерфейса" (с. 357) для каждого из этих применений. В результате может выясниться, как разделить класс еще сильнее.

Если большой класс является классом GUI, может потребоваться переместить его данные и поведение в отдельный объект предметной области. Это может потребовать хранения дублированных данных в двух местах и поддерживать их согласованность. Рефакторинг "Дублирование видимых данных" (с. 207) предлагает способ, которым можно это осуществить. В данном случае, особенно при использовании старых компонентов Abstract Windows Toolkit (AWT), после этого можно удалить класс GUI и заменить его компонентами Swing.

Длинный список параметров

Ранее при обучении программированию все необходимые подпрограмме данные рекомендовалось передавать в виде параметров. Это можно понять, потому что альтернативой были глобальные переменные, а глобальные переменные — это одна большая головная боль. Благодаря объектам ситуация изменилась, так как если нужны какие-то данные, их всегда можно запросить у другого объекта. Поэтому, работая с объектами, следует передавать методу только то, что ему нужно, чтобы иметь возможность получить все необходимые ему данные самостоятельно. Значительная часть информации, необходимой методу, находится в классе, которому он принадлежит. В объектно-ориентированных программах списки параметров обычно гораздо короче, чем в традиционных программах.

Это правильно, потому что в длинных списках параметров трудно разобраться. Они зачастую противоречивы и сложны в применении, а кроме того, их приходится вечно изменять по мере возникновения необходимости в новых данных. Если же передавать методам объекты, то изменений потребуется меньше, так как для получения новых данных, скорее всего, хватит пары запросов.

Рефакторинг "Замена параметра вызовом метода" (с. 308) пригоден тогда, когда можно получить данные в одном параметре путем вызова метода объекта, который уже известен. Этот объект может быть полем или другим параметром. Рефакторинг "Сохранение всего объекта" (с. 305) позволяет заменить набор данных, получаемых от объекта, самим этим объектом. Если же имеется ряд элементов данных без логического объекта, можно воспользоваться рефакторингом "Введение объекта параметра" (с. 312).

Есть одно важное исключение, когда такие изменения вносить не следует. Это случай, когда мы не хотим создавать явную зависимость между вызываемым и более крупным объектами. В таких случаях разумно распаковать данные

и передавать их по отдельности в виде параметров, но отдавать себе отчет о стоимости этого решения. Если список параметров оказывается слишком длинным или модификации происходят слишком часто, лучше пересмотреть структуру зависимостей.

Расходящиеся изменения

Мы структурируем программы, чтобы облегчить внесение в них изменений; в конце концов, программное обеспечение тем и отличается от аппаратного обеспечения, что его можно изменять. Планируя изменение, мы хотим перейти в определенную точку программы и внести изменения именно в ней. Если сделать это не удается, то здесь пахнет сразу двумя тесно связанными проблемами.

Расходящиеся (divergent) изменения имеют место тогда, когда один класс часто изменяется различными путями по разным причинам. Если глядя на класс, вы замечаете про себя, что вот эти три метода придется изменять для каждой новой базы данных, а вот эти четыре метода — при каждом появлении нового финансового требования, то это может означать, что вместо одного класса лучше иметь два. Так каждый класс будет иметь свою точно определенную зону ответственности и изменяться только в соответствии с изменениями в этой зоне. Не исключено, что это выяснится лишь после добавления нескольких баз данных или финансовых требований. При каждом вызванном новыми условиями изменении должен изменяться только один класс. Для приведения кода в порядок следует определить все, что модифицируется по данной конкретной причине, а затем применить рефакторинг “Извлечение класса” (с. 169).

Стрельба дробью

“Стрельба дробью” подобна расходящимся изменениям, но представляет собой их противоположность. Унюхать ее можно, когда при выполнении любых изменений приходится вносить множество мелких модификаций в большое число классов. Когда изменения разбросаны повсюду, их трудно искать и можно пропустить важное изменение.

В этой ситуации следует свести все изменения в один класс, использовав рефакторинги “Перенос метода” (с. 162) и “Перенос поля” (с. 166). Если подходящего кандидата среди имеющихся классов нет, создайте новый класс. Часто можно применить рефакторинг “Встраивание класса” (с. 174) и поместить целый пакет методов в один класс. При этом появится определенное количество расходящихся изменений, но вы легко с ними справитесь.

Расходящиеся изменения имеют место при наличии одного класса, в котором выполняется много изменений разных типов, а “стрельба дробью” — это одно изменение, затрагивающее много классов. В обоих случаях желательно добиться того, чтобы между распространенными изменениями и классами было взаимно однозначное отношение.

Завистливые функции

Смысл объектов в том, что они вместе с данными хранят и процедуры для их обработки. Классический пример запаха — метод, который больше интересуется не тем классом, в котором он располагается, а некоторым другим. Чаще всего предметом зависти являются данные. Мы много раз сталкивались с методом, вызывающим с полдюжины методов доступа к данным другого объекта, чтобы вычислить некоторое значение. К счастью, лечение очевидно: метод следует перенести в другое место с помощью рефакторинга “Перенос метода” (с. 162). Иногда завистью страдает только часть метода; в таком случае сначала примените к этой части рефакторинг “Извлечение метода” (с. 132), а уже затем рефакторинг “Перенос метода” (с. 162).

Конечно, не все ситуации так очевидны. Часто метод использует функции нескольких классов, так в какой из них его следует поместить? Мы используем эвристику, заключающуюся с определении того, в каком классе находится больше всего данных, и перемещении метода к этим данным. Этот шаг зачастую оказывается легче, если использовать рефакторинг “Извлечение метода” (с. 132), чтобы разбить метод на части, которые будут размешены в разных местах.

Конечно, могут быть сложные схемы, нарушающие это правило. На ум сразу приходят проектные шаблоны “Стратегия” и “Визитер” [9]. Еще одним примером является проектный шаблон “Самоделегирование” [6]. С его помощью можно бороться с запахом расходящихся изменений. Фундаментальное практическое правило гласит: то, что изменяется одновременно, лучше хранить в одном месте. Данные и функции, использующие эти данные, как правило, изменяются вместе, но бывают и исключения. Сталкиваясь с ними, мы перемещаем поведение так, чтобы изменения осуществлялись в одном месте. Проектные шаблоны “Стратегия” и “Визитер” позволяют легко изменять поведение, потому что они изолируют небольшое количество поведения, которое должно быть перекрыто, ценой увеличения косвенности.

Группы данных

Данные — как дети: они любят сбиваться в тесные группы. Часто можно видеть, как одни и те же три-четыре элемента данных встречаются во множестве мест:

поля в паре классов, параметры в нескольких методах. Данные, встречающиеся совместно, имеет смысл превращать в отдельный класс. Сначала следует найти, где группы данных встречаются в качестве полей, и, применяя к ним рефакторинг "Извлечение класса" (с. 169), преобразовать группы данных в объект. Затем следует обратить внимание на сигнатуры методов и применить рефакторинг "Введение объекта параметра" (с. 312) или "Сохранение всего объекта" (с. 305) для сокращения их объема. Непосредственной выгодой от этого являются сокращение многих списков параметров и упрощение вызовов методов. Не беспокойтесь о том, что некоторые группы данных используют лишь часть полей нового объекта. Заменив несколько полей новым объектом, вы оказываетесь в выигрыше.

Хорошая проверка заключается в том, чтобы удалить одно из значений данных и посмотреть, сохранят ли при этом смысл остальные данные. Если нет, то это верный признак того, что данные лучше объединить в один объект.

Сокращение списков полей и параметров, несомненно, удаляет некоторые запахи, но если у вас есть объекты, можно добиться и приличных запахов. Можно поискать завистливые функции с поведением, которое стоит переместить в новые классы, еще до того, как они станут полезными членами программы.

Одержимость примитивами

Большинство программных сред имеет две разновидности данных. Типы записей позволяют структурировать данные в значимые группы. Стандартными строительными блоками при этом служат примитивные типы. С записями всегда связаны определенные накладные расходы. Записи могут представлять таблицы в базах данных, но их создание может оказаться неудобным, если они нужны лишь в паре случаев.

Одно из ценных свойств объектов заключается в том, что они затушевывают или вообще стирают границу между примитивными и большими классами. Нетрудно написать маленькие классы, неотличимые от встроенных типов языка. В Java есть примитивы для чисел, но строки и даты, которые во многих других средах являются примитивами, здесь представляют собой классы.

Те, кто занимается объектами недавно, обычно не любят использовать маленькие объекты для маленьких задач, такие как, например, денежные классы, объединяющие численное значение и валюту, диапазоны с верхней и нижней границами или специализированные строки наподобие телефонных номеров или почтовых индексов. Выбраться в мир объектов помогает рефакторинг "Замена значения данных объектом" (с. 194), примененный к отдельным значениям данных. Если значение данного представляет собой код типа, воспользуйтесь рефакторингом

"Замена кода типа классом" (с. 236), если это значение не влияет на поведение. Если у вас имеются условные инструкции, зависящие от кода типа, то самое время прибегнуть к рефакторингу "Замена кода типа подклассами" (с. 241) или "Замена кода типа состоянием/стратегией" (с. 245).

При наличии группы полей, которые должны работать совместно, применяйте рефакторинг "Извлечение класса" (с. 169). При наличии примитивов в списках параметров воспользуйтесь рефакторингом "Введение объекта параметра" (с. 312). Если же обнаружатся массивы, попробуйте рефакторинг "Замена массива объектом" (с. 204).

Инструкции switch

Одним из наиболее очевидных признаков объектно-ориентированного кода является относительная немногочисленность инструкций `switch` (или `case`). Основная проблема, связанная с применением `switch`, по сути представляет собой проблему дублирования. Часто одна и та же инструкция `switch` оказывается разбросанной по разным местам программы. При добавлении в нее нового варианта приходится искать все эти инструкции `switch` и изменять их. Объектно-ориентированная концепция полиморфизма предоставляет элегантный способ справиться с этой проблемой.

В большинстве случаев, заметив блок `switch`, следует подумать об использовании полиморфизма. Задача заключается в том, чтобы выяснить, где должен применяться полиморфизм. Часто инструкция `switch` работает с кодами типов, переключая поведение для разных типов. При этом необходим метод или класс, хранящий значение кода типа. Воспользуйтесь рефакторингом "Извлечение метода" (с. 132) для выделения инструкции `switch`, а затем рефакторингом "Перенос метода" (с. 162) для ее вставки в класс, где требуется полиморфизм. В этот момент вы должны решить, чем именно воспользоваться — рефакторингом "Замена кода типа подклассами" (с. 241) или рефакторингом "Замена кода типа состоянием/стратегией" (с. 245). Разобравшись в структуре наследования, можно применить рефакторинг "Замена условной инструкции полиморфизмом" (с. 271).

Если имеется малое количество вариантов инструкции `switch`, оказывающих влияние на один метод, и не предполагается их изменение, то применение полиморфизма оказывается излишним. В таком случае хорошим выбором может быть рефакторинг "Замена параметра явными методами" (с. 302). Если одним из вариантов является нулевой, подумайте о применении рефакторинга "Введение нулевого объекта" (с. 276).

Параллельные иерархии наследования

Параллельные иерархии наследования в действительности представляют собой частный случай стрельбы дробью. В этом случае всякий раз, когда вы создаете подкласс одного из классов, вам приходится создавать еще и подкласс другого класса. Этот запах можно распознать по тому, что префиксы имен классов в двух разных иерархиях классов оказываются одинаковыми.

Общая стратегия устранения дублирования состоит в том, чтобы заставить экземпляры одной иерархии ссылаться на экземпляры другой. С помощью рефакторингов “Перенос метода” (с. 162) и “Перенос поля” (с. 166) можно устранить излишнюю иерархию.

Ленивый класс

Сопровождение и понимание каждого создаваемого класса влечет определенные затраты. Класс, существование которого не окупается выполняемыми им функциями, должен быть ликвидирован. Часто это класс, создание которого было оправданным в свое время, но уменьшившийся в результате рефакторинга. Это может быть класс, добавленный для планировавшейся модификации, но которая так и не была осуществлена. В любом случае следует дать классу возможность умереть с честью. Если у вас есть подклассы с недостаточной функциональностью, попробуйте применить рефакторинг “Свертывание иерархии” (с. 360). Компоненты, являющиеся практически бесполезными, должны быть подвергнуты рефакторингу “Встраивание класса” (с. 174).

Теоретическая общность

Брайан Фут (Brian Foote) предложил название “теоретическая общность” (speculative generality) для запаха, к которому мы очень чувствительны. Он возникает, когда говорят о том, что в будущем, наверное, потребуется возможность делать то или иное, и хотят обеспечить набор механизмов для работы с вещами, которые пока что не нужны. Получающуюся в результате программу труднее понимать и сопровождать. Если бы все эти механизмы использовались, их наличие было бы оправданным, а без этого они только мешают, так что лучше от них избавиться.

Если у вас есть абстрактные классы, не приносящие большой пользы, избавляйтесь от них путем применения рефакторинга “Свертывание иерархии” (с. 360). Излишнее делегирование можно устранить с помощью рефакторинга

"Встраивание класса" (с. 174). Методы с неиспользуемыми параметрами должны быть подвергнуты рефакторингу "Удаление параметра" (с. 294). Методы со странными абстрактными именами необходимо вернуть на землю путем рефакторинга "Переименование метода" (с. 290).

Теоретическая общность может наблюдаться, когда единственными пользователями метода или класса являются тестовые примеры. Найдя такой метод или класс, удалите его и тестовый пример, его проверяющий. Если для тестового примера имеется вспомогательный метод или класс, осуществляющий разумные функции, его, конечно, следует оставить.

Временное поле

Иногда выясняется, что в некотором объекте атрибут устанавливается только в определенных обстоятельствах. Такой код труден для понимания, поскольку вы ожидаете, что объекту нужны все его переменные. Можно сломать голову, пытаясь понять, для чего нужна некоторая переменная, если найти, где она используется, никак не удается.

С помощью рефакторинга "Извлечение класса" (с. 169) создайте приют для бедных осиротевших переменных. Поместите в него весь код, работающий с ними. Возможно, вам удастся удалить условный код с помощью рефакторинга "Введение нулевого объекта" (с. 276) для создания альтернативного компонента для случая, когда переменные являются некорректными.

Обычно временные поля возникают, когда сложному алгоритму требуется несколько переменных. Программист, который реализовывал алгоритм, не захотел пересылать большой список параметров (да и кто бы захотел?), поэтому он разместил их в полях. Но поля корректны только во время работы алгоритма; в других контекстах они лишь вводят в заблуждение. В таком случае можно применить рефакторинг "Извлечение класса" (с. 169) к переменным и методам, в которых требуются эти поля. Новый объект является объектом метода [6].

Цепочки сообщений

Цепочки сообщений возникают, когда клиент запрашивает у одного объекта другой, у того клиент, в свою очередь, запрашивает еще один объект, у которого запрашивается еще один... и т. д. Это может выглядеть как длинный ряд методов типа `getThis` или как последовательность временных переменных. Такие последовательности вызовов означают, что клиент связан с навигацией по структуре классов. Любые изменения промежуточных связей приводят к необходимости модификации клиента.

Здесь можно применить рефакторинг “Сокрытие делегирования” (с. 176), причем его можно использовать в различных местах цепочки. В принципе можно выполнить его для каждого объекта цепочки, но это часто превращает каждый промежуточный объект в посредника. Зачастую лучшей альтернативой оказывается выяснение, для чего используется конечный объект. Посмотрите, нельзя ли с помощью рефакторинга “Извлечение метода” (с. 132) взять использующую его часть кода и с помощью рефакторинга “Перенос метода” (с. 162) сместить его вниз по цепочке. Если несколько клиентов одного из объектов цепочки хотят пройти остальную часть пути, добавьте для этого соответствующий метод.

Некоторые программисты рассматривают любую цепочку вызовов методов как нечто ужасное. Авторы же относятся к этому явлению со спокойной взвешенной умеренностью. По крайней мере, в данном случае.

Посредник

Одной из основных характеристик объектов является инкапсуляция — сокрытие внутренних деталей реализации от внешнего мира. Инкапсуляции часто сопутствует делегирование. Например, вы договариваетесь с директором о встрече. Он делегирует это сообщение своему календарю и дает вам ответ. Все хорошо и правильно. Совершенно не важно, использует ли директор ежедневник, электронное устройство или своего секретаря, чтобы вести расписание личных встреч.

Однако такой подход может завести слишком далеко. Например, мы просматриваем интерфейс класса и обнаруживаем, что половина методов делегирует обработку другому классу. В таком случае нужно воспользоваться рефакторингом “Удаление посредника” (с. 179) и общаться с объектом, который действительно знает, что происходит. При наличии нескольких методов, которые не выполняют большой объем работ, их можно поместить в вызывающий метод с помощью рефакторинга “Встраивание метода” (с. 139). При наличии дополнительного поведения с помощью рефакторинга “Замена делегирования наследованием” (с. 371) можно преобразовать посредник в подкласс реального объекта. Это позволит вам расширить поведение, не прибегая ко всему этому делегированию.

Неуместная близость

Временами классы оказываются в слишком интимных отношениях и чаще, чем это следует, погружаются в закрытые части один другого. Мы не ханжи, когда это касается людей, но считаем, что классы должны следовать строгим пуританским правилам.

Чрезмерно интимничающие классы нужно разделять так же, как в прежние времена это делали с молодыми влюбленными. С помощью рефакторингов “Перенос метода” (с. 162) и “Перенос поля” (с. 166) необходимо разделить части кода, чтобы уменьшить их близость. Посмотрите, нельзя ли прибегнуть к рефакторингу “Замена двунаправленной связи однонаправленной” (с. 219). Если у классов есть общие интересы, воспользуйтесь рефакторингом “Извлечение класса” (с. 169), чтобы поместить общую деятельность в безопасное место и превратить их в добропорядочные классы. Можно также применить рефакторинг “Замена наследования делегированием” (с. 369), позволив другому классу выступать в качестве промежуточного звена.

Наследование зачастую приводит к чрезмерной близости. Подклассы всегда знают о своих родителях больше, чем последним хотелось бы. Если пришло время покинуть родительский дом, примените рефакторинг “Замена наследования делегированием” (с. 369).

Альтернативные классы с разными интерфейсами

Примените рефакторинг “Переименование метода” (с. 290) ко всем методам, выполняющим одни и те же действия, но имеющим разные сигнатуры. Однако часто этого оказывается недостаточно. В таких случаях классы делают все еще недостаточно. Продолжайте применять рефакторинг “Перенос метода” (с. 162) для перемещения поведения в классы, пока протоколы не станут одинаковыми. Если для этого придется выполнить избыточное перемещение кода, можно попробовать компенсировать это применением рефакторинга “Извлечение суперкласса” (с. 352).

Неполный библиотечный класс

Цель применения объектов зачастую поясняется как повторное использование кода. Мы считаем, что важность этого аспекта сильно переоценивается (нам достаточно просто использования). Но не будем отрицать, что программирование во многом основывается на применении библиотечных классов, благодаря которым никто не сможет утверждать, что мы забыли, как работают алгоритмы, скажем, сортировки.

Разработчики библиотечных классов не всеведущи, и их нельзя за это осуждать; в конце концов, зачастую проект становится понятным лишь тогда, когда он почти готов, так что перед разработчиками библиотек действительно стоит нелегкая задача. Беда в том, что часто считается дурным тоном (и обычно

оказывается невозможным) модифицировать библиотечный класс, чтобы он выполнял какие-то желательные нам действия. Это означает, что испытанная тактика вроде рефакторинга "Перенос метода" (с. 162) оказывается бесполезной.

У нас есть пара специализированных инструментов для выполнения этой работы. Если в библиотечный класс надо включить всего лишь пару новых методов, используйте рефакторинг "Введение внешнего метода" (с. 181). Если дополнительная функциональность достаточно велика, необходимо применить рефакторинг "Введение локального расширения" (с. 183).

Классы данных

Такие классы содержат поля, методы для получения и установки значений этих полей и ничего больше. Такие классы — молчаливые хранилища данных, которыми другие классы наверняка манипулируют излишне обстоятельно. На ранних этапах в этих классах могут быть открытые поля, и тогда необходимо немедленно, пока их никто не обнаружил, применить рефакторинг "Инкапсуляция поля" (с. 224). При наличии полей коллекций проверьте, инкапсулированы ли они должным образом, и если нет, то примените рефакторинг "Инкапсуляция коллекции" (с. 226). Рефакторинг "Сокрытие метода" (с. 320) применяется ко всем полям, значение которых не должно изменяться.

Посмотрите, как методы доступа к полям используются другими классами. Попробуйте использовать рефакторинг "Перенос метода" (с. 162) для перемещения поведения в класс данных. Если метод не удается переместить целиком, воспользуйтесь рефакторингом "Извлечение метода" (с. 132), чтобы создать метод, который можно переместить. Через некоторое время можно начать применять рефакторинг "Сокрытие метода" (с. 320) к методам получения и установки значений полей.

Классы данных подобны детям. В качестве начальной точки они годятся, но чтобы участвовать в работе в качестве взрослых объектов, они должны принять на себя определенную ответственность.

Отказ от наследства

Подклассам полагается наследовать методы и данные своих родителей. Но что делать, если это наследство им не нравится или не требуется? И получив все эти дары, они пользуются только малой их частью.

Обычно это означает неправильно продуманную иерархию. Необходимо создать новый ("братский") класс на одном уровне с потомком и использовать рефакторинги "Опускание метода" (с. 345) и "Опускание поля" (с. 346), чтобы вынести в него все неиспользуемые методы. Благодаря этому в суперклассе будет содержаться только та функциональность, которая используется. Часто вы можете встретить совет делать все суперклассы абстрактными.

Мы не будем советовать такие крайности; как минимум этот совет годится не всегда. Мы постоянно обращаемся к созданию подклассов для повторного использования части функций и считаем это совершенно нормальным стилем программирования. Небольшой запах обычно остается, но не такой уж сильный. Поэтому мы говорим, что если не принятое наследство вызывает какие-то проблемы, следуйте традиционному совету. Но не следует думать, что так надо делать всегда. В девяти случаях из десяти запах слишком слабый, чтобы требовалось любой ценой от него избавиться.

Запах отвергнутого наследства становится сильнее, если подкласс повторно использует функции суперкласса, но не желает поддерживать его интерфейс. Мы не возражаем против отказа от реализаций, но отказ от интерфейса нас возмущает. В этом случае не возитесь с иерархией; ее лучше разрушить с помощью рефакторинга "Замена наследования делегированием" (с. 371).

Комментарии

Не волнуйтесь, мы не станем утверждать, что писать комментарии не нужно. В нашей обонятельной аналогии комментарии издают не дурной, а вовсе даже приятный запах. Мы упомянули комментарии потому, что часто они играют роль дезодоранта. Удивительно часто встречается код с обильными комментариями, которые появились в нем лишь потому, что код плохой.

Комментарии приводят нас к плохому коду, издающему всевозможные дурные запахи, о которых мы писали в этой главе. Первым действием должно быть удаление этих запахов с помощью рефакторинга. После этого комментарии часто оказываются ненужными.

Если для объяснения действий блока кода нужен комментарий, попробуйте применить рефакторинг "Извлечение метода" (с. 132). Если метод уже выделен, но комментарий для объяснения его действий остается необходимым, воспользуйтесь рефакторингом "Переименование метода" (с. 290). А если требуется изложить некоторые правила, касающиеся необходимого состояния системы, примените рефакторинг "Введение утверждения" (с. 284).

Почувствовав необходимость написать комментарий, попробуйте сначала изменить структуру кода так, чтобы любые комментарии стали ненужными.

Комментарии полезны, когда вы не знаете, что делать. Помимо описания происходящего, комментарии могут отмечать те места, в которых вы не уверены. Комментарии — хорошее место пояснить, *почему* вы поступаете именно так. Эта информация пригодится тем, кто будет работать с вашим кодом в будущем.

Глава 4

Создание тестов

Важным предварительным условием проведения рефакторинга является наличие надежных тестов. Даже если вам повезло и у вас есть средство, автоматизирующее рефакторинг, без тестов все равно не обойтись. Еще не скоро наступит время, когда всевозможные методы рефакторинга будут выполняться автоматически, с помощью специального инструментария.

Я не считаю это серьезным недостатком. Мой опыт показывает, что хорошие тесты значительно увеличивают скорость программирования, даже если оно не связано с рефакторингом. Это оказалось сюрпризом для меня и противоречит интуиции многих программистов, поэтому стоит разобраться, почему это так.

Важность самотестируемого кода

Если выяснить, на что в основном уходит время большинства программистов, то окажется, что написание кода в действительности занимает лишь небольшую его часть. Некоторое время уходит на уяснение задачи, другая часть — на проектирование, но львиную долю занимает отладка. Уверен, что каждый программист может припомнить долгие часы отладки, часто до поздней ночи, или рассказать историю о том, как на поиски какой-то ничтожной ошибки ушел целый день (а то и больше). Исправить ошибку обычно удается довольно быстро, но ее поиск превращается в кошмар. При этом при исправлении ошибки всегда существует возможность внести новую, которая проявится гораздо позднее, и утечет немало воды, прежде чем вы ее обнаружите.

Событием, подтолкнувшим меня на путь создания самотестируемого кода, стал разговор на OOPSLA в 1992 году. Некто (припоминается, это был Дэйв Томас (Dave Thomas)) небрежно заметил: "Классы должны содержать тесты для самих себя". Мне пришло в голову, что это неплохой способ организации тестирования. Я истолковал эти слова так, что в каждом классе должен быть свой метод (например, с именем `test`), с помощью которого класс может протестировать сам себя.

Тогда я занимался постепенной разработкой (incremental development), а потому попытался по завершении каждого шага добавлять в классы тестирующие методы. Проект был совсем небольшим, и каждый шаг занимал у нас примерно

неделю. Выполнять тесты стало достаточно просто, но хотя и их было легко запускать, само тестирование оставалось весьма утомительным, потому что каждый тест выводил на консоль результаты, которые приходилось проверять. Я человек достаточно ленивый, так что готов как следует потрудиться, лишь бы избавиться от лишней работы. Очевидно, что можно не смотреть на экран в поисках некоторой информации, которая должна быть выведена, а заставить заниматься этим компьютер. Все, что требовалось, — это поместить ожидаемые данные в код теста и провести в нем сравнение. После этого можно было просто вызывать тестирующий метод каждого класса, и если все было в порядке, то он просто выводил на экран сообщение OK. Так классы стали самотестируемыми.

Тесты должны быть полностью автоматизированы и проверять свои результаты.

После этого выполнение тестов ничуть не труднее, чем компиляция, так что я начал выполнять их при каждой компиляции. Вскоре обнаружилось, что производительность моего труда резко возросла. Я понял, что перестал тратить много времени на отладку! Если я допускал ошибку, перехватываемую предыдущим тестом, это обнаруживалось, как только я запускал этот тест. Поскольку раньше тест работал, я понимал, что ошибка находится там, где я внес изменения после предыдущего тестирования. Тесты я запускал часто, буквально каждые несколько минут, так что всегда знал, где именно находится ошибка — в том коде, который я только что написал. Этот код был еще свеж в памяти, мал по размеру, так что найти ошибку было легко. Часы, которые ранее приходилось тратить на поиск ошибок, превратились в считанные минуты. Такое мощное средство обнаружения ошибок появилось у меня не только благодаря тому, что я писал классы с самотестированием, но и потому, что часто выполнял тестирование.

Придя к таким выводам, я стал тестировать свой код еще более агрессивно. Я стал добавлять тесты сразу же после написания небольших фрагментов функций, не дожидаясь завершения этапа разработки. За день, как правило, добавлялась пара новых функций и тесты для них. На отладку я стал тратить не более нескольких минут в день.

Набор тестов является мощным детектором ошибок, резко уменьшающим время их поиска.

Убедить других последовать тем же путем, конечно же, было нелегко. Тестам необходим большой объем кода. Самотестирование кажется бессмысленным, пока

не убедишься сам, насколько оно ускоряет работу. К тому же многие не просто совершенно не умеют писать тесты, но даже никогда о них и не задумываются. Проводить тестирование вручную — занятие не из веселых, но если его автоматизировать, то написание тестов может даже доставлять удовольствие.

Фактически очень полезно писать тесты еще до начала программирования. Когда требуется добавить новую функциональность, начинайте с создания теста. Это не так глупо, как может показаться. Когда вы пишете тест, то спрашиваете себя, что нужно сделать для добавления этой функциональности. При написании теста вы сосредоточиваетесь на интерфейсе, а не на реализации, что всегда хорошо. Кроме того, это означает наличие точного критерия завершения кодирования — в конечном итоге тест должен успешно проходиться.

Идея частого тестирования составляет важную часть экстремального программирования [3]. Название вызывает представление о программистах из числа скорых и небрежных хакеров. Однако экстремальные программисты очень привержены частому тестированию. Они стремятся разрабатывать программы как можно быстрее и знают, что двигаться вперед с большой скоростью позволяет только частое тестирование.

Но прекратим эту бесполезную дискуссию. Хотя я уверен, что написание самотестируемого кода крайне выгодно для всех, эта книга все же посвящена другой теме, а именно — рефакторингу. Рефакторинг требует тестов. Тому, кто собирается заниматься рефакторингом, просто необходимо писать тесты. В этой главе дается беглое представление о том, как это делается при работе на языке программирования Java. Книга посвящена не тестированию, так что я не стану погружаться в детали. Но что касается тестирования, я твердо убедился, что даже малое его количество может принести очень большую пользу.

Как и все остальное в этой книге, подход к тестированию описан с применением большого количества примеров. При разработке кода лично я пишу тесты прямо по ходу работы; но зачастую, когда над рефакторингом приходится работать вместе с другими людьми, приходится иметь дело с большим объемом кода при отсутствии самотестирования. Поэтому, прежде чем применить рефакторинг, нужно сделать код самотестируемым.

Стандартной идиомой Java для тестирования является тестирующая функция `main`. Идея заключается в наличии в каждом классе тестирующей функции `main`, предназначенной для проверки работоспособности класса. Это разумное соглашение (хотя и принимается далеко не всеми), но оно может оказаться неудобным. Дело в том, что при таком соглашении сложно запускать много тестов. Другой подход заключается в создании отдельных тестирующих классов, работа которых должна облегчить тестирование.

Каркас тестирования JUnit

Я пользуюсь каркасом тестирования JUnit — системой с открытым исходным кодом, которую разработали Эрих Гамма (Erich Gamma) и Кент Бек [4]. Он очень прост, но при этом позволяет делать все необходимое для тестирования. В данной главе с его помощью я разрабатываю тесты для некоторых классов ввода-вывода.

Для начала я создаю класс `FileReaderTester` для тестирования класса, читающего файлы. Все классы, содержащие тесты, должны создаваться как подклассы класса контрольного примера каркаса тестирования. В среде используется составной проектный шаблон (composite pattern) [9], позволяющий объединять тесты в наборы (рис. 4.1).

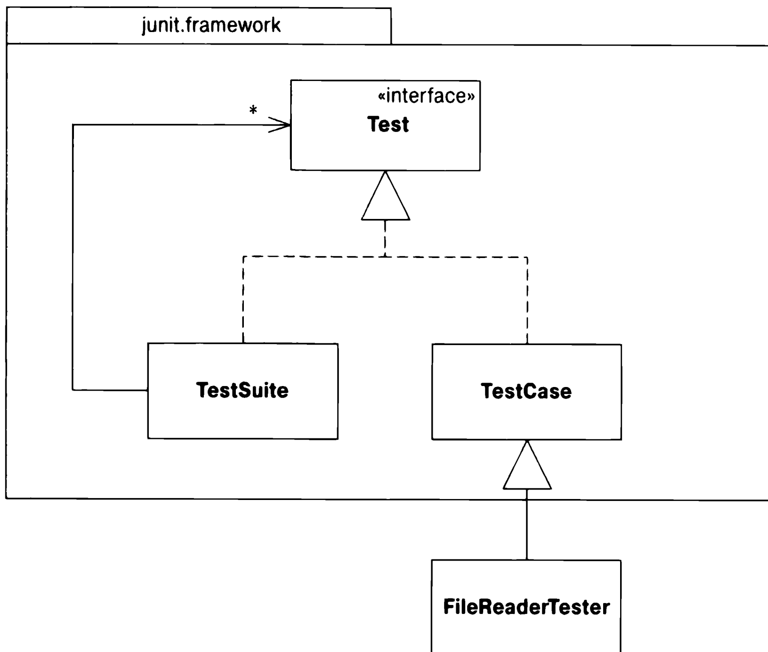


Рис. 4.1. *Составная структура тестов*

В этих наборах могут содержаться простые контрольные примеры или другие наборы контрольных примеров. Благодаря этому легко строить большие наборы тестов и выполнять их автоматически.

```
class FileReaderTester extends TestCase {
    public FileReaderTester(String name) {
        super(name);
    }
}
```

Новый класс должен иметь конструктор. После этого я начинаю добавление исходного текста в тест. Первая задача — подготовка тестовых данных. По сути, это объекты, работающие в качестве образцов для тестирования. Поскольку будет происходить чтение файла, нужно подготовить тестовый файл, например такой:

```
Bradman    99.94  52  80 10 6996 334   29
Pollock    60.97  23  41  4 2256 274    7
Headley    60.83  22  40  4 2256 270*  10
Sutcliffe  60.73  54  84  9 4555 194   16
```

В классе тестового примера есть два метода работы с тестовыми данными: `setUp` создает объекты, а `tearDown` удаляет их. Оба они реализованы как нулевые методы контрольного примера. Удалять объекты обычно нет необходимости, так как этим занимается сборщик мусора, но будет разумно использовать этот метод для закрытия файла:

```
class FileReaderTester...
    protected void setUp() {
        try {
            _input = new FileReader("data.txt");
        } catch (FileNotFoundException e) {
            throw new RuntimeException("Ошибка открытия файла");
        }
    }

    protected void tearDown() {
        try {
            _input.close();
        } catch (IOException e) {
            throw new RuntimeException("Ошибка закрытия файла");
        }
    }
```

Теперь, когда тестовые данные готовы, можно начать писать тесты. Сначала проверяем метод чтения. Для этого я считываю несколько символов, а затем проверяю их корректность:

```
public void testRead() throws IOException {
    char ch = '&';
    for (int i = 0; i < 4; i++)
        ch = (char) _input.read();
    assert('d' == ch);
}
```

Автоматическое тестирование представляет собой вызов `assert`. Если выражение внутри `assert` истинно, все в порядке. В противном случае генерируется сообщение об ошибке. Позже я покажу, как это делает каркас, но сначала опишу, как выполнить тест.

Первый шаг состоит в создании набора тестов. Для этого создаем метод под названием suite:

```
class FileReaderTester...
    public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTest(new FileReaderTester("testRead"));
        return suite;
    }
```

В этом наборе тестов имеется только один объект контрольного примера, экземпляр FileReaderTester. При создании контрольного примера я передаю конструктору строковый аргумент с именем метода, который хочу тестировать. В результате создается объект, тестирующий один этот метод. Тест работает с тестируемым объектом с помощью возможностей рефлексии Java. Вы можете просмотреть исходный текст каркаса и разобраться, как это происходит. Я же отношусь к этому как к магии.

Для выполнения теста я использую отдельный класс TestRunner. Есть две версии класса TestRunner: одна с мощным графическим интерфейсом, а вторая — с простым консольным интерфейсом, которую можно вызывать в main:

```
class FileReaderTester...
    public static void main(String[] args) {
        junit.textui.TestRunner.run(suite());
    }
```

Этот код создает класс, выполняющий тесты, и указывает ему, что следует протестировать класс FileReaderTester. После выполнения теста выводится следующее:

```
.
Time: 0.110

OK(1 tests)
```

JUnit выводит точку для каждого выполняемого теста, чтобы можно было следить за происходящим. Каркас также сообщает, сколько времени выполнялся тест. После этого, если тест успешно пройден, выводится сообщение OK и количество выполненных тестов. Я могу выполнить тысячу тестов, но если все будет в порядке, я увижу только одно это сообщение. Такая простая обратная связь важна для самотестируемого кода. Если ее не будет, никто не будет запускать тесты достаточно часто. Зато при ее наличии можно запустить кучу тестов, отправиться на обед или на совещание, а вернувшись проверить полученный результат.

Выполняйте тесты почаще. Выполняйте их при каждой компиляции, как минимум каждый тест ежедневно.

При проведении рефакторинга выполняются лишь несколько тестов для проверки кода, который модифицируется. Можно выполнять лишь часть тестов, так как требуется достаточно быстрое тестирование, так как в противном случае оно будет замедлять работу, и может возникнуть соблазн отказаться от тестов. Не поддавайтесь этому соблазну — возмездие *неминуемо.*

А что будет, если что-то пойдет не так, как надо? Давайте специально внесем ошибку:

```
public void testRead() throws IOException {
    char ch = '&';
    for (int i = 0; i < 4; i++)
        ch = (char) _input.read();
    assert('2' == ch); // Преднамеренная ошибка
}
```

Результат теперь имеет следующий вид:

```
.F
Time: 0.220

!!!FAILURES!!!
Test Results:
Run: 1 Failures: 1 Errors: 0
There was 1 failure:
1) FileReaderTester.testRead
test.framework.AssertionFailedError
```

Каркас тестирования предупреждает об ошибке и сообщает, какой из тестов оказался неудачным. Такое сообщение об ошибке не слишком информативно, и его можно улучшить с помощью `assert` другого вида.

```
public void testRead() throws IOException {
    char ch = '&';
    for (int i = 0; i < 4; i++)
        ch = (char) _input.read();
    assertEquals('m', ch);
}
```

В большинстве вызовов `assert` проверяется равенство двух значений, поэтому каркас тестирования для такого случая имеет `assertEquals`. Его наличие достаточно удобно: для сравнения объектов эта конструкция применяет вызовы `equals()`, а для сравнения значений — оператор ==, о чем я часто забываю. Кроме того, при этом выводится более содержательное сообщение об ошибке:

```
.F
Time: 0.170

!!!FAILURES!!!
Test Results:
Run: 1 Failures: 1 Errors: 0
There was 1 failure:
1) FileReaderTester.testRead "expected: "m" but was: "d""
```

Должен отметить, что при создании тестов я часто начинаю с того, что проверяю их реакцию на неприятности. Существующий код я изменяю так, чтобы тест завершался неудачей (если этот код мне доступен), либо помещаю неверное ожидаемое значение в `assert`. Я поступаю так, чтобы проверить, действительно ли выполняется тест и действительно ли он проверяет то, что должен (вот почему я предпочитаю по возможности изменить тестируемый код). Можете считать это паранойей, но несложно запутаться, если тесты проверяют не то, что должны по вашим представлениям.

Помимо неудачного выполнения тестов (утверждений, оказывающихся ложными), каркас перехватывает ошибки (неожиданные исключения). Если, например, закрыть поток и попытаться после этого выполнить из него чтение, будет сгенерировано исключение. Это можно проверить с помощью кода:

```
public void testRead() throws IOException {
    char ch = '&';
    _input.close();

    for (int i = 0; i < 4; i++)
        ch = (char) _input.read(); // Генерация исключения

    assertEquals('m', ch);
}
```

При выполнении этого кода получается следующий результат:

```
.E
Time: 0.110

!!!FAILURES!!!
Test Results:
Run: 1 Failures: 0 Errors: 1
There was 1 error:
1) FileReaderTester.testRead
java.io.IOException: Stream closed
```

Полезно различать неудачные тесты и исключения, потому что и обнаруживаются они по-разному, и процесс их отладки разный.

JUnit включает удачно сделанный графический интерфейс (рис. 4.2). Индикатор имеет зеленый цвет, если все тесты прошли успешно, и красный цвет, если возникли какие-то отказы. Можно постоянно держать этот интерфейс открытым, а среда тестирования будет автоматически компоновать любые изменения, сделанные в коде. Так получается очень удобный способ выполнения тестов.

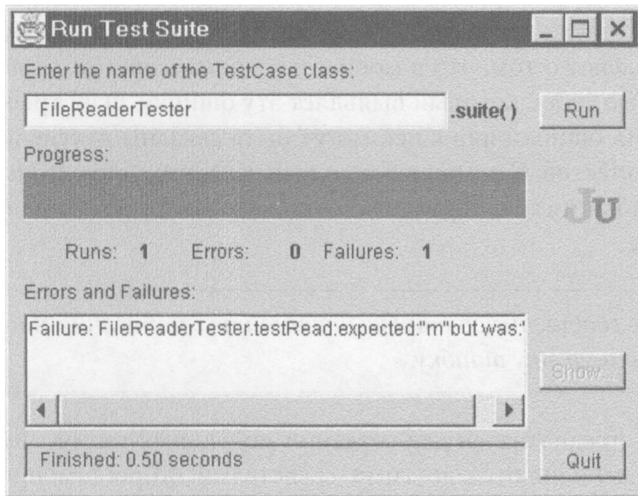


Рис. 4.2. *Графический интерфейс пользователя в JUnit*

Модульные и функциональные тесты

Описываемый каркас используется для тестирования модулей, поэтому я хочу остановиться на различии между тестированием модулей и тестированием функциональности. Тесты, о которых здесь идет речь, — *модульные тесты* (unit tests). Я пишу их для повышения продуктивности своей работы. Побочный эффект, который при этом достигается, — удовлетворение отдела контроля качества. Модульные тесты достаточно сильно локализованы. Каждый класс теста работает внутри одного пакета. Он проверяет интерфейсы с другими пакетами, но исходит из того, что за пределами проверяемого модуля все корректно работает.

Функциональные тесты (functional tests) — это совсем другое дело. Их пишут для проверки работоспособности системы в целом. Такие тесты гарантируют потребителю качество продукта, при этом вопрос о производительности труда программиста остается за рамками. Разрабатывать такие тесты должна отдельная команда программистов из тех, для кого обнаружить ошибку — одно удовольствие. При написании таких тестов используются мощные инструменты и технологии.

Обычно в функциональных тестах стремятся рассматривать систему в целом как единый черный ящик. В системе с графическим интерфейсом пользователя

работа выполняется через этот интерфейс. Для программ, изменяющих файлы или базы данных, эти тесты проверяют, какие результаты будут получены при вводе определенных данных.

Когда тестировщик (или пользователь) обнаруживает ошибку в программном обеспечении, для ее исправления требуются по меньшей мере две вещи. Естественно, нужно модифицировать код программы и устранить ошибку, но это не все. Необходимо также добавить модульный тест, обнаруживающий эту ошибку. Когда мне сообщают о том, что в моем коде найдена ошибка, я начинаю с написания модульного теста, который выявляет эту ошибку. Если требуется уточнить область действия ошибки или с ней могут быть связаны другие неприятности, я пишу несколько тестов. С помощью модульных тестов я определяю точное местоположение ошибки и гарантирую, что ошибка такого рода больше не проскочит через мои тесты.

Получив сообщение об ошибке, начинайте с написания модульного теста, выявляющего эту ошибку.

Каркас JUnit предназначен для создания модульных тестов. Функциональные тесты обычно выполняются другими средствами. Хорошим примером служат средства тестирования на основе графического интерфейса пользователя. Однако зачастую имеет смысл написание собственных инструментов тестирования для конкретного приложения, с помощью которых легче организовать контрольные примеры, чем при использовании одних лишь сценариев графического интерфейса пользователя. С помощью JUnit можно осуществлять и функциональное тестирование, но обычно это не самый эффективный и удобный путь. При рефакторинге я полагаюсь на испытанных друзей программистов — модульные тесты.

Добавление новых тестов

Продолжим добавлять тесты. Я следую стилю, при котором все действия, выполняемые классом, рассматриваются и проверяются при всех условиях, которые могут вызвать отказ. Это не то же самое, что тестирование всех открытых методов, пропагандируемое рядом программистов. Тестирование должно управляться рисками. Наша задача — найти ошибки, которые могут возникнуть сейчас или в будущем. Поэтому я, например, не проверяю методы доступа, осуществляющие лишь чтение или запись поля: они настолько просты, что вряд ли в них обнаружится ошибка.

Это важный момент, так как попытка написать слишком много тестов обычно приводит к тому, что их оказывается недостаточно. Я прочел не одну книгу о тестировании, вызывавшую одно лишь желание любой ценой уклониться от той необъятной работы, которую предлагалось проделать. Всеохватывающее тестирование контрпродуктивно, поскольку заставляет считать, что тестирование всегда связано с чрезмерным трудом. Однако тестирование приносит ощутимую пользу, даже если осуществляется в небольшом объеме. Главное для решения проблемы тестирования — тестировать в основном код, возможность ошибок в котором вызывает наибольшее беспокойство. При этом подходе усилия, затраченные на тестирование, дают максимальную выгоду.

Лучше написать и выполнить неполные тесты, чем не выполнить полные тесты.

Сейчас я рассматриваю метод чтения. Что еще он должен делать? Так, из исходного текста видно, что он возвращает -1 при достижении конца файла (на мой взгляд, не очень удачное решение... но, пожалуй, программистам на С оно должно казаться вполне естественным). Выполним тест. Текстовый редактор сообщает, что в файле 141 символ, поэтому тест будет таким:

```
public void testReadAtEnd() throws IOException {
    int ch = -1234;

    for (int i = 0; i < 141; i++)
        ch = _input.read();

    assertEquals("чтение за концом", -1, ch, _input.read());
}
```

Для выполнения теста его надо добавить в набор тестов:

```
public static Test suite() {
    TestSuite suite = new TestSuite();
    suite.addTest(new FileReaderTester("testRead"));
    suite.addTest(new FileReaderTester("testReadAtEnd"));
    return suite;
}
```

Этот набор выполняет все тесты, из которых состоит (два контрольных примера). Каждый контрольный пример выполняет `setUp`, код тестирующего метода и в заключение `tearDown`. Выполнение `setUp` и `tearDown` необходимо для изоляции тестов одного от другого, так, что их можно выполнять в любом порядке.

Помнить о необходимости добавления тестов в метод suite — сплошная головная боль. К счастью, Эрих Гамма и Кент Бек ленивы, как и я, а потому предоставили возможность обойтись без этого. Специальный конструктор набора тестов принимает класс в качестве параметра. Этот конструктор формирует набор тестов, содержащий контрольный пример для каждого метода, имя которого начинается с test. Следуя этому соглашению, можно заменить функцию main следующей:

```
public static void main(String[] args) {
  junit.textui.TestRunner.run(new TestSuite(FileReaderTester.class));
}
```

В результате каждый написанный мною тест будет включен в набор тестов.

Главная хитрость в тестах состоит в том, чтобы найти граничные условия. В случае чтения границами будут первый символ, последний символ и символ, следующий за последним:

```
public void testReadBoundaries()throws IOException {
    assertEquals("чтение первого символа", 'B', _input.read());
    int ch;

    for (int i = 1; i < 140; i++)
        ch = _input.read();

    assertEquals("чтение последнего символа", '6', _input.read());
    assertEquals("чтение за концом", -1, _input.read());
}
```

Обратите внимание, что в assert можно поместить сообщение, выводимое в случае неудачного теста.

Подумайте о граничных условиях, которые могут быть неправильно обработаны, и сосредоточьте на них свои усилия.

Другую часть поиска границ составляет выявление особых условий, способных привести к неудаче теста. В случае работы с файлами такими условиями могут оказаться пустые файлы:

```
public void testEmptyRead() throws IOException {
    File empty = new File("empty.txt");
    FileOutputStream out = new FileOutputStream(empty);
    out.close();
    FileReader in = new FileReader(empty);
    assertEquals(-1, in.read());
}
```

В этом случае я создаю дополнительные тестовые данные. Если позже мне понадобится пустой файл, его можно будет поместить в обычный набор тестов, перенеся соответствующий код в `setUp`.

```
protected void setUp() {
    try {
        _input = new FileReader("data.txt");
        _empty = newEmptyFile();
    } catch (IOException e) {
        throw new RuntimeException(e.toString());
    }
}

private FileReader newEmptyFile() throws IOException {
    File empty = new File("empty.txt");
    FileOutputStream out = new FileOutputStream(empty);
    out.close();
    return new FileReader(empty);
}

public void testEmptyRead() throws IOException {
    assertEquals(-1, _empty.read());
}
```

Что произойдет, если выполнять чтение за концом файла? Снова должно возвратиться значение `-1`, и я добавляю еще один соответствующий тест:

```
public void testReadBoundaries()throws IOException {
    assertEquals("read first char", 'B', _input.read());
    int ch;

    for (int i = 1; i < 140; i++)
        ch = _input.read();

    assertEquals("read last char", '6', _input.read());
    assertEquals("read at end", -1, _input.read());
    assertEquals("read past end", -1, _input.read());
}
```

Обратите внимание, как я играю роль врага для моего кода. Я активно стараюсь добиться его неверной работы! Такие действия кажутся мне одновременно и продуктивными, и забавными (они доставляют удовольствие темной стороне моей натуры).

Выполняя тесты, не забывайте проверять, происходят ли ожидаемые ошибки в надлежащем порядке. При попытке чтения потока после его закрытия должно генерироваться исключение `IOException`. Это тоже должно быть проверено:

```
public void testReadAfterClose() throws IOException {
    _input.close();

    try {
        _input.read();
        fail("Нет исключения при чтении за концом файла");
    } catch (IOException io) {}
}
```

Все прочие исключения, отличные от IOException, генерируют ошибку обычным путем.

Не забывайте проверять генерацию исключений в случае возникновения проблем.

Для того чтобы протестировать интерфейс некоторых классов, придется потрудиться, но в процессе вы получите действительное понимание интерфейса класса. В частности, благодаря этому будет легче разобраться в условиях возникновения ошибок и в граничных условиях. Это еще одно преимущество создания тестов во время написания кода или даже раньше.

По мере добавления тестирующих классов их можно объединять в наборы, помещая в создаваемые для этой цели другие тестирующие классы. Это делается легко, потому что комплект тестов может содержать в себе другие комплекты тестов. Таким образом, можно создать главный тестирующий класс:

```
class MasterTester extends TestCase {
    public static void main(String[] args) {
        junit.textui.TestRunner.run(suite());
    }
    public static Test suite() {
        TestSuite result = new TestSuite();
        result.addTest(new TestSuite(FileReaderTester.class));
        result.addTest(new TestSuite(FileWriterTester.class));
        // И так далее...
        return result;
    }
}
```

Когда нужно остановиться? Наверняка вы не раз слышали, что тестирование не доказывает отсутствие ошибок в программе. Это верно, но это не мешает тестированию повысить скорость работы программиста. Было предложено немало правил, призванных гарантировать тестируемость всех мыслимых

комбинаций данных. Ознакомиться с ними полезно, но не стоит принимать их слишком всерьез. Они могут уменьшить выгоду, приносимую тестированием, а кроме того, имеется опасность, что попытка написать слишком много тестов оставит в душе программиста такой след, что в итоге он вообще перестанет писать тесты. Необходимо сконцентрироваться на наиболее подозрительных местах. Изучите код и определите, где он становится сложным. Изучите функцию и установите области, где могут возникнуть ошибки. Тесты не выявят все ошибки, но в процессе рефакторинга помогут лучше понять программу и благодаря этому найти больше ошибок. Я уже говорил, что всегда начинаю рефакторинг с создания комплекта тестов; по мере продвижения вперед я обязательно пополняю этот комплект.

Опасения по поводу того, что тестирование не выявит все ошибки, не должно мешать писать тесты, которые выявят большинство ошибок.

Одна из проблем, связанных с объектами, заключается в том, что наследование и полиморфизм затрудняют тестирование, поскольку приходится проверять слишком много разных комбинаций. Если у вас есть три сотрудничающих абстрактных класса и каждый имеет по три подкласса, то получается девять альтернативных классов и двадцать семь возможных комбинаций взаимодействий. Я не стараюсь любой ценой проверять все возможные комбинации взаимодействия, но пытаюсь проверить хотя бы все варианты классов. Если варианты классов достаточно независимы один от другого, я обычно не пытаюсь перепробовать все комбинации. Всегда есть риск что-то пропустить, но лучше потратить разумное время на то, чтобы выявить большинство ошибок, чем потратить вечность, пытаясь найти их все.

Разница между кодом тестов и производственным кодом заключается в том, что в том, чтобы копировать и редактировать тестовый код, нет ничего ненормального. При работе с большими комбинациями взаимодействий и альтернативами классов я часто это делаю — это позволяет мне быстро создавать тесты (к которым затем также можно применить рефакторинг).

Надеюсь, в этой главе вы получили неплохое представление о том, как пишутся тесты. Можно было бы рассказывать на эту тему значительно больше, но это отвлекло бы нас от главной задачи — рефакторинга. Создайте хороший детектор ошибок и запускайте его почаще. Это — прекрасный инструмент для любых видов разработок и необходимое предварительное условие для проведения рефакторинга.

Глава 5

На пути к каталогу рефакторингов

В главах 5–12 представлен исходный каталог методов рефакторинга. Они написаны на основе моего опыта рефакторинга за несколько последних лет. Этот каталог не является исчерпывающим или бесспорным, он должен всего лишь обеспечить надежную стартовую точку для выполнения рефакторинга пользователем.

Формат описания рефакторинга

При описании рефакторингов в этой и других главах соблюдается стандартный формат. Описание каждого метода состоит из пяти частей, как показано ниже.

- Первым идет **название**. Оно играет роль при создания списка методов рефакторинга и используется повсюду в книге.
- За названием следуют **краткая сводка** ситуаций, в которых применим данный метод, и краткие сведения о том, что именно он делает. Это позволяет быстрее находить необходимый метод.
- **Мотивация** описывает, почему следует пользоваться этим методом рефакторинга и когда не следует его применять.
- **Техника** подробно, шаг за шагом, описывает, как применять данный рефакторинг.
- **Примеры** содержат иллюстрацию (очень простого) применения метода, демонстрирующую его действие.

Краткая сводка содержит формулировку проблемы, решение которой обеспечивает данный метод, а также краткое описание его действий и набросок простого примера, показывающего ситуацию до и после выполнения рефакторинга. Иногда я использую здесь код, а иногда — диаграмму на унифицированном языке моделирования (UML), в зависимости от того, что лучше передает суть данного метода. (Все диаграммы UML в этой книге используют реализацию, описанную

в книге [8].) Если вы уже встречались с данным рефакторингом, достаточно бегло взглянуть на пример, чтобы получить хорошее представление о рефакторинге. Если же нет, то вам следует проработать пример получше, чтобы хорошо понять смысл рефакторинга.

Описание техники основано на моих заметках о том, как выполнять данный рефакторинг. Они необходимы мне после того, как я некоторое время не пользовался данным рефакторингом, чтобы вспомнить его. Поэтому данный раздел, как правило, краток и не поясняет, почему следует выполнять те или иные шаги. Более подробные пояснения можно найти в примере. Таким образом, техника представляет собой просто краткие заметки, к которым можно обратиться, когда данный рефакторинг вам знаком, но следует освежить память (так, по крайней мере, использую их я). При проведении рефакторинга впервые вам, вероятно, потребуется полностью прочесть пример.

Я описывал технику таким образом, чтобы каждый шаг рефакторинга был как можно мельче. Особое внимание при проведении рефакторинга уделяется вопросам осторожности, поэтому рефакторинг следует выполнять маленькими шажками, проводя после каждого из них тестирование. На практике я обычно выполняю более крупные шаги, чем описано здесь; столкнувшись с ошибкой, я отступаю назад и двигаюсь более мелкими шажками. При описании шагов можно найти ряд ссылок на особые случаи. Таким образом, описание шагов выступает также в качестве контрольного списка (зачастую я и сам забываю об этих вещах).

Примеры просты до смешного. Их задача — помочь разъяснению рефакторинга, но при этом как можно меньше отвлекать читателя на не относящиеся к делу подробности (надеюсь, что по этой причине их упрощенность простительна; они, несомненно, не могут служить примерами добротного проектирования бизнес-объектов). Я уверен, что, разобравшись, вы сможете применить рефакторинг к своим более сложным исходным текстам. Для некоторых совсем уж простых методов рефакторинга примеры отсутствуют, поскольку вряд ли от них могло бы быть много пользы.

Не забывайте, что эти примеры включены для иллюстрации только одного обсуждаемого метода рефакторинга. В большинстве случаев в получаемом в результате рефакторинга коде сохраняются проблемы, для решения которых требуется применение других методов рефакторинга. В ряде случаев, когда методы рефакторинга часто комбинируются один с другим, я переношу примеры из одного рефакторинга в другой. При этом чаще всего я оставляю код таким, каким он получается после первого рефакторинга. Это сделано для того, чтобы каждый рефакторинг был автономным (ведь каталог рефакторингов, в первую очередь, должен служить справочником).

Не следует принимать эти примеры как советы о том, как писать тот или иной класс. Примеры приведены только для иллюстрации рефакторинга и ни для чего более. Например, обратите внимание, что в примерах для представления денежных величин используется тип `double`. Это сделано для упрощения примеров, поскольку конкретное представление не играет роли при проведении рефакторинга. На самом же деле я настоятельно не рекомендую прибегать к типу `double` для денежных величин в коммерческих программах. Для решения этой задачи я пользуюсь проектным шаблоном "Количество" [7].

Во время написания этой книги в коммерческих проектах чаще всего применялась версия языка программирования Java 1.1, поэтому большинство примеров ориентировано на нее; это особенно заметно в случае использования коллекций. Когда я завершал эту книгу, более широко доступной стала версия Java 2. Тем не менее я не вижу необходимости переделывать все примеры, потому что с точки зрения рефакторинга коллекции имеют второстепенное значение. Но все же некоторые методы рефакторинга, такие как "Инкапсуляция коллекции" (с. 226), в Java 2 осуществляются иначе. В таких (довольно редких) случаях я излагаю обе версии — как для Java 2, так и для Java 1.1.

Я использую **`полужирный моноширинный`** шрифт для выделения изменений кода, если они окружены кодом, оставшимся неизмененным, и могут быть не замечены читателем. Однако полужирным шрифтом выделяются далеко не все внесенные изменения, поскольку неумеренное применение этого шрифта помешало бы восприятию материала.

Поиск ссылок

Во многих методах рефакторинга требуется найти все ссылки на метод, поле или класс. Привлекайте для этой работы компьютер. С его помощью риск пропустить ссылку снижается, а поиск выполняется значительно быстрее, чем при обычном просмотре кода.

В большинстве языков компьютерные программы представляют собой просто текстовые файлы, поэтому лучше всего прибегнуть к подходящему виду текстового поиска. Во многих средах программирования можно осуществлять поиск как в одном файле, так и в группе файлов одновременно.

Не прибегайте к автоматическому поиску и замене. Для каждой ссылки убедитесь, что она и в самом деле указывает на то, что вы хотите заменить. Можно воспользоваться шаблоном поиска, но я всегда мысленно проверяю правильность выполняемой замены. Если в разных классах есть одноименные методы или в одном классе — методы с разными сигнатурами, вероятность ошибки оказывается слишком большой.

В сильно типизированных языках помощь в поиске может оказать компилятор. Часто можно просто удалить прежнюю функцию и позволить компилятору искать висячие ссылки. Этот подход хорош тем, что компилятор найдет их все. Но и с этим подходом связаны некоторые проблемы.

Прежде всего, компилятор запутается, если некоторая функция определена в иерархии наследования несколько раз. Особенно это касается поиска неоднократно перекрывавшихся методов. Работая с иерархией, используйте текстовый поиск, чтобы узнать, не определен ли интересующий вас метод и в других классах.

Вторая проблема заключается в том, что компилятор может оказаться слишком медлительным для выполнения этой работы. В таком случае сначала обратитесь к текстовому поиску; компилятор будет использоваться для перепроверки полученных вами результатов. Этот способ действует, только если надо удалить метод. Однако зачастую нужно разобраться, как он используется, и решить, что с ним делать дальше. В таких случаях применяйте текстовый поиск.

Третья проблема связана с тем, что компилятор не может перехватить использование API рефлексии Java. Кстати, это одна из причин, по которым применять рефлексию следует с особой осторожностью. Если в системе используется API рефлексии Java, необходимы текстовый поиск и дополнительное усиление тестирования. В некоторых местах я предлагаю компилировать без тестирования — обычно там, где ошибки будут перехвачены компилятором. Но в случае применения рефлексии Java этот способ не работает, так что все компиляции должны тестироваться.

Некоторые среды Java, в частности VisualAge от IBM, предоставляют возможности, аналогичные Refactoring Browser для Smalltalk. Здесь вместо текстового поиска для нахождения ссылок можно использовать меню. В таких средах код хранится не в текстовых файлах, а в базе данных в оперативной памяти. Если у вас есть возможность такой работы, приучите себя к ней: часто указанные способности сред превосходят по своим возможностям текстовый поиск.

Насколько зрелы предлагаемые методы рефакторинга

Любой автор, пишущий на технические темы, вынужден выбирать момент для публикации своих идей. Чем раньше они будут опубликованы, тем скорее ими смогут воспользоваться другие. Но если опубликовать слишком рано, пока идеи еще “полусырые”, они могут не сработать и даже послужить источником проблем для тех, кто попытается ими воспользоваться.

Основы технологии рефакторинга, т.е. внесение небольших изменений и частое тестирование, проверена на протяжении многих лет, особенно в сообществе программистов Smalltalk. Поэтому я уверен, что фундаментальная идея рефакторинга прочно и надежно устоялась.

Методы же рефакторинга, описываемые в данной книге, представляют собой мои заметки по поводу тех разновидностей рефакторинга, которые применялись мною на практике. Но есть разница между применением рефакторинга и сведением его к чисто механической последовательности некоторых шагов. Например, могут встречаться проблемы, возникающие лишь при весьма специфических обстоятельствах. Не могу сказать, что те, кто выполнял описанные здесь шаги, часто встречались с такого рода проблемами. Применять рефакторинг бездумно не следует. Помните, что, используя рефакторинг, необходимо адаптировать его к своим условиям. Если вы столкнетесь с интересной проблемой, напишите мне по электронной почте, и я постараюсь поставить других в известность об этой ситуации.

Еще один аспект рефакторинга, о котором необходимо помнить, — методы рефакторинга описываются в предположении, что все программное обеспечение выполняется одним потоком. Со временем, надеюсь, будут описаны методы рефакторинга для параллельного и распределенного программирования. Эти методы будут иными. Например, в программах, выполняемых последовательно, не стоит беспокоиться о частоте вызова метода: вызов метода обходится дешево. Однако в распределенных программах межпроцессные вызовы следует выполнять как можно реже. Для таких видов программирования есть свои разновидности рефакторинга, которые выходят за рамки данной книги.

Многие методы рефакторинга, например “Замена кода типа состоянием/стратегией” (с. 245) или “Формирование шаблонного метода” (с. 361), состоят во введении в систему проектных шаблонов. В книге [9] сказано, что “проектные шаблоны... предоставляют целевые объекты для проведения рефакторинга”. Существует естественная связь между проектными шаблонами и рефакторингом. Шаблоны представляют собой цели; рефакторинг же дает методы их достижения. В данной книге нет методов рефакторинга для всех известных проектных шаблонов, даже для всех шаблонов из упомянутой книги. В этом отношении приводимый в данной книге каталог не полон. Надеюсь, что когда-нибудь этот пробел будет исправлен.

Применяя методы рефакторинга, помните, что они представляют собой лишь начало пути. Вы наверняка найдете, что их недостаточно. Я публикую их прямо сейчас только потому, что считаю их полезными, несмотря на их несовершенство. Надеюсь, что они послужат неплохой отправной точкой и повысят ваши возможности эффективного рефакторинга, т.е. послужат вам тем же, чем они служат для меня.

Думаю, что по мере приобретения опыта применения рефакторинга вы начнете разрабатывать собственные методы. Надеюсь, приведенные примеры побудят вас к этому и дадут начальные представления о том, как это делается. Я отдаю себе отчет в том, что методов рефакторинга существует гораздо больше, чем описано мною, а потому я буду рад сообщениям о ставших известными вам новых видах рефакторинга.

Глава 6

Составление методов

Большая часть моих рефакторингов состоит в составлении методов, корректно оформляющих код. Проблемы по большей части обусловлены наличием слишком длинных методов, зачастую с большим количеством информации, спрятанной за сложной логикой, которую они обычно включают. Основным методом рефакторинга в этой ситуации является рефакторинг "Извлечение метода" (с. 132), в результате которого фрагмент кода становится отдельным методом. Рефакторинг "Встраивание метода" (с. 139), по сути, представляет собой противоположную процедуру, заменяющую вызов метода кодом, содержащимся в его теле. Этот рефакторинг обычно требуется после проведения нескольких извлечений, когда я вижу, что некоторые из полученных методов больше не выполняют адекватную работу, или если необходима реорганизация способа разделения кода на методы.

Главная проблема рефакторинга "Извлечение метода" (с. 132) связана с обработкой локальных переменных и, прежде всего, с наличием временных переменных. Работая над методом, я обычно с помощью рефакторинга "Замена временной переменной запросом" (с. 141) стараюсь избавиться от как можно большего количества временных переменных. Если временная переменная используется для нескольких целей, я сначала применяю рефакторинг "Расщепление временной переменной" (с. 149), чтобы облегчить последующую ее замену.

Однако иногда временные переменные оказывается слишком сложно заменить. Тогда требуется прибегнуть к рефакторингу "Замена метода объектом методов" (с. 155). Он позволяет разбить на части даже самый запутанный метод, но его цена — добавление нового класса для выполнения задачи.

С параметрами обычно проблем меньше, чем с временными переменными, — если только им не присваиваются значения. В противном случае нужно выполнить рефакторинг "Удаление присваиваний параметрам" (с. 152).

Когда метод разделен, становится легче понять, как он работает. Иногда находится возможность улучшить алгоритм, сделать его более понятным. Тогда я применяю рефакторинг "Замена алгоритма" (с. 159).

Извлечение метода (Extract Method)

Имеется фрагмент кода, который можно сгруппировать.

Преобразуем фрагмент кода в метод, название которого поясняет его назначение.

```
void printOwing(double amount) {
    printBanner();

    // Вывод деталей
    System.out.println("name: " + _name);
    System.out.println("amount " + amount);
}
```

```
void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}

void printDetails(double amount) {
    System.out.println("name:" + _name);
    System.out.println("amount" + amount);
}
```

Мотивация

Это один из наиболее часто выполняемых мною разновидностей рефакторинга. Если я нахожу метод, кажущийся слишком длинным, или код, который для пояснения его предназначения требует комментариев, я преобразую этот фрагмент кода в отдельный метод.

Я предпочитаю короткие методы с осмысленными именами, и тому есть объяснения. Во-первых, вероятность использования небольшого метода другими методами выше, чем в случае большого громоздкого метода. Во-вторых, методы более высокого уровня начинают выглядеть как набор комментариев. Кроме того, при “мелкой грануляции” упрощается замена методов.

Если раньше вы пользовались большими методами, вам понадобится некоторое время, чтобы привыкнуть к малым методам. Особенно полезны малые методы, обладающие хорошими самодокументируемыми именами. Иногда меня спрашивают, какой длины должно быть имя метода. Мне кажется, важна не длина, а семантическое расстояние между именем и телом метода. Если выделение части кода в отдельный метод делает код более понятным, выполняйте его, даже если имя метода окажется длиннее, чем выделенный код.

Техника

- Создайте новый метод и дайте ему имя, соответствующее предназначению метода (тому, что он делает, а не как).

  ⇛ *Когда предполагаемый для выделения код очень прост (например, выводит отдельное сообщение или вызывает одну функцию), то его стоит извлекать и выделять в отдельный метод, только если имя нового метода лучше раскрывает назначение кода. Если вы не можете придумать содержательное имя, не извлекайте такой код.*

- Скопируйте код, подлежащий извлечению, из исходного метода в создаваемый.
- Найдите в извлеченном коде все обращения к переменным, имеющим локальную область видимости в исходном методе. Ими являются локальные переменные и параметры метода.
- Найдите временные переменные, которые используются только внутри этого выделенного кода. Если такие переменные есть, объявите их как временные переменные во вновь создаваемом методе.
- Проверьте, не модифицирует ли выделенный код какие-либо из упомянутых переменных с локальной областью видимости. Если модифицируется только одна переменная, попробуйте превратить извлекаемый код в вызов другого метода, результат которого присваивается этой переменной. Если это затруднительно или таких переменных несколько, то извлечь метод как он есть не представляется возможным. В этом случае попробуйте сначала выполнить рефакторинг “Расщепление временной переменной” (с. 149), а затем снова выделить метод. Временные переменные можно устранить с помощью рефакторинга “Замена временной переменной запросом” (с. 141) (см. обсуждение в примерах).
- Передайте переменные с локальной областью видимости, чтение которых осуществляется в извлекаемом коде, во вновь создаваемый метод в качестве параметров.
- По завершении работы со всеми локальными переменными выполните компиляцию.
- Замените в исходном методе извлеченный код вызовом вновь созданного метода.

  ⇛ *Если в созданный метод перемещены какие-либо временные переменные, найдите их объявления вне извлеченного кода. Если таковые имеются, их можно удалять.*

- Выполните компиляцию и тестирование.

Пример: локальных переменных нет

В простейшем случае рефакторинг “Извлечение метода” выполняется тривиально. Рассмотрим следующий метод:

```
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;
    // Вывод баннера
    System.out.println("****************************");
    System.out.println("*** Задолженность клиента ***");
    System.out.println("****************************");

    // Расчет задолженности
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    // Вывод детальной информации
    System.out.println("Имя: " + _name);
    System.out.println("Количество " + outstanding);
}
```

Код, выводящий баннер, легко извлечь с помощью копирования и вставки и вызова вновь созданного метода:

```
void printOwing() {

    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // Расчет задолженности
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    // Вывод детальной информации
    System.out.println("Имя: " + _name);
    System.out.println("Количество " + outstanding);
}

void printBanner() {
    // Вывод баннера
    System.out.println("****************************");
    System.out.println("*** Задолженность клиента ***");
    System.out.println("****************************");
}
```

Пример: локальные переменные есть

В чем проблема в этом случае? В наличии локальных переменных: параметров, передаваемых в исходный метод, и временных переменных, объявленных в исходном методе. Локальные переменные находятся в области видимости исходного метода, поэтому, когда я использую данный рефакторинг, мне приходится выполнять дополнительную работу. В некоторых случаях они могут просто не дать мне возможности выполнить рефакторинг.

Простейший случай — когда происходит только чтение значений локальных переменных, но не их изменение. В этом случае можно просто передавать их в качестве параметров. Пусть, например, у нас есть следующий метод:

```
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // Расчет задолженности
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    // Вывод детальной информации
    System.out.println("Имя:" + _name);
    System.out.println("Количество " + outstanding);
}
```

Вывод детальной информации можно выделить в метод с одним параметром:

```
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // Расчет задолженности
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    printDetails(outstanding);
}

void printDetails(double outstanding) {
    System.out.println("Имя:" + _name);
    System.out.println("Количество " + outstanding);
}
```

Такой способ может быть использован для любого количества локальных переменных.

Этот же способ работает и если локальная переменная является объектом, и вызывается метод, модифицирующий переменную. В таком случае также можно просто передать объект в качестве параметра. Другие действия требуются только тогда, когда действительно выполняется присваивание значения локальной переменной.

Пример: присваивание локальной переменной

Наибольшие трудности возникают, когда нужно присвоить локальным переменным новые значения. Сейчас мы говорим только о временных переменных. Если вы увидите присваивание параметру, то должны сразу же применить рефакторинг “Удаление присваиваний параметрам” (с. 152).

Есть два варианта присваивания временным переменным. В простейшем случае временная переменная используется лишь внутри извлекаемого кода. Тогда временную переменную можно переместить в этот извлекаемый код. Другой же случай — это использование переменной и вне извлекаемого кода. В таком случае необходимо обеспечить возврат извлекаемым кодом измененного значения переменной. Проиллюстрирую сказанное на следующем методе:

```
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // Расчет задолженности
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    printDetails(outstanding);
}
```

Теперь я извлекаю расчет:

```
void printOwing() {
    printBanner();
    double outstanding = getOutstanding();
    printDetails(outstanding);
}

double getOutstanding() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;
```

```
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    return outstanding;
}
```

Переменная перечисления используется только в извлекаемом коде, поэтому ее можно перенести в новый метод. Переменная `outstanding` используется в обоих местах, поэтому выделенный метод должен ее возвращать. После компиляции и тестирования, следующих за выделением, я переименовываю возвращаемое значение:

```
double getOutstanding() {
    Enumeration e = _orders.elements();
    double result = 0.0;

    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        result += each.getAmount();
    }

    return result;
}
```

Переменная `outstanding` в нашем случае инициализируется совершенно очевидным начальным значением, поэтому ее можно просто инициализировать в выделяемом методе. Если с переменной выполняются какие-то иные сложные действия, следует передать ее последнее значение в качестве параметра. Соответствующий исходный текст может иметь следующий вид:

```
void printOwing(double previousAmount) {

    Enumeration e = _orders.elements();
    double outstanding = previousAmount * 1.2;

    printBanner();

    // Расчет задолженности
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    printDetails(outstanding);
}
```

Извлечение метода при этом будет выглядеть следующим образом:

```
void printOwing(double previousAmount) {
    double outstanding = previousAmount * 1.2;
    printBanner();
    outstanding = getOutstanding(outstanding);
    printDetails(outstanding);
}

double getOutstanding(double initialValue) {
    double result = initialValue;
    Enumeration e = _orders.elements();

    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        result += each.getAmount();
    }

    return result;
}
```

После компиляции и тестирования я делаю инициализацию переменной `outstanding` более понятной:

```
void printOwing(double previousAmount) {
    printBanner();
    double outstanding = getOutstanding(previousAmount * 1.2);
    printDetails(outstanding);
}
```

Вы можете спросить "А что делать, если надо вернуть не одну, а несколько переменных?"

Есть ряд способов сделать это. Обычно лучше всего выбрать для извлечения другой код. Лично я предпочитаю, чтобы метод возвращал только одно значение, поэтому я попытался бы организовать возврат разных значений разными методами. (Если язык, с которым вы работаете, допускает возврат значений через параметры, этим можно воспользоваться. Но я предпочитаю работать с одиночными возвращаемыми значениями, насколько это возможно.)

Часто временных переменных оказывается настолько много, что извлекать методы становится очень трудно. Тогда я начинаю с того, что пытаюсь сократить количество временных переменных с помощью рефакторинга "Замена временной переменной запросом" (с. 141). Если он не помогает, я пробую рефакторинг "Замена метода объектом методов" (с. 155), которому безразлично количество временных переменных и то, что вы с ними делаете.

Встраивание метода (Inline Method)

Тело метода столь же понятно, как и его название.

Поместите тело метода в код, который его вызывает, и удалите метод.

```
int getRating() {
    return (moreThanFiveLateDeliveries()) ? 2 : 1;
}
boolean moreThanFiveLateDeliveries() {
    return _numberOfLateDeliveries > 5;
}
```

```
int getRating() {
    return (_numberOfLateDeliveries > 5) ? 2 : 1;
}
```

Мотивация

Данная книга учит использовать короткие методы с названиями, отражающими их предназначение, что дает более понятный и простой код. Но иногда встречаются методы, тела которых так же очевидны, как и названия, — либо изначально, либо после рефакторинга. В этом случае можно избавиться от метода. Косвенность может быть полезной, но излишняя косвенность мешает.

Еще одна ситуация, в которой можно применить указанный рефакторинг, — наличие группы методов, представляющейся плохо разделенной. Их все можно встроить в один большой метод, а затем выполнить извлечение методов другим способом. Кент Бек полагает полезным выполнение указанной операции перед применением рефакторинга “Замена метода объектом методов” (с. 155). Вы встраиваете различные вызовы, выполняемые методом, поведение которого вы хотите получить в объекте метода. Но проще переместить один метод, чем перемещать метод вместе с вызываемыми им методами.

Я обычно использую рефакторинг “Встраивание метода”, когда нахожу в коде слишком много косвенности, и выясняю, что каждый метод просто выполняет делегирование другому методу, и во всем этом делегировании очень легко заблудиться. Косвенность оправданна далеко не всегда и далеко не в любых количествах. Встраивание позволяет собрать полезное и убрать ненужное.

Техника

- Убедитесь, что метод не является полиморфным.
  ⇛ *Избегайте встраивания, если метод перекрыт в подклассах — они не смогут перекрыть отсутствующий метод.*
- Найдите все вызовы метода.
- Замените каждый вызов телом метода.
- Выполните компиляцию и тестирование.
- Удалите объявление метода.

Приведенное описание рефакторинга создает впечатление простоты, но в общем случае это не так. Я мог бы посвятить многие страницы обработке рекурсии, множественным точкам возврата, встраиванию в другие объекты при отсутствии функций доступа и т.п. Я не делаю этого потому, что при наличии таких сложностей данный рефакторинг лучше не применять.

Встраивание временной переменной (Inline Temp)

Имеется временная переменная, которой один раз присваивается значение простого выражения, и эта переменная мешает проведению других рефакторингов.

Замените все ссылки на переменную выражением.

```
double basePrice = anOrder.basePrice();
returrn(basePrice > 1000)
```

```
return (anOrder.basePrice() > 1000)
```

Мотивация

Чаще всего встраивание переменной производится в ходе рефакторинга “Замена временной переменной запросом” (с. 141), поэтому настоящая мотивация имеется в его описании. Встраивание временной переменной выполняется отдельно только тогда, когда обнаруживается временная переменная, которой присваивается значение, возвращаемое вызовом метода. Часто эта переменная не несет никаких неприятностей, и можно оставить ее в покое. Но если она мешает другим рефакторингам, например рефакторингу “Извлечение метода” (с. 132), ее лучше встроить.

Техника

- Убедитесь, что в правой части присваивания нет никаких побочных действий.
- Объявите временную переменную с ключевым словом `final`, если это еще не сделано, и скомпилируйте код.
  ⇒ *Так вы убедитесь, что значение этой переменной присваивается ей ровно один раз.*
- Найдите все ссылки на временную переменную и замените их правой частью присваивания.
- Выполняйте компиляцию и тестирование после каждого внесения изменений.
- Удалите объявление и присваивание данной переменной.
- Выполните компиляцию и тестирование.

Замена временной переменной запросом (Replace Temp with Query)

Временная переменная используется для хранения результата выражения.

Преобразуйте выражение в метод. Замените все ссылки на временную переменную новым методом, который после этого может быть использован и в других методах.

```
double basePrice = _quantity * _itemPrice;

if (basePrice > 1000)
    return basePrice * 0.95;
else
    return basePrice * 0.98;
```

```
if (basePrice() > 1000)
    return basePrice() * 0.95;
else
    return basePrice() * 0.98;

...
double basePrice() {
    return _quantity * _itemPrice;
}
```

Мотивация

Проблема в том, что эти переменные являются временными и локальными. Будучи видны лишь в контексте метода, в котором используются, они ведут к увеличению размеров методов, потому что только так можно их достичь. После замены временной переменной методом запроса получить содержащуюся в ней информацию может любой метод класса. Это помогает получению качественного кода класса.

Данный рефакторинг часто является необходимым шагом перед проведением рефакторинга "Извлечение метода" (с. 132). Локальные переменные затрудняют извлечение, поэтому замените как можно больше переменных вызовами методов.

Простыми случаями данного рефакторинга являются случаи, когда присваивание временным переменным осуществляется однократно, и случаи, когда выражение, участвующее в присваивании, не имеет побочных действий. Прочие ситуации сложнее, но, тем не менее, разрешимы. Облегчить положение может предварительное выполнение рефакторингов "Расщепление временной переменной" (с. 149) и "Разделение запроса и модификатора" (с. 296). Если временная переменная используется для накопления результата (например, при суммировании в цикле), необходимо скопировать эту логику в методе запроса.

Техника

Действия в простом случае.

- Найдите временную переменную, присваивание которой выполняется один раз.
  - ⇒ *Если значение временной переменной присваивается несколько раз, попробуйте воспользоваться рефакторингом "Расщепление временной переменной" (с. 149).*
- Объявите временную переменную с ключевым словом `final`.
- Скомпилируйте код.
  - ⇒ *Так вы убедитесь, что значение этой переменной присваивается ей ровно один раз.*
- Выделите правую часть присваивания в метод.
  - ⇒ *Сначала сделайте метод закрытым (`private`). Позднее для него может быть найдено дополнительное применение; тогда его защиту можно будет ослабить.*
  - ⇒ *Выделенный метод должен быть свободен от побочных действий, например он не должен модифицировать какие-либо объекты. Если это не так, воспользуйтесь рефакторингом "Разделение запроса и модификатора" (с. 296).*

- Выполните компиляцию и тестирование.
- Воспользуйтесь для временной переменной рефакторингом “Встраивание временной переменной” (с. 140).

Временные переменные часто используются для накопления в циклах. Цикл может быть полностью выделен в метод, что позволит убрать несколько строк отвлекающего внимание кода. Иногда в цикле суммируются несколько величин. В таком случае повторите цикл для каждой временной переменной по отдельности, чтобы иметь возможность заменить ее запросом. Цикл должен быть очень простым, так что дублирование кода не представляет никакой опасности.

Вопросы снижения производительности оставим до лучших времен. В девяти случаях из десяти они не существенны. Если же производительность важна, то она может быть повышена на этапе оптимизации. Когда код имеет ясную структуру, легче найти более мощные оптимизирующие решения, которые без рефакторинга могли бы остаться незамеченными. Если дела будут совсем плохи, всегда можно легко вернуться ко временным переменным.

Пример

Начну с простого метода:

```
double getPrice() {
    int basePrice = _quantity * _itemPrice;
    double discountFactor;

    if (basePrice > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;

    return basePrice * discountFactor;
}
```

Я намерен поочередно заменить обе временные переменные.

Хотя в данном случае все и так понятно, я могу проверить, действительно ли значения временным переменным присваиваются только один раз, объявив их с ключевым словом `final`:

```
double getPrice() {
    final int basePrice = _quantity * _itemPrice;
    final double discountFactor;

    if (basePrice > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;

    return basePrice * discountFactor;
}
```

При компиляции компилятор сообщит мне о возможных проблемах. Это первое действие, потому что при возникновении проблем на этом этапе от проведения данного рефакторинга следует воздержаться. Временные переменные заменяются поочередно. Сначала я извлекаю правую часть присваивания:

```
double getPrice() {
    final int basePrice = basePrice();
    final double discountFactor;

    if (basePrice > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;

    return basePrice * discountFactor;
}

private int basePrice() {
    return _quantity * _itemPrice;
}
```

Я выполняю компиляцию и тестирование, а затем провожу рефакторинг "Встраивание временной переменной" (с. 140). Сначала я заменяю первую ссылку на временную переменную:

```
double getPrice() {
    final int basePrice = basePrice();
    final double discountFactor;

    if (basePrice() > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;

    return basePrice * discountFactor;
}
```

После этого я выполняю компиляцию и тестирование, а затем — следующую замену. Поскольку она будет последней, удаляется и объявление временной переменной:

```
double getPrice() {
    final double discountFactor;

    if (basePrice() > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;

    return basePrice() * discountFactor;
}
```

Теперь я точно так же могу извлечь `discountFactor`:

```
double getPrice() {
    final double discountFactor = discountFactor();
    return basePrice() * discountFactor;
}

private double discountFactor() {
    if (basePrice() > 1000) return 0.95;
    else return 0.98;
}
```

Обратите внимание, что извлечь `discountFactor` без предварительной замены `basePrice` запросом было бы достаточно трудно.

В итоге я получаю следующий вид метода `getPrice`:

```
double getPrice() {
    return basePrice() * discountFactor();
}
```

Введение поясняющей переменной (Introduce Explaining Variable)

Имеется сложное выражение.

Результат выражения или его части помещается во временную переменную, имя которой поясняет его назначение.

```
if ((platform.toUpperCase().indexOf("MAC") > -1) &&
    (browser.toUpperCase().indexOf("IE") > -1) &&
    wasInitialized() && resize > 0) {
    // Некоторые действия
}
```

```
final boolean isMacOs =
    platform.toUpperCase().indexOf("MAC") > -1;
final boolean isIEBrowser =
    browser.toUpperCase().indexOf("IE") > -1;
final boolean wasResized = resize > 0;

if (isMacOs && isIEBrowser &&
    wasInitialized() && wasResized) {
    // Некоторые действия
}
```

Мотивация

Выражения могут быть очень сложными для чтения. В таких случаях имеет смысл с помощью временных переменных превратить выражение в нечто, лучше поддающееся пониманию и управлению.

Особую ценность рассматриваемый рефакторинг имеет для условной логики, когда для каждого пункта условия можно объяснить, что он означает, просто использовав временную переменную с хорошо подобранным именем. Другим примером служит длинный алгоритм, шаги которого можно разъяснить с помощью временной переменной.

Это очень распространенный вид рефакторинга, но должен признаться, что сам я применяю его лишь изредка. Почти всегда, если это возможно, я предпочитаю ему рефакторинг "Извлечение метода" (с. 132). Временная переменная полезна только в контексте одного метода. Метод же можно использовать как всюду в объекте, так и в других объектах. Однако бывают ситуации, когда локальные переменные затрудняют применение рефакторинга "Извлечение метода" (с. 132), и тогда я обращаюсь к рефакторингу "Введение поясняющей переменной" (с. 145).

Техника

- Объявите временную переменную с ключевым словом `final` и установите ее значением результат части сложного выражения.
- Замените эту часть выражения значением временной переменной.

  ⇒ *Если часть повторяется, все повторения можно заменять по одному.*
- Выполните компиляцию и тестирование.
- Повторите эти действия для других частей выражения.

Пример

Начну с простого вычисления:

```
double price() {
    // price = базовая цена - скидка + доставка
    return _quantity * _itemPrice -
           Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
           Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
```

Код несложный, но его можно сделать еще понятнее. Для начала я определю базовую цену как количество, умноженное на цену. Эту часть расчета можно превратить во временную переменную:

```
double price() {
    // price = базовая цена - скидка + доставка
    final double basePrice = _quantity * _itemPrice;
    return basePrice -
           Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
           Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
```

Количество, умноженное на цену, используется и далее, поэтому и там можно использовать введенную временную переменную:

```
double price() {
    // price = базовая цена - скидка + доставка
    final double basePrice = _quantity * _itemPrice;
    return basePrice -
           Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
           Math.min(basePrice * 0.1, 100.0);
}
```

Теперь возьмемся за скидку:

```
double price() {
    // price = базовая цена - скидка + доставка
    final double basePrice = _quantity * _itemPrice;
    final double quantityDiscount =
            Math.max(0, _quantity - 500) * _itemPrice * 0.05;
    return basePrice - quantityDiscount +
           Math.min(basePrice * 0.1, 100.0);
}
```

Наконец разберемся с доставкой. После этого можно удалить комментарий, потому сейчас в нем нет ничего такого, о чем не говорил бы сам код:

```
double price() {
    final double basePrice = _quantity * _itemPrice;
    final double quantityDiscount =
            Math.max(0, _quantity - 500) * _itemPrice * 0.05;
    final double shipping = Math.min(basePrice * 0.1, 100.0);
    return basePrice quantityDiscount + shipping;
}
```

Пример с извлечением метода

Я для данного примера все же не стал бы создавать поясняющие переменные, а прибегнул бы к рефакторингу “Извлечение метода” (с. 132). Для этого я снова начинаю с кода

```
double price() {
    // price = базовая цена - скидка + доставка
    return _quantity * _itemPrice -
           Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
           Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
```

На этот раз я извлекаю метод для базовой цены:

```
double price() {
    // price = базовая цена - скидка + доставка
    return basePrice() -
           Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
           Math.min(basePrice() * 0.1, 100.0);
}
private double basePrice() {
    return _quantity * _itemPrice;
}
```

Продолжая пошагово, я получаю следующее:

```
double price() {
    return basePrice() - quantityDiscount() + shipping();
}

private double quantityDiscount() {
    return Math.max(0, _quantity - 500) * _itemPrice * 0.05;
}

private double shipping() {
    return Math.min(basePrice() * 0.1, 100.0);
}

private double basePrice() {
    return _quantity * _itemPrice;
}
```

Я предпочитаю рефакторинг “Извлечение метода” (с. 132), поскольку в результате методы становятся доступными для любой другой части объекта, нуждающейся в них. Сначала я делаю их `private`, но всегда могу ослабить ограничения, если методы понадобятся другому объекту. По моему опыту обычно для рефакторинга “Извлечение метода” (с. 132) требуется не больше усилий, чем для рефакторинга “Введение поясняющей переменной” (с. 145).

Когда же я применяю рефакторинг “Введение поясняющей переменной”? Тогда, когда рефакторинг “Извлечение метода” (с. 132) действительно требует большего количества труда. Если я работаю с алгоритмом, использующим множество локальных переменных, применение рефакторинга “Извлечение метода” (с. 132) может оказаться непростым делом. В такой ситуации я выбираю рефакторинг

"Введение поясняющей переменной", который позволяет мне лучше понять работу алгоритма. Когда логика становится более доступной для моего понимания, я могу применить рефакторинг "Замена временной переменной запросом" (с. 141). Временная переменная также оказывается полезной, если в конечном итоге я прибегаю к рефакторингу "Замена метода объектом методов" (с. 155).

Расщепление временной переменной (Split Temporary Variable)

Имеется временная переменная, которой многократно присваивается значение, но переменная не является ни переменной цикла, ни переменной для накопления результата.

Для каждого присваивания создается отдельная временная переменная.

```
double temp = 2 * (_height + _width);
System.out.println(temp);
temp = _height * _width;
System.out.println(temp);
```

```
final double perimeter = 2 * (_height + _width);
System.out.println(perimeter);
final double area = _height * _width;
System.out.println(area);
```

Мотивация

Временные переменные создаются с различными целями. Иногда эти цели естественным образом приводят к тому, что значение временной переменной присваивается несколько раз. Переменные цикла [6] изменяются при каждой его итерации (например, `i` в `for(int i=0; i<10; i++)`. Временные переменные для накопления аккумулируют некоторое значение, получаемое при выполнении метода.

Другие временные переменные часто используются для хранения результата большого фрагмента кода, чтобы облегчить ссылки на него в будущем. Переменным такого рода значение должно присваиваться только один раз. Если значение присваивается им многократно, то это свидетельствует о выполнении переменными нескольких задач в одном методе. Все переменные, выполняющие несколько функций, должны быть заменены отдельными переменными — по одной для каждой из этих функций. Использование одной и той же переменной для решения разных задач затрудняет чтение кода.

Техника

- Измените имя временной переменной в объявлении и первом присваивании значения.

  ⇛ *Если последующие присваивания имеют вид* `i=i+выражение`, *то это — временная переменная для накопления данных, и ее расщеплять не надо. Операторы для работы с накопительными временными переменными обычно включают сложение, конкатенацию строк, вывод в поток или добавление в коллекцию.*

- Объявите новую временную переменную с ключевым словом `final`.
- Измените все ссылки на временную переменную вплоть до второго присваивания.
- Объявите временную переменную в месте второго присваивания.
- Выполните компиляцию и тестирование.
- Повторяйте эти шаги переименования и изменения ссылок до очередного присваивания.

Пример

В этом примере вычисляется расстояние, на которое перемещается объект. В стартовой позиции на него воздействует некоторая первоначальная сила. После некоторой задержки вступает в действие вторая сила, придающая дополнительное ускорение. Обычные законы движения позволяют вычислить пройденное расстояние, как показано ниже:

```
double getDistanceTravelled(int time) {
    double result;
    double acc = _primaryForce / _mass;
    int primaryTime = Math.min(time, _delay);
    result = 0.5 * acc * primaryTime * primaryTime;
    int secondaryTime = time - _delay;

    if (secondaryTime > 0) {
        double primaryVel = acc * _delay;
        acc = (_primaryForce + _secondaryForce) / _mass;
        result += primaryVel * secondaryTime
                + 0.5 * acc * secondaryTime * secondaryTime;
    }

    return result;
}
```

Маленькая неуклюжая функция. Для нас представляет интерес то, что переменной acc дважды присваивается значение. Она выполняет две функции: содержит первоначальное ускорение, вызванное первой силой, и последующее ускорение, вызванное обеими силами вместе. Я хочу ее расщепить.

Начнем с изменения имени переменной и объявления новой переменной как final. После этого будут изменены все ссылки на временную переменную, начиная с этой точки и до очередного присваивания. При следующем присваивании я объявлю прежнюю переменную:

```
double getDistanceTravelled(int time) {
    double result;
    final double primaryAcc = _primaryForce / _mass;
    int primaryTime = Math.min(time, _delay);
    result = 0.5 * primaryAcc * primaryTime * primaryTime;
    int secondaryTime = time - _delay;

    if (secondaryTime > 0) {
        double primaryVel = primaryAcc * _delay;
        double acc = (_primaryForce + _secondaryForce) / _mass;
        result += primaryVel * secondaryTime
                + 0.5 * acc * secondaryTime * secondaryTime;
    }

    return result;
}
```

Я выбрал новое имя для представления первого применения временной переменной. Я объявляю переменную как final, чтобы гарантировать однократность присваивания. После этого можно объявить первоначальную переменную в месте второго присваивания. После этих действий можно выполнить компиляцию и тестирование, — все должно корректно работать.

Последующие действия начинаются со второго присваивания временной переменной. Они окончательно удаляют первоначальное имя переменной, заменяя его новой временной переменной для второй задачи.

```
double getDistanceTravelled(int time) {
    double result;
    final double primaryAcc = _primaryForce / _mass;
    int primaryTime = Math.min(time, _delay);
    result = 0.5 * primaryAcc * primaryTime * primaryTime;
    int secondaryTime = time - _delay;

    if (secondaryTime > 0) {
        double primaryVel = primaryAcc * _delay;
        final double secondaryAcc = (_primaryForce +
                                      _secondaryForce) / _mass;
```

```
        result += primaryVel * secondaryTime + 0.5 *
                  secondaryAcc * secondaryTime * secondaryTime;
    }
    return result;
}
```

Я уверен, что вы сможете предложить для этого примера и другие методы рефакторинга. Займитесь этим на досуге — это неплохая тренировка.

Удаление присваиваний параметрам (Remove Assignments to Parameters)

Код выполняет присваивание параметру.

Воспользуйтесь вместо этого временной переменной.

```
int discount(int inputVal, int quantity, int yearToDate) {
    if (inputVal > 50) inputVal -= 2;
```

```
int discount(int inputVal, int quantity, int yearToDate) {
    int result = inputVal;
    if (inputVal > 50) result -= 2;
```

Мотивация

Начнем с определения термина "присваивание параметру". Это означает, что если в качестве значения параметра передается объект с именем foo, то после присваивания параметру foo изменится и станет ссылаться на другой объект. Никаких проблем при выполнении операций над переданным объектом нет — я делаю это постоянно. Я возражаю только против изменения foo, которое делает его ссылкой на совершенно иной объект:

```
void aMethod(Object foo) {
    foo.modifyInSomeWay(); // OK
    foo = anotherObject;   // Возможны неприятности
```

Проблема — в отсутствии ясности и смешении передачи по значению и передачи по ссылке. В Java используется передача исключительно по значению, и данное описание основано именно на этом способе.

При передаче по значению изменение параметра не отражается в вызвавшем коде. Это может смутить тех, кто привык к передаче по ссылке.

Замешательство может вызвать и тело метода. Код будет значительно понятнее, если параметр используется только для представления того, что передано, поскольку это последовательное его использование.

Не присваивайте значений параметрам, а увидев код, который это делает, примените рефакторинг “Удаление присваиваний параметрам”.

Данное правило не является обязательным для других языков, в которых применяются выходные параметры, но даже в таких языках я предпочитаю как можно реже пользоваться последними.

Техника

- Создайте для параметра временную переменную.
- Замените все обращения к параметру, осуществляемые после присваивания, обращениями ко временной переменной.
- Измените присваивание так, чтобы оно выполнялось для временной переменной.
- Выполните компиляцию и тестирование.

  ⇒ *Если осуществляется передача по ссылке, проверьте, не используется ли параметр в вызывающем методе снова. Посмотрите также, сколько передается по ссылке параметров, которым присваиваются в дальнейшем используемые значения. Попробуйте добиться, чтобы метод возвращал единственное значение. Если необходимо возвратить несколько значений, попытайтесь преобразовать группу данных в объект или создать отдельные методы.*

Пример

Начну со следующей простой программы:

```
int discount(int inputVal, int quantity, int yearToDate) {
    if (inputVal > 50) inputVal -= 2;
    if (quantity > 100) inputVal -= 1;
    if (yearToDate > 10000) inputVal -= 4;
    return inputVal;
}
```

Замена параметра временной переменной приводит к следующему коду:

```
int discount(int inputVal, int quantity, int yearToDate) {
    int result = inputVal;
    if (inputVal > 50) result -= 2;
    if (quantity > 100) result -= 1;
    if (yearToDate > 10000) result -= 4;
    return result;
}
```

Убедиться в выполнении данного соглашения можно с помощью ключевого слова final:

```
int discount(final int inputVal, final int quantity,
             final int yearToDate) {
    int result = inputVal;
    if (inputVal > 50) result -= 2;
    if (quantity > 100) result -= 1;
    if (yearToDate > 10000) result -= 4;
    return result;
}
```

Откровенно говоря, я редко пользуюсь ключевым словом final, поскольку оно мало дает для улучшения понятности коротких методов. Я прибегаю к нему в длинных методах, чтобы обнаружить, не изменяется ли где-нибудь интересующий меня параметр.

Передача по значению в Java

Передача по значению, применяемая в Java, часто приводит к путанице. Java строго придерживается передачи по значению, поэтому программа

```
class Param {
    public static void main(String[] args) {
        int x = 5;
        triple(x);
        System.out.println("x после triple: " + x);
    }
    private static void triple(int arg) {
        arg = arg * 3;
        System.out.println("arg в triple: " + arg);
    }
}
```

выводит такие результаты:

```
arg в triple: 15
x после triple: 5
```

Неразбериха начинается при использовании объектов. Допустим, мы создаем и изменяем дату:

```
class Param {
    public static void main(String[] args) {
        Date d1 = new Date("1 Apr 98");
        nextDateUpdate(d1);
```

```
        System.out.println("d1 после nextDay: " + d1);
        Date d2 = new Date("1 Apr 98");
        nextDateReplace(d2);
        System.out.println("d2 после nextDay: " + d2);
    }
    private static void nextDateUpdate(Date arg) {
        arg.setDate(arg.getDate() + 1);
        System.out.println("arg в    nextDay: " + arg);
    }
    private static void nextDateReplace(Date arg) {
        arg = new Date(arg.getYear(), arg.getMonth(), arg.getDate() +
1);
        System.out.println("arg в    nextDay: " + arg);
    }
}
```

Эта программа выводит следующие результаты:

```
arg в    nextDay: Thu Apr 02 00: 00: 00 EST 1998
d1 после nextDay: Thu Apr 02 00: 00: 00 EST 1998
arg в    nextDay: Thu Apr 02 00: 00: 00 EST 1998
d2 после nextDay: Wed Apr 01 00: 00: 00 EST 1998
```

Суть в том, что ссылка на объект передается по значению. Это позволяет модифицировать объект, но присваивание параметру другого объекта не влияет на первый.

В Java 1.1 и выше можно пометить параметр как `final`, что препятствует выполнению присваивания параметру, но сохраняет возможность модификации объекта, на который ссылается переменная. Я всегда рассматриваю параметры как константы, но редко помечаю их таковыми в списке параметров.

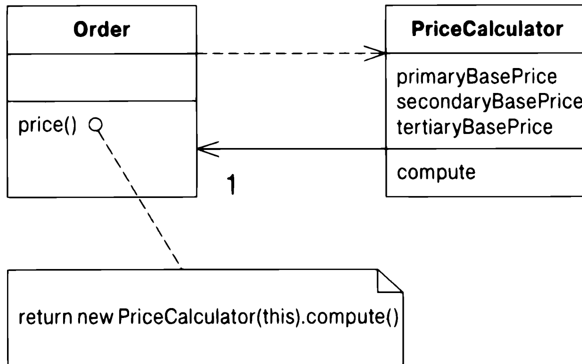
Замена метода объектом методов (Replace Method with Method Object)

Имеется длинный метод, локальные переменные в котором используются так, что это не позволяет применить рефакторинг “Извлечение метода”.

Преобразуйте метод в отдельный объект так, чтобы локальные переменные стали его полями. После этого можно разложить метод на несколько методов того же объекта.

```
class Order...
    double price() {
        double primaryBasePrice;
```

```
    double secondaryBasePrice;
    double tertiaryBasePrice;
    // Длинные вычисления;
    ...
}
```



Мотивация

На всем протяжении книги я подчеркиваю прелесть небольших методов. Разделив большой код на отдельные методы, его можно сделать значительно более понятным.

Декомпозицию метода затрудняет наличие локальных переменных. При большом их количестве декомпозиция может оказаться сложной задачей. И хотя рефакторинг "Замена временной переменной запросом" (с. 141) может облегчить ее решение, иногда и его оказывается недостаточно. В этом случае нужно порыться в ящике с инструментами и поискать в нем *объект методов* (method object) [6].

Рассматриваемый рефакторинг "Замена метода объектом методов" превращает локальные переменные в поля объекта методов. Затем к этому объекту применяется рефакторинг "Извлечение метода" (с. 132), создающий новые методы, на которые разделяется исходный метод.

Техника

Бессовестно заимствована из книги [6].

- Создайте новый класс и назовите его так же, как метод.
- Создайте в новом классе поле с модификатором `final` для объекта, который содержит исходный метод (исходный объект) и поля для всех временных переменных и параметров метода.

- Создайте для нового класса конструктор, принимающий исходный объект и все параметры.
- Создайте в новом классе метод с именем compute (вычислить).
- Скопируйте в compute тело исходного метода. Для вызовов методов исходного объекта используйте поле исходного объекта.
- Выполните компиляцию.
- Замените старый метод другим, который создает новый объект и вызывает его метод compute.

Теперь начинается самое интересное. Поскольку все локальные переменные стали полями, можно беспрепятственно разложить метод, при этом не нуждаясь в передаче каких-либо параметров.

Пример

Для хорошего примера потребовалась бы целая глава, поэтому я продемонстрирую этот рефакторинг на методе, для которого он не нужен. (Не стоит спрашивать о логике этого метода — она была придумана по ходу дела.)

```
Class Account
    int gamma(int inputVal, int quantity, int yearToDate) {
        int importantValue1 = (inputVal * quantity) + delta();
        int importantValue2 = (inputVal * yearToDate) + 100;
        if ((yearToDate - importantValue1) > 100)
            importantValue2 -= 20;
        int importantValue3 = importantValue2 * 7;
        // И так далее...
        return importantValue3 - 2 * importantValue1;
    }
```

Чтобы превратить этот метод в объект, я сначала объявляю новый класс. В нем создаются поле с модификатором final для исходного объекта и поля для каждого параметра и каждой временной переменной в методе.

```
class Gamma...
    private final Account _account;
    private int inputVal;
    private int quantity;
    private int yearToDate;
    private int importantValue1;
    private int importantValue2;
    private int importantValue3;
```

Обычно я использую в именах полей символ подчеркивания в качестве префикса, но сейчас я оставляю их такими, какие они есть.

Добавляю конструктор:

```
Gamma(Account source, int inputValArg, int quantityArg,
      int yearToDateArg) {
    _account = source;
    inputVal = inputValArg;
    quantity = quantityArg;
    yearToDate = yearToDateArg;
}
```

Теперь можно переместить исходный метод. При этом я должен модифицировать все вызовы так, чтобы они выполнялись с использованием поля _account:

```
int compute() {
    importantValue1 = (inputVal * quantity) + _account.delta();
    importantValue2 = (inputVal * yearToDate) + 100;
    if ((yearToDate - importantValue1) > 100)
        importantValue2 -= 20;
    int importantValue3 = importantValue2 * 7;
    // И так далее...
    return importantValue3 - 2 * importantValue1;
}
```

После этого старый метод изменяется таким образом, чтобы делегировать работу объекту методов:

```
int gamma(int inputVal, int quantity, int yearToDate) {
    return new Gamma(this, inputVal, quantity, yearToDate).compute();
}
```

Вот, в сущности, и весь рефакторинг. Его преимущество в том, что теперь можно легко применить рефакторинг "Извлечение метода" (с. 132) к методу compute, не беспокоясь о передаче аргументов:

```
int compute() {
    importantValue1 = (inputVal * quantity) + _account.delta();
    importantValue2 = (inputVal * yearToDate) + 100;
    importantThing();
    int importantValue3 = importantValue2 * 7;
    // И так далее...
    return importantValue3 - 2 * importantValue1;
}

void importantThing() {
    if ((yearToDate - importantValue1) > 100)
        importantValue2 -= 20;
}
```

Замена алгоритма (Substitute Algorithm)

Желательно заменить алгоритм более понятным.
Замените тело метода новым алгоритмом.

```
String foundPerson(String[] people) {
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals("Don")) {
            return "Don";
        }
        if (people[i].equals("John")) {
            return "John";
        }
        if (people[i].equals("Kent")) {
            return "Kent";
        }
    }
    return "";
}
```

```
String foundPerson(String[] people) {
    List candidates =
        Arrays.asList(new String[]
               {"Don", "John", "Kent"});
    for (int i = 0; i < people.length; i++)
        if (candidates.contains(people[i]))
            return people[i];
    return "";
}
```

Мотивация

Обычно есть множество способов выполнить одну и ту же работу, причем одни из них более простые и понятные, чем другие. Это касается и алгоритмов. Если выясняется, что есть более понятный способ сделать что-либо, следует заменить им сложный способ. Рефакторинг позволяет разделять сложные вещи на более простые, но иногда наступает момент, когда надо просто взять алгоритм целиком и заменить его более простым. Это случается, когда вы знакомитесь с задачей поближе и обнаруживаете, что ее можно решить и более простым способом. Иногда с этим сталкиваешься, когда находишь библиотеку, в которой есть функции, дублирующие ваш код.

Иногда, когда требуется изменить алгоритм, чтобы он решал задачу, немного отличающуюся от первоначальной, имеет смысл начать с его замены кодом, в который потом будет проще внести необходимые изменения.

Перед тем как приступить к этому действию, убедитесь, что дальнейшая декомпозиция метода невозможна. Заменять большой и сложный алгоритм — очень трудная задача, которая становится реально выполнимой только после существенного его упрощения.

Техника

- Подготовьте свой вариант алгоритма. Добейтесь, чтобы он компилировался.
- Выполните свои тесты для нового алгоритма. Если результаты работы совпадают с теми, которые были для старого алгоритма, на этом можно остановиться.
- Если результаты иные, используйте старый алгоритм для сравнения при тестировании и отладке.

  ⇒ *Выполните каждый контрольный пример со старым и новым алгоритмами и сравните результаты. Это поможет вам увидеть, какие контрольные примеры приводят к проблемам и к каким именно.*

Глава 7

Перенос функциональности между объектами

Одним из наиболее фундаментальных решений, принимаемых при проектировании объектов, является решение о месте размещения ответственности за те или иные действия. Я работаю с объектами свыше десяти лет, но и сейчас не всегда могу сделать правильный выбор с первого раза. Раньше это беспокоило меня, но сейчас я знаю, что рефакторинг позволяет мне в любой момент изменить первоначальное решение.

Часто эти проблемы решаются с помощью простых рефакторингов “Перенос метода” (с. 162) и “Перенос поля” (с. 166), обеспечивающих перенос поведения. Если требуется выполнить оба рефакторинга, то лучше начать с рефакторинга “Перенос поля”.

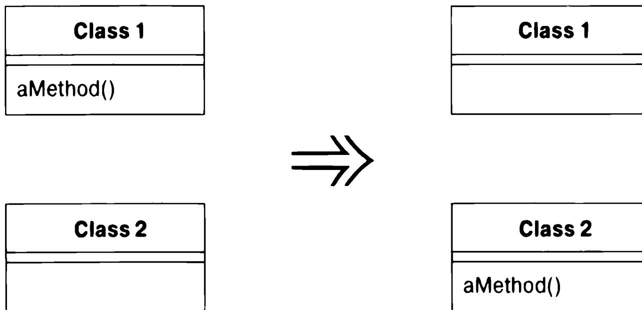
Зачастую в классах имеется слишком много ответственности. В этом случае я применяю рефакторинг “Извлечение класса” (с. 169) для разделения этой ответственности на части. Если у некоторого класса ответственность слишком мала, я соединяю его с другим классом с помощью рефакторинга “Встраивание класса” (с. 174). Если используется некоторый другой класс, часто удобно скрыть этот факт с помощью рефакторинга “Сокрытие делегирования” (с. 176). Иногда сокрытие класса, к которому выполняется делегирование, приводит к перманентным изменениям интерфейса, и тогда следует воспользоваться рефакторингом “Удаление посредника” (с. 179).

Два последних рефакторинга, описываемые в этой главе, “Введение внешнего метода” (с. 181) и “Введение локального расширения” (с. 183), представляют собой частные случаи, которыми я пользуюсь только тогда, когда мне недоступен исходный код класса, но я хочу переместить функциональность в этот класс. Если таких методов всего лишь два-три, я применяю рефакторинг “Введение внешнего метода” (с. 181); если же методов больше, то я пользуюсь рефакторингом “Введение локального расширения” (с. 183).

Перенос метода (Move Method)

Метод использует (или будет использовать) функциональность иного класса (или используется ею), отличного от того, в котором он определен.

Создайте новый метод с аналогичным телом в том классе, который чаще всего им используется. Замените тело прежнего метода простым делегированием или полностью удалите его.



Мотивация

Перенос методов — хлеб и масло рефакторинга. Я перемещаю методы, если в классах сосредоточено слишком много функциональности или классы слишком сильно связаны взаимодействием между собой. Перенося методы, можно сделать классы более простыми и добиться более ясной реализации функций.

Обычно я просматриваю все методы класса, пытаясь найти метод, который чаще обращается не к объекту, в котором находится, а к некоторому другому. Это полезно делать после перемещения каких-либо полей. Найдя метод, который стоит перенести, я рассматриваю, какие методы его вызывают, какими методами он пользуется, и ищу переопределяющие его методы в иерархии классов. Я прикидываю, следует ли продолжать работу на основе объекта, с которым активнее всего взаимодействует данный метод.

Не всегда просто принять такое решение. Если я не уверен в необходимости перемещения данного метода, я рассматриваю другие методы. Бывает, что принять решение о перемещении некоторого другого метода проще. В принципе выбор метода не так уж важен. Если принять решение трудно, то, вероятно, оно не играет особой роли. Я принимаю то решение, которое подсказывает инстинкт; в конце концов, это решение всегда можно будет изменить.

Техника

- Изучите всю функциональность, используемую исходным методом, которая определена в исходном классе, и определите, не следует ли переместить и ее.

  ⇛ *Если некоторая функциональность используется только методом, который вы собираетесь переместить, то вполне можно переместить и ее. Если же эта функциональность используется и другими методами, подумайте, не стоит ли переместить их все. Иногда проще переместить сразу группу методов, чем по одному.*

- Проверьте наличие в подклассах и суперклассах других объявлений этого метода.

  ⇛ *Если есть другие объявления, перемещение может оказаться невозможным, если только полиморфизм не будет отражен в целевом классе.*

- Объявите метод в целевом классе.

  ⇛ *Можно выбрать другое имя метода, более подходящее для целевого класса.*

- Скопируйте код исходного метода в целевой. Настройте метод для работы в новом окружении.

  ⇛ *Если методу нужен его исходный объект, следует определить способ обращения к нему из целевого метода. Если в целевом классе соответствующего механизма нет, передайте ссылку на исходный объект новому методу в качестве параметра.*

  ⇛ *Если метод содержит обработчики исключений, решите, какому из классов логичнее передать обработку исключений. Если эта ответственность должна лежать на исходном классе, оставьте обработчики исключений в нем.*

- Выполните компиляцию целевого класса.
- Определите способ обращения к целевому объекту из исходного.

  ⇛ *Может иметься поле или метод, представляющий целевой объект. Если же их нет, посмотрите, насколько сложно создать метод для этого. Если это трудно, создайте в исходном объекте новое поле, в котором будет храниться ссылка на целевой объект. Такая модификация может стать постоянной или сохраняться только до тех пор, пока рефакторинг не позволит удалить это поле.*

- Сделайте исходный метод делегирующим.
- Выполните компиляцию и тестирование.

- Определите, следует ли удалить исходный метод или сохранить его в качестве делегирующего свою функциональность.

  ⇒ *При наличии большого количества обращений проще оставить исходный метод в качестве делегирующего.*

- Если исходный метод удаляется, замените все обращения к нему обращениями ко вновь созданному методу.

  ⇒ *Выполнять компиляцию и тестирование можно после исправления каждого обращения, хотя обычно проще заменить все ссылки сразу путем поиска и замены.*

- Выполните компиляцию и тестирование.

Пример

Класс банковского счета иллюстрирует применение данного рефакторинга.

```
class Account...
    double overdraftCharge() {
        if (_type.isPremium()) {
            double result = 10;
            if (_daysOverdrawn>7) result += (_daysOverdrawn-7)*0.85;
            return result;
        }
        else return _daysOverdrawn * 1.75;
    }

    double bankCharge() {
        double result = 4.5;
        if (_daysOverdrawn > 0) result += overdraftCharge();
        return result;
    }
    private AccountType _type;
    private int _daysOverdrawn;
```

Представим себе, что будет добавлено несколько новых типов счетов со своими правилами начисления платы за превышение кредита (овердрафт). Я хочу переместить метод начисления этой платы в соответствующий тип счета.

Прежде всего, посмотрим, какие функции использует метод `overdraftCharge`, и решим, следует ли перенести один метод или сразу группу методов. В данном случае надо, чтобы поле `_daysOverdrawn` оставалось в исходном классе, потому что оно будет разным для отдельных счетов.

После этого я копирую тело метода в класс типа счета и настраиваю его для работы на новом месте.

```
class AccountType...
    double overdraftCharge(int daysOverdrawn) {
        if (isPremium()) {
            double result = 10;
            if (daysOverdrawn>7) result += (daysOverdrawn-7)*0.85;
            return result;
        } else return daysOverdrawn * 1.75;
    }
```

В данном случае настройка означает удаление `_type` из вызовов соответствующих методов и некоторые действия с той функциональностью, которая остается нужной. Если мне требуется некоторая функциональная возможность исходного класса, можно выбрать один из четырех вариантов: 1) перенести ее в целевой класс, 2) создать или использовать ссылку на исходный класс из целевого, 3) передать исходный объект в качестве параметра метода, 4) если интересующая возможность представляет собой переменную, передать ее в виде параметра.

Здесь я передал переменную как параметр.

Когда метод настроен и компилируется, можно заменить тело исходного метода простым делегированием:

```
class Account...
    double overdraftCharge() {
        return _type.overdraftCharge(_daysOverdrawn);
    }
```

Теперь можно вновь выполнить компиляцию и тестирование.

Сейчас весь код можно оставить в таком виде, но можно и полностью удалить метод из исходного класса. Для этого нужно найти все места, где он вызывается, и заменить их вызовом вновь созданного метода:

```
class Account...
    double bankCharge() {
        double result = 4.5;
        if (_daysOverdrawn > 0)
                 result += _type.overdraftCharge(_daysOverdrawn);
        return result;
    }
```

После замены исходного текста во всех точках вызова можно полностью удалить объявление метода в классе `Account`. Компиляцию и тестирование можно проводить как после каждого удаления, так и после всех удалений. Если метод не является закрытым, нужно проверить, не используют ли его другие классы. В строго типизированном языке компиляция после удаления объявления из исходного класса обнаруживает все огрехи, которые могли быть пропущены.

В нашем примере метод обращался к единственному полю, поэтому я смог передать его как переменную. Но если бы метод вызывал другой метод класса `Account`, я не смог бы так поступить. В подобных случаях приходится передавать весь исходный объект:

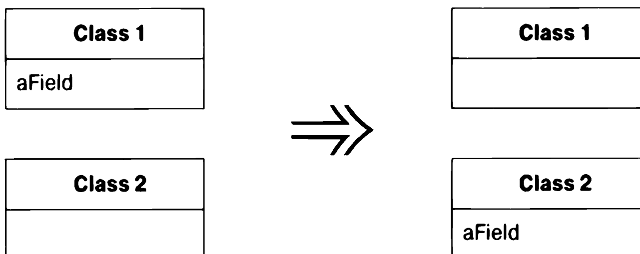
```
class AccountType...
    double overdraftCharge(Account account) {
        if (isPremium()) {
            double result = 10;
            if (account.getDaysOverdrawn() > 7)
                result += (account.getDaysOverdrawn()-7)*0.85;
            return result;
        } else return account.getDaysOverdrawn() * 1.75;
    }
```

Я передаю исходный объект и тогда, когда требуется несколько возможностей класса (хотя, если их оказывается слишком много, лучше выполнить дополнительный рефакторинг; обычно приходится заняться декомпозицией и вернуть некоторые части обратно).

Перенос поля (Move Field)

Поле используется (или будет использоваться) другим классом чаще, чем классом, в котором оно определено.

Создайте в целевом классе новое поле и измените код всех его использований.



Мотивация

Перенос состояний и поведения между классами является самой сутью рефакторинга. По мере разработки системы выявляется необходимость в новых классах и распределении ответственности между ними. Корректное и разумное проектное решение через неделю может оказаться никуда не годным. Но проблема не в этом, а в том, чтобы не оставить эту ситуацию навечно.

Я рассматриваю возможность переноса поля, когда вижу, что оно используется большим количеством методов в другом классе, чем в своем (это использование может быть косвенным, через методы доступа). Можно принять решение о переносе методов; конкретное решение зависит от интерфейса. Но если представляется разумным оставить методы на месте, я перемещаю поле.

Другим основанием для переноса поля может быть выполнение рефакторинга "Извлечение класса" (с. 169). В этом случае сначала переносятся поля, а затем методы.

Техника

- Если поле открытое, выполните рефакторинг "Инкапсуляция поля" (с. 224).

  ⇒ *Если вы собираетесь перенести методы, часто обращающиеся к полю, или к полю обращается много методов, может оказаться полезным рефакторинг "Самоинкапсуляция поля"* *(с. 190)*.

- Выполните компиляцию и тестирование.
- Создайте в целевом классе поле с методами для чтения и установки значений.
- Скомпилируйте целевой класс.
- Определите способ обращения к целевому объекту из исходного.

  ⇒ *Целевой класс может быть доступным через уже имеющиеся поля или методы. Если нет, посмотрите, нельзя ли создать для этого метод. Если нет, следует создать в исходном объекте новое поле, в котором будет храниться ссылка на целевой объект. Такая модификация может стать постоянной или сохраниться до тех пор, пока рефакторинг не позволит удалить это поле.*

- Удалите поле из исходного класса.
- Замените все ссылки на исходное поле обращениями к соответствующему методу в целевом классе.

  ⇒ *Чтение переменной замените обращением к методу получения значения в целевом объекте; для присваивания переменной замените обращение к переменной обращением к методу установки значения в целевом объекте.*

  ⇒ *Если поле не является закрытым, поищите ссылки на него во всех подклассах исходного класса.*

- Выполните компиляцию и тестирование.

Пример

Ниже представлена часть класса Account:

```
class Account...
    private AccountType _type;
    private double _interestRate;
    double interestForAmount_days(double amount, int days) {
        return _interestRate * amount * days / 365;
    }
```

Я хочу переместить поле процентной ставки в класс типа счета. Есть несколько методов, обращающихся к этому полю; одним из них является interestForAmount_days.

Далее создаются поле и методы доступа к нему в целевом классе:

```
class AccountType...
    private double _interestRate;

    void setInterestRate(double arg) {
        _interestRate = arg;
    }

    double getInterestRate() {
        return _interestRate;
    }
```

На этом этапе можно скомпилировать новый класс.

После этого я переадресую методы исходного класса на использование целевого класса и удаляю поле процентной ставки из исходного класса. Это поле надо удалить, чтобы гарантировать, что переадресация действительно происходит. Так компилятор поможет обнаружить методы, которые были пропущены при переадресации.

```
private double _interestRate;

double interestForAmount_days(double amount, int days) {
    return _type.getInterestRate() * amount * days / 365;
}
```

Пример: использование самоинкапсуляции

Если имеется много методов, которые используют поле процентной ставки, можно начать с рефакторинга “Самоинкапсуляция поля” (с. 190):

```
class Account...
    private AccountType _type;
    private double _interestRate;
```

```
    double interestForAmount_days(double amount, int days) {
        return getInterestRate() * amount * days / 365;
    }
    private void setInterestRate(double arg) {
        _interestRate = arg;
    }
    private double getInterestRate() {
        return _interestRate;
    }
```

Таким образом, необходимо выполнить только переадресацию методов доступа:

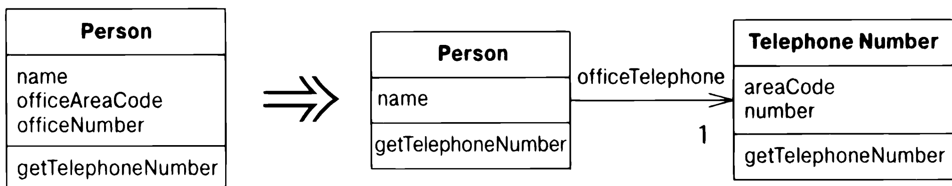
```
double interestForAmountAndDays(double amount, int days) {
    return getInterestRate() * amount * days / 365;
}

private void setInterestRate(double arg) {
    _type.setInterestRate(arg);
}

private double getInterestRate() {
    return _type.getInterestRate();
}
```

Позднее при желании можно выполнить переадресацию клиентов методов доступа, чтобы они непосредственно использовали новый объект. Применение самоинкапсуляции позволяет выполнять рефакторинг более мелкими шагами. Это удобно, когда класс подвергается значительной переделке. В частности, это упрощает применение рефакторинга "Перенос метода" (с. 162) для переноса методов в другой класс. Если они обращаются к методам доступа, то такие ссылки изменять не обязательно.

Извлечение класса (Extract Class)

Некоторый класс выполняет работу, которую следует поделить между двумя классами.

Создайте новый класс и переместите в него соответствующие поля и методы из старого класса.



Мотивация

Вероятно, вы слышали или читали, что класс должен представлять собой точно очерченную абстракцию, выполняя несколько конкретных обязанностей. На практике классы подвержены разрастанию. То здесь добавится несколько операций, то там появятся новые данные. Добавляя новую функциональность в класс, вам кажется, что для нее нет смысла заводить отдельный класс, но по мере того, как функции растут и плодятся, класс становится слишком сложным и раздутым.

Вы получаете класс с огромным количеством методов и данных, который оказывается слишком велик для понимания. Вы подумываете о том, чтобы разделить его на части, и в конце концов делаете это. Хорошим признаком является сочетание подмножества данных с подмножеством методов. Другой хороший признак — наличие подмножеств данных, которые обычно либо вместе изменяются, либо находятся в тесной зависимости одно от другого. Полезно задать себе вопрос о том, что будет, если удалить часть данных или метод. Какие другие данные или методы при этом потеряют смысл?

Одним из признаков, часто проявляющихся в дальнейшем во время разработки, служит способ создания подтипов класса. Вы можете обнаружить, что выделение подтипов оказывает воздействие лишь на некоторую функциональность или что для одной функциональности выделение подтипов выполняется иначе, чем для других.

Техника

- Определите, как будут разделены ответственности класса.
- Создайте новый класс, выражающий отделяемую ответственность.
  - ⇛ *Если ответственность прежнего класса перестает соответствовать его названию, переименуйте класс.*
- Организуйте связь старого и нового классов.
  - ⇛ *Может потребоваться двусторонняя связь; но не создавайте обратную связь, пока она не станет необходимой.*
- Примените рефакторинг “Перенос поля” (с. 166) ко всем полям, которые желательно перенести.
- После каждого переноса выполните компиляцию и тестирование.
- Примените рефакторинг “Перенос метода” (с. 162) ко всем методам, переносимым из старого класса в новый. Начните с методов более низкого уровня (вызываемых, а не вызывающих) и постепенно идите к методам более высокого уровня.
- После каждого переноса выполняйте компиляцию и тестирование.

- Пересмотрите интерфейсы каждого класса и сократите их.
  - ⇛ *Создав двустороннюю связь, подумайте, нельзя ли превратить ее в односторонную.*
- Определите, должен ли новый класс быть виден всем. Если да, то решите, как это должно быть сделано — в виде объекта ссылки или объекта с неизменяемым значением.

Пример

Начну с простого класса, описывающего персону:

```
class Person...
    public String getName() {
        return _name;
    }
    public String getTelephoneNumber() {
        return ("(" + _officeAreaCode + ") " +
                _officeNumber);
    }
    String getOfficeAreaCode() {
        return _officeAreaCode;
    }
    void setOfficeAreaCode(String arg) {
        _officeAreaCode = arg;
    }
    String getOfficeNumber() {
        return _officeNumber;
    }
    void setOfficeNumber(String arg) {
        _officeNumber = arg;
    }

    private String _name;
    private String _officeAreaCode;
    private String _officeNumber;
```

Здесь мы можем выделить в отдельный класс функции, относящиеся к телефонным номерам. Начнем с определения класса телефонного номера:

```
class TelephoneNumber {
}
```

Это очень просто! Теперь создаем ссылку из класса `Person` к классу телефонного номера:

```
class Person
    private TelephoneNumber _officeTelephone = new TelephoneNumber();
```

Теперь я применяю рефакторинг “Перенос метода” (с. 162) к одному из полей:

```
class TelephoneNumber {
    String getAreaCode() {
        return _areaCode;
    }
    void setAreaCode(String arg) {
        _areaCode = arg;
    }
    private String _areaCode;
}
class Person...
    public String getTelephoneNumber() {
        return ("(" + getOfficeAreaCode() + ") " + _officeNumber);
    }
    String getOfficeAreaCode() {
        return _officeTelephone.getAreaCode();
    }
    void setOfficeAreaCode(String arg) {
        _officeTelephone.setAreaCode(arg);
    }
```

После этого я могу перенести другое поле и применить рефакторинг “Перенос метода” (с. 162) к номеру телефона:

```
class Person...
    public String getName() {
        return _name;
    }
    public String getTelephoneNumber() {
        return _officeTelephone.getTelephoneNumber();
    }
    TelephoneNumber getOfficeTelephone() {
        return _officeTelephone;
    }

    private String _name;
    private TelephoneNumber _officeTelephone = new TelephoneNumber();

class TelephoneNumber...
        public String getTelephoneNumber() {
        return ("(" + _areaCode + ") " + _number);
    }
    String getAreaCode() {
        return _areaCode;
    }
    void setAreaCode(String arg) {
        _areaCode = arg;
    }
    String getNumber() {
```

```
        return _number;
    }
    void setNumber(String arg) {
        _number = arg;
    }
    private String _number;
    private String _areaCode;
```

Теперь необходимо решить, какой новый класс должен быть доступен для клиентов. Можно полностью скрыть класс, предоставив соответствующие делегирующие методы, а можно сделать класс открытым. Можно открыть класс лишь для некоторых клиентов (например, находящихся в моем пакете).

Решив сделать класс общедоступным, следует рассмотреть опасности, связанные с псевдонимами. Если я открою телефонный номер, а клиент изменит код зоны, что произойдет? Такое изменение может выполнить клиент косвенный — клиент клиента клиента.

Возможны следующие варианты.

1. Допускается, что любой объект может изменить любую часть телефонного номера. В таком случае телефонный номер становится объектом ссылки, и следует рассмотреть возможность применения рефакторинга "Замена значения ссылкой" (с. 198). В этом случае доступ к телефонному номеру обеспечивает экземпляр класса `Person`.
2. Я не желаю, чтобы кто-либо мог изменить телефонный номер иначе, кроме как посредством методов экземпляра класса `Person`. Можно сделать телефонный номер неизменяемым или обеспечить к нему неизменяемый интерфейс.
3. Существует возможность копировать телефонный номер перед тем, как предоставлять его, но это может привести к недоразумениям, потому что программисты могут решить, что в состоянии изменить его значение. При частой передаче телефонного номера у клиентов могут также возникать проблемы с псевдонимами.

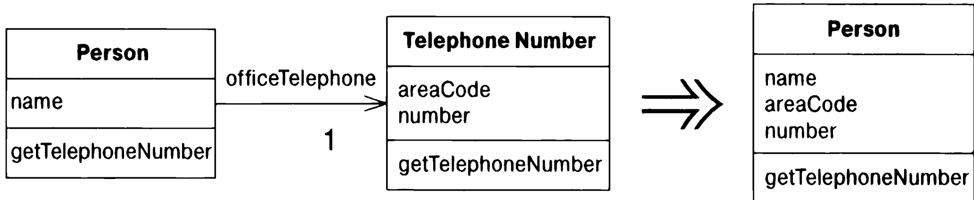
Для повышения живучести параллельной программы часто используется рефакторинг "Извлечение класса" (с. 169), потому что он позволяет устанавливать раздельные блокировки для двух получающихся классов. Если блокировать оба объекта не требуется, то этот рефакторинг становится не обязательным. Подробнее об этом сказано в разделе 3.3 книги [11].

Однако здесь имеется опасность. Если необходимо обеспечить совместную блокировку обоих объектов, вы попадаете в область транзакций и других разновидностей совместных блокировок. Как описывается в разделе 8.1 книги [11], это сложная область, требующая применения более мощного аппарата, что обычно мало оправданно. Транзакции очень полезны, но большинству программистов не стоит браться за написание их менеджеров.

Встраивание класса (Inline Class)

Класс выполняет слишком мало действий.

Переместите его функциональность в другой класс и удалите исходный.



Мотивация

Данный рефакторинг, по сути, противоположен рефакторингу “Извлечение класса” (с. 169). Я прибегаю к нему, когда класс становится мало полезным и его имеет смысл удалить. Зачастую это происходит из-за рефакторинга, оставившего в классе мало функциональности. В этом случае следует встроить данный класс в другой, выбрав в качестве целевого тот класс, который чаще всего использует данный.

Техника

- Объявите открытый протокол исходного класса в поглощающем классе. Делегируйте все новые методы исходному классу.

  ⇛ *Если для методов исходного класса имеет смысл отдельный интерфейс, выполните перед встраиванием рефакторинг “Извлечение интерфейса” (с. 357).*

- Перенесите все ссылки из исходного класса в поглощающий класс.

  ⇛ *Объявите исходный класс закрытым, чтобы удалить все обращения к нему из-за пределов пакета. Измените также имя исходного класса, чтобы компилятор помог вам обнаружить все обращения к исходному классу.*

- Выполните компиляцию и тестирование.
- С помощью рефакторингов “Перенос метода” (с. 162) и “Перенос поля” (с. 166) переносите одну за другой функциональности исходного класса, пока в нем ничего не останется.
- Отслужите короткую и скромную панихиду по исходному классу.

Пример

Создав класс из телефонного номера, я теперь хочу вернуть его обратно. Начну с отдельных классов:

```
class Person...
    public String getName() {
    return _name;
    }
    public String getTelephoneNumber(){
    return _officeTelephone.getTelephoneNumber();
    }
    TelephoneNumber getOfficeTelephone() {
    return _officeTelephone;
    }

    private String _name;
    private TelephoneNumber _officeTelephone = new TelephoneNumber();

class TelephoneNumber...
    public String getTelephoneNumber() {
        return ("(" + _areaCode + ") " + _number);
    }
    String getAreaCode() {
        return _areaCode;
    }
    void setAreaCode(String arg) {
        _areaCode = arg;
    }
    String getNumber() {
        return _number;
    }
    void setNumber(String arg) {
        _number = arg;
    }

    private String _number;
    private String _areaCode;
```

Начну с объявления в классе `Person` всех видимых методов класса телефонного номера:

```
class Person...
    String getAreaCode() {
        return _officeTelephone.getAreaCode();
    }
    void setAreaCode(String arg) {
        _officeTelephone.setAreaCode(arg);
    }
    String getNumber() {
```

```
        return _officeTelephone.getNumber();
    }
    void setNumber(String arg) {
        _officeTelephone.setNumber(arg);
    }
```

Теперь я нахожу клиентов класса телефонного номера и переключаю их на использование интерфейса класса `Person`; так что

```
Person martin = new Person();
martin.getOfficeTelephone().setAreaCode("781");
```

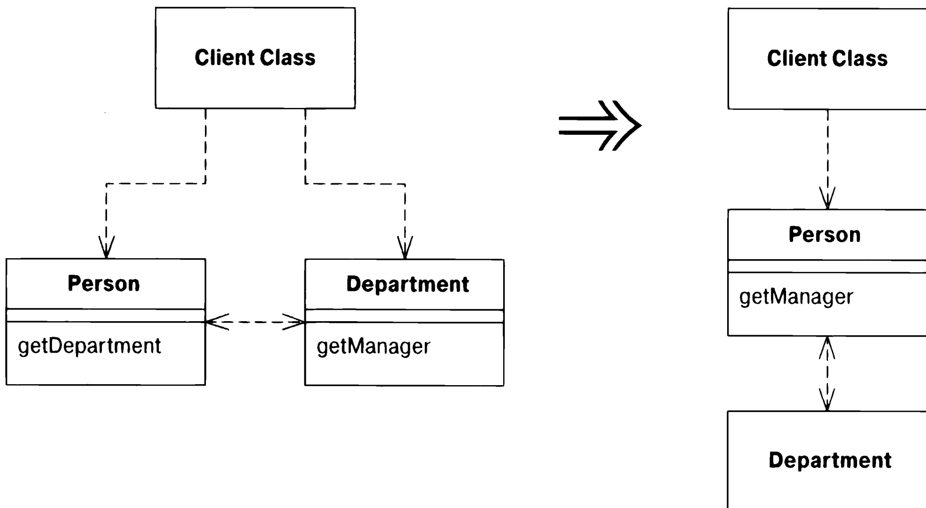
превращается в

```
Person martin = new Person();
martin.setAreaCode("781");
```

Теперь можно повторять рефакторинги “Перенос метода” (с. 162) и “Перенос поля” (с. 166), пока от класса телефонного номера не останется пустое место.

Сокрытие делегирования (Hide Delegate)

Клиент обращается к делегируемому классу объекта.
Создайте на сервере методы, скрывающие делегирование.



Мотивация

Одним из ключевых (если не самым главным) свойством объектов является инкапсуляция. Инкапсуляция означает, что объекты должны иметь о других частях системы меньше информации. Тогда при модификации некоторых частей системы требуется учитывать ее влияние на меньшее количество объектов, что упрощает внесение изменений.

Всякий, кто занимался объектами, знает, что поля следует скрывать, несмотря на то что Java позволяет делать их открытыми. По мере накопления опыта работы с объектами появляется понимание того, что инкапсулировать можно все более и более широкий круг вещей.

Если клиент вызывает метод, определенный для одного из полей объекта-сервера, клиент должен быть осведомлен о соответствующем объекте-делегате. Если этот делегат изменяется, может потребоваться модифицировать код клиента. От этой зависимости можно избавиться, поместив в сервер простой делегирующий метод, скрывающий делегирование (рис. 7.1). В таком случае изменения ограничиваются сервером и не распространяются на клиента.

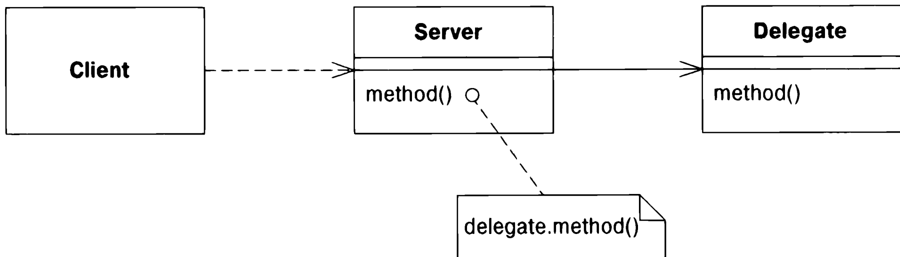


Рис. 7.1. *Простое делегирование*

Может иметь смысл применение рефакторинга “Извлечение класса” (с. 169) к некоторым или всем клиентам сервера. Если скрыть делегирование от всех клиентов, можно полностью удалить упоминание о делегате из интерфейса сервера.

Техника

- Для каждого метода класса-делегата создайте простой делегирующий метод на сервере.
- Модифицируйте код клиента так, чтобы он обращался к серверу.

  ⇛ *Если клиент и сервер находятся в разных пакетах, рассмотрите возможность ограничения доступа к методу делегата областью видимости пакета.*

- После настройки каждого метода выполните компиляцию и тестирование.
- Если после этого доступ к делегату не нужен ни одному клиенту, удалите с сервера метод обращения к делегату.
- Выполните компиляцию и тестирование.

Пример

Начну с классов, представляющих работника и его отдел:

```
class Person {
    Department _department;

    public Department getDepartment() {
        return _department;
    }
    public void setDepartment(Department arg) {
        _department = arg;
    }
}

class Department {
    private String _chargeCode;
    private Person _manager;

    public Department(Person manager) {
        _manager = manager;
    }

    public Person getManager() {
        return _manager;
    }
    ...
```

Если клиенту требуется узнать, кто является руководителем некоторого работника, он должен сначала узнать, в каком отделе работает этот человек:

```
manager = john.getDepartment().getManager();
```

Так клиент получает сведения о работе класса `Department` и о том, что в нем хранятся данные о руководителе. Эту связь можно ослабить, скрыв от клиента класс `Department` путем создания простого делегирующего метода в классе `Person`:

```
public Person getManager() {
    return _department.getManager();
}
```

Теперь необходимо модифицировать код всех клиентов `Person`, чтобы они использовали новый метод:

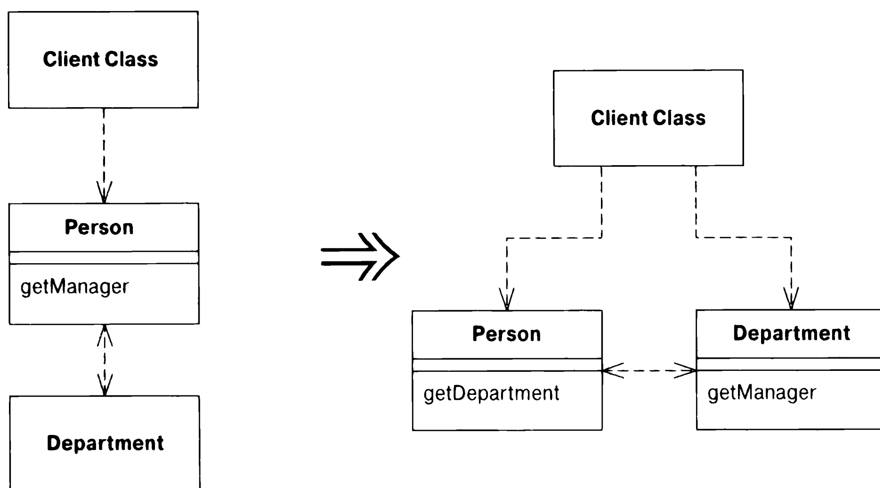
```
manager = john.getManager();
```

Выполнив изменения всех методов `Department` и всех клиентов `Person`, можно полностью удалить метод доступа `getDepartment` из класса `Person`.

Удаление посредника (Remove Middle Man)

Класс выполняет слишком много простого делегирования.

Заставьте клиента обращаться непосредственно к делегату.



Мотивация

При описании мотивации рефакторинга "Сокрытие делегирования" (с. 176) я говорил о преимуществах инкапсуляции при использовании делегирования. Однако имеются определенные неудобства, связанные с тем, что при желании клиента использовать новую функцию делегата в сервер необходимо добавлять простой делегирующий метод. Добавление большого количества методов оказывается утомительным; кроме того, в результате класс сервера становится просто посредником, так что может настать момент, когда клиенту лучше вызывать делегата непосредственно.

Какой именно должна быть мера сокрытия делегирования, сказать трудно. К счастью, это не так и важно благодаря наличию рефакторингов "Сокрытие

делегирования" (с. 176) и "Удаление посредника" (с. 179). Настройку системы можно осуществлять по мере надобности. По мере развития системы меняется и уровень сокрытия. Инкапсуляция, удовлетворявшая полгода назад, может оказаться неудобной в настоящий момент. Смысл рефакторинга в том, что надо не раскаиваться в сделанных ошибках, а просто выполнить необходимые исправления.

Техника

- Создайте метод доступа к делегату.
- Для каждого использования клиентом метода делегата удалите этот метод из сервера и замените его вызов в клиенте вызовом метода делегата.
- После обработки каждого метода выполняйте компиляцию и тестирование.

Пример

В качестве примера я воспользуюсь классами Person и Department, развернув ситуацию в обратную сторону. Начнем с класса Person, скрывающего делегирование классу Department:

```
class Person...
    Department _department;
    public Person getManager() {
        return _department.getManager();
    }

class Department...
    private Person _manager;
    public Department(Person manager) {
        _manager = manager;
    }
```

Чтобы найти руководителя некоторого работника, клиент делает запрос

```
manager = john.getManager();
```

Это простая конструкция, инкапсулирующая Department. Однако при наличии множества такого рода методов в классе Person оказывается слишком много простых делегирований. В этом случае посредника лучше удалить. Для этого сначала создается метод обращения к делегату:

```
class Person...
    public Department getDepartment() {
        return _department;
    }
```

После этого я поочередно рассматриваю каждый метод. Я нахожу клиентов, использующих этот метод `Person`, и изменяю их таким образом, чтобы они сначала получали делегата:

```
manager = john.getDepartment().getManager();
```

После этого можно убрать метод `getManager` из класса `Person`. Компиляция покажет, не пропустил ли я что-либо.

Может оказаться удобным сохранить некоторые из делегирований. Возможно, следует скрыть делегирование от одних клиентов, но показать другим. В этом случае также придется оставить некоторые простые делегирования.

Введение внешнего метода (Introduce Foreign Method)

Используемому серверу требуется дополнительный метод, но модифицировать класс невозможно.

Создайте в классе клиента метод, которому в качестве первого аргумента передается экземпляр класса сервера.

```
Date newStart = new Date(previousEnd.getYear(),
                         previousEnd.getMonth(),
                         previousEnd.getDate() + 1);
```

```
Date newStart = nextDay(previousEnd);

private static Date nextDay(Date arg) {
    return new Date(arg.getYear(), arg.getMonth(),
                    arg.getDate() + 1);
}
```

Мотивация

Достаточно распространенная ситуация: есть прекрасный класс с отличными сервисами. Затем оказывается, что нужен еще один сервис, но класс его не предоставляет. Мы ругаемся последними словами, но ничего не можем поделать — если бы была возможность модифицировать исходный код, мы дописали бы новый метод, но такой возможности нет. В этом случае приходится идти обходными путями, чтобы запрограммировать отсутствующий метод в клиенте.

Если клиентский класс использует этот метод единственный раз, то такое дополнительное кодирование не представляет особой проблемы и даже может вообще не требовать работы с исходным классом. Однако если метод используется многократно, его приходится кодировать снова и снова. Повторение кода — корень всех зол в программах, поэтому такой повторяющийся код следует выделить в отдельный метод. При проведении данного рефакторинга можно указать, что этот метод в действительности должен находиться в исходном классе, сделав его внешним методом.

Если выяснится, что для класса сервера создается много внешних методов или что многим классам требуется один и тот же внешний метод, следует применить другой рефакторинг, "Введение локального расширения" (с. 183).

Не нужно забывать о том, что внешние методы представляют собой обходной путь. Старайтесь помещать методы туда, где им надлежит находиться. Если проблема связана с правами владельца кода, отправьте внешний метод владельцу класса сервера и попросите его реализовать этот метод для вас.

Техника

- Создайте в классе клиента метод, выполняющий нужные вам действия.
  - ⇒ *Создаваемый метод не должен пользоваться возможностями класса клиента. Если ему требуется какое-то значение, передайте его в качестве параметра.*
- Сделайте первым параметром метода экземпляр класса сервера.
- В комментарии к методу отметьте, что это внешний метод, который на самом деле должен располагаться на сервере.
  - ⇒ *Благодаря этому вы сможете позднее, если появится возможность переноса метода, найти такие внешние методы с помощью простого текстового поиска.*

Пример

Имеется код, в котором нужно открыть новый период выставления счетов. Первоначально код выглядит так:

```
Date newStart = new Date(previousEnd.getYear(),
                         previousEnd.getMonth(),
                         previousEnd.getDate() + 1);
```

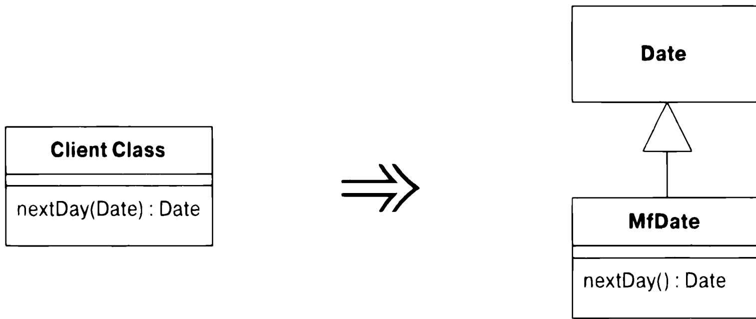
Код в правой части присваивания можно выделить в метод, который будет внешним для `Date`:

```
Date newStart = nextDay(previousEnd);

private static Date nextDay(Date arg) {
    // Внешний метод, должен находиться в классе Date
    return new Date(arg.getYear(),arg.getMonth(),arg.getDate()+1);
}
```

Введение локального расширения (Introduce Local Extension)

Используемому классу сервера требуется несколько дополнительных методов, но класс недоступен для модификации.

Создайте новый класс с необходимыми дополнительными методами. Сделайте его подклассом или оболочкой для исходного класса.



Мотивация

Увы, авторы классов не всеведущи и не могут предвидеть, какие методы могут потребоваться. Если чего-то не хватает и есть возможность модифицировать исходный код, то лучше всего добавить в класс новые методы. Однако иногда исходный код модифицировать невозможно. Если нужны лишь один-два новых метода, можно прибегнуть к рефакторингу “Введение внешнего метода” (с. 181). Однако, если их больше, ситуация может выйти из-под контроля, так что лучше все их объединить в подходящем месте. Очевидным методом решения этой задачи является стандартная объектно-ориентированная методика создания подклассов или оболочек. Я называю такой подкласс или оболочку локальным расширением.

Локальное расширение представляет собой отдельный класс, являющийся подтипом класса, расширением которого он является. Это означает, что он умеет делать все то же, что и исходный класс, но при этом имеет дополнительные возможности. Вместо работы с исходным классом следует создать экземпляр локального расширения и пользоваться им.

Используя локальное расширение, вы поддерживаете принцип упаковки методов и данных в корректно сформированные блоки. Если же размещать код в других классах, а не в расширении, то это приведет к усложнению других классов и сложностям повторного использования этих методов.

При выборе "подкласс или оболочка?" я обычно склоняюсь к подклассу, так как это означает меньший объем работы. Наибольшее препятствие при использовании подклассов заключается в том, что они должны использоваться во время создания объектов. Если вы можете управлять процессом создания объектов, никаких проблем не возникает. Они появляются, если локальное расширение необходимо применять позднее, после того как объекты уже созданы. При работе с подклассами приходится создавать новый объект подкласса. Если при этом в наличии есть другие объекты, ссылающиеся на старый, в результате мы получаем два объекта, содержащие оригинальные данные. Если оригинал неизменяемый, то проблем не возникает, поскольку можно использовать копию. Но если оригинал может изменяться, то возникает проблема, связанная с тем, что изменения одного объекта не отражаются в другом. В этом случае следует прибегнуть к оболочке, — тогда изменения, вносимые посредством локального расширения, воздействуют на исходный объект и наоборот.

Техника

- Создайте класс расширения в качестве подкласса или оболочки оригинала.
- Добавьте к расширению преобразующие конструкторы.
  ⇛ *Такой конструктор принимает в качестве аргумента оригинальный объект. В случае подкласса вызывается соответствующий конструктор суперкласса; в случае оболочки аргумент присваивается полю делегата.*
- Разместите в расширении новые функции.
- Где требуется, замените оригинал расширением.
- При наличии внешних методов для исходного класса переместите их в расширение.

Примеры

Мне приходилось заниматься такими вещами, работая с Java 1.0.1 и классом даты. Пока не появился класс календаря в Java 1.1, при работе с датой представлялась масса возможностей использовать расширения. Этот опыт пригодится сейчас в качестве примера.

Начнем с выбора между подклассом и оболочкой. Применение подклассов более очевидно:

```
Class mfDate extends Date {
    public nextDay()...
    public dayOfYear()...
```

В оболочке же используется делегирование:

```
class mfDate {
    private Date _original;
```

Пример: использование подкласса

Сначала создается новая дата в виде подкласса оригинала:

```
class MfDateSub extends Date
```

Затем осуществляется замена дат расширением. Конструкторы оригинала повторяются с помощью простого делегирования:

```
public MfDateSub(String dateString) {
    super(dateString);
};
```

Далее добавим преобразующий конструктор, принимающий в качестве аргумента оригинал:

```
public MfDateSub(Date arg) {
    super(arg.getTime());
}
```

Теперь можно добавить в расширение новую функциональность и с помощью рефакторинга “Перенос метода” (с. 162) перенести в расширение имеющиеся внешние методы:

```
client class...
    private static Date nextDay(Date arg) {
        // Внешний метод, должен быть в классе Date
        return new Date(arg.getYear(), arg.getMonth(),
                        arg.getDate() + 1);
    }
```

превращается в

```
class MfDate...
    Date nextDay() {
        return new Date(getYear(), getMonth(), getDate() + 1);
    }
```

Пример: использование оболочки

Начнем с объявления класса-оболочки:

```
class mfDate {
    private Date _original;
}
```

При использовании оболочки конструкторы создаются иначе. Исходные конструкторы реализованы с помощью простого делегирования:

```
public MfDateWrap(String dateString) {
    _original = new Date(dateString);
};
```

Преобразующий конструктор теперь просто присваивает переменную экземпляра:

```
public MfDateWrap(Date arg) {
    _original = arg;
}
```

Затем нужно выполнить скучную работу по делегированию всех методов исходного класса. Я покажу только два метода:

```
public int getYear() {
    return _original.getYear();
}
public boolean equals(MfDateWrap arg) {
    return (toDate().equals(arg.toDate()));
}
```

По окончании этой работы можно добавить в новый класс специальные методы для дат с помощью рефакторинга “Перенос метода” (с. 162):

```
client class...
private static Date nextDay(Date arg) {
    // Внешний метод, должен быть в классе Date
    return new Date(arg.getYear(), arg.getMonth(),
                    arg.getDate() + 1);
}
```

превращается в

```
class MfDate...
    Date nextDay() {
        return new Date(getYear(), getMonth(), getDate() + 1);
    }
```

Особые проблемы в случае применения оболочек возникают при использовании методов, принимающих в качестве аргумента оригинал, например:

```
public boolean after (Date arg)
```

Поскольку оригинал изменять нельзя, работать с `after` можно только одним способом:

```
aWrapper.after(aDate)           // Можно заставить работать
aWrapper.after(anotherWrapper)  // Можно заставить работать
aDate.after(aWrapper)           // Работать не будет
```

Цель этого вида перекрытия — скрыть от пользователя класса факт применения оболочки. Это правильная стратегия, потому что пользователю оболочки не нужно беспокоиться о том, оболочка ли это, и у него должна быть возможность одинаково работать с обоими классами. Однако скрыть эту информацию полностью я не смогу. Проблема вызывается некоторыми системными методами, например `equals`. Хотя, казалось бы, в идеале можно заменить `equals` в `MfDateWrap`:

```
public boolean equals (Date arg) // Возможны проблемы
```

Это опасно, поскольку, хотя так можно адаптировать класс к своим задачам, в других частях Java-системы предполагается симметричность равенства, т.е. если `a.equals(b)`, то `b.equals(a)`. При нарушении этого соглашения может возникнуть целый ряд странных ошибок. Единственный способ их избежать — модифицировать `Date`, но ведь, если бы это можно было сделать, не потребовался бы и данный рефакторинг! Поэтому в таких ситуациях приходится демонстрировать тот факт, что применяется оболочка, и для проверки равенства выбирать метод с другим именем:

```
public boolean equalsDate(Date arg)
```

Избежать проверки типа неизвестного объекта можно путем предоставления версии данного метода как для `Date`, так и для `MfDateWrap`:

```
public boolean equalsDate (MfDateWrap arg)
```

При работе с подклассами эта проблема не возникает, если не перекрывать операции. Если же я выполняю перекрытие, возникает путаница с поиском метода. Обычно я не перекрываю методы в расширениях, а только добавляю их.

Глава 8

Организация данных

В этой главе обсуждаются некоторые методы рефакторинга, которые предназначены для упрощения работы с данными. Многие считают рефакторинг "Самоинкапсуляция поля" (с. 190) излишним. Уже давно идут споры о том, как объект должен осуществлять доступ к собственным данным: непосредственно или через методы доступа. Иногда методы доступа нужны, и их можно получить с помощью упомянутого рефакторинга. Я, как правило, выбираю непосредственный доступ, потому что если мне понадобится такой рефакторинг, его очень легко выполнить.

Одно из полезных свойств объектных языков состоит в том, что они разрешают определять новые типы, которые позволяют делать больше, чем простые типы данных в обычных языках. Однако, чтобы научиться работать с этой возможностью, требуется некоторое время. Часто мы сначала используем простые данные-значения и лишь потом понимаем, что удобнее было бы работать с объектом. Рефакторинг "Замена значения данных объектом" (с. 194) позволяет превратить молчаливые данные в красноречивые объекты. Если эти объекты представляют собой экземпляры, которые могут понадобиться во многих местах программы, их можно превратить в объекты-ссылки с помощью рефакторинга "Замена значения ссылкой" (с. 198).

Если очевидно, что объект выступает в качестве структуры данных, ее можно сделать более понятной с помощью рефакторинга "Замена массива объектом" (с. 204). Во всех этих случаях объект представляет собой лишь первый шаг. Реальные преимущества достигаются при применении рефакторинга "Перенос метода" (с. 162), позволяющего добавить поведение в новые объекты.

Магические числа, т.е. числа с особым значением, также представляют собой проблему. Когда я только начинал программировать, меня предупреждали, что пользоваться ими не следует. Тем не менее они продолжают появляться в программах и сегодня, и чтобы избавиться от них, я использую рефакторинг "Замена магического числа символической константой" (с. 223).

Связи между объектами могут быть одно- или двусторонними. Односторонние связи проще, но иногда для новой функциональности требуется применить рефакторинг "Замена однонаправленной связи двунаправленной" (с. 216). Рефакторинг же "Замена двунаправленной связи однонаправленной" (с. 219) устраняет излишнюю сложность, когда выясняется, что двунаправленная связь больше не нужна.

Я часто сталкивался с ситуациями, когда классы графического интерфейса выполняют бизнес-логику, т.е. занимаются несвойственной им деятельностью. Для переноса поведения в классы предметной области необходимо там же хранить данные, и поддерживать графический интерфейс пользователя с помощью рефакторинга "Дублирование видимых данных" (с. 207). Я против дублирования данных, но в этом случае мы имеем дело с исключением.

Одним из ключевых принципов объектно-ориентированного программирования является инкапсуляция. Если некоторые открытые данные повсюду демонстрируют себя, рефакторинг "Инкапсуляция поля" (с. 224) обеспечит им достойное прикрытие. Если данные представляют собой коллекцию, следует воспользоваться рефакторингом "Инкапсуляция коллекции" (с. 226), поскольку для коллекций требуется особый протокол. Если обнажена целая запись, выберите рефакторинг "Замена записи классом данных" (с. 235).

Особого обращения требует такой вид данных, как код типа — специальное значение, которое указывает некоторую особенность типа экземпляра. Зачастую эти данные представляют собой перечисление; иногда они реализуются как статические неизменяемые целочисленные значения. Если код имеется только для информации и не меняет поведение класса, можно воспользоваться рефакторингом "Замена кода типа классом" (с. 236), который ведет к лучшей проверке типа и закладывает фундамент для переноса поведения в дальнейшем. Если код типа оказывает влияние на поведение класса, можно попытаться применить рефакторинг "Замена кода типа подклассами" (с. 241). Если это не удается, возможно, подойдет более сложный (но и более гибкий) рефакторинг "Замена кода типа состоянием/стратегией" (с. 245).

Самоинкапсуляция поля (Self Encapsulate Field)

При непосредственном обращении к полю работа с ним становится затруднительной.

Создайте методы получения и установки значения поля и обращайтесь к полю только через них.

```
private int _low, _high;
boolean includes(int arg) {
    return arg >= _low && arg <= _high;
}
```

```
private int _low, _high;
boolean includes(int arg) {
    return arg >= getLow() && arg <= getHigh();
}
int getLow() {
    return _low;
}
int getHigh() {
    return _high;
}
```

Мотивация

Что касается обращения к полям, то здесь существуют два подхода. Первый из них учит, что в классе, в котором определена переменная, обращаться к ней можно непосредственно. Другая же школа утверждает, что даже внутри класса следует всегда использовать методы доступа к полям (косвенный доступ к переменной). Между сторонниками разных подходов постоянно возникают горячие дискуссии. (См. обсуждение в работах [1] (с. 413) и [6].)

По сути, преимущество *косвенного обращения к переменной* состоит в том, что оно позволяет подклассу перекрыть метод получения информации и обеспечивает большую гибкость при работе с данными, например отложенную инициализацию, при которой переменная инициализируется лишь при необходимости ее использования.

Преимущество *непосредственного обращения к переменной* заключается в легкости чтения кода. Не нужно все время вспоминать — "это просто метод получения значения переменной".

Мой взгляд на эту проблему двойственный. Обычно я готов действовать так, как считают правильным остальные участники команды. Однако, действуя в одиночку, я предпочитаю сначала использовать непосредственное обращение к переменной, пока оно не является препятствием. При возникновении неудобств этого подхода я перехожу на косвенное обращение к переменным. Рефакторинг позволяет легко изменить свое решение.

Важнейший повод применить данный рефакторинг возникает, когда при обращении к полю суперкласса необходимо заменить обращение к переменной вычислением значения в подклассе. Первым шагом для этого является самоинкапсуляция поля. Затем можно заменить методы получения и установки значений теми, которые вам необходимы.

Техника

- Создайте методы получения и установки значения поля.
- Найдите все обращения к полю и замените их методами получения или установки значений.
  ⇒ *Замените чтение поля вызовом метода получения значения; замените присваивание полю вызовом метода установки значения.*
  ⇒ *Для проверки можно привлечь компилятор, временно изменив название поля.*
- Сделайте поле `private`.
- Проверьте, все ли обращения найдены и заменены.
- Выполните компиляцию и тестирование.

Пример

Возможно, следующий код слишком прост для примера, но зато он не потребовал много времени для написания:

```
class IntRange {

    private int _low, _high;

    boolean includes(int arg) {
        return arg >= _low && arg <= _high;
    }

    void grow(int factor) {
        _high = _high * factor;
    }

    IntRange(int low, int high) {
        _low = low;
        _high = high;
    }
```

Для самоинкапсуляции я определяю методы получения и установки значений (если таковых еще нет) и применяю их:

```
class IntRange {

    boolean includes(int arg) {
        return arg >= getLow() && arg <= getHigh();
    }

    void grow(int factor) {
        setHigh(getHigh() * factor);
    }
```

```
    private int _low, _high;

    int getLow() {
        return _low;
    }

    int getHigh() {
        return _high;
    }

    void setLow(int arg) {
        _low = arg;
    }

    void setHigh(int arg) {
        _high = arg;
    }
```

При использовании самоинкапсуляции необходимо проявлять осторожность, применяя метод установки значения в конструкторе. Часто предполагается, что метод установки будет использован только после создания объекта, так что его поведение во время инициализации может быть иным. В таких случаях я предпочитаю использовать в конструкторе непосредственное обращение или отдельный метод инициализации:

```
IntRange(int low, int high) {
    initialize(low, high);
}

private void initialize(int low, int high) {
    _low = low;
    _high = high;
}
```

Важность этих действий проявится при создании подкласса, например

```
class CappedRange extends IntRange {

    CappedRange(int low, int high, int cap) {
        super(low, high);
        _cap = cap;
    }

    private int _cap;

    int getCap() {
        return _cap;
    }
```

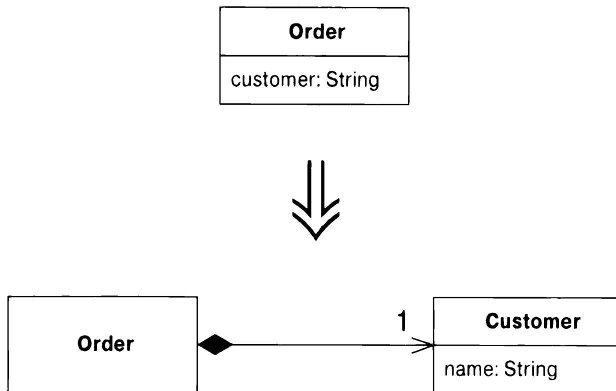
```
    int getHigh() {
        return Math.min(super.getHigh(), getCap());
    }
}
```

Можно целиком перекрыть поведение `IntRange` в отношение `_cap`, не меняя при этом поведение исходного класса.

Замена значения данных объектом (Replace Data Value with Object)

Имеется элемент данных, которому требуются дополнительные данные или поведение.

Преобразуйте элемент данных в объект.



Мотивация

Зачастую на ранних этапах разработки принимается решение о представлении простых фактов в виде простых элементов данных. Затем, в процессе разработки, обнаруживается, что эти простые элементы на самом деле куда сложнее. Телефонный номер вначале может быть представлен в виде строки, но позднее выясняется, что у него должно быть особое поведение, например в виде форматирования, извлечения кода города и т.п. Если таких элементов всего лишь пару, соответствующие методы можно поместить в объект, который ими владеет, но такой код начинает быстро попахивать дублированием и завистью к функциям. Если вы учуяли такой запах, преобразуйте данные в объект.

Техника

- Создайте класс для значения. Предусмотрите в нем final поле того же типа, что и у значения в исходном классе. Добавьте метод чтения и конструктор, принимающий поле в качестве аргумента.
- Выполните компиляцию.
- Измените тип поля в исходном классе на новый класс.
- Измените метод получения значения в исходном классе так, чтобы он вызывал метод чтения нового класса.
- Если поле упоминается в конструкторе исходного класса, присвойте ему значение с помощью конструктора нового класса.
- Измените метод получения значения так, чтобы он создавал новый экземпляр нового класса.
- Выполните компиляцию и тестирование.
- Для нового объекта может потребоваться рефакторинг “Замена значения ссылкой” (с. 198).

Пример

Начну с примера класса заказа order, в котором клиент customer хранится в виде строки, а затем преобразую клиент в объект, чтобы хранить в нем такие данные, как адрес или оценку кредитоспособности клиента, а также добавлять полезную функциональность для работы с этой информацией.

```
class Order...

    public Order(String customer) {
        _customer = customer;
    }

    public String getCustomer() {
        return _customer;
    }

    public void setCustomer(String arg) {
        _customer = arg;
    }

    private String _customer;
```

Вот пример клиентского кода, использующего этот класс:

```
private static int numberOfOrdersFor(Collection orders,
                                     String customer) {
    int result = 0;
    Iterator iter = orders.iterator();

    while (iter.hasNext()) {
        Order each = (Order) iter.next();

        if (each.getCustomer().equals(customer))
            result++;
    }

    return result;
}
```

Сначала я создаю новый класс клиента. В нем будет иметься объявленное как final поле для строкового атрибута (такое же, как то, которое используется в настоящий момент в заказе). Я назову его name и одновременно создам метод получения значения этого поля и конструктор, в котором используется этот атрибут:

```
class Customer {
    public Customer(String name) {
        _name = name;
    }
    public String getName() {
        return _name;
    }
    private final String _name;
}
```

Теперь я изменю тип поля клиента и методы, которые к нему обращаются, таким образом, чтобы они использовали ссылки на класс Customer. Метод получения значения и конструктор очевидны. Для метода установки значения я создаю новый объект Customer:

```
class Order...
    public Order(String customer) {
        _customer = new Customer(customer);
    }
    public String getCustomer() {
        return _customer.getName();
    }
    private Customer _customer;

    public void setCustomer(String arg) {
        _customer = new Customer(arg);
    }
```

Метод установки создает новый объект Customer потому, что прежний строковый атрибут был объектом-значением, и в настоящее время Customer тоже является объектом-значением. Это значит, что в каждом заказе имеется собственный объект Customer. Как правило, такие объекты-значения должны быть неизменяемыми, благодаря чему удается избежать некоторых неприятных ошибок, связанных с псевдонимами. Позже мне потребуется, чтобы Customer стал объектом-ссылкой, но это произойдет уже в результате другого рефакторинга. А пока самое время выполнить компиляцию и тестирование.

Теперь я рассмотрю методы заказа, работающие с клиентом, и выполню некоторые изменения, призванные прояснить новое положение вещей. К методу получения значения я применю рефакторинг “Переименование метода” (с. 290), чтобы сделать очевидным, что он возвращает имя, а не объект:

```
public String getCustomerName() {
    return _customer.getName();
}
```

Что касается конструктора и метода установки значения, их сигнатуры изменять не требуется, но имена аргументов следует изменить:

```
public Order(String customerName) {
    _customer = new Customer(customerName);
}
public void setCustomer(String customerName) {
    _customer = new Customer(customerName);
}
```

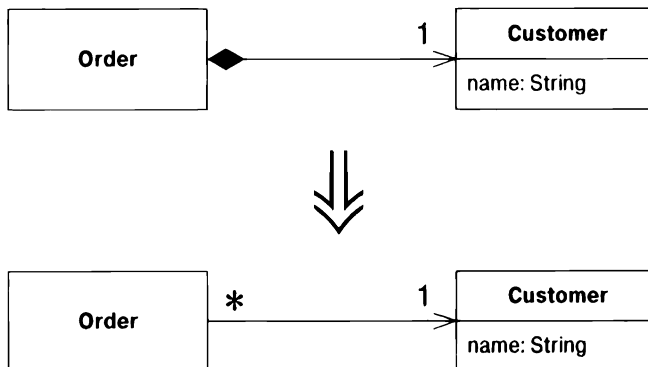
Далее при проведении следующего рефакторинга может потребоваться добавить новый конструктор и метод установки, которому передается существующий клиент.

На этом данный рефакторинг завершается, но, как и во многих других случаях, требуется сделать еще одну вещь. Если нужно добавить к клиенту оценку кредитоспособности, адрес или иную информацию, то в настоящий момент сделать это нельзя, потому что класс Customer действует как объект-значение. Каждый заказ имеет собственный объект Customer. Чтобы придать Customer эти атрибуты, необходимо применить рефакторинг “Замена значения ссылкой” (с. 198), чтобы все заказы для одного и того же клиента использовали один и тот же объект Customer. Рассматривающийся здесь пример не завершен — мы займемся им при описании следующего рефакторинга.

Замена значения ссылкой (Change Value to Reference)

Есть класс с множеством одинаковых экземпляров одного класса, которые желательно заменить одним объектом.

Превратите объект в объект-ссылку.



Мотивация

Во многих системах можно выполнить полезную классификацию объектов как объектов-ссылок и объектов-значений. *Объекты-ссылки* — это такие объекты, как клиент или счет. Каждый объект обозначает один объект реального мира, и равными считаются совпадающие объекты. *Объекты-значения* — это, например, дата или денежная сумма. Такие объекты полностью определены значениями своих данных. Нет никаких возражений против существования дублей объектов-значений, так что в системе могут быть сотни объектов со значением, скажем, "01.01.2000". Вы должны иметь возможность проверять, равны ли два объекта-значения, а для этого следует перекрыть метод `equals` (а также метод `hashCode`).

Выбор между ссылкой и значением не всегда очевиден. Зачастую изначально имеется простое значение с небольшим объемом неизменяемых данных. Затем возникает необходимость добавить изменяемые данные и гарантировать передачу этих изменений при всех обращениях к объекту. В таком случае объект-значение необходимо превратить в объект-ссылку.

Техника

- Выполните рефакторинг "Замена конструктора фабричным методом" (с. 321).
- Выполните компиляцию и тестирование.
- Решите, какой объект отвечает за обращение к объектам.

  ⇛ *Им может быть статический словарь или объект реестра.*

  В качестве точки доступа для нового объекта можно использовать несколько объектов.
- Определите, как будут создаваться объекты — заранее или динамически, по мере надобности.

  ⇛ *Если объекты создаются предварительно и извлекаются из памяти, необходимо обеспечить их загрузку перед тем, как они потребуются.*
- Измените фабричный метод так, чтобы он возвращал объект-ссылку.

  ⇛ *Если объекты создаются заранее, необходимо решить, как обрабатывать ошибки при запросе несуществующего объекта.*

  Может потребоваться применить к фабрике рефакторинг "Переименование метода" (с. 290), чтобы указать, что она возвращает существующие объекты.
- Выполните компиляцию и тестирование.

Пример

Продолжу с места, где мы остановились при описании рефакторинга "Замена значения данных объектом" (с. 194). Итак, имеется следующий класс клиента:

```
class Customer {
    public Customer(String name) {
        _name = name;
    }
    public String getName() {
        return _name;
    }
    private final String _name;
}
```

Его используют класс заказа:

```
class Order...
    public Order(String customerName) {
        _customer = new Customer(customerName);
    }
```

```
    public void setCustomer(String customerName) {
        _customer = new Customer(customerName);
    }
    public String getCustomerName() {
        return _customer.getName();
    }
    private Customer _customer;
```

и некоторый клиентский код:

```
private static int numberOfOrdersFor(Collection orders,
                                     String customer) {
    int result = 0;
    Iterator iter = orders.iterator();
    while (iter.hasNext()) {
        Order each = (Order) iter.next();
        if (each.getCustomerName().equals(customer))
            result++;
    }
    return result;
}
```

В настоящий момент это — объект-значение. У каждого заказа собственный объект `customer`, даже если концептуально это один и тот же клиент. Я хочу внести изменения таким образом, чтобы при наличии нескольких заказов одного и того же клиента в них использовался бы один и тот же экземпляр класса `Customer`. В данном случае это означает, что для каждого имени клиента должен существовать только один объект.

Я начинаю с рефакторинга “Замена конструктора фабричным методом” (с. 321). Его применение позволит мне контролировать процесс создания объектов, что окажется важным в дальнейшем. Я определяю фабричный метод в клиенте:

```
class Customer {
    public static Customer create(String name) {
        return new Customer(name);
    }
```

Затем я заменяю вызовы конструктора обращением к фабрике:

```
class Order {
    public Order(String customer) {
        _customer = Customer.create(customer);
    }
```

После этого я делаю конструктор закрытым:

```
class Customer {
    private Customer(String name) {
        _name = name;
    }
```

Теперь мне надо решить, как выполнять обращение к клиентам. Я предпочитаю использовать другой объект. Это удобно при работе с объектами наподобие пунктов заказа. Заказ отвечает за предоставление доступа к пунктам. Правда, в нашей ситуации такого объекта не видно. В таком случае я обычно создаю объект реестра, который должен служить точкой доступа. Для простоты здесь я сохраню его в виде статического поля в классе `Customer`, делая последний точкой доступа:

```
private static Dictionary _instances = new Hashtable();
```

Затем я принимаю решение о том, как именно создавать клиенты — “на лету” по запросу или заранее. Воспользуюсь последним способом. При запуске моего приложения я загружаю используемые клиенты (их список можно взять из базы данных или из файла). Для простоты я использую “прошитый” код, который впоследствии всегда можно будет изменить с помощью рефакторинга “Замена алгоритма” (с. 159).

```
class Customer...
    static void loadCustomers() {
        new Customer("Lemon Car Hire").store();
        new Customer("Associated Coffee Machines").store();
        new Customer("Bilston Gasworks").store();
    }
    private void store() {
        _instances.put(this.getName(), this);
    }
```

Теперь я модифицирую фабричный метод так, чтобы он возвращал заранее созданный клиент:

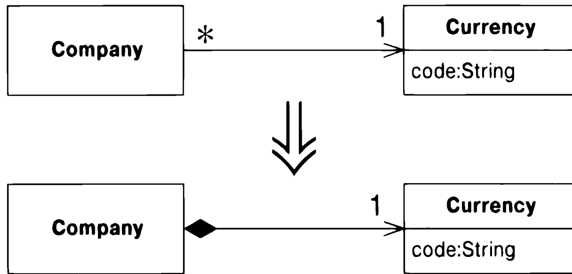
```
public static Customer create(String name) {
    return (Customer) _instances.get(name);
}
```

Поскольку метод `create` всегда возвращает уже существующий клиент, это должно быть очевидно из его имени, так что я прибегаю к рефакторингу “Переименование метода” (с. 290).

```
class Customer...
    public static Customer getNamed(String name) {
        return (Customer) _instances.get(name);
    }
```

Замена ссылки значением (Change Reference to Value)

Имеется небольшой неизменяемый и неудобный для управления объект-ссылка. *Превратите его в объект-значение.*



Мотивация

Как и в случае рефакторинга "Замена значения ссылкой" (с. 198), выбор между объектом-ссылкой и объектом-значением не всегда очевиден. Такого рода решения часто приходится менять на противоположные.

Переход от ссылки к значению может быть вызван проблемами, возникающими при работе с объектом-ссылкой. Объектами-ссылками необходимо тем или иным образом управлять. Всегда приходится запрашивать нужный объект у контроллера. Связи в памяти тоже могут оказаться неудобными в работе. В особенности полезны объекты-значения в распределенных и параллельных системах.

Важное свойство объектов-значений заключается в том, что они должны быть неизменяемыми. При каждом запросе, возвращающем значение одного из них, должен получаться один и тот же результат. Если это так, то не возникает проблем при наличии многих объектов, представляющих одну и ту же сущность. Если же значение изменяемое, то необходимо гарантировать при изменении любого из объектов соответствующее изменение всех остальных объектов, представляющих ту же самую сущность. Это настолько трудно, что проще сделать объект объектом-ссылкой.

Важно внести ясность в смысл слова *неизменяемый* (immutable). Если имеется класс, представляющий денежную сумму и содержащий тип валюты и ее количество, то обычно это объект с неизменяемым значением. Это не значит, что ваша зарплата не может измениться. Это значит, что для изменения зарплаты необходимо заменить существующий объект-значение денежной суммы другим объектом-значением денежной суммы, а не изменять значение в существующем объекте. Ваши связи с объектами зарплаты могут измениться, но сам объект-значение не изменяется.

Техника

- Убедитесь, что преобразуемый объект неизменяемый или может стать неизменяемым.

  ⇛ *Если в настоящее время объект не является неизменяемым, добейтесь с помощью рефакторинга "Удаление метода установки значения" (с. 317), чтобы он стал таковым.*

  Если преобразуемый объект нельзя сделать неизменяемым, данный рефакторинг следует отменить.
- Создайте методы `equals` и `hash`.
- Выполните компиляцию и тестирование.
- Рассмотрите возможность удаления фабричных методов и превращения конструктора в открытый метод.

Пример

Начну с класса валюты:

```
class Currency...
    private String _code;

    public String getCode() {
        return _code;
    }
    private Currency(String code) {
        _code = code;
    }
```

Вся функциональность этого класса состоит в хранении и возврате кода валюты. Это объект-ссылка, поэтому для получения экземпляра необходимо использовать код

```
Currency usd = Currency.get("USD");
```

Класс `Currency` поддерживает список экземпляров. Я не могу просто обратиться к конструктору (вот почему он закрытый).

```
new Currency("USD").equals(new Currency("USD")) // false
```

Чтобы преобразовать его в объект-значение, важно удостовериться в неизменяемости объекта. Если объект изменяем, не пытайтесь выполнить этот рефакторинг, поскольку изменяемое значение приводит к бесконечным изматывающим ссылкам.

В нашем случае объект неизменяем, поэтому следующим шагом будет определение метода `equals`:

```
public boolean equals(Object arg) {
    if (!(arg instanceof Currency)) return false;
    Currency other = (Currency) arg;
    return (_code.equals(other._code));
}
```

Определяя `equals`, я должен определить и метод `hashCode`. Самый простой способ сделать это — взять хеш-коды всех полей, используемых в методе `equals`, и выполнить над ними операцию побитового исключающего или (^). В нашем случае поле только одно, так что все очень просто:

```
public int hashCode() {
    return _code.hashCode();
}
```

После замены методов можно выполнить компиляцию и тестирование. Я должен заменить оба метода, так как иначе любая коллекция на основе хеширования, такая как `Hashtable`, `HashSet` или `HashMap`, может повести себя некорректно.

Теперь я могу создать сколько угодно одинаковых объектов. Я могу избавиться от любого поведения контроллера в классе и фабричном методе, и использовать конструктор, который теперь может быть открытым.

```
new Currency("USD").equals(new Currency("USD")) // true
```

Замена массива объектом (Replace Array with Object)

Имеется массив, некоторые элементы которого представляют собой разные сущности.

Замените массив объектом, в котором есть поле для каждого элемента.

```
String[] row = new String[3];
row [0] = "Liverpool";
row [1] = "15";
```

```
Performance row = new Performance();
row.setName("Liverpool");
row.setWins("15");
```

Мотивация

Массивы — распространенная структура организации данных, но их следует применять лишь для хранения коллекций сходных объектов в определенном порядке. Однако иногда можно столкнуться с хранением в массивах ряда различных объектов. Соглашения наподобие "первый элемент массива содержит имя человека" трудно запоминаются. Используя объекты, такую информацию можно передать с помощью имен полей или методов, а не запоминать ее и не полагаться на точность комментариев. Можно также инкапсулировать данную информацию и добавить к ней поведение с помощью рефакторинга "Перенос метода" (с. 162).

Техника

- Создайте новый класс для представления данных, содержащихся в массиве. Организуйте в нем открытое поле для массива.
- Модифицируйте пользователей массива так, чтобы они работали с новым классом.
- Выполните компиляцию и тестирование.
- Для каждого элемента массива по одному добавьте методы получения и установки значений. Именуйте методы доступа в соответствии с назначением элементов массива. Модифицируйте код клиентов так, чтобы они использовали созданные методы доступа. После каждого изменения выполняйте компиляцию и тестирование.
- После замены всех обращений к массиву методами сделайте массив закрытым.
- Выполните компиляцию.
- Для каждого элемента массива создайте поле в классе и измените методы доступа так, чтобы они использовали это поле.
- После изменения каждого элемента выполните компиляцию и тестирование.
- После замены всех элементов массива полями удалите массив.

Пример

Начну с массива, хранящего название спортивной команды, количество выигрышей и количество поражений, который определен следующим образом:

```
String[] row = new String[3];
```

Предполагается примерно следующее его использование в коде:

```
row [0] = "Liverpool";
row [1] = "15";
String name = row[0];
int wins = Integer.parseInt(row[1]);
```

Превращение массива в объект начинается с создания класса:

```
class Performance {}
```

На первом этапе в новый класс вводится открытый член-данные (да, это плохо, но все это будет исправлено).

```
public String[] _data = new String[3];
```

Далее я нахожу места, в которых создается массив и выполняется обращение к нему. Вместо кода создания массива я пишу

```
Performance row = new Performance();
```

Код, использующий массив, заменяется следующим:

```
row._data[0] = "Liverpool";
row._data[1] = "15";

String name = row._data[0];
int wins = Integer.parseInt(row._data[1]);
```

Теперь я по одному добавляю методы получения и установки с более содержательными именами. Начну с названия команды:

```
class Performance...
    public String getName() {
        return _data[0];
    }
    public void setName(String arg) {
        _data[0] = arg;
    }
```

Код, использовавший имя команды, теперь изменяется таким образом, чтобы он использовал методы доступа:

```
row.setName("Liverpool");
row._data [1] = "15";

String name = row.getName();
int wins = Integer.parseInt(row._data[1]);
```

То же можно проделать и со вторым элементом. Для упрощения я могу инкапсулировать приведение типа данных:

```
class Performance...
    public int getWins() {
        return Integer.parseInt(_data[1]);
    }
    public void setWins(String arg) {
        _data[1] = arg;
    }

Код клиента...
    row.setName("Liverpool");
    row.setWins("15");

    String name = row.getName();
    int wins = row.getWins();
```

Проделав эти действия для всех элементов, можно сделать массив закрытым:

```
private String[] _data = new String[3];
```

На этом главная часть рефакторинга — замена интерфейса — завершена. Однако полезно также заменить массив внутри класса. Это можно сделать, добавив поля для каждого элемента массива и изменив соответствующим образом методы доступа:

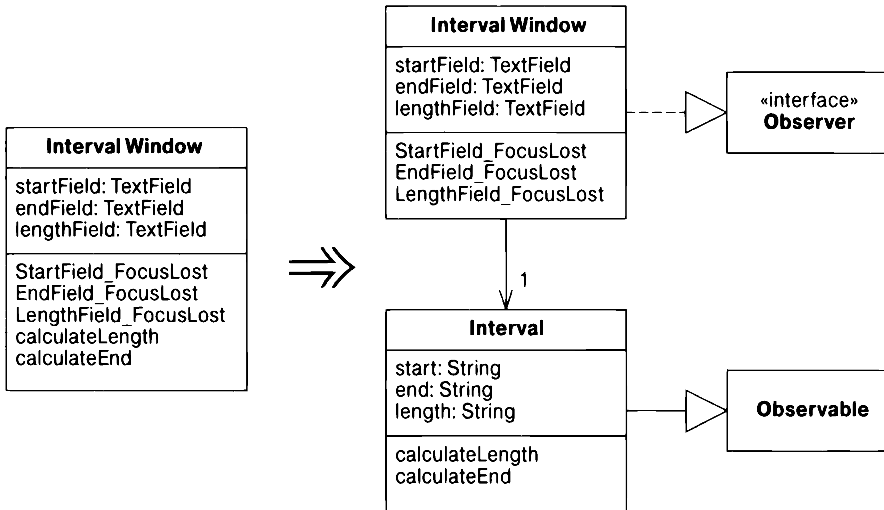
```
class Performance...
    public String getName() {
        return _name;
    }
    public void setName(String arg) {
        _name = arg;
    }

    private String _name;
```

По окончании данного преобразования для всех элементов массива последний можно удалить.

Дублирование видимых данных (Duplicate Observed Data)

Имеются данные предметной области приложения, доступные только в управляющем элементе графического интерфейса пользователя, доступ к которым требуется методам предметной области.

Скопируйте данные в объект предметной области. Создайте объект-наблюдатель, который будет синхронизировать эти два набора данных.



Мотивация

В хорошо организованной многоуровневой системе код интерфейса пользователя отделен от кода бизнес-логики. Так делается по ряду причин. Вы можете захотеть предоставить несколько различных интерфейсов пользователя для одной и той же бизнес-логики. Интерфейс пользователя, выполняющий обе функции, становится слишком сложным. Сопровождение и изменение объектов предметной области оказываются более простыми, если они отделены от графического интерфейса пользователя. Наконец разные части приложения могут разрабатывать разные программисты.

Хотя поведение можно легко разделить на части, это зачастую нельзя сделать с данными. Данные могут встраиваться в управляющий элемент графического интерфейса пользователя и иметь тот же смысл, что и данные в модели предметной области. Каркасы пользовательского интерфейса, начиная с MVC (model-view-controller — модель-представление-контроллер), используют многоуровневую систему, которая обеспечивает механизмы, предоставляющие эти данные и поддерживающие их синхронность.

Столкнувшись с кодом, разработанным на основе двухуровневого подхода, в котором бизнес-логика встроена в интерфейс пользователя, необходимо разделить поведение на части. В основном эта задача решается с помощью декомпозиции и перемещения методов. Однако просто переместить данные нельзя; их надо скопировать и обеспечить механизм синхронизации копий.

Техника

- Сделайте класс представления наблюдателем для класса предметной области [9].

  ⇛ *Если класса предметной области не существует, создайте его.*

  Если нет связи от класса представления к классу предметной области, поместите класс предметной области в поле класса представления.

- Примените рефакторинг “Самоинкапсуляция поля” (с. 190) к данным предметной области в классе графического интерфейса пользователя.
- Выполните компиляцию и тестирование.
- Добавьте вызов метода установки значения в обработчик события для обновления компонента текущим значением с использованием непосредственного доступа.

  ⇛ *Поместите метод в обработчик события, который обновляет значение компонента на основе его текущего значения. В этом нет особой необходимости; вы просто устанавливаете значение равным его текущему значению, но благодаря использованию метода установки вы обеспечиваете возможность выполнения любого необходимого поведения.*

  ⇛ *При внесении этих изменений не пользуйтесь методом получения значения компонента, вместо этого прибегая к непосредственному доступу. Позже метод получения будет извлекать значение из предметной области, которое изменяется только при выполнении метода установки.*

  Убедитесь, что механизм обработки событий срабатывает при выполнении тестового кода.

- Выполните компиляцию и тестирование.
- Определите данные и методы доступа в классе предметной области.

  ⇛ *Убедитесь, что метод установки в предметной области запускает механизм уведомления в шаблоне наблюдателя.*

  ⇛ *Используйте в предметной области тот же тип данных, что и в представлении (обычно это строка). Тип данных будет преобразован при проведении следующего рефакторинга.*

- Перенаправьте методы доступа для записи в поле предметной области.
- Измените метод обновления наблюдателя так, чтобы он копировал данные из поля объекта предметной области в элемент графического интерфейса пользователя.
- Выполните компиляцию и тестирование.

Пример

Начну с окна, представленного на рис. 8.1. Его поведение очень простое. Как только изменяется значение в одном из текстовых полей, тут же обновляются и остальные поля. При изменении полей Start или End вычисляется значение поля length; при изменении поля length вычисляется новое значение поля End.

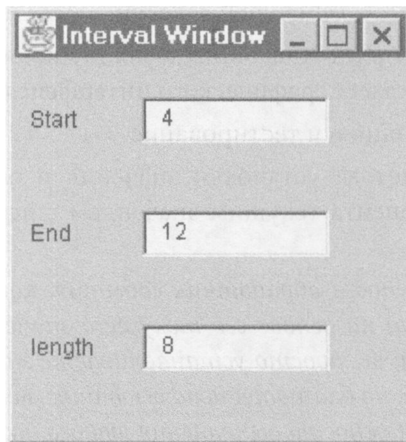


Рис. 8.1. *Простое окно графического интерфейса пользователя*

Все методы находятся в единственном классе `IntervalWindow`. Поля реагируют на потерю фокуса ввода.

```
public class IntervalWindow extends Frame...
    java.awt.TextField _startField;
    java.awt.TextField _endField;
    java.awt.TextField _lengthField;

    class SymFocus extends
        java.awt.event.FocusAdapter {
        public void
        focusLost(java.awt.event.FocusEvent event) {
            Object object = event.getSource();
            if (object == _startField)
                StartField_FocusLost(event);
            else if (object == _endField)
                EndField_FocusLost(event);
            else if (object == _lengthField)
                LengthField_FocusLost(event);
        }
    }
```

На потерю фокуса начальным полем слушатель реагирует вызовом `StartField_FocusLost` и вызовами `EndField_FocusLost` и `LengthField_Focus_Lost` при потере фокуса другими полями. Эти методы обработки событий имеют следующий вид:

```
void StartField_FocusLost(java.awt.event.FocusEvent event) {
    if (isNotInteger(_startField.getText()))
        _startField.setText("0");
    calculateLength();
}

void EndField_FocusLost(java.awt.event.FocusEvent event) {
    if (isNotInteger(_endField.getText()))
        _endField.setText("0");
    calculateLength();
}

void LengthField_FocusLost(java.awt.event.FocusEvent event) {
    if (isNotInteger(_lengthField.getText()))
        _lengthField.setText("0");
    calculateEnd();
}
```

Если вам интересно, почему я спроектировал окно таким образом, отвечу, что это простейший способ, который был предложен мне моей интегрированной средой разработки (Cafe).

При появлении нецифрового символа в любом поле в него вносится нуль и вызывается соответствующая подпрограмма вычислений:

```
void calculateLength() {
    try {
        int start = Integer.parseInt(_startField.getText());
        int end = Integer.parseInt(_endField.getText());
        int length = end - start;
        _lengthField.setText(String.valueOf(length));
    } catch (NumberFormatException e) {
        throw new RuntimeException("Ошибка - некорректный формат числа");
    }
}
void calculateEnd() {
    try {
        int start = Integer.parseInt(_startField.getText());
        int length = Integer.parseInt(_lengthField.getText());
        int end = start + length;
        _endField.setText(String.valueOf(end));
    } catch (NumberFormatException e) {
        throw new RuntimeException("Ошибка - некорректный формат числа");
    }
}
```

Моей задачей становится отделение внутренней логики от логики графического интерфейса пользователя. По существу это означает перенос calculateLength и calculateEnd в отдельный класс предметной области. Чтобы это сделать, я должен обращаться к данным, *минуя* класс окна. Этого можно добиться, дублируя данные в классе предметной области и синхронизируя их с данными графического интерфейса пользователя. Эта задача и описывается рефакторингом "Дублирование видимых данных".

В данный момент класса предметной области еще нет, поэтому надо создать соответствующий (пустой) класс:

```
class Interval extends Observable {}
```

В классе окна должна быть ссылка на этот новый класс предметной области.

```
private Interval _subject;
```

Затем следует соответствующим образом инициализировать это поле и сделать окно наблюдателем интервала. Для этого поместим в конструктор окна следующий код:

```
_subject = new Interval();
_subject.addObserver(this);
update(_subject, null);
```

Я предпочитаю помещать этот код в конце конструктора класса. Вызов update гарантирует, что при дублировании данных в классе предметной области графический интерфейс пользователя инициализируется данными из этого класса. Для этого я должен объявить, что окно реализует интерфейс Observer:

```
public class IntervalWindow extends Frame implements Observer
```

Для реализации наблюдателя я должен создать метод обновления update. Пока что он может быть пустым:

```
public void update(Observable observed, Object arg) {
}
```

Сейчас можно выполнить компиляцию и тестирование. Никакая реальная модификация еще выполнена, но ошибки можно допустить и в простейших действиях.

Теперь пора заняться переносом полей. Как всегда, я вношу изменения в поля по одному. Начну с поля конца интервала. Сначала следует применить рефакторинг "Самоинкапсуляция поля" (с. 190). Текстовые поля обновляются с помощью методов getText и setText. Я создаю методы доступа, которые вызывают указанные методы:

```
String getEnd() {
    return _endField.getText();
}
void setEnd(String arg) {
    _endField.setText(arg);
}
```

Далее я нахожу все ссылки на `_endField` и заменяю их соответствующими методами доступа:

```
void calculateLength() {
    try {
        int start = Integer.parseInt(_startField.getText());
        int end = Integer.parseInt(getEnd());
        int length = end - start;
        _lengthField.setText(String.valueOf(length));
    } catch (NumberFormatException e) {
       throw new RuntimeException("Ошибка - некорректный формат числа");
    }
}
void calculateEnd() {
    try {
        int start = Integer.parseInt(_startField.getText());
        int length = Integer.parseInt(_lengthField.getText());
        int end = start + length;
        setEnd(String.valueOf(end));
    } catch (NumberFormatException e) {
       throw new RuntimeException("Ошибка - некорректный формат числа");
    }
}
void EndField_FocusLost(java.awt.event.FocusEvent event) {
    if (isNotInteger(getEnd()))
        setEnd("0");
    calculateLength();
}
```

Это обычный процесс рефакторинга “Самоинкапсуляция поля” (с. 190). Однако при работе с графическим интерфейсом пользователя возникает проблема. Пользователь может изменить значение поля непосредственно, не вызывая `setEnd`. Поэтому следует поместить вызов `setEnd` в обработчик события графического интерфейса пользователя. Этот вызов заменяет старое значение поля `End` текущим. Конечно, сейчас он ничего не делает, но тем самым гарантируется, что ввод пользователя будет пропущен через метод установки:

```
void EndField_FocusLost(java.awt.event.FocusEvent event) {
    setEnd(_endField.getText());
    if (isNotInteger(getEnd()))
        setEnd("0");
    calculateLength();
}
```

В этом вызове я не использую getEnd, а обращаюсь к полю непосредственно. Я делаю это потому, что после продолжения рефакторинга getEnd будет получать значение от объекта предметной области, а не от поля. Сейчас вызов метода означал бы, что всякий раз, когда пользователь менял бы значение поля, этот код возвращал бы его обратно, так что здесь мне необходим непосредственный доступ. Теперь я могу выполнить компиляцию и протестировать инкапсулированное поведение.

Далее я добавлю в класс предметной области поле конца:

```
class Interval...
    private String _end = "0";
```

Я инициализирую его тем же значением, которым оно инициализируется в графическом интерфейсе пользователя. Теперь я добавляю методы получения и установки:

```
class Interval...
    String getEnd() {
        return _end;
    }
    void setEnd(String arg) {
        _end = arg;
        setChanged();
        notifyObservers();
    }
```

Поскольку я использую проектный шаблон наблюдателя, я должен добавить код уведомления в метод установки. Я использую строку, а не (более логичное) числовое значение. Просто я хочу, чтобы изменений было как можно меньше. Когда данные будут успешно продублированы, я смогу сменить тип данных внутреннего представления на целочисленный.

Теперь перед дублированием можно еще раз выполнить компиляцию и тестирование. Вся проведенная работа существенно снижает риск предстоящего сложного шага.

Сначала я модифицирую методы доступа IntervalWindow, чтобы использовать в них Interval.

```
class IntervalWindow...
    String getEnd() {
        return _subject.getEnd();
    }
    void setEnd(String arg) {
        _subject.setEnd(arg);
    }
```

Мне также нужно обновить update, чтобы гарантировать реакцию графического интерфейса пользователя на уведомление:

```
class IntervalWindow...
    public void update(Observable observed, Object arg) {
        _endField.setText(_subject.getEnd());
    }
```

Это еще одно место, где необходим непосредственный доступ. Вызов метода установки привел бы здесь к бесконечной рекурсии.

Теперь можно вновь выполнить компиляцию и тестирование и убедиться, что данные дублируются корректно.

Далее нужно повторить эти действия для двух оставшихся полей, после чего с помощью рефакторинга “Перенос метода” (с. 162) перенести calculateEnd и calculateLength в класс интервала. Так будет получен класс предметной области, содержащий все поведение и данные предметной области отдельно от кода графического интерфейса пользователя.

После этого можно попытаться избавиться от класса графического интерфейса пользователя вовсе. Если это старый класс AWT (Abstract Windows Toolkit), то внешний вид можно улучшить с использованием Swing (причем Swing лучше работает в плане координации). Построить графический интерфейс пользователя Swing можно поверх класса предметной области. Успешно сделав это, можно избавиться от старого класса графического интерфейса пользователя.

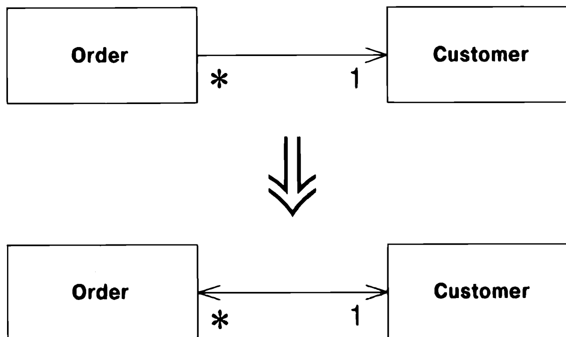
Использование слушателей событий

Рефакторинг “Дублирование видимых данных” применим и тогда, когда вы используете слушателей событий вместо наблюдателя и наблюдаемого объекта. В таком случае вам надо создать слушателя и событие в модели предметной области (либо использовать классы AWT, если вас не смущает внесение зависимостей). Объект предметной области должен регистрировать слушателей точно так же, как это делает наблюдаемый класс, и посылать им событие при изменениях, как в методе update. После этого окно интервала может использовать внутренний класс для реализации интерфейса слушателя и вызова соответствующих методов обновления.

Замена однонаправленной связи двунаправленной (Change Unidirectional Association to Bidirectional)

Имеется два класса, взаимно использующие функциональность один другого, но между ними есть связь только в одном направлении.

Добавьте обратные указатели и измените модификаторы так, чтобы они обновляли оба множества.



Мотивация

Иногда приходится сталкиваться с ситуацией, когда изначально имеются два класса, один из которых обращается к другому. Со временем обнаруживается, что клиенту класса, к которому выполняются обращения, нужно знать объекты, которые к нему обращаются. Фактически это означает перемещение по указателям в обратном направлении. Однако в имеющемся виде это невозможно, потому что указатели по своей природе односторонние. Зачастую удается просто обойти эту проблему, найдя другой маршрут. Это может вызвать некоторое увеличение объема вычислений, но оно оправданно, и в вашем классе может быть создан метод, использующий это поведение. Однако иногда такой способ затруднителен, так что желательно организовать двустороннюю ссылку, которую часто называют *обратным указателем* (back pointer). Увы, без навыков работы с обратными указателями в них легко запутаться; однако если привыкнуть к этой идиоме, она перестает казаться сложной.

Сложность и запутанность этой идиомы требует проведения частого тестирования, по крайней мере в период освоения данной технологии. Обычно я не утруждаю себя тестированием методов доступа (здесь риск ошибиться не очень велик), так что данный рефакторинг — один из немногих, для которых добавляются тесты.

Этот рефакторинг использует обратные указатели для реализации двунаправленности. Другие методы, такие как объекты-связи (link objects), требуют применения других рефакторингов.

Техника

- Добавьте поле для обратного указателя.
- Определите, какой класс будет управлять связями.
- Создайте вспомогательный метод на подчиненном конце связи. Дайте ему имя, ясно указывающее ограниченность его применения.
- Если существующий модификатор находится на управляющем конце, модифицируйте его так, чтобы он обновлял обратные указатели.
- Если существующий модификатор находится на управляемом конце, создайте управляющий метод на управляющем конце и вызывайте его из существующего модификатора.

Пример

Вот простая программа, в которой заказ “обращается” к клиенту:

```
class Order...
    Customer getCustomer() {
        return _customer;
    }
    void setCustomer(Customer arg) {
        _customer = arg;
    }
    Customer _customer;
```

В классе `Customer` нет ссылки на `Order`.

Я начинаю рефакторинг с добавления поля в класс `Customer`. Поскольку у клиента может быть несколько заказов, это поле будет коллекцией. Для реализации ограничения, чтобы один и тот же заказ не мог встречаться в коллекции клиента дважды, в качестве коллекции выбирается множество:

```
class Customer {
    private Set _orders = new HashSet();
```

Теперь нужно решить, какой из классов ответственен за связь. Я предпочитаю возлагать ответственность на один класс, так как это позволяет хранить всю логику управления связью в одном месте. Процедура принятия решения выглядит следующим образом.

1. Если оба объекта являются объектами ссылок, а связь представляет собой связь "один ко многим", то управляющим следует делать объект, содержащий одну ссылку. (То есть если у одного клиента несколько заказов, то связью управляет заказ.)
2. Если один объект является компонентом другого (связь "целое–часть"), то управлять связью должен составной объект.
3. Если оба объекта представляют собой объекты ссылок, а связь представляет собой связь "многие ко многим", то управляющим можно произвольно выбирать как класс заказа, так и класс клиента.

Поскольку в нашем случае за связь отвечает заказ, я должен добавить к клиенту вспомогательный метод, предоставляющий непосредственный доступ к коллекции заказов. С его помощью модификатор заказа будет синхронизировать оба множества указателей. Я дам этому методу имя `friendOrders`, чтобы указать, что его следует применять только в данном особом случае. Я также ограничу его область видимости пакетом (если это возможно). Если же второй класс находится в другом пакете, то я буду вынужден объявить его с модификатором видимости `public`:

```
class Customer...
    Set friendOrders() {
        /** Должен использоваться только в Order
            при модификации связи */
        return _orders;
    }
```

Теперь я изменяю модификатор так, чтобы он обновлял обратные указатели:

```
class Order...
    void setCustomer(Customer arg) ...
        if (_customer != null) _customer.friendOrders().remove(this);
        _customer = arg;
        if (_customer != null) _customer.friendOrders().add(this);
    }
```

Точный код в управляющем модификаторе зависит от кратности связи. Если `customer` не может быть `null`, можно обойтись без соответствующей проверки его значения, но аргумент надо проверять. Базовая схема, впрочем, всегда одинакова: сначала от нового объекта требуется, чтобы он уничтожил указатель на исходный объект, затем ваш указатель устанавливается на новый объект, после чего от нового объекта требуется, чтобы он добавил ссылку на ваш объект.

Если вы хотите модифицировать связь через `Customer`, он может вызывать управляющий метод:

```
class Customer...
    void addOrder(Order arg) {
        arg.setCustomer(this);
    }
```

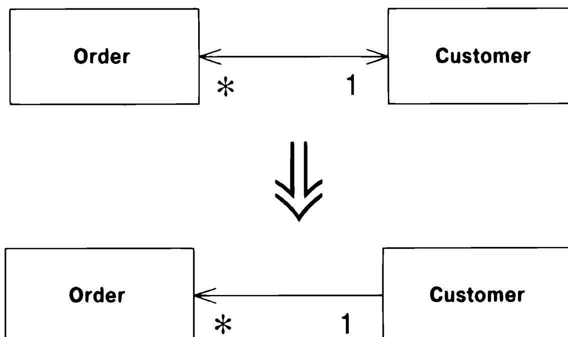
Если же у заказа может быть несколько клиентов, то это случай отношения “многие ко многим”, и методы при этом имеют следующий вид:

```
class Order... // Управляющие методы
    void addCustomer(Customer arg) {
        arg.friendOrders().add(this);
        _customers.add(arg);
    }
    void removeCustomer(Customer arg) {
        arg.friendOrders().remove(this);
        _customers.remove(arg);
    }

class Customer...
    void addOrder(Order arg) {
        arg.addCustomer(this);
    }
    void removeOrder(Order arg) {
        arg.removeCustomer(this);
    }
```

Замена двунаправленной связи однонаправленной (Change Bidirectional Association to Unidirectional)

Имеется двунаправленная связь, но одному из классов больше не нужны функции другого класса.

Удалить ненужный конец связи.



Мотивация

Двунаправленные связи удобны, но имеют свою цену. Этой ценой является дополнительная сложность поддержки двунаправленных связей и корректности создания и удаления объектов. Двунаправленные связи непривычны многим программистам, и потому зачастую оказываются источником ошибок.

Большое количество двусторонних ссылок может оказаться источником "зомби" — объектов, которые не нужны и должны быть уничтожены, но все еще существуют из-за наличия ссылок на них.

Двунаправленные связи вносят взаимозависимость между двумя классами. Любое изменение в одном классе может вызвать изменение в другом. Если классы находятся в разных пакетах, взаимозависимость возникает между пакетами. Наличие большого числа взаимозависимостей приводит к сильно связанным системам, в которых любые малозначительные изменения вызывают множество непредсказуемых последствий.

Двунаправленными связями следует пользоваться только тогда, когда это действительно необходимо. Если вы обнаружили, что двунаправленная связь больше не несет никакой нагрузки, обрубите лишний конец.

Техника

- Изучите все места чтения поля с указателем, которые вы хотите удалить, и убедитесь в том, что удаление можно выполнить без ущерба для системы.
  - ⇒ *Рассмотрите непосредственное чтение, а также методы, которые вызывают эти читающие методы.*
  - ⇒ *Посмотрите, нельзя ли определить другой объект без использования указателя. Если это возможно, то, применив рефакторинг "Замена алгоритма" (с. 159) к методу получения значения поля, можно разрешить клиентам пользоваться им даже при отсутствии указателя.*
  - ⇒ *Рассмотрите возможность передачи объекта в качестве аргумента для всех методов, использующих это поле.*
- Если клиентам требуется использовать метод доступа к полю, выполните рефакторинг "Самоинкапсуляция поля" (с. 190) с последующим рефакторингом "Замена алгоритма" (с. 159) над методом доступа. Выполните компиляцию и тестирование.
- Если метод доступа к полю клиентам не нужен, поочередно модифицируйте клиентов таким образом, чтобы они получали объект в поле другим способом. Выполняйте компиляцию и тестирование после каждой модификации.

- Когда в приложении не останется кода, читающего значение поля, удалите весь код обновления поля и само поле.

  ⇒ *Если присваивание значения полю происходит во многих местах, выполните рефакторинг "Самоинкапсуляция поля" (с. 190), чтобы все присваивания использовали один и тот же метод установки значения поля. Выполните компиляцию и тестирование. Модифицируйте метод установки значения таким образом, чтобы у него было пустое тело. Выполните компиляцию и тестирование. Если они пройдут удачно, удалите поле, метод установки и все вызовы этого метода.*

- Выполните компиляцию и тестирование.

Пример

Начну с того места, где я завершил пример при описании рефакторинга "Замена однонаправленной связи двунаправленной" (с. 216). Итак, у нас имеются клиент и заказ с двусторонней связью:

```
class Order...
    Customer getCustomer() {
        return _customer;
    }
    void setCustomer(Customer arg) {
        if (_customer != null) _customer.friendOrders().remove(this);

        _customer = arg;

        if (_customer != null) _customer.friendOrders().add(this);
    }
    private Customer _customer;

class Customer...
    void addOrder(Order arg) {
        arg.setCustomer(this);
    }
    private Set _orders = new HashSet();
    Set friendOrders() {
        /** Должен использоваться только в Order */
        return _orders;
    }
```

Я выяснил, что в моем приложении заказы невозможны, пока нет клиента, так что я решил разорвать связь от заказа к клиенту.

Самое трудное в данном рефакторинге — это проверка возможности его проведения. Необходимо точно убедиться в безопасности выполнения рефакторинга; выполнить сам рефакторинг довольно легко. Проблема в наличии кода,

зависящего от наличия поля клиента. Чтобы удалить поле, надо предоставить этому коду альтернативу.

Я начинаю с изучения всех операций чтения поля и всех методов, которые эти операции используют. Нет ли иного способа предоставить объект клиента? Зачастую это означает, что надо передать клиент в качестве аргумента операции. Вот упрощенный пример: код

```
class Order...
    double getDiscountedPrice() {
    return getGrossPrice() * (1 - _customer.getDiscount());
}
```

заменяется кодом

```
class Order...
    double getDiscountedPrice(Customer customer) {
        return getGrossPrice() * (1 - customer.getDiscount());
    }
```

Особенно хорошо это работает, когда поведение вызывается клиентом, поскольку передать себя в качестве аргумента для него легче всего. Таким образом, код

```
class Customer...
    double getPriceFor(Order order) {
        Assert.isTrue(_orders.contains(order));
        // Рефакторинг "Введение утверждения"
        return order.getDiscountedPrice();
```

принимает вид

```
class Customer...
    double getPriceFor(Order order) {
        Assert.isTrue(_orders.contains(order));
        return order.getDiscountedPrice(this);
    }
```

Еще одна альтернатива, которую следует рассмотреть — изменение метода доступа к данным, которое позволит получить клиент, не используя поле. Тогда к телу метода `Order.getCustomer` можно будет применить рефакторинг “Замена алгоритма” (с. 159) и сделать что-то наподобие следующего:

```
Customer getCustomer() {
    Iterator iter = Customer.getInstances().iterator();
    while (iter.hasNext()) {
        Customer each = (Customer)iter.next();
        if (each.containsOrder(this)) return each;
    }
    return null;
}
```

Медленно, но работает. В контексте базы данных работа такого кода может даже не быть такой уж медленной, если применить запрос к базе данных. Если в классе заказа есть методы, работающие с полем клиента, то с помощью рефакторинга “Самоинкапсуляция поля” (с. 190) их можно изменить так, чтобы они использовали метод `getCustomer`.

Если метод доступа к полю сохранен, то согласно интерфейсу связь остается двунаправленной, но согласно реализации она оказывается однонаправленной. Я удаляю обратный указатель, но сохраняю взаимозависимость между двумя классами.

При замене метода получения все остальные изменения я оставляю на более позднее время. В противном случае я должен поочередно изменить вызывающие методы так, чтобы они получали клиент из другого источника. После внесения каждого такого изменения я выполняю компиляцию и тестирование. На практике этот процесс обычно осуществляется довольно быстро. Если бы этот процесс был сложным, я не использовал бы данный рефакторинг.

Избавившись от чтения поля, можно заняться записью в него. Для этого надо лишь убрать все присваивания полю, а затем удалить последнее. Поскольку никто его больше не читает, это удаление не должно иметь никакого значения для работоспособности программы.

Замена магического числа символической константой (Replace Magic Number with Symbolic Constant)

Есть числовой литерал, имеющий определенный смысл.

Создайте константу, присвойте ей имя в соответствии со смыслом константы и замените ею число.

```
double potentialEnergy(double mass, double height) {
    return mass * 9.81 * height;
}
```

```
double potentialEnergy(double mass, double height) {
    return mass * GRAVITATIONAL_CONSTANT * height;
}
static final double GRAVITATIONAL_CONSTANT = 9.81;
```

Мотивация

Магические числа являются одной из древнейших проблем программирования. Это числа с особыми значениями, смысл которых обычно не очевиден. Магические числа очень неприятны, когда нужно обратиться к логически одному и тому же числу в разных местах кода. Особым кошмаром становится их изменение. Но даже если эти числа не требуется изменять, понять, как они работают и что означают, можно лишь с большим трудом.

Многие языки программирования позволяют объявлять и использовать константы. При этом не наносится ущерб производительности, зато значительно улучшается удобочитаемость кода.

Прежде чем выполнять данный рефакторинг, всегда имеет смысл поискать возможную альтернативу. Посмотрите, как именно используется магическое число. Зачастую удается найти лучший способ его применения. Если оно представляет собой код типа, попробуйте выполнить рефакторинг “Замена кода типа классом” (с. 236). Если же это число — размер массива, при обходе массива используйте вместо него `anArray.length`.

Техника

- Объявите константу и инициализируйте ее значением магического числа.
- Найдите все вхождения магического числа в исходный текст.
- Проверьте, согласуется ли магическое число с использованием константы; если согласуется, замените магическое число константой.
- Выполните компиляцию.
- После замены всех магических чисел выполните компиляцию и тестирование. Все должно работать так, как будто ничего и не изменялось.

  ⇛ *Хорошим тестом будет проверка легкости изменения константы. Для этого может потребоваться изменить некоторые ожидаемые результаты в соответствии с новым значением. Этот способ применим не всегда, но если его можно использовать, это удачный метод проверки.*

Инкапсуляция поля (Encapsulate Field)

Имеется открытое поле.

Сделайте его закрытым и предоставьте методы доступа к нему.

```
private int _low, _high;
boolean includes(int arg) {
    return arg >= _low && arg <= _high;
}
```

```
private int _low, _high;
boolean includes(int arg) {
    return arg >= getLow() && arg <= getHigh();
}
int getLow() {
    return _low;
}
int getHigh() {
    return _high;
}
```

Мотивация

Одним из базовых принципов объектно-ориентированного программирования является инкапсуляция, или сокрытие данных. Он гласит, что данные никогда не должны быть общедоступными. Если сделать данные открытыми, другие объекты смогут читать и изменять их значения без ведома объекта — владельца этих данных. Таким образом, данные оказываются отделенными от поведения.

Это плохо, потому что снижает модульность программы. Когда данные и использующее их поведение сгруппированы, модифицировать код становится проще, поскольку весь изменяемый код расположен в одном месте, а не разбросан по всей программе.

Данный рефакторинг начинает процесс сокрытия данных и создания методов доступа, но это лишь первый шаг. Класс, в котором есть только методы доступа, — это “немой” класс, который не позволяет использовать преимущества объектно-ориентированного программирования, так что такой объект оказывается попросту потерянным. Выполнив описываемый в этом разделе рефакторинг, я ищу методы, которые используют вновь созданные методы доступа, в надежде перенести их в новый объект с помощью рефакторинга “Перенос метода” (с. 162).

Техника

- Создайте методы получения и установки значений поля.
- Найдите за пределами класса весь клиентский код, обращающийся к данному полю. Если клиент использует значение поля, замените обращение к

нему вызовом метода получения этого значения. Если клиент изменяет значение поля, замените его вызовом метода установки значения поля.

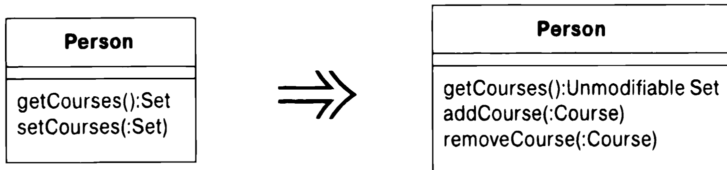
⇛ *Если поле представляет собой объект, а клиент вызывает метод, модифицирующий этот объект, — это следует рассматривать как использование поля, а не изменение его значения. Метод установки значения следует применять только для замены присваивания.*

- После каждого изменения выполните компиляцию и тестирование.
- После модификации всех клиентов объявите поле закрытым.
- Выполните компиляцию и тестирование.

Инкапсуляция коллекции (Encapsulate Collection)

Метод возвращает коллекцию.

Сделайте возвращаемое значение представлением, доступным только для чтения, и предоставьте методы добавления/удаления элементов.



Мотивация

Часто класс содержит коллекцию экземпляров. Эта коллекция может быть массивом, списком, множеством или вектором. Зачастую при этом имеются методы получения и установки значений для коллекции.

Однако коллекции должны использовать протокол, немного отличающийся от используемого другими типами данных. Метод получения не должен возвращать сам объект коллекции, потому что это позволило бы клиентам изменять содержимое коллекции без ведома класса-владельца. Кроме того, такое раскрытие клиентам внутреннего строения структур данных объекта является излишним. Метод получения для такого многозначного атрибута, как коллекция, должен возвращать значение, которое не позволит ни изменять коллекцию, ни получать лишнюю информацию об используемой для нее структуре. Как это сделать, зависит от используемой версии Java.

Кроме того, не должно быть метода, который выполнял бы присваивание коллекции; вместо этого класс следует обеспечить операциями добавления и

удаления элементов. Благодаря такому подходу объект-владелец получает контроль над добавлением и удалением элементов коллекции.

Описанный протокол обеспечивает корректную инкапсуляцию коллекции, уменьшающую связывание владеющего ею класса с клиентами.

Техника

- Создайте методы для добавления и удаления элементов коллекции.
- Присвойте полю в качестве начального значения пустую коллекцию.
- Выполните компиляцию.
- Найдите код, вызывающий методы установки значений. Измените метод установки значения так, чтобы он использовал операции добавления и удаления элементов (или сделайте так, чтобы клиенты сами вызывали эти операции).
  - ⇒ *Методы установки используются в двух случаях: когда коллекция пуста и когда метод заменяет непустую коллекцию.*

    Может оказаться желательным применение рефакторинга “Переименование метода” (с. 290). Измените имя с обозначающего установку, на имя, свидетельствующее об инициализации или замене.
- Выполните компиляцию и тестирование.
- Найдите код, вызывающий методы получения значений и при этом модифицирующий коллекцию. Измените его так, чтобы он использовал методы добавления и удаления элементов. После каждого изменения выполняйте компиляцию и тестирование.
- Когда все вызовы метода получения значения, модифицирующие коллекцию, будут заменены, измените метод получения так, чтобы он возвращал представление, доступное только для чтения.
  - ⇒ *В Java 2 это достигается с помощью соответствующего немодифицируемого представления коллекции.*
  - ⇒ *В Java 1.1 должна возвращаться копия коллекции.*
- Выполните компиляцию и тестирование.
- Найдите код, работающий с методом получения значения коллекции. Поищите код, который должен находиться в объекте-владельце. С помощью рефакторингов “Извлечение метода” (с. 132) и “Перенос метода” (с. 162) перенесите этот код в объект-владелец.

Для Java 2 на этом работа завершена. Но в Java 1.1 клиенты могут предпочесть использовать перечисление. Чтобы предоставить им перечисление, выполните следующее.

- Измените имя текущего метода получения значения и создайте новый метод получения значения, возвращающий перечисление. Найдите пользователей старого метода и измените их код так, чтобы они использовали один из новых методов.

  ⇛ *Если такой переход слишком большой, примените к старому методу получения рефакторинг "Переименование метода" (с. 290), создайте новый метод, возвращающий перечисление, и измените код вызывающих объектов, чтобы в нем использовался новый метод.*

- Выполните компиляцию и тестирование.

Примеры

В Java 2 добавлена целая новая группа классов для работы с коллекциями. При этом не просто добавлены новые классы, но изменился и сам стиль работы с коллекциями. Поэтому способ инкапсуляции коллекции различен в случае использования коллекций Java 2 и коллекций Java 1.1. Я начну с описания подхода Java 2, поскольку полагаю, что более функциональные коллекции Java 2 вскоре вытеснят коллекции Java 1.1.

Пример: Java 2

Некто слушает курс лекций. Наш курс довольно прост:

```
class Course...
    public Course(String name, boolean isAdvanced) {
        ...
    };
    public boolean isAdvanced() {
        ...
    };
```

Я не намерен обременять класс `Course` чем-то еще, так как нас интересует класс `Person`:

```
class Person...
    public Set getCourses() {
        return _courses;
    }
    public void setCourses(Set arg) {
        _courses = arg;
    }
    private Set _courses;
```

При использовании данного интерфейса клиенты добавляют курсы с помощью кода наподобие следующего:

```
Person kent = new Person();
Set s = new HashSet();
s.add(new Course("Smalltalk Programming", false));
s.add(new Course("Appreciating Single Malts", true));
kent.setCourses(s);
Assert.equals(2, kent.getCourses().size());
Course refact = new Course("Refactoring", true);
kent.getCourses().add(refact);
kent.getCourses().add(new Course("Brutal Sarcasm", false));
Assert.equals(4, kent.getCourses().size());
kent.getCourses().remove(refact);
Assert.equals(3, kent.getCourses().size());
```

Клиент, желающий узнать об углубленных курсах, может сделать это следующим образом:

```
Iterator iter = person.getCourses().iterator();
int count = 0;
while (iter.hasNext()) {
    Course each = (Course)iter.next();
    if (each.isAdvanced()) count ++;
}
```

Первое, что я хочу сделать, — это создать корректные методы модификации этой коллекции и выполнить компиляцию:

```
class Person
    public void addCourse(Course arg) {
        _courses.add(arg);
    }
    public void removeCourse(Course arg) {
        _courses.remove(arg);
    }
```

Жизнь станет проще, если я также проинициализирую поле:

```
private Set _courses = new HashSet();
```

Теперь рассмотрим пользователей метода установки значения. Если клиентов много и этот метод интенсивно используется, необходимо заменить тело метода установки так, чтобы в нем использовались операции добавления и удаления элементов. Сложность этого процесса зависит от того, каким образом используется метод установки значения. Возможны два варианта. В простейшем случае клиент использует метод установки для инициализации значений, т.е. до того, как будет

вызван метод установки значения, никаких курсов не имеется. В этом случае я изменяю тело метода установки значения так, чтобы он использовал метод добавления:

```
class Person...
    public void setCourses(Set arg) {
        Assert.isTrue(_courses.isEmpty());
        Iterator iter = arg.iterator();
        while (iter.hasNext()) {
            addCourse((Course) iter.next());
        }
    }
```

После такого изменения тела метода имеет смысл с помощью рефакторинга "Переименование метода" (с. 290) сделать намерения более ясными.

```
public void initializeCourses(Set arg) {
    Assert.isTrue(_courses.isEmpty());
    Iterator iter = arg.iterator();
    while (iter.hasNext()) {
        addCourse((Course) iter.next());
    }
}
```

В общем случае я должен сначала прибегнуть к методу удаления и убрать все элементы из коллекции, после чего добавить новые. Однако я обнаружил, что такая ситуация встречается достаточно редко (как зачастую бывает с общими случаями).

Если известно, что нет никакого дополнительного поведения при добавлении элементов во время инициализации, можно убрать цикл и использовать метод `addAll`.

```
public void initializeCourses(Set arg) {
    Assert.isTrue(_courses.isEmpty());
    _courses.addAll(arg);
}
```

Просто присвоить значение множеству нельзя, даже если предыдущее множество было пустым. Если клиент модифицирует множество после его передачи в качестве аргумента, это является нарушением принципа инкапсуляции. Поэтому я должен сделать копию множества.

Если клиенты просто создают множество и пользуются методом установки значения, я могу заставить их использовать методы добавления и удаления элементов непосредственно и полностью удалить метод установки значения. Например, код

```
Person kent = new Person();
Set s = new HashSet();
s.add(new Course("Smalltalk Programming", false));
s.add(new Course("Appreciating Single Malts", true));
kent.initializeCourses(s);
```

превращается в код

```
Person kent = new Person();
kent.addCourse(new Course("Smalltalk Programming", false));
kent.addCourse(new Course("Appreciating Single Malts", true));
```

Теперь я начинаю поиск пользователей метода получения значения. В первую очередь, меня интересуют случаи модификации коллекции с применением этого метода, например:

```
kent.getCourses().add(new Course ("Brutal Sarcasm", false));
```

Я должен заменить такой код вызовом нового метода:

```
kent.addCourse(new Course ("Brutal Sarcasm", false));
```

После того как я выполню такую замену везде в коде, проверить, не пропущено ли что-то и не выполняется ли где-нибудь модификация посредством метода получения значения, можно, изменив его тело так, чтобы метод возвращал немодифицируемое представление:

```
public Set getCourses() {
    return Collections.unmodifiableSet(_courses);
}
```

Теперь коллекция инкапсулирована. Никто не сможет изменить элементы коллекции, кроме как посредством методов класса `Person`.

Перемещение поведения в класс

Итак, у меня есть корректный интерфейс. Теперь я хочу взглянуть на пользователей метода получения значения и найти код, который должен находиться в классе `Person`. Такой код, как

```
Iterator iter = person.getCourses().iterator();
int count = 0;
while (iter.hasNext()) {
    Course each = (Course) iter.next();
    if (each.isAdvanced()) count ++;
}
```

лучше разместить в классе Person, потому что он использует только данные класса Person. Сначала я применю к этому коду рефакторинг “Извлечение метода” (с. 132):

```
int numberOfAdvancedCourses(Person person) {
    Iterator iter = person.getCourses().iterator();
    int count = 0;
    while (iter.hasNext()) {
        Course each = (Course) iter.next();
        if (each.isAdvanced()) count ++;
    }
    return count;
}
```

Затем с помощью рефакторинга “Перенос метода” (с. 162) я перенесу его в класс Person:

```
class Person...
    int numberOfAdvancedCourses() {
        Iterator iter = getCourses().iterator();
        int count = 0;
        while (iter.hasNext()) {
            Course each = (Course) iter.next();
            if (each.isAdvanced()) count ++;
        }
        return count;
    }
```

Часто в коде можно встретить такой вызов:

```
kent.getCourses().size()
```

Его можно заменить более удобочитаемым:

```
kent.numberOfCourses()

class Person...
    public int numberOfCourses() {
        return _courses.size();
    }
```

Несколько лет назад я был бы обеспокоен тем, что такое перемещение поведения в Person ведет к разбуханию класса. Но практика показывает, что обычно никаких неприятностей при этом не возникает.

Пример: Java 1.1

Во многих отношениях ситуация в случае Java 1.1 аналогична ситуации с Java 2. Я воспользуюсь тем же примером, но в этот раз с вектором:

```
class Person...
    public Vector getCourses() {
        return _courses;
    }
    public void setCourses(Vector arg) {
        _courses = arg;
    }
    private Vector _courses;
```

И вновь я начинаю с создания методов модификации и инициализации поля:

```
class Person
    public void addCourse(Course arg) {
        _courses.addElement(arg);
    }
    public void removeCourse(Course arg) {
        _courses.removeElement(arg);
    }
    private Vector _courses = new Vector();
```

Я могу изменить `setCourses` так, чтобы он инициализировал вектор:

```
public void initializeCourses(Vector arg) {
    Assert.isTrue(_courses.isEmpty());
    Enumeration e = arg.elements();
    while (e.hasMoreElements()) {
        addCourse((Course) e.nextElement());
    }
}
```

Я изменяю пользователей метода получения значения так, чтобы они использовали методы модификации, так что код

```
kent.getCourses().addElement(new Course ("Brutal Sarcasm", false));
```

превращается в

```
kent.addCourse(new Course ("Brutal Sarcasm", false));
```

Однако последний шаг будет иным, потому что у векторов нет немодифицируемой версии:

```
class Person...
    Vector getCourses() {
        return (Vector) _courses.clone();
    }
```

Итак, коллекция инкапсулирована. Теперь никто не сможет изменить элементы коллекции, иначе как через методы Person.

Пример: инкапсуляция массивов

Применение массивов крайне распространено, в особенности — программистами, которые не знакомы с коллекциями. Лично я работаю с массивами редко, потому что предпочитаю более богатые возможности коллекций. Поэтому при инкапсуляции я часто перехожу от массивов к коллекциям.

На этот раз я начну с массива строк для навыков (skills):

```
String[] getSkills() {
    return _skills;
}
void setSkills(String[] arg) {
    _skills = arg;
}
String[] _skills;
```

Я снова начинаю с создания методов модификации. Поскольку клиенту, вероятно, потребуется изменять значение в определенной позиции, мне нужна операция установки значения конкретного элемента:

```
void setSkill(int index, String newSkill) {
    _skills[index] = newSkill;
}
```

Установить значения для всего массива можно с помощью следующей операции:

```
void setSkills(String[] arg) {
    _skills = new String[arg.length];
    for (int i = 0; i < arg.length; i++)
        setSkill(i, arg[i]);
}
```

Здесь возможен ряд ловушек, если над удаляемыми элементами должны выполняться какие-то действия. Ситуация усложняется, если массив, переданный в качестве аргумента, отличается по размеру от оригинального массива. Это — еще один довод в пользу коллекций.

Теперь я могу начать поиск пользователей метода получения значения. Строку

```
kent.getSkills()[1] = "Refactoring";
```

можно заменить строкой

```
kent.setSkill(1,"Refactoring");
```

Когда все изменения будут выполнены, я могу модифицировать метод получения значения таким образом, чтобы он возвращал копию:

```
String[] getSkills() {
    String[] result = new String[_skills.length];
    System.arraycopy(_skills, 0, result, 0, _skills.length);
    return result;
}
```

Теперь самое время заменить массив списком:

```
class Person...
    String[] getSkills() {
        return (String[]) _skills.toArray(new String[0]);
    }
    void setSkill(int index, String newSkill) {
        _skills.set(index, newSkill);
    }
    List _skills = new ArrayList();
```

Замена записи классом данных (Replace Record with Data Class)

Необходим интерфейс для структуры записи в традиционной программной среде.

Создайте для записи немой объект данных.

Мотивация

Структуры записей — распространенная возможность программных сред. Для их применения в объектно-ориентированных программах могут быть разные причины. Возможно, вы копируете старую программу или вынуждены работать со структурированной записью посредством традиционного API, или это запись базы данных. В таких случаях бывает полезно создать интерфейсный класс для работы с таким внешним элементом. Проще всего сделать класс подобным внешней записи. Другие поля и методы можно поместить в этот класс позже. Менее очевидным, но более убедительным случаем является массив, в котором элемент для каждого индекса имеет собственный смысл. В этом случае вы можете использовать рефакторинг “Замена массива объектом” (с. 204).

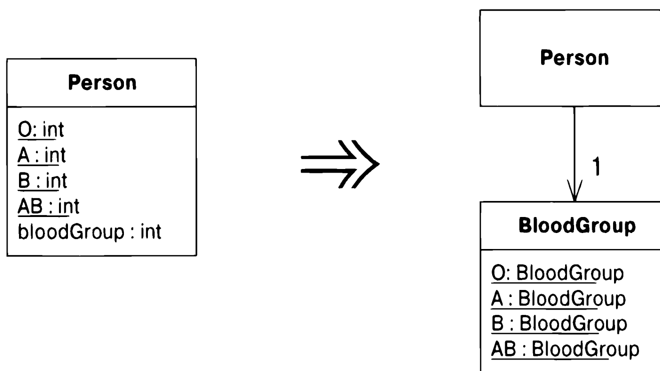
Техника

- Создайте класс, который будет представлять запись.
- Создайте в классе для каждого элемента данных закрытое поле с методом получения и методом установки значения.

Теперь у вас есть немой объект данных. В нем пока нет поведения, но дальнейшие рефакторинги быстро исправят ситуацию.

Замена кода типа классом (Replace Type Code with Class)

У класса есть код числового типа, который не влияет на его поведение.
Замените число новым классом.



Мотивация

Числовые коды типа, или перечисления, часто встречаются в С-подобных языках. Символические имена делают их удобочитаемыми. Проблема в том, что символическое имя является не более чем псевдонимом: компилятор видит в нем обозначаемое этим именем число. Проверка типа компилятором использует числовое значение, а не символическое имя. Все методы, принимающие в качестве аргумента код типа, ожидают число, и ничто не может заставить их использовать символическое имя. Это снижает удобочитаемость кода и может стать источником ошибок.

Если заменить число классом, компилятор сможет выполнять проверку типа класса. Предоставив для класса фабричные методы, можно статически проверять, что создаются только допустимые экземпляры и что эти экземпляры передаются корректным объектам.

Однако, прежде чем применять данный рефакторинг, необходимо рассмотреть возможность другой замены кодов типа. Заменяйте код типа классом только тогда, когда этот код типа представляет собой чистые данные, т.е. не приводит к различному поведению внутри конструкции `switch`. Java может выполнять инструкцию `switch` только для целых чисел, а не для произвольных классов, поэтому замена может оказаться неудачной. Еще важнее, что все инструкции выбора `switch` должны быть удалены с помощью рефакторинга "Замена условной инструкции полиморфизмом" (с. 271). Для выполнения этого рефакторинга код типа сначала следует обработать с помощью рефакторинга "Замена кода типа подклассами" (с. 241) или "Замена кода типа состоянием/стратегией" (с. 245).

Даже если код типа не вызывает изменения поведения в зависимости от его значения, может обнаружиться поведение, которое лучше поместить в класс кода типа, так что будьте готовы к применению одного-двух рефакторингов "Перенос метода" (с. 162).

Техника

- Создайте новый класс для кода типа.

  ⇛ *В классе должны быть поле кода, соответствующее коду типа, и метод получения его значения. Класс должен иметь статические переменные для допустимых экземпляров класса и статический метод, возвращающий соответствующий экземпляр из аргумента и основанный на оригинальном коде.*

- Модифицируйте реализацию исходного класса так, чтобы она использовала новый класс.

  ⇛ *Поддерживайте старый интерфейс, но измените статические поля так, чтобы для генерации кодов использовался новый класс. Измените другие методы так, чтобы они получали числовые значения кода из нового класса.*

- Выполните компиляцию и тестирование.

  ⇛ *После этого новый класс может выполнять проверки кодов времени выполнения.*

- Для каждого метода исходного класса, в котором используется этот код, создайте новый метод, который будет использовать вместо этого новый класс.

  ⇛ *Для методов, в которых код участвует в качестве аргумента, требуются новые методы, использующие в качестве аргумента новый класс. Для методов, которые возвращают код, требуется создать новые методы, возвращающие экземпляр нового класса. Зачастую разумно*

применить рефакторинг "Переименование метода" (с. 290) к старому методу доступа к значению, прежде чем создавать новый, — чтобы сделать понятнее программу, использующую старый код.

- По одному измените клиенты исходного класса так, чтобы они использовали новый интерфейс.
- После модификации каждого клиента выполняйте компиляцию и тестирование.

  ⇛ *Может потребоваться изменить несколько методов, прежде чем вы добьетесь согласованности, достаточной для компиляции и тестирования.*
- Удалите прежний интерфейс, использующий коды, а также статические объявления этих кодов.
- Выполните компиляцию и тестирование.

Пример

Рассмотрим моделирование группы крови человека с помощью кода типа:

```
class Person {
    public static final int O = 0;
    public static final int A = 1;
    public static final int B = 2;
    public static final int AB = 3;

    private int _bloodGroup;

    public Person(int bloodGroup) {
        _bloodGroup = bloodGroup;
    }

    public void setBloodGroup(int arg) {
        _bloodGroup = arg;
    }

    public int getBloodGroup() {
        return _bloodGroup;
    }
}
```

Я начинаю с создания нового класса группы крови, экземпляры которого содержат числовой код типа:

```
class BloodGroup {
    public static final BloodGroup O  = new BloodGroup(0);
    public static final BloodGroup A  = new BloodGroup(1);
    public static final BloodGroup B  = new BloodGroup(2);
```

```
    public static final BloodGroup AB = new BloodGroup(3);
    private static final BloodGroup[] _values = {O, A, B, AB};

    private final int _code;

    private BloodGroup(int code) {
        _code = code;
    }

    public int getCode() {
        return _code;
    }

    public static BloodGroup code(int arg) {
        return _values[arg];
    }
}
```

Затем я модифицирую класс `Person` так, чтобы в нем использовался новый класс:

```
class Person {
    public static final int O  = BloodGroup.O.getCode();
    public static final int A  = BloodGroup.A.getCode();
    public static final int B  = BloodGroup.B.getCode();
    public static final int AB = BloodGroup.AB.getCode();

    private BloodGroup _bloodGroup;

    public Person(int bloodGroup) {
        _bloodGroup = BloodGroup.code(bloodGroup);
    }

    public int getBloodGroup() {
        return _bloodGroup.getCode();
    }

    public void setBloodGroup(int arg) {
        _bloodGroup = BloodGroup.code(arg);
    }
}
```

Теперь в классе группы крови есть проверка типа на этапе выполнения. Для того чтобы получить реальную выгоду от внесенных изменений, я должен изменить пользователей класса `Person`, чтобы они использовали класс группы крови вместо целочисленных значений.

Сначала я применяю рефакторинг “Переименование метода” (с. 290) к методу доступа к группе крови в классе `Person`, чтобы максимально прояснить новое положение дел:

```
class Person...
    public int getBloodGroupCode() {
        return _bloodGroup.getCode();
    }
```

Затем я добавляю новый метод получения значения, который использует новый класс:

```
public BloodGroup getBloodGroup() {
    return _bloodGroup;
}
```

Я также создаю новый конструктор и метод установки значения, использующие новый класс:

```
public Person(BloodGroup bloodGroup) {
    _bloodGroup = bloodGroup;
}

public void setBloodGroup(BloodGroup arg) {
    _bloodGroup = arg;
}
```

Теперь я приступаю к работе с клиентами класса `Person`. Главное — не спешить и работать с клиентами по одному и продвигаться вперед небольшими шагами. Проблема в том, что каждый клиент может потребовать внесения своих изменений, отличных от изменений другого клиента, а это усложняет задачу. Должны быть заменены все ссылки на статические переменные. Таким образом, код

```
Person thePerson = new Person(Person.A)
```

превращается в код

```
Person thePerson = new Person(BloodGroup.A);
```

Обращения к методу получения значения должны использовать новый метод, так что код

```
thePerson.getBloodGroupCode()
```

превращается в

```
thePerson.getBloodGroup().getCode()
```

То же самое относится и к методам установки значений, так что код

```
thePerson.setBloodGroup(Person.AB)
```

превращается в

```
thePerson.setBloodGroup(BloodGroup.AB)
```

Проделав это для всех клиентов класса Person, я могу удалить метод получения значения, конструктор, статические определения и методы установки значения, использующие целочисленные значения:

```
class Person ...
    public static final int O = BloodGroup.O.getCode();
    public static final int A = BloodGroup.A.getCode();
    public static final int B = BloodGroup.B.getCode();
    public static final int AB = BloodGroup.AB.getCode();
    public Person(int bloodGroup) {
        _bloodGroup = BloodGroup.code(bloodGroup);
    }
    public int getBloodGroupCode() {
        return _bloodGroup.getCode();
    }
    public void setBloodGroup(int arg) {
        _bloodGroup = BloodGroup.code(arg);
    }
```

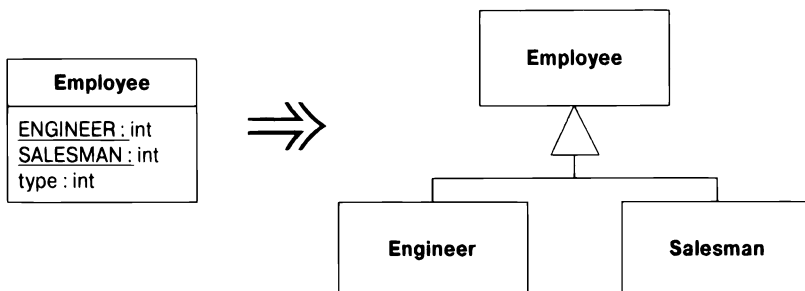
Если ни один из клиентов не использует числовой код, можно также объявить закрытыми методы класса группы крови, использующие целочисленный код:

```
class BloodGroup...
    private int getCode() {
        return _code;
    }
    private static BloodGroup code(int arg) {
        return _values[arg];
    }
```

Замена кода типа подклассами (Replace Type Code with Subclasses)

Имеется неизменяемый код типа, влияющий на поведение класса.

Замените код типа подклассами.



Мотивация

Если код типа не оказывает влияния на поведение, можно воспользоваться рефакторингом “Замена кода типа классом” (с. 236). Однако если этот код типа влияет на поведение, то лучше всего организовать вариантное поведение с помощью полиморфизма.

Обычно на такую ситуацию указывает наличие условных инструкций типа `case` — это могут быть переключатели `switch` или конструкции наподобие `if-then-else`. Они проверяют значение кода типа и в зависимости от результата выполняют различные действия. К таким фрагментам кода следует применить рефакторинг “Замена условной инструкции полиморфизмом” (с. 271). Чтобы данный рефакторинг работал, код типа следует заменить структурой наследования с полиморфным поведением. В такой иерархии наследования имеется класс с подклассами для каждого кода типа.

Простейший способ организации такой структуры — применение описываемого здесь рефакторинга. Нужно взять класс, имеющий данный код типа, и создать подкласс для каждого конкретного кода. Однако бывают случаи, когда сделать это нельзя. Один из них — когда код типа изменяется после создания объекта. Во втором случае класс с кодом типа уже имеет подклассы по каким-то иным другим причинам. В обоих случаях следует использовать рефакторинг “Замена кода типа состоянием/стратегией” (с. 245).

Описываемый здесь рефакторинг служит, в первую очередь, в качестве вспомогательного этапа, обеспечивающего выполнение рефакторинга “Замена условной инструкции полиморфизмом” (с. 271). Побудительной причиной для применения рассматриваемого здесь рефакторинга “Замена кода типа подклассами” служит наличие условных инструкций. Если таковых нет, более правильным и менее критичным будет применение рефакторинга “Замена кода типа классом” (с. 236).

Еще одной причиной применения рефакторинга “Замена кода типа подклассами” является наличие функций, относящихся только к объектам с определенными кодами типа. Выполнив данный рефакторинг, можно будет использовать рефакторинги “Опускание метода” (с. 345) и “Опускание поля” (с. 346), чтобы яснее указать, что данные возможности относятся только к некоторым ситуациям.

Преимущество рассматриваемого рефакторинга в том, что он перемещает знание о вариантном поведении из клиентов класса в сам класс. При добавлении новых вариантов поведения все, что нужно сделать, — это добавить подкласс. В отсутствие полиморфизма я был бы вынужден искать все условные инструкции и все их изменять. Поэтому данный рефакторинг в особенности полезен тогда, когда варианты поведения постоянно изменяются.

Техника

- Выполните самоинкапсуляцию кода типа.
  - ⇛ *Если код типа передается конструктору, необходимо заменить конструктор фабричным методом.*
- Для каждого значения кода типа создайте подкласс. Перекройте метод получения значения кода типа в подклассе так, чтобы он возвращал правильное значение.
  - ⇛ *Это значение жестко прошивается в операторе* `return` *(например,* `return 1`*). Такое решение может показаться непривлекательным, но это не более чем временная мера, пока не будут заменены все инструкции* `case`.
- После замены каждого значения кода типа подклассом выполните компиляцию и тестирование.
- Удалите поле кода типа из суперкласса. Объявите методы доступа к коду типа как абстрактные.
- Выполните компиляцию и тестирование.

Пример

Воспользуюсь скучным и нереалистичным примером с зарплатой служащего:

```
class Employee...
    private int _type;
    static final int ENGINEER = 0;
    static final int SALESMAN = 1;
    static final int MANAGER = 2;

    Employee(int type) {
        _type = type;
    }
```

Первый шаг состоит в применении к коду типа рефакторинга “Самоинкапсуляция поля” (с. 190):

```
int getType() {
    return _type;
}
```

Поскольку конструктор `Employee` использует код типа в качестве параметра, я должен заменить его фабричным методом:

```
Employee create(int type) {
    return new Employee(type);
}

private Employee(int type) {
    _type = type;
}
```

Теперь я могу приступить к преобразованию Engineer в подкласс. Сначала создаются подкласс и перекрывающий метод для кода типа:

```
class Engineer extends Employee {
    int getType() {
        return Employee.ENGINEER;
    }
}
```

Мне также нужно изменить фабричный метод так, чтобы он создавал соответствующий объект:

```
class Employee
    static Employee create(int type) {
        if (type == ENGINEER) return new Engineer();
        else return new Employee(type);
    }
```

Затем я продолжаю работать с каждым кодом по очереди до тех пор, пока все коды типа не будут заменены подклассами. После этого я могу избавиться от поля кода типа в Employee и сделать getType абстрактным методом. Теперь фабричный метод имеет следующий вид:

```
abstract int getType();

static Employee create(int type) {
    switch (type) {
        case ENGINEER:
            return new Engineer();
        case SALESMAN:
            return new Salesman();
        case MANAGER:
            return new Manager();
        default:
            throw new IllegalArgumentException(
                    "Неверное значение кода типа");
    }
}
```

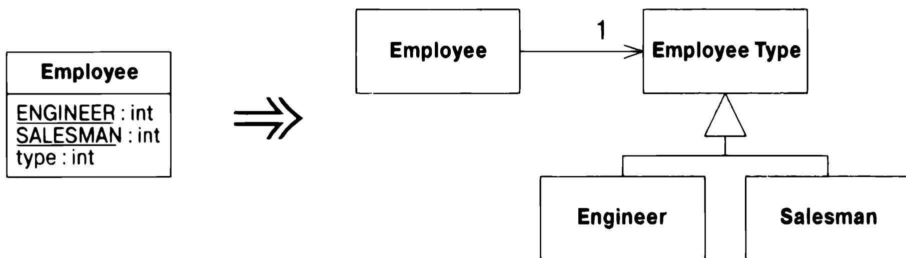
Конечно, мне очень хотелось бы убрать эту инструкцию switch, но она только одна и используется только при создании объекта.

Само собой разумеется, что после создания подклассов нужно выполнить рефакторинги “Опускание метода” (с. 345) и “Опускание поля” (с. 346) для всех методов и полей, которые имеют отношение только к конкретным типам служащих.

Замена кода типа состоянием/стратегией (Replace Type Code with State/Strategy)

Имеется код типа, влияющий на поведение класса, но применить создание подклассов невозможно.

Замените код типа объектом состояния.



Мотивация

Этот рефакторинг похож на рефакторинг “Замена кода типа подклассами” (с. 241), но может быть применен тогда, когда код типа изменяется во время жизни объекта или если созданию подклассов мешают иные причины. В нем применяется проектный шаблон состояния или стратегии [9].

Состояние и стратегия очень похожи между собой, так что рефакторинг в обоих случаях будет одинаков. Выберите тот проектный шаблон, который лучше подходит в данной конкретной ситуации. Если вы пытаетесь упростить отдельный алгоритм с помощью рефакторинга “Замена условной инструкции полиморфизмом” (с. 271), то лучшим выбором будет использование стратегии. Если же вы намерены переместить данные, специфические для состояния, и представляете себе объект как изменяющий состояние, используйте проектный шаблон состояния.

Техника

- Выполните самоинкапсуляцию кода типа.
- Создайте новый класс и дайте ему имя в соответствии с предназначением кода типа. Это и есть объект состояния.

- Добавьте подклассы к объекту состояния, по одному для каждого кода типа.

  ⇒ *Проще добавить все подклассы одновременно, чем по одному.*
- Создайте в объекте состояния абстрактный запрос для возврата кода типа. Создайте перекрывающие запросы для каждого подкласса объекта состояния, возвращающие корректный код типа.
- Выполните компиляцию.
- Создайте в старом классе поле для нового объекта состояния.
- Настройте запрос кода типа в исходном классе так, чтобы он делегировался объекту состояния.
- Настройте методы установки значения кода типа в исходном классе так, чтобы они присваивали экземпляр соответствующего подкласса объекта состояния.
- Выполните компиляцию и тестирование.

Пример

Я снова обращаюсь к утомительному и бессмысленному примеру зарплаты служащих:

```
class Employee {

    private int _type;
    static final int ENGINEER = 0;
    static final int SALESMAN = 1;
    static final int MANAGER = 2;

    Employee(int type) {
        _type = type;
    }
```

Вот пример условного поведения, которое может использовать эти коды:

```
int payAmount() {
    switch (_type) {
        case ENGINEER:
            return _monthlySalary;
        case SALESMAN:
            return _monthlySalary + _commission;
        case MANAGER:
            return _monthlySalary + _bonus;
        default:
            throw new RuntimeException("Неверный код служащего");
    }
}
```

Предполагается, что это замечательная, прогрессивная компания, позволяющая менеджерам вырастать до инженеров. Поэтому код типа может изменяться, и применять подклассы я не могу. Мой первый шаг, как обычно, состоит в самоинкапсуляции кода типа:

```
Employee(int type) {
    setType(type);
}

int getType() {
    return _type;
}

void setType(int arg) {
    _type = arg;
}

int payAmount() {
    switch (getType()) {
        case ENGINEER:
            return _monthlySalary;
        case SALESMAN:
            return _monthlySalary + _commission;
        case MANAGER:
            return _monthlySalary + _bonus;
        default:
            throw new RuntimeException("Неверный код служащего");
    }
}
```

Теперь я объявляю класс состояния. Я объявляю абстрактный класс и предоставляю абстрактный метод возврата кода типа:

```
abstract class EmployeeType {
    abstract int getTypeCode();
}
```

Далее я создаю подклассы:

```
class Engineer extends EmployeeType {

    int getTypeCode() {
        return Employee.ENGINEER;
    }
}
class Manager extends EmployeeType {

    int getTypeCode() {
        return Employee.MANAGER;
    }
```

```
}
class Salesman extends EmployeeType {

    int getTypeCode() {
        return Employee.SALESMAN;
    }
}
```

Я компилирую то, что получилось, и все сделанное так тривиально, что все легко и правильно компилируется. Теперь я фактически подключаю подклассы к Employee, изменяя методы доступа к коду типа:

```
class Employee...
    private EmployeeType _type;

    int getType() {
        return _type.getTypeCode();
    }

    void setType(int arg) {
        switch (arg) {
            case ENGINEER:
                _type = new Engineer();
                break;
            case SALESMAN:
                _type = new Salesman();
                break;
            case MANAGER:
                _type = new Manager();
                break;
            default:
                throw new IllegalArgumentException(
                        "Неверный код служащего");
        }
    }
```

Это означает, что у меня получилась конструкция switch. После завершения рефакторинга она окажется единственной в коде и будет выполняться только при изменении типа. Я могу также воспользоваться рефакторингом “Замена конструктора фабричным методом” (с. 321), чтобы создать фабричные методы для разных случаев. Я могу удалить все прочие инструкции case с помощью рефакторинга “Замена условной инструкции полиморфизмом” (с. 271).

Мне нравится завершать описанный в этом разделе рефакторинг перемещением в новый класс всей информации о кодах типов и подклассах. Сначала я копирую определения кодов типов в тип служащего, создаю фабричный метод для типов служащих и настраиваю метод установки значения для служащего:

```
class Employee...
    void setType(int arg) {
        _type = EmployeeType.newType(arg);
    }

class EmployeeType...
    static EmployeeType newType(int code) {
        switch (code) {
            case ENGINEER:
                return new Engineer();
            case SALESMAN:
                return new Salesman();
            case MANAGER:
                return new Manager();
            default:
                throw new IllegalArgumentException(
                        "Неверный код служащего");
        }
    }
    static final int ENGINEER = 0;
    static final int SALESMAN = 1;
    static final int MANAGER = 2;
```

Теперь я удаляю определения кодов типа из `Employee` и заменяю их ссылками на тип служащего:

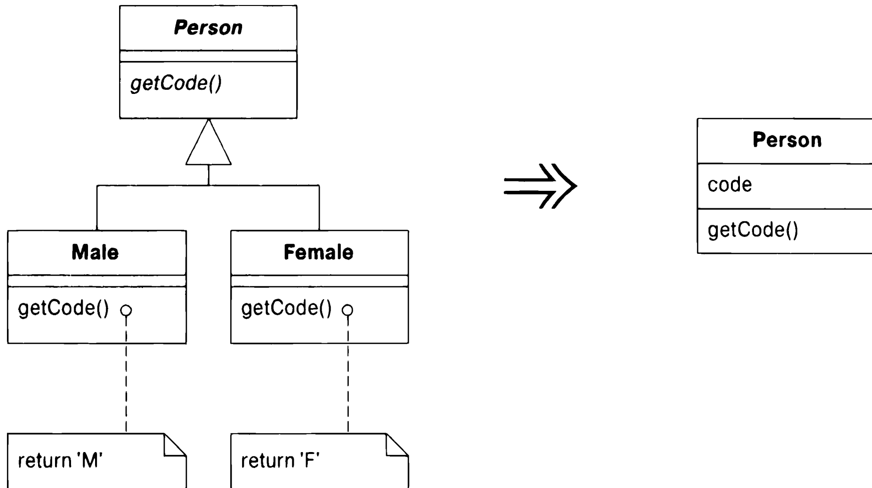
```
class Employee...
    int payAmount() {
        switch (getType()) {
            case EmployeeType.ENGINEER:
                return _monthlySalary;
            case EmployeeType.SALESMAN:
                return _monthlySalary + _commission;
            case EmployeeType.MANAGER:
                return _monthlySalary + _bonus;
            default:
                throw new RuntimeException(
                        "Неверный код служащего");
        }
    }
```

Сейчас я готов применить к `payAmount` рефакторинг “Замена условной инструкции полиморфизмом” (с. 271).

Замена подкласса полями (Replace Subclass with Fields)

Имеются подклассы, различающиеся только методами, возвращающими константные данные.

Замените методы полями в суперклассе и удалите подклассы.



Мотивация

Цель создания подклассов — добавление новых функций или изменение поведения. Одной из форм вариантного поведения является константный метод [6]. Константный метод — это метод, который возвращает жестко закодированное значение. Он может быть полезен в подклассах, возвращающих различные значения в методах доступа к значению. Вы определяете метод доступа в суперклассе и реализуете его с помощью различных значений в подклассах.

Хотя константные методы полезны, подкласс, состоящий только из них, делает недостаточно для оправдания своего существования. Такие подклассы можно полностью удалять, добавляя поля в суперклассе. Таким образом убирается дополнительная сложность, связанная с подклассами.

Техника

- Примените к подклассам рефакторинг “Замена конструктора фабричным методом” (с. 321).
- Если некоторый код обращается к подклассам, замените его обращением к суперклассу.

- Для каждого константного метода объявите в суперклассе соответствующее поле final.
- Объявите защищенный конструктор суперкласса, инициализирующий поля.
- Добавьте или модифицируйте конструкторы подклассов так, чтобы они вызывали новый конструктор суперкласса.
- Выполните компиляцию и тестирование.
- Реализуйте каждый константный метод в суперклассе таким образом, чтобы он возвращал поле, и удалите метод из подкласса.
- После каждого такого удаления выполняйте компиляцию и тестирование.
- После удаления всех методов подклассов примените рефакторинг "Встраивание метода" (с. 139), чтобы встроить конструктор в фабричный метод суперкласса.
- Выполните компиляцию и тестирование.
- Удалите подкласс.
- Выполните компиляцию и тестирование.
- Повторяйте встраивание конструктора и удаление подклассов до тех пор, пока не удалите их все.

Пример

Начну с класса Person и подклассов, ориентированных на пол:

```
abstract class Person {

    abstract boolean isMale();
    abstract char getCode();
    ...

class Male extends Person {
    boolean isMale() {
        return true;
    }
    char getCode() {
        return 'M';
    }
}

class Female extends Person {
    boolean isMale() {
        return false;
    }
```

```
        char getCode() {
            return 'F';
        }
    }
```

Единственное различие между подклассами здесь в том, что в них имеются реализации абстрактного метода, возвращающие жестко "прошитые" константы. Я удалю эти "ленивые" подклассы [6].

Сначала я должен применить рефакторинг "Замена конструктора фабричным методом" (с. 321). В данном случае мне нужен фабричный метод для каждого подкласса:

```
class Person...
    static Person createMale() {
        return new Male();
    }
    static Person createFemale() {
        return new Female();
    }
```

Затем я заменяю вызовы вида

```
Person kent = new Male();
```

такими:

```
Person kent = Person.createMale();
```

После того как я заменю все эти вызовы, в коде не должно остаться обращений к подклассам. Это можно проверить с помощью текстового поиска, а сделав класс закрытым, можно убедиться, что к ним нет обращений извне пакета.

Теперь я объявляю поля для каждого константного метода в суперклассе:

```
class Person...
    private final boolean _isMale;
    private final char _code;
```

Далее я добавляю в суперкласс защищенный конструктор:

```
class Person...
    protected Person(boolean isMale, char code) {
        _isMale = isMale;
        _code = code;
    }
```

Затем я добавляю конструкторы, вызывающие этот новый конструктор:

```
class Male...
    Male() {
        super(true, 'M');
    }

class Female...
    Female() {
        super(false, 'F');
    }
```

Теперь можно выполнить компиляцию и тестирование. Поля создаются и инициализируются, но пока что не используются. Теперь я ввожу в игру поля, размещая в суперклассе методы доступа к ним и удаляя методы подклассов:

```
class Person...
    boolean isMale() {
        return _isMale;
    }
class Male...
    boolean isMale() {
        return true;
    }
```

Это можно делать это по одному полю и подклассу за раз; если вы чувствуете уверенность в себе, можете сделать это со всеми полями сразу.

В конечном итоге все подклассы оказываются пустыми, поэтому я убираю модификатор `abstract` с класса `Person` и с помощью рефакторинга “Встраивание метода” (с. 139) встраиваю конструктор подкласса в суперкласс:

```
class Person
    static Person createMale() {
        return new Person(true, 'M');
    }
```

После компиляции и тестирования класс `Male` удаляется. Та же процедура выполняется и для класса `Female`.

Глава 9

Упрощение условных выражений

Логика условного выполнения имеет тенденцию становиться со временем все более сложной и запутанной, поэтому целый ряд рефакторингов направлен на то, чтобы ее упростить. Основным рефакторингом при этом является рефакторинг "Декомпозиция условного оператора" (с. 256), предназначенный для разложения условной инструкции на части. Его значение заключается в том, что логика переключения отделяется от деталей происходящего.

Прочие рефакторинги в этой главе относятся к другим важным случаям. Рефакторинг "Консолидация условного выражения" (с. 258) применяйте тогда, когда имеется несколько проверок и все они выполняют одни и те же действия. Рефакторинг "Консолидация дублирующихся условных фрагментов" (с. 260) позволяет удалить дублирование в коде условной инструкции.

При работе с кодом, разработанным с применением принципа "единственной точки выхода", часто обнаруживаются управляющие флаги, которые позволяют условной логике действовать согласно этому принципу. Здесь я применяю рефакторинг "Замена вложенных условных операторов граничным оператором" (с. 267) для прояснения частных случаев условных инструкций и рефакторинг "Удаление управляющего флага" (с. 262) для избавления от запутывающих управляющих флагов.

В объектно-ориентированных программах количество условных инструкций обычно меньше, чем в процедурных, потому что значительную часть условного поведения берет на себя полиморфизм. Полиморфизм обладает тем преимуществом, что вызывающему коду не требуется знать об условном поведении, а потому облегчается расширение условий. В результате в объектно-ориентированных программах редко встречаются конструкции `switch`. Если же таковые все же имеются, они являются главными кандидатами для выполнения рефакторинга "Замена условной инструкции полиморфизмом" (с. 271).

Одним из наиболее полезных, хотя и менее очевидных, применений полиморфизма является рефакторинг "Введение нулевого объекта" (с. 276), позволяющий избавиться от проверок на нулевое значение.

Декомпозиция условного оператора (Decompose Conditional)

Имеется сложная условная цепочка проверок (if-then-else).
Выделите методы из условия и частей `then` *и* `else`.

```
if (date.before (SUMMER_START) || date.after(SUMMER_END))
    charge = quantity * _winterRate + _winterServiceCharge;
else
    charge = quantity * _summerRate;
```

```
if (notSummer(date))
    charge = winterCharge(quantity);
else charge = summerCharge(quantity);
```

Мотивация

Зачастую сложность программы обусловлена сложностью условной логики. При написании кода для проверки условий и выполнения в зависимости от результата проверки разных действий мы быстро приходим к созданию довольно длинного метода. Уже сама по себе длина метода затрудняет его чтение, но если в нем есть сложные условные выражения, трудностей становится еще больше. Обычно проблема связана с тем, что код (как в проверках условий, так и в действиях), говоря о том, что происходит, легко маскирует причину, по которой это происходит.

Как и в любом большом блоке кода, свои намерения можно сделать более понятными, если выполнить декомпозицию кода и заменить его фрагменты вызовами методов, имена которых раскрывают их назначение. Для кода с условными операторами выгода будет еще большей, если проделать это как для условия, так и для всех ветвей. Таким способом можно ясно выделить условие и четко указать, что лежит в основе ветвления. Кроме того, подчеркиваются причины организации ветвления.

Техника

- Выделите условие в собственный метод.
- Выделите части `then` и `else` в собственные методы.

Сталкиваясь с вложенной условной инструкцией, я сначала обычно смотрю, не стоит ли выполнить рефакторинг "Замена вложенных условных операторов граничным оператором" (с. 267). Если в нем нет особого смысла, я выполняю декомпозицию каждой условной инструкции.

Пример

Предположим, что нам требуется вычислить плату за какую-то услугу, для которой есть отдельно зимний и летний тарифы:

```
if (date.before(SUMMER_START) || date.after(SUMMER_END))
    charge = quantity * _winterRate + _winterServiceCharge;
else charge = quantity * _summerRate;
```

Я выделяю условие и каждую ветвь следующим образом:

```
if (notSummer(date))
    charge = winterCharge(quantity);
else charge = summerCharge(quantity);

private boolean notSummer(Date date) {
    return date.before(SUMMER_START) || date.after(SUMMER_END);
}

private double summerCharge(int quantity) {
    return quantity * _summerRate;
}

private double winterCharge(int quantity) {
    return quantity * _winterRate + _winterServiceCharge;
}
```

Здесь для ясности показан конечный результат всего рефакторинга в целом. Однако на практике я провожу каждое выделение по отдельности, выполняя после него компиляцию и тестирование.

Многие программисты в таких ситуациях не выделяют части, формирующие условие. Условия зачастую кажутся очень короткими и не стоящими такого труда. Но несмотря на краткость условия, нередко существует большой разрыв между смыслом кода и его телом. Даже в данном маленьком примере при чтении `notSummer(date)` я получаю более понятную информацию о происходящем, чем из исходного кода. При чтении оригинального исходного текста приходится разбираться в том, что же делает тот или иной фрагмент кода. В данном случае это сделать нетрудно, но даже здесь выделенный метод более похож на комментарий.

Консолидация условного выражения (Consolidate Conditional Expression)

Имеется ряд проверок условий, дающих одинаковый результат.

Объедините их в одно условное выражение и выделите его.

```
double disabilityAmount() {
    if (_seniority < 2) return 0;
    if (_monthsDisabled > 12) return 0;
    if (_isPartTime) return 0;
    // Вычисление оплаты больничного
```

```
double disabilityAmount() {
    if (isNotEligableForDisability()) return 0;
    // Вычисление оплаты больничного
```

Мотивация

Иногда встречается ряд проверок условий, в котором все проверки различны, но результирующее действие оказывается одним и тем же. Встретившись с таким поведением, следует с помощью логических операций "И" и "ИЛИ" объединить проверки в одну, возвращающую единственный результат.

Объединение условного кода важно по двум причинам. Проверка становится более понятной, показывая, что на самом деле выполняется только одна проверка, в которой логически соединяются результаты других проверок. Последовательность проверок приводит к тому же результату, но выглядит как ряд отдельных проверок, которые случайно собрались вместе. Второе основание для проведения рефакторинга заключается в том, что он зачастую подготавливает почву для рефакторинга "Извлечение метода" (с. 132). Извлечение условия — одно из наиболее полезных для прояснения кода действий. Оно заменяет выполняемые инструкции описанием причины, по которой они выполняются.

Обоснование консолидации условных выражений одновременно указывает причины, по которым ее не следует выполнять. Если вы считаете, что проверки действительно независимы и не должны рассматриваться как единая проверка, не выполняйте этот рефакторинг. Имеющийся код уже раскрывает ваш замысел.

Техника

- Убедитесь, что условные выражения не содержат побочных действий.
  ⇛ *Если побочные действия есть, вы не сможете выполнить данный рефакторинг.*
- Замените последовательность условий одним условием с помощью логических операторов.
- Выполните компиляцию и тестирование.
- Рассмотрите возможность применения к условию рефакторинга "Извлечение метода" (с. 132).

Пример: логическое "ИЛИ"

Имеется следующий код:

```
double disabilityAmount() {
    if (_seniority < 2) return 0;
    if (_monthsDisabled > 12) return 0;
    if (_isPartTime) return 0;
    // Вычисление оплаты больничного
    ...
```

Мы видим ряд условных инструкций, приводящих к одному и тому же результату. Проверки для такого кода эквивалентны выражению с использованием оператора ИЛИ:

```
double disabilityAmount() {
    if ((_seniority < 2) || (_monthsDisabled > 12) || (_isPartTime))
        return 0;
    // Вычисление оплаты больничного
    ...
```

Теперь я вижу, что, применив рефакторинг "Извлечение метода" (с. 132), я могу сообщить о том, что именно проверяет условная инструкция (нетрудоспособность не оплачивается):

```
double disabilityAmount() {
    if (isNotEligibleForDisability()) return 0;

    // Вычисление оплаты больничного
    ...
}
boolean isNotEligibleForDisability() {
    return ((_seniority < 2) ||
            (_monthsDisabled > 12) ||
            (_isPartTime));
}
```

Пример: логическое И

Предыдущий пример демонстрировал применение ИЛИ, но то же самое можно делать и в случае И. Пусть ситуация имеет следующий вид:

```
if (onVacation())
    if (lengthOfService() > 10)
        return 1;
return 0.5;
```

Этот код нужно заменить следующим:

```
if (onVacation() && lengthOfService() > 10)
    return 1;
else
    return 0.5;
```

Зачастую в результате преобразований получается комбинированное выражение, содержащее операторы И, ИЛИ и НЕ. В таких случаях условия бывают весьма запутанными, так что я стараюсь упростить их с помощью рефакторинга "Извлечение метода" (с. 132).

Если рассматриваемая процедура только проверяет условие и возвращает значение, ее можно превратить в одно выражение `return` с помощью тернарного оператора, так что

```
if (onVacation() && lengthOfService() > 10)
    return 1;
else
    return 0.5;
```

превращается в

```
return (onVacation() && lengthOfService() > 10) ? 1 : 0.5;
```

Консолидация дублирующихся условных фрагментов (Consolidate Duplicate Conditional Fragments)

Один и тот же фрагмент кода присутствует во всех ветвях условного выражения.

Вынесите его за пределы выражения.

```
if (isSpecialDeal()) {
    total = price * 0.95;
    send();
```

```
} else {
    total = price * 0.98;
    send();
}
```

```
if (isSpecialDeal())
    total = price * 0.95;
else
    total = price * 0.98;
send();
```

Мотивация

Иногда оказывается, что во всех ветвях условной инструкции выполняется один и тот же фрагмент кода. Этот фрагмент следует переместить за пределы условной инструкции. В результате код становится понятнее, и сразу ясно, что меняется, а что остается постоянным.

Техника

- Выявите код, который выполняется одинаково, независимо от значения условия.
- Если общий код находится в начале, поместите его перед условной инструкцией.
- Если общий код находится в конце, поместите его после условной инструкции.
- Если общий код находится в средине, посмотрите, не выполняет ли код, находящийся до или после него, какие-либо модификации. Если это так, то общий код можно переместить вперед или назад до конца. После этого можно выполнить такое же перемещение, как и для кода, находящегося в конце или в начале.
- Если код состоит из нескольких выражений, его стоит выделить в метод.

Пример

Указанная ситуация обнаруживается, например, в следующем коде:

```
if (isSpecialDeal()) {
    total = price * 0.95;
```

```
    send();
} else {
    total = price * 0.98;
    send();
}
```

Поскольку метод send выполняется в любом случае, следует вынести его из условной инструкции:

```
if (isSpecialDeal())
    total = price * 0.95;
else
    total = price * 0.98;
send();
```

Та же ситуация может возникать и с исключениями. Если код повторяется после оператора, вызывающего исключение, в блоке try и во всех блоках catch, такой код можно переместить в блок final.

Удаление управляющего флага (Remove Control Flag)

Имеется переменная, действующая в качестве управляющего флага для ряда логических выражений.

Используйте вместо нее break или return.

Мотивация

Встретившись с рядом условных выражений, часто можно обнаружить управляющий флаг, используемый для определения окончания просмотра:

```
Установить done равным false
Пока не done
    Если (условие)
        Выполнить некоторые действия
        Установить done равным true
    Следующая итерация цикла
```

От таких управляющих флагов больше проблем, чем пользы. Их наличие диктуется правилами структурного программирования, согласно которым в процедурах должны быть одна входная и одна выходная точка. Я согласен с тем, что должна быть одна входная точка (этого требуют современные языки программирования), но требование одной выходной точки приводит к сильно запутанным условным инструкциям, в коде которых используются такие неуклюжие флаги. Чтобы выбраться из сложного условного оператора, в языках есть инструкции

break и continue. Часто бывает удивительно, что можно сделать, избавившись от такого управляющего флага. А главное — действительное назначение условной инструкции становится при этом гораздо понятнее.

Техника

Очевидный способ справиться с управляющими флагами предоставляют имеющиеся в Java операторы break и continue.

- Найдите значение управляющего флага, при котором происходит выход из логической инструкции.
- Замените присваивания значения для выхода оператором break, а присваивание значения для продолжения работы — оператором continue.
- Выполняйте компиляцию и тестирование после каждой замены.

Другой подход, применимый также в языках без операторов break и continue, заключается в следующем.

- Извлеките логику в метод.
- Найдите значение управляющего флага, при котором происходит выход из логической инструкции.
- Замените присваивание значения выхода оператором return.
- Выполняйте компиляцию и тестирование после каждой замены.

Даже в языках, в которых есть break или continue, я предпочитаю применять извлечение и return. Оператор return ясно сигнализирует, что никакой код в методе больше не выполняется. При наличии кода такого рода зачастую в любом случае требуется извлечение данного фрагмента.

Следите за тем, не несет ли управляющий флаг заодно и информацию о результате. Если да, то либо управляющий флаг придется оставить, либо, если вы извлекли метод, это значение можно возвращать из него.

Пример: замена простого управляющего флага оператором break

Приведенная далее функция проверяет, не содержится ли в списке лиц кто-либо из подозрительных персон, имена которых жестко прошиты в коде:

```
void checkSecurity(String[] people) {
    boolean found = false;
    for (int i = 0; i < people.length; i++) {
        if (! found) {
            if (people[i].equals("Don")) {
                sendAlert();
```

```
                found = true;
            }
            if (people[i].equals("John")) {
                sendAlert();
                found = true;
            }
        }
    }
}
```

В такой ситуации заметить управляющий флаг очень легко. Я вижу фрагмент, в котором переменной `found` присваивается значение `true`. Я начинаю по одному вводить операторы `break`

```
void checkSecurity(String[] people) {
    boolean found = false;
    for (int i = 0; i < people.length; i++) {
        if (! found) {
            if (people[i].equals("Don")) {
                sendAlert();
                break;
            }

            if (people[i].equals("John")) {
                sendAlert();
                found = true;
            }
        }
    }
}
```

до тех пор, пока не будут заменены все присваивания:

```
void checkSecurity(String[] people) {
    boolean found = false;
    for (int i = 0; i < people.length; i++) {
        if (! found) {
            if (people[i].equals("Don")) {
                sendAlert();
                break;
            }
            if (people[i].equals("John")) {
                sendAlert();
                break;
            }
        }
    }
}
```

После этого можно убрать все обращения к управляющему флагу:

```
void checkSecurity(String[] people) {
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals("Don")) {
            sendAlert();
            break;
        }
        if (people[i].equals("John")) {
            sendAlert();
            break;
        }
    }
}
```

Пример: использование оператора `return`, возвращающего значение управляющего флага

Другой стиль этого рефакторинга использует оператор `return`. Проиллюстрирую это вариантом кода, в котором управляющий флаг выступает в качестве возвращаемого значения:

```
void checkSecurity(String[] people) {
    String found = "";
    for (int i = 0; i < people.length; i++) {
        if (found.equals("")) {
            if (people[i].equals("Don")) {
                sendAlert();
                found = "Don";
            }
            if (people[i].equals("John")) {
                sendAlert();
                found = "John";
            }
        }
    }
    someLaterCode(found);
}
```

Здесь переменная `found` служит двум целям: указывает результат и действует в качестве управляющего флага. Увидев подобный код, я предпочитаю извлечь фрагмент, определяющий `found`, в отдельный метод:

```
void checkSecurity(String[] people) {
    String found = foundMiscreant(people);
    someLaterCode(found);
}

String foundMiscreant(String[] people) {
```

```
    String found = "";
    for (int i = 0; i < people.length; i++) {
        if (found.equals("")) {
            if (people[i].equals("Don")) {
                sendAlert();
                found = "Don";
            }
            if (people[i].equals("John")) {
                sendAlert();
                found = "John";
            }
        }
    }
    return found;
}
```

Теперь я могу заменить управляющий флаг оператором `return`

```
String foundMiscreant(String[] people) {
    String found = "";
    for (int i = 0; i < people.length; i++) {
        if (found.equals("")) {
            if (people[i].equals("Don")) {
                sendAlert();
                return "Don";
            }
            if (people[i].equals("John")) {
                sendAlert();
                found = "John";
            }
        }
    }

    return found;
}
```

пока не избавлюсь от управляющего флага:

```
String foundMiscreant(String[] people) {
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals("Don")) {
            sendAlert();
            return "Don";
        }
        if (people[i].equals("John")) {
            sendAlert();
            return "John";
        }
    }
    return "";
}
```

Применение оператора `return` возможно и тогда, когда нет необходимости в возврате значения; в таком случае используется оператор `return` без аргумента.

Конечно, здесь могут возникнуть проблемы, связанные с побочными действиями функции. Поэтому я планирую применить рефакторинг "Разделение запроса и модификатора" (с. 296). Я продолжу работу с этим примером в соответствующем разделе.

Замена вложенных условных операторов граничным оператором (Replace Nested Conditional with Guard Clauses)

Метод использует условное поведение, которое не дает возможности понять нормальный путь выполнения.

Используйте граничные операторы для всех особых случаев.

```
double getPayAmount() {
    double result;
    if (_isDead) result = deadAmount();
    else {
        if (_isSeparated) result = separatedAmount();
        else {
            if (_isRetired) result = retiredAmount();
            else result = normalPayAmount();
        };
    }
    return result;
};
```

```
double getPayAmount() {
    if (_isDead) return deadAmount();
    if (_isSeparated) return separatedAmount();
    if (_isRetired) return retiredAmount();
    return normalPayAmount();
};
```

Мотивация

Условные выражения чаще всего имеют один из двух видов. В первом случае это проверка, при которой любая выбранная альтернатива является частью нормального поведения. Во втором случае мы имеем дело с ситуацией, когда одна

альтернатива условной инструкции представляет собой нормальное поведение, а вторая — некоторые необычные условия.

Эти виды условных операторов имеют разное назначение, и оно должно быть очевидно из кода. Если обе части представляют собой нормальное поведение, используйте условие с ветвями `then` и `else`. Если же некоторое условие ведет к ненормальному поведению, то после проверки условия выполните `return`, если условие истинно. Такого рода проверка часто называется *граничным оператором* (guard clause [6]).

Главный смысл описываемого здесь рефакторинга — в придании коду выразительности. При использовании конструкции `if-then-else` ветви `then` и `else` получают равный вес. Это говорит читателю кода, что обе ветви обладают равными вероятностью и важностью. Граничный же оператор гласит: "Такое происходит редко, и если оно все же произошло, то надо выполнить вот такие действия и выйти".

Я часто прибегаю к этому рефакторингу, когда работаю с программистом, которого учили, что в методе должны быть единственная точка входа и единственная точка выхода. Единственная точка входа обеспечивается самими современными языками программирования, но в правиле единственной точки выхода на самом деле никакой пользы нет. Главным принципом должна быть ясность: если метод понятнее, когда в нем единственная точка выхода, делайте ее единственной; в противном же случае не следует к этому стремиться.

Техника

- Для каждой проверки добавьте граничный оператор.

  ⇛ *Граничный оператор осуществляет возврат или генерирует исключение.*

- Выполняйте компиляцию и тестирование после каждого добавления граничного оператора.

  ⇛ *Если все граничные операторы возвращают одно и то же значение, примените рефакторинг "Консолидация дублирующихся условных фрагментов" (с. 260).*

Пример

Представьте себе работу системы начисления зарплаты с особыми правилами для служащих, которые умерли, проживают раздельно или вышли на пенсию. Такие случаи необычны, но могут встретиться.

Допустим, я вижу такой код:

```
double getPayAmount() {
    double result;
    if (_isDead) result = deadAmount();
    else {
        if (_isSeparated) result = separatedAmount();
        else {
            if (_isRetired) result = retiredAmount();
            else result = normalPayAmount();
        };
    }
    return result;
};
```

Проверки в этом исходном тексте маскируют выполнение обычных действий, поэтому при использовании граничных операторов код станет яснее. Буду вводить их по одному, начиная сверху:

```
double getPayAmount() {
    double result;
    if (_isDead) return deadAmount();
    if (_isSeparated) result = separatedAmount();
    else {
        if (_isRetired) result = retiredAmount();
        else result = normalPayAmount();
    };
    return result;
};
```

Продолжаю делать поочередные замены:

```
double getPayAmount() {
    double result;
    if (_isDead) return deadAmount();
    if (_isSeparated) return separatedAmount();
    if (_isRetired) result = retiredAmount();
    else result = normalPayAmount();
    return result;
};
```

и затем

```
double getPayAmount() {
    double result;
    if (_isDead) return deadAmount();
    if (_isSeparated) return separatedAmount();
    if (_isRetired) return retiredAmount();
    result = normalPayAmount();
    return result;
};
```

После этих изменений временная переменная `result` не оправдывает своего существования, поэтому убираю ее:

```
double getPayAmount() {
    if (_isDead) return deadAmount();
    if (_isSeparated) return separatedAmount();
    if (_isRetired) return retiredAmount();
    return normalPayAmount();
};
```

Вложенный условный код часто пишут программисты, которых учили, что в методе должна быть единственная точка выхода. Я считаю, что это слишком упрощенный подход. Если метод не представляет для меня дальнейшего интереса, я указываю на это путем выхода из него. Заставляя читателя рассматривать пустой блок `else`, вы только мешаете пониманию кода.

Пример: обращение условий

Рецензируя рукопись этой книги, Джошуа Кериевски (Joshua Kerievsky) заметил, что зачастую данный рефакторинг выполняется с обращением условных выражений. Он любезно предоставил пример, который позволил мне не перенапрягать собственное воображение:

```
public double getAdjustedCapital() {
    double result = 0.0;
    if (_capital > 0.0) {
        if (_intRate > 0.0 && _duration > 0.0) {
            result = (_income / _duration) * ADJ_FACTOR;
        }
    }
    return result;
}
```

Я снова поочередно выполняю замену, но на этот раз при вставке граничного оператора условие делается обратным:

```
public double getAdjustedCapital() {
   double result = 0.0;
    if (_capital <= 0.0) return result;
    if (_intRate > 0.0 && _duration > 0.0) {
        result = (_income / _duration) * ADJ_FACTOR;
    }
    return result;
}
```

Поскольку следующий условный оператор немного сложнее, я обращу его за два шага. Сначала я добавляю отрицание:

```
public double getAdjustedCapital() {
    double result = 0.0;
    if (_capital <= 0.0) return result;
    if (!(_intRate > 0.0 && _duration > 0.0)) return result;
    result = (_income / _duration) * ADJ_FACTOR;
    return result;
}
```

Когда в условной инструкции я вижу такие отрицания, у меня просто "сносит крышу", так что я тут же упрощаю его следующим образом:

```
public double getAdjustedCapital() {
    double result = 0.0;
    if (_capital <= 0.0) return result;
    if (_intRate <= 0.0 || _duration <= 0.0) return result;
    result = (_income / _duration) * ADJ_FACTOR;
    return result;
}
```

В таких ситуациях я стараюсь использовать в операторах `return` явные значения. Благодаря этому можно легко увидеть возвращаемый результат (впрочем, здесь я подумал бы о применении рефакторинга "Замена магического числа символической константой" (с. 223)).

```
public double getAdjustedCapital() {
    double result = 0.0;
    if (_capital <= 0.0) return 0.0;
    if (_intRate <= 0.0 || _duration <= 0.0) return 0.0;
    result = (_income / _duration) * ADJ_FACTOR;
    return result;
}
```

После этого можно полностью удалить временную переменную:

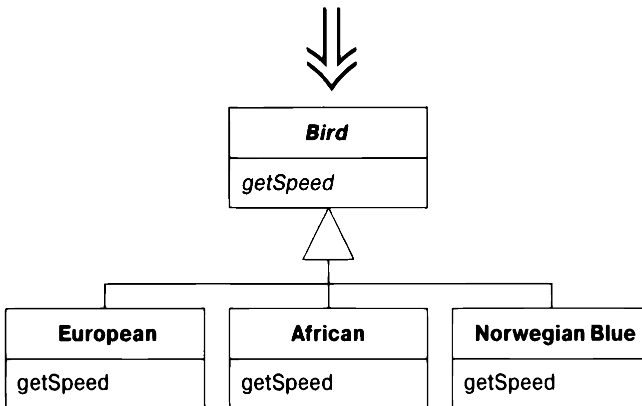
```
public double getAdjustedCapital() {
    if (_capital <= 0.0) return 0.0;
    if (_intRate <= 0.0 || _duration <= 0.0) return 0.0;
    return (_income / _duration) * ADJ_FACTOR;
}
```

Замена условной инструкции полиморфизмом (Replace Conditional with Polymorphism)

Имеется условная инструкция, обеспечивающая разное поведение в зависимости от типа объекта.

Переместите каждую ветвь условной инструкции в перекрытый метод подкласса. Сделайте исходный метод абстрактным.

```
double getSpeed() {
    switch (_type) {
        case EUROPEAN:
            return getBaseSpeed();
        case AFRICAN:
            return getBaseSpeed()getLoadFactor() *
                   _numberOfCoconuts;
        case NORWEGIAN_BLUE:
            return (_isNailed) ? 0 : getBaseSpeed(_voltage);
    }
    throw new RuntimeException("Сюда мы не должны попасть!");
}
```



Мотивация

Одним из наиболее впечатляющих терминов объектного программирования является *полиморфизм*. Полиморфизм позволяет избежать написания явных условных инструкций, когда есть объекты, поведение которых зависит от их типа.

В результате оказывается, что операторы `switch`, выполняющие переключение в зависимости от кода типа, или инструкции `if-then-else`, выполняющие выбор действий в зависимости от строки типа, в объектно-ориентированных программах встречаются значительно реже.

Полиморфизм дает множество преимуществ. Наибольшая отдача от него имеет место тогда, когда один и тот же набор условий появляется во многих местах программы. Если необходимо ввести новый тип, то приходится отыскивать и изменять все соответствующие условные инструкции. Но при использовании подклассов достаточно создать новый подкласс и обеспечить в нем соответствующие

методы. Клиенты класса не должны ничего знать о подклассах, благодаря чему сокращается количество зависимостей в системе и упрощается ее модификация.

Техника

Прежде чем применять данный рефакторинг, следует создать необходимую иерархию наследования. Такая иерархия может уже иметься как результат ранее проведенного рефакторинга. Если такой иерархии нет, ее надо создать.

Чтобы создать иерархию наследования, можно воспользоваться рефакторингом "Замена кода типа подклассами" (с. 241) или рефакторингом "Замена кода типа состоянием/стратегией" (с. 245). Более простым вариантом является создание подклассов, поэтому лучше выбирать его. Однако, если код типа изменяется после того, как создан объект, применять создание подклассов нельзя, так что следует прибегнуть к проектному шаблону "состояния/стратегии". Этот шаблон должен использоваться и тогда, когда подклассы данного класса уже создаются по иным причинам. Не забывайте, что если несколько операторов `case` выполняют переключение для одного и того же кода типа, для этого кода типа нужно создать лишь структуру наследования.

Теперь можно предпринять атаку на условную инструкцию. Код, на который мы нацелились, может представлять собой инструкцию `switch`, а может быть и инструкцией `if`.

- Если условная инструкция является частью более крупного метода, разделите условную инструкцию на части и примените рефакторинг "Извлечение метода" (с. 132).
- При необходимости воспользуйтесь рефакторингом "Перенос метода" (с. 162), чтобы поместить условную инструкцию на вершину иерархии наследования.
- Выберите один из подклассов. Создайте метод подкласса, перекрывающий метод условной инструкции. Скопируйте тело этой ветви условной инструкции в метод подкласса и выполните необходимые настройки.

  ⇛ *Для этого может потребоваться сделать некоторые закрытые члены надкласса защищенными.*
- Выполните компиляцию и тестирование.
- Удалите скопированную ветвь из условной инструкции.
- Выполните компиляцию и тестирование.
- Повторяйте эти действия с каждой ветвью условной инструкции, пока все они не превратятся в методы подкласса.
- Сделайте метод суперкласса абстрактным.

Пример

Я обращусь к скучному и упрощенному примеру расчета зарплаты служащих. Здесь применяются классы, полученные после применения рефакторинга “Замена кода типа состоянием/стратегией” (с. 245), поэтому объекты имеют такой вид, как показано на рис. 9.1 (см. пример из главы 8, “Организация данных”).

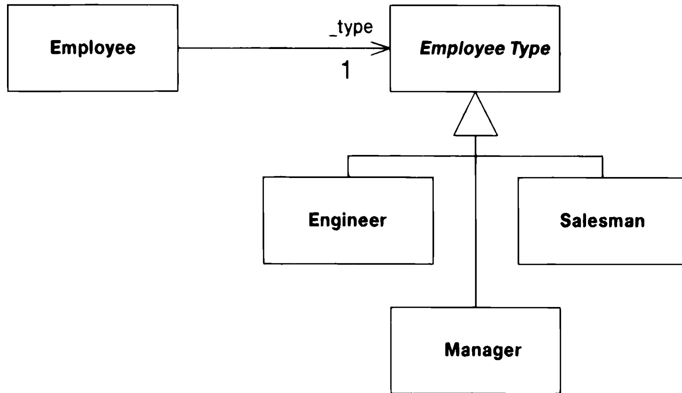


Рис. 9.1. *Структура наследования*

```
class Employee...
    int payAmount() {
        switch (getType()) {
            case EmployeeType.ENGINEER:
                return _monthlySalary;
            case EmployeeType.SALESMAN:
                return _monthlySalary + _commission;
            case EmployeeType.MANAGER:
                return _monthlySalary + _bonus;
            default:
                throw new RuntimeException("Incorrect Employee");
        }
    }
    int getType() {
        return _type.getTypeCode();
    }
    private EmployeeType _type;

abstract class EmployeeType...
    abstract int getTypeCode();

class Engineer extends EmployeeType...
    int getTypeCode() {
        return Employee.ENGINEER;
    }

... и другие подклассы
```

Инструкция switch уже извлечена, так что здесь мне делать нечего. Я должен только переместить ее в EmployeeType, поскольку это тот класс, для которого будут созданы подклассы.

```
class EmployeeType...
    int payAmount(Employee emp) {
        switch (getTypeCode()) {
            case ENGINEER:
                return emp.getMonthlySalary();
            case SALESMAN:
                return emp.getMonthlySalary() + emp.getCommission();
            case MANAGER:
                return emp.getMonthlySalary() + emp.getBonus();
            default:
                throw new RuntimeException("Incorrect Employee");
        }
    }
```

Так как мне нужны данные из объекта Employee, я должен передать его в качестве аргумента. Некоторые из этих данных можно переместить в объект типа Employee, но это уже тема другого рефакторинга.

Если компиляция проходит успешно, я изменяю метод payAmount в классе Employee так, чтобы он делегировал обработку новому классу:

```
class Employee...
    int payAmount() {
        return _type.payAmount(this);
    }
```

Теперь можно заняться инструкцией выбора. Это будет похоже на расправу мальчишки с насекомым — я буду отрывать ему “лапки” одну за другой. Сначала я скопирую ветвь Engineer в класс Engineer.

```
class Engineer...
    int payAmount(Employee emp) {
        return emp.getMonthlySalary();
    }
```

Этот новый метод замещает весь оператор case для инженеров. Поскольку я параноик, то иногда на всякий случай ставлю в операторе case ловушку:

```
class EmployeeType...
    int payAmount(Employee emp) {
        switch (getTypeCode()) {
            case ENGINEER:
                throw new RuntimeException("Должен быть перекрыт");
            case SALESMAN:
                return emp.getMonthlySalary()+emp.getCommission();
```

```
            case MANAGER:
                return emp.getMonthlySalary()+emp.getBonus();
            default:
                throw new RuntimeException("Неверный тип");
        }
    }
```

Я продолжаю выполнение, пока не будут удалены все "лапки"-ветви:

```
class Salesman...
    int payAmount(Employee emp) {
        return emp.getMonthlySalary() + emp.getCommission();
    }

class Manager...
    int payAmount(Employee emp) {
        return emp.getMonthlySalary() + emp.getBonus();
    }
```

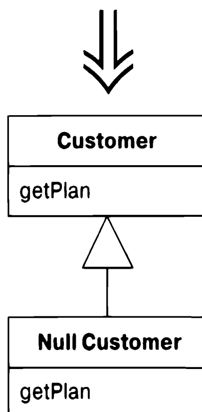
А теперь я делаю метод суперкласса абстрактным:

```
class EmployeeType...
    abstract int payAmount(Employee emp);
```

Введение нулевого объекта (Introduce Null Object)

В исходном тексте имеются многократные проверки на нулевое значение.
Замените нулевое значение нулевым объектом.

```
if (customer == null) plan = BillingPlan.basic();
else plan = customer.getPlan();
```



Мотивация

Сущность полиморфизма в том, что вместо запроса типа у объекта и выполнения того или иного поведения в зависимости от ответа вы просто вызываете требуемое поведение. Объект же, в зависимости от своего типа, делает именно то, что нужно. Все это не так просто, когда значением поля является нулевой объект. Предоставлю слово Рону Джеффризу.

Рон Джеффриз (Ron Jeffries)

Мы впервые стали применять проектный шаблон нулевого объекта, когда Рич Гарзанити обнаружил, что перед тем как послать объекту сообщение, в коде очень часто проверяется, существует ли этот объект. Мы сперва выясняли, не равен ли объект персоны значению `null`. Если объект существовал, мы обращались к нему за значением `rate`. Это делалось во множестве мест, и это стало нас сильно раздражать.

Поэтому мы создали объект отсутствующей персоны, который сообщал, что у него нулевое значение `rate` (мы называем наши нулевые объекты *отсутствующими*). Вскоре у отсутствующего объекта было уже множество методов, включая `rate`. Сейчас в нашем проекте более 80 классов нулевых объектов.

Чаще всего мы применяем нулевые объекты при выводе информации. Например, когда выводится информация о персоне, у соответствующего объекта может отсутствовать любой из примерно 20 атрибутов. Если бы мы позволили им быть нулевыми, вывод информации стал бы очень запутанным. Вместо этого мы подключаем различные нулевые объекты, которые корректно отображают информацию о себе. В результате мы избавились от большого объема процедурного кода.

Наиболее остроумно мы применяем нулевой объект для отсутствующего сеанса Gemstone. Мы используем базу данных Gemstone для производственного кода, но разработку предпочитаем вести без нее и сбрасываем новый код в Gemstone примерно раз в неделю или около того. В коде есть несколько мест, где необходимо регистрироваться в сеансе Gemstone. При работе без Gemstone мы просто подключаем отсутствующий сеанс Gemstone. Он выглядит так же, как реальный, но позволяет вести разработку и тестирование, не ощущая отсутствия базы данных.

Еще одно полезное применение нулевого объекта — для отсутствующего буфера. Буфер представляет собой коллекцию значений из платежной ведомости, которые приходится часто суммировать или обходить в цикле. Если какого-то буфера не существует, возвращается отсутствующий буфер, который действует так же, как пустой. Отсутствующий буфер знает, что у него нулевое сальдо и нет значений. При таком подходе нам удается избежать создания десятков пустых буферов для каждого из тысяч служащих.

Интересная особенность применения нулевых объектов состоит в том, что при этом почти никогда не возникают аварийные ситуации. Поскольку нулевой объект отвечает на те же сообщения, что и реальный объект, система в целом ведет себя обычным образом. Из-за этого иногда трудно заметить или локализовать проблему, потому что все работает нормально. Конечно, начав изучение объектов, вы где-нибудь да обнаружите нулевой объект, которого там быть не должно.

Помните, что нулевые объекты константны — в них никогда ничего не меняется. Соответственно, мы реализуем их с использованием проектного шаблона "Синглтон" [9]. Например, при каждом запросе отсутствующего лица вы будете получать один и тот же экземпляр этого класса.

Подробнее о проектном шаблоне "Нулевой объект" можно прочесть в работе [15].

Техника

- Создайте подкласс исходного класса, который будет действовать в качестве нулевой версии класса. Создайте операцию `isNull` в исходном и нулевом классах. В исходном классе она должна возвращать значение `false`, а в нулевом классе — `true`.
  - ⇛ *Может оказаться удобным создание явного нулевого интерфейса для метода* `isNull`.
  - ⇛ *Альтернативой может быть использование проверочного интерфейса для проверки на нулевое значение.*
- Выполните компиляцию.
- Найдите все места, где при запросе исходного объекта может возвращаться нулевое значение, и измените их таким образом, чтобы вместо этого возвращался нулевой объект.
- Найдите все места, где переменная типа исходного класса сравнивается с `null`, и замените код сравнения вызовом `isNull`.
  - ⇛ *Это можно сделать, заменяя поочередно каждый исходный класс вместе с его клиентами и выполняя компиляцию и тестирование после каждой замены.*
  - ⇛ *Может оказаться полезным использовать несколько вызовов* `assert`*, проверяющих на равенство нулю в местах, где значение* `null` *теперь встречаться не должно.*
- Выполните компиляцию и тестирование.

- Найдите случаи вызова клиентами операции для ненулевого значения и добавьте альтернативное поведение для нулевого объекта.
- Для каждого из этих случаев перекройте операции в нулевом классе некоторым альтернативным поведением.
- Удалите проверку условия там, где используется такое перекрытое поведение, и выполните компиляцию и тестирование.

Пример

Коммунальное предприятие знает свои участки: дома и квартиры, пользующиеся его услугами. У участка всегда есть пользователь (customer).

```
class Site...
    Customer getCustomer() {
        return _customer;
    }
    Customer _customer;
```

Пользователь обладает рядом функциональных возможностей. Я приведу три из них.

```
class Customer...
    public String getName() {
        ...
    }
    public BillingPlan getPlan() {
        ...
    }
    public PaymentHistory getHistory() {
        ...
    }
```

У истории платежей `PaymentHistory` есть собственные функции, например количество недель просрочки в прошлом году:

```
public class PaymentHistory...
    int getWeeksDelinquentInLastYear()
```

Показанные мною методы доступа к значениям позволяют клиентам получать эти данные. Однако иногда у участка нет пользователя. Кто-то мог уехать, а о въехавшем сведений пока нет. Раз такое может произойти, следует гарантировать обработку нулевого значения любым кодом, который использует объект `customer`. Приведу несколько примеров:

```
Customer customer = site.getCustomer();
BillingPlan plan;
```

```
if (customer == null) plan = BillingPlan.basic();
else plan = customer.getPlan();

...

String customerName;
if (customer == null) customerName = "occupant";
else customerName = customer.getName();

...

int weeksDelinquent;
if (customer == null) weeksDelinquent = 0;
else weeksDelinquent =
        customer.getHistory().getWeeksDelinquentInLastYear();
```

В таких ситуациях может иметься множество клиентов `site` и `customer`, и все они должны производить проверки на нулевое значение, выполняя одинаковые действия при обнаружении такового. Пожалуй, пришло время создать нулевой объект.

Первым делом я создаю нулевой класс для `customer` и модифицирую класс `Customer` так, чтобы он поддерживал запрос проверки на нулевое значение:

```
class NullCustomer extends Customer {
    public boolean isNull() {
        return true;
    }
}

class Customer...
    public boolean isNull() {
        return false;
    }

    protected Customer() {} //Требуется для NullCustomer
```

Если возможности модифицировать класс `Customer` нет, можно воспользоваться тестирующим интерфейсом (см. ниже раздел “Пример: тестирующий интерфейс”).

При желании можно сигнализировать об использовании нулевого объекта посредством интерфейса:

```
interface Nullable {
    boolean isNull();
}
class Customer implements Nullable
```

Для создания нулевых клиентов я хочу добавить фабричный метод. Таким образом, клиентам не обязательно будет знать о нулевом классе:

```
class Customer...
    static Customer newNull() {
        return new NullCustomer();
    }
```

Теперь наступает трудный момент. Я должен возвращать этот новый нулевой объект вместо нулевого значения и заменять проверки вида `foo == null` проверками `foo.isNull()`. Полезно найти все места, где запрашивается `customer`, и изменить их таким образом, чтобы возвращать вместо нулевого значения нулевого пользователя.

```
class Site...
    Customer getCustomer() {
        return (_customer == null) ?
               Customer.newNull() :
               _customer;
    }
```

Я должен также изменить все случаи использования этого значения, чтобы делать в них проверку с помощью вызова `isNull()`, а не сравнения с `null`.

```
Customer customer = site.getCustomer();
BillingPlan plan;

if (customer.isNull()) plan = BillingPlan.basic();
else plan = customer.getPlan();

...

String customerName;
if (customer.isNull()) customerName = "occupant";
else customerName = customer.getName();

...

int weeksDelinquent;
if (customer.isNull()) weeksDelinquent = 0;
else weeksDelinquent =
        customer.getHistory().getWeeksDelinquentInLastYear();
```

Несомненно, это самая сложная часть данного рефакторинга. Для каждого заменяемого источника `null` необходимо найти все случаи проверки на нулевое значение и отредактировать их. Если объект интенсивно передается в разные методы, их может быть нелегко отследить. Необходимо найти все переменные типа `customer` и все места их использования. К сожалению, эту процедуру

трудно разделить на мелкие шаги. Иногда я обнаруживаю источник, который используется лишь в нескольких местах, и тогда можно заменить его. Но чаще всего приходится делать много объемистых изменений. Вернуться к старой версии обработки нулевых объектов не так уж сложно, потому что обращения к `isNull` можно найти без особого труда (хотя это все равно достаточно трудоемкая процедура).

Когда этот этап завершен и успешно выполнены компиляция и тестирование, можно вздохнуть с облегчением. Теперь начинается приятная часть работы. В данный момент я ничего не выигрываю от применения `isNull` вместо сравнения с `null`. Преимущества появятся тогда, когда я перемещу поведение в нулевой объект `customer` и уберу условные операторы. Эти шаги можно выполнять по одному. Начну с имени. В настоящий момент у меня имеется следующий код клиента:

```
String customerName;
if (customer.isNull()) customerName = "occupant";
else customerName = customer.getName();
```

Я добавляю к нулевому классу подходящий метод получения имени:

```
class NullCustomer...
    public String getName() {
        return "occupant";
    }
```

Теперь можно убрать условный код:

```
String customerName = customer.getName();
```

Те же действия можно проделать с любым другим методом, в котором есть разумный общий ответ на запрос. Я могу также выполнить соответствующие действия для модификаторов. Так что клиентский код

```
if (! customer.isNull())
    customer.setPlan(BillingPlan.special());
```

можно заменить кодом

```
customer.setPlan(BillingPlan.special());

class NullCustomer...
    public void setPlan(BillingPlan arg) {}
```

Помните, что такое перемещение поведения оправдано только тогда, когда большинству клиентов нужен один и тот же ответ. Обратите внимание: я сказал "большинству", а не "всем". Клиенты, которым нужен ответ, отличный

от стандартного, могут по-прежнему выполнять проверку с помощью вызова isNull. Выигрыш имеет место тогда, когда многим клиентам требуется одно и то же; они могут просто полагаться на поведение по умолчанию нулевого класса.

Данный пример содержит несколько иной случай, а именно — код клиента, в котором используется результат обращения к клиенту:

```
if (customer.isNull()) weeksDelinquent = 0;
else weeksDelinquent =
        customer.getHistory().getWeeksDelinquentInLastYear();
```

С этим можно справиться, создав нулевую историю платежей:

```
class NullPaymentHistory extends PaymentHistory...
    int getWeeksDelinquentInLastYear() {
        return 0;
    }
```

Я модифицирую нулевой клиент так, чтобы при запросе он возвращал нулевой класс:

```
class NullCustomer...
    public PaymentHistory getHistory() {
        return PaymentHistory.newNull();
    }
```

И снова я могу удалить условный код:

```
int weeksDelinquent =
    customer.getHistory().getWeeksDelinquentInLastYear();
```

Зачастую оказывается, что одни нулевые объекты возвращают другие нулевые объекты.

Пример: тестирующий интерфейс

Тестирующий интерфейс служит альтернативой определению метода isNull. При этом подходе я создаю нулевой интерфейс, в котором не определены никакие методы:

```
interface Null {}
```

Затем я реализую этот нулевой интерфейс в своих нулевых объектах:

```
class NullCustomer extends Customer implements Null...
```

После этого проверка на "нулевость" выполняется с помощью оператора instanceof:

```
aCustomer instanceof Null
```

Обычно, заметив на горизонте оператор instanceof, я с ужасом убегаю, но в данном случае его применение вполне оправданно. Его особое преимущество в том, что при этом не нужно изменять класс customer. Это позволяет нам пользоваться нулевым объектом даже тогда, когда доступ к исходному коду customer отсутствует.

Другие особые случаи

При выполнении данного рефакторинга можно иметь несколько разновидностей нулевых объектов. Часто имеется разница между отсутствием customer (новое здание, в котором никто не живет) и отсутствием сведений о customer (кто-то в теремке живет, но кто — неизвестно). В такой ситуации можно создать отдельные классы для разных нулевых случаев. Иногда нулевые объекты могут содержать фактические данные, например регистрировать пользование услугами неизвестным жильцом, чтобы впоследствии, когда станет известно, кто является жильцом, выставить ему счет.

В сущности, здесь должен применяться более крупный проектный шаблон, именуемый "особым", или "частным случаем" (special case). Класс частного случая — это отдельный экземпляр класса с особым поведением. Таким образом, неизвестный клиент UnknownCustomer и отсутствующий клиент NoCustomer будут частными случаями Customer. Такие частные случаи часто встречаются среди чисел. В Java у чисел с плавающей точкой есть особые значения для положительной и отрицательной бесконечностей и для "не числа" (NaN). Польза от таких значений в том, что благодаря им сокращается объем кода, обрабатывающего ошибки. Операции над числами с плавающей точкой не генерируют исключения. Выполнение любой операции, в которой участвует NaN, дает в качестве результата тоже NaN, подобно тому как методы доступа к нулевым объектам обычно возвращают нулевые объекты.

Введение утверждения (Introduce Assertion)

Некоторая часть кода предполагает определенное состояние программы.

Сделайте предположение явным с помощью оператора утверждения.

```
double getExpenseLimit() {
    // Должен иметься либо предел цены, либо основной проект
```

```
    return (_expenseLimit != NULL_EXPENSE) ?
           _expenseLimit :
           _primaryProject.getMemberExpenseLimit();
}
```

```
double getExpenseLimit() {
    Assert.isTrue(_expenseLimit != NULL_EXPENSE ||
                  _primaryProject != null);
    return (_expenseLimit != NULL_EXPENSE) ?
           _expenseLimit :
           _primaryProject.getMemberExpenseLimit();
}
```

Мотивация

Часто некоторые разделы кода могут работать только при выполнении определенных условий. Например, вычисление квадратного корня требует, чтобы входное значение было неотрицательным. Для объекта это может быть предположение о том, что хотя бы одно поле в группе полей содержит значение.

Такие предположения часто оказываются несформулированными и могут быть выяснены только при детальном изучении алгоритма. Иногда предположения о таких нормальных условиях указаны в комментариях. Но лучше всего, когда такое предположение явно формулируется в виде утверждения.

Утверждение (assertion) представляет собой условную конструкцию, которая всегда должна выполняться. Сбой утверждения свидетельствует об ошибке в программе, а потому он должен всегда приводить к необрабатываемому исключению. Утверждения никогда не должны использоваться другими частями системы. В действительности утверждения обычно полностью удаляются из готового кода. Поэтому важно указать, что нечто представляет собой утверждение.

Утверждения способствуют организации сообщений и отладки. Их коммуникативная роль состоит в том, что они помогают читателю понять, какие предположения делает код. Во время отладки утверждения помогают перехватывать ошибки ближе к моменту их возникновения. По моим наблюдениям, при написании самотестирующегося кода помощь утверждений для отладки невелика, но я очень ценю коммуникативную роль утверждений.

Техника

Поскольку утверждения не должны оказывать влияния на работу системы, их добавление всегда сохраняет поведение.

- Когда вы видите, что некоторое условие предполагается всегда выполненным, добавьте утверждение, в котором это явно сформулировано.

  ⇒ *Следует иметь в наличии класс утверждения, который можно использовать для такого поведения.*

Не злоупотребляйте утверждениями. Не следует стараться проверить с их помощью все условия, которые, по вашему мнению, должны быть истинны в некотором фрагменте кода. Используйте утверждения только для проверки того, что *обязано* быть истинным. Чрезмерное применение утверждений может привести к дублированию логики, что мешает поддержке кода. Логика, охватывающая предположение, полезна, потому что заставляет заново обдумать часть кода. Если код работает и без утверждений, то от последних больше путаницы и трудностей при последующем изменении кода, чем пользы.

Всегда проверяйте, будет ли работоспособен код в случае невыполнения утверждения. Если это так, такое утверждение следует удалить.

Остерегайтесь дублировать код в утверждениях — продублированный код в утверждениях пахнет ничуть не лучше, чем в любом другом месте. Не бойтесь выполнять рефакторинг "Извлечение метода" (с. 132), чтобы избежать дублирования.

Пример

Вот простая история с пределами затрат. Служащим может назначаться индивидуальный предел затрат. Если служащие участвуют в основном проекте, то могут пользоваться пределами затрат этого проекта. У них не обязан быть предел затрат или основной проект, но что-то из этих двух должно иметься в обязательном порядке. Такое предположение предполагается выполненным в коде, который использует пределы затрат:

```
class Employee...
    private static final double NULL_EXPENSE = -1.0;
    private double _expenseLimit = NULL_EXPENSE;
    private Project _primaryProject;
    double getExpenseLimit() {
        return (_expenseLimit != NULL_EXPENSE) ?
                _expenseLimit :
                _primaryProject.getMemberExpenseLimit();
    }
    boolean withinLimit(double expenseAmount) {
        return (expenseAmount <= getExpenseLimit());
    }
```

В этом коде сделано неявное допущение, что у служащего есть предел затрат — либо проекта, либо индивидуальный. Такое утверждение следует формулировать в коде явно:

```
double getExpenseLimit() {
    Assert.isTrue(_expenseLimit != NULL_EXPENSE ||
                  _primaryProject != null);
    return (_expenseLimit != NULL_EXPENSE) ?
           _expenseLimit :
           _primaryProject.getMemberExpenseLimit();
}
```

Это утверждение ни в коей мере не меняет поведение программы. В любом случае, если утверждение не выполнено, возникает исключение времени выполнения: либо исключение из-за нулевого указателя в withinLimit, либо исключение времени выполнения в Assert.isTrue. В ряде ситуаций такое утверждение помогает обнаружить ошибку, потому что находится ближе к месту ее возникновения. Однако утверждение главным образом сообщает о том, как работает код и какие предположения он делает.

Я часто применяю рефакторинг "Извлечение метода" (с. 132) к условному выражению внутри утверждения. С его помощью либо устраняется дублирование кода, либо проясняется смысл условия.

Одна из сложностей применения утверждений в Java связана с отсутствием простого механизма их вставки. Утверждения должны легко удаляться, чтобы не оказывать влияния на производительность окончательной версии кода. Конечно, наличие такого вспомогательного класса, как Assert, облегчает ситуацию. К сожалению, любое выражение в утверждении выполняется, что бы ни случилось. Единственным способом не допустить этого является код наподобие следующего:

```
double getExpenseLimit() {
    Assert.isTrue(Assert.ON &&
                  (_expenseLimit != NULL_EXPENSE ||
                   _primaryProject != null));
    return (_expenseLimit != NULL_EXPENSE) ?
           _expenseLimit :
           _primaryProject.getMemberExpenseLimit();
}
```

или

```
double getExpenseLimit() {
    if (Assert.ON)
        Assert.isTrue(_expenseLimit != NULL_EXPENSE ||
                      _primaryProject != null);
    return (_expenseLimit != NULL_EXPENSE) ?
           _expenseLimit :
           _primaryProject.getMemberExpenseLimit();
}
```

Если Assert.ON представляет собой константу, компилятор должен обнаружить и удалить "мертвый" код при ложном значении этой константы. Однако добавление такого условного выражения запутывает код, так что многие программисты предпочитают просто использовать Assert, а затем, при создании производственной версии кода, удалять все строки, в которых есть assert (например, с помощью Perl).

В классе Assert должны быть разные методы с "говорящими" именами. Помимо isTrue, в нем могут иметься такие методы, как equals или shouldNeverReachHere.

Глава 10

Упрощение вызовов методов

Суть объектов заключена в интерфейсах. Главным при разработке хорошего объектно-ориентированного программного обеспечения является создание простых в понимании и применении интерфейсов. В данной главе изучаются рефакторинги, упрощающие интерфейсы.

Часто самое простое и важное, что вы можете сделать, — это дать методу другое имя. Присваивание "говорящих" имен является ключевым элементом коммуникации. Если вам понятно, как работает программа, не надо бояться применить рефакторинг "Переименование метода" (с. 290), чтобы поделиться этим пониманием. Кроме того, можно (и нужно) переименовывать переменные и классы. В общем случае такие переименования выполняются с помощью простой текстовой замены, так что я не добавлял для них особых методов рефакторинга.

В интерфейсах существенную роль играют параметры. Поэтому распространенными рефакторингами являются "Добавление параметра" (с. 292) и "Удаление параметра" (с. 294). Программисты-новички, не имеющие опыта работы с объектами, часто используют длинные списки параметров, типичные для других сред разработки. Объекты позволяют получать короткие списки параметров, а ряд рефакторингов позволяет сделать их еще короче. При передаче нескольких значений из объекта можно применить рефакторинг "Сохранение всего объекта" (с. 305) и свести все значения к одному объекту. Если такой объект не существует, его можно создать с помощью рефакторинга "Введение объекта параметра" (с. 312). Если же данные могут быть получены из объекта, к которому у метода уже есть доступ, параметры можно исключить с помощью рефакторинга "Замена параметра вызовом метода" (с. 308). Если параметры используются для определения условного поведения, можно выполнить рефакторинг "Замена параметра явными методами" (с. 302). Несколько похожих методов можно объединить в один, добавляя параметр с помощью рефакторинга "Параметризация метода" (с. 300).

Дуг Ли (Doug Lea) предупредил меня о проведении рефакторингов, которые приводят к сокращению списков параметров. Обычно такие рефакторинги выполняются таким образом, чтобы передаваемые параметры были неизменяемыми, такими как встроенные объекты или объекты-значения. Обычно длинные

списки параметров можно заменить неизменяемыми объектами; в противном же случае рефакторинги из этой группы следует выполнять с особой осторожностью.

На протяжении многих лет я придерживаюсь одного очень ценного соглашения — о четком разделении методов, изменяющих состояние (модификаторов), и методов, запрашивающих состояние (запросов). Множество раз я сам сталкивался с неприятностями (или видел, как попадают в беду другие) из-за смешения этих функций. Поэтому, увидев такое смешение, я немедленно использую рефакторинг "Разделение запроса и модификатора" (с. 296), чтобы избавиться от такого смешения.

Хорошие интерфейсы показывают не более того, что необходимо показать, и ничего лишнего. Скрывая некоторые вещи, можно улучшить интерфейс. Конечно, должны быть скрыты все данные (я думаю, нет необходимости напоминать об этом), а также все методы, которые можно скрыть. При выполнении рефакторинга часто возникает необходимость что-то временно открыть, а затем скрыть с помощью рефакторинга "Сокрытие метода" (с. 320) или "Удаление метода установки значения" (с. 317).

Особенное неудобство в Java и C++ представляют собой конструкторы, требующие знания класса создаваемого объекта. Зачастую знать его необязательно. Необходимость в знании можно устранить, применив рефакторинг "Замена конструктора фабричным методом" (с. 321).

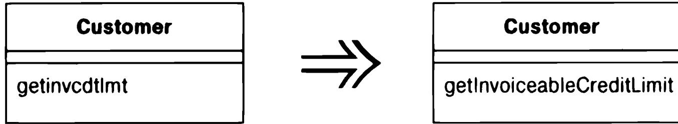
Жизнь программистов на языке программирования Java отравляет также преобразование типов. Старайтесь избавлять пользователей классов от необходимости выполнять нисходящее преобразование, если вы в состоянии разместить его в некотором ином месте с помощью рефакторинга "Инкапсуляция нисходящего приведения типа" (с. 325).

В Java, как и во многих других современных языках программирования, имеется механизм обработки исключений ситуаций, делающий более простой обработку ошибок. Не привыкшие к нему программисты для сообщения о возникших неприятностях часто используют коды ошибок. Чтобы воспользоваться новым механизмом обработки исключений, можно применить рефакторинг "Замена кода ошибки исключением" (с. 327). Однако бывает, что исключения не являются правильным решением, и тогда следует выполнить рефакторинг "Замена исключения проверкой" (с. 332).

Переименование метода (Rename Method)

Имя метода не раскрывает его назначения.

Измените имя метода.



Мотивация

Важной частью пропагандируемого мною стиля программирования является разделение больших и сложных процедур на небольшие методы. Если делать это неверно, придется изрядно помучиться, выясняя, что же делают полученные маленькие методы. Избежать таких мучений помогают хорошие имена для этой "мелочи". Методам следует давать имена, ясно раскрывающие их назначение. Хороший способ для этого — представить, каким должен быть комментарий к методу, и преобразовать этот комментарий в имя метода.

Жизнь такова, что удачное имя может прийти в голову не сразу. В такой ситуации может возникнуть соблазн забросить это занятие — ну ведь не в имени же счастье! Не слушайте эти голоса — это вас соблазняет демон Путаник, не слушайте его. Если вы видите, что у метода плохое имя, обязательно измените его. Помните, что ваш код предназначен, в первую очередь, для человека, и только потом — для компьютера. Человеку требуются хорошие имена. Вспомните, сколько времени вы пытались разобраться в коде с плохими именами, чтобы понять, чем же занят метод `s4x27uv`, и насколько проще было бы, окажись у него более понятное имя. Создание хороших имен — это искусство, требующее практики; овладение этим искусством — важный шаг на пути к превращению в действительно хорошего программиста. То же утверждение справедливо и в отношении других элементов сигнатуры метода. Если переупорядочение способно пояснить суть, выполните его (см. рефакторинги "Добавление параметра" (с. 292) и "Удаление параметра" (с. 294)).

Техника

- Выясните, где реализована сигнатура метода — в суперклассе или подклассе. Выполните приведенные шаги для каждой из реализаций.
- Объявите новый метод с новым именем. Скопируйте тело прежнего метода в метод с новым именем и осуществите необходимую настройку.
- Выполните компиляцию.
- Измените тело прежнего метода так, чтобы в нем вызывался новый метод.

  ⇛ *Если ссылок на метод мало, этот шаг вполне можно пропустить.*
- Выполните компиляцию и тестирование.

- Найдите все обращения к прежнему методу и замените их вызовами нового. Выполняйте компиляцию и тестирование после каждой замены.
- Удалите старый метод.
  - ⇒ *Если старый метод является частью интерфейса и его нельзя удалить, сохраните его и документируйте как устаревший.*
- Выполните компиляцию и тестирование.

Пример

Пусть имеется метод для получения номера телефона человека:

```
public String getTelephoneNumber() {
    return ("(" + _officeAreaCode + ") " + _officeNumber);
}
```

Я хочу переименовать метод в `getOfficeTelephoneNumber`. Я начинаю с создания нового метода и копирования тела старого метода в новый метод. Старый метод изменяется так, чтобы вызывать новый:

```
class Person...
    public String getTelephoneNumber() {
        return getOfficeTelephoneNumber();
    }

    public String getOfficeTelephoneNumber() {
        return ("(" + _officeAreaCode + ") " + _officeNumber);
    }
```

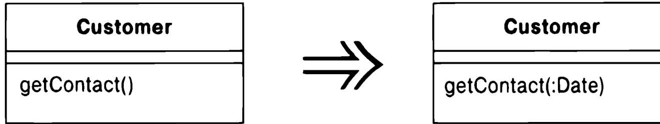
Теперь я нахожу места вызова старого метода и изменяю их так, чтобы вместо старого вызывался новый метод. После внесения всех изменений прежний метод можно удалить.

Та же процедура осуществляется, когда нужно добавить или удалить параметр.

Если мест, где вызывается метод, немного, я сразу заменяю их обращением к новому методу без использования старого в качестве делегирующего. Если в результате тесты будут показывать проблемы, я выполню откат и заново произведу изменения, теперь уже без лишней поспешности.

Добавление параметра (Add Parameter)

Метод нуждается в дополнительной информации от вызывающего метода.
Добавьте параметр, который может передать эту информацию.



Мотивация

Это очень распространенный рефакторинг, и вы наверняка уже имели с ним дело. Его мотивация проста — изменить метод, причем это изменение требует дополнительной информации, которая ранее в метод не передавалась, так что вы добавляете новый параметр.

Фактически я хочу не столько рассказать о том, когда этот рефакторинг нужен, сколько о том, когда его не следует проводить. Зачастую добавлению параметра имеются более предпочтительные альтернативы, которые не приводят к увеличению списка параметров. Длинные списки параметров плохо пахнут, потому что их трудно запоминать и они часто содержат группы данных.

Взгляните на уже имеющиеся параметры. Может, вы можете запросить у одного из этих объектов необходимую информацию? Если нет, то не будет ли разумно создать в одном из них метод, предоставляющий эту информацию? Для чего будет использована эта информация? Не лучше было бы иметь данное поведение в другом объекте — в том, который обладает этой информацией? Рассмотрите имеющиеся параметры и подумайте о них вместе с новым параметром. Не лучше ли будет выполнить рефакторинг "Введение объекта параметра" (с. 312)?

Я не утверждаю, что параметры не нужно добавлять ни при каких обстоятельствах; я сам часто делаю это, но не следует забывать и об альтернативных возможностях.

Техника

Техника данного рефакторинга очень похожа на технику рефакторинга "Переименование метода" (с. 290).

- Выясните, где реализуется сигнатура метода — в суперклассе или в подклассе. Выполните эти шаги для каждой из реализаций.
- Объявите новый метод с добавленным параметром. Скопируйте тело старого метода в метод с новым именем и осуществите необходимую настройку.

  ⇛ *Если требуется одновременно добавить несколько параметров, проще сделать это сразу.*
- Выполните компиляцию.
- Измените тело прежнего метода так, чтобы в нем вызывался новый метод.

⇛ *Если ссылок на метод мало, этот шаг вполне можно пропустить.*

⇛ *В качестве значения параметра можно передать любое значение, но обычно используется* `null` *для параметра-объекта и необычное значение для встроенных типов. Часто полезно использовать числа, отличные от нуля, чтобы было проще их отследить.*

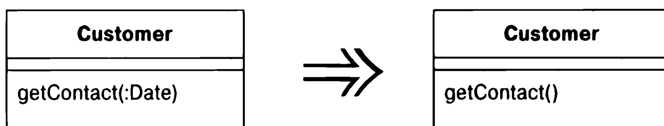
- Выполните компиляцию и тестирование.
- Найдите все вызовы прежнего метода и замените их вызовами нового метода. Выполняйте компиляцию и тестирование после каждой замены.
- Удалите старый метод.

⇛ *Если старый метод является частью интерфейса и его нельзя удалить, сохраните его и документируйте как устаревший.*

- Выполните компиляцию и тестирование.

Удаление параметра (Remove Parameter)

Параметр больше не используется телом метода.
Удалите его.



Мотивация

Программисты часто добавляют параметры и очень неохотно их удаляют. Ведь лишний параметр не вызывает никаких проблем, а в будущем может снова понадобиться.

Учтите, что это вас опять соблазняет уже упоминавшийся демон Путаник; гоните его вон! Наличие параметра указывает, что данная информация необходима; разные значения играют свою роль. Тот, кто вызывает ваш метод, должен озаботиться передачей правильных значений. Не удалив лишний параметр, вы обеспечите лишней работой всех, кто использует данный метод. Это нехороший подход; тем более что удаление параметров представляет собой простейший рефакторинг.

Нужно проявлять особую осторожность, если этот метод — полиморфный. В этом случае может оказаться, что данный параметр используется в других реализациях метода, и тогда удалять его не следует. Для использования в таких случаях можно добавить отдельный метод, но нужно изучить, как им пользуются вызывающие методы, чтобы решить, стоит ли это делать. Если вызывающим методам известно, что они имеют дело с некоторым подклассом, и они выполняют дополнительные действия для поиска значения этого параметра либо пользуются информацией об иерархии классов, чтобы выяснить, нельзя ли обойтись значением `null`, добавьте еще один метод без этого параметра. Если же им не требуется знать о том, какой метод какому классу принадлежит, оставьте вызывающие методы в счастливом неведении.

Техника

Техника данного рефакторинга очень похожа на технику рефакторингов "Переименование метода" (с. 290) и "Добавление параметра" (с. 292).

- Выясните, реализуется ли сигнатура метода в суперклассе или подклассе. Выясните, использует ли класс или суперкласс этот параметр. Если да, не выполняйте этот рефакторинг.
- Объявите новый метод без данного параметра. Скопируйте тело прежнего метода в метод с новым именем и осуществите необходимую настройку.

  ⇛ *Если требуется удалить несколько параметров, проще удалить их сразу все.*
- Выполните компиляцию.
- Измените тело старого метода так, чтобы в нем вызывался новый метод.

  ⇛ *Если вызовов метода немного, этот шаг вполне можно пропустить.*
- Выполните компиляцию и тестирование.
- Найдите все вызовы прежнего метода и замените их вызовами нового метода. Выполняйте компиляцию и тестирование после каждой замены.
- Удалите старый метод.

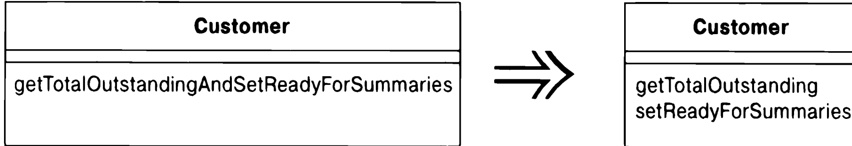
  ⇛ *Если старый метод является частью интерфейса и его нельзя удалить, сохраните его и документируйте как устаревший.*
- Выполните компиляцию и тестирование.

Поскольку я чувствую себя вполне уверенно при добавлении и удалении параметров, я часто делаю это за один этап сразу для большой группы.

Разделение запроса и модификатора (Separate Query from Modifier)

Имеется метод, не только возвращающий значение, но еще и изменяющий состояние объекта.

Создайте два метода: один для запроса, второй — для модификации.



Мотивация

Если есть функция, которая возвращает значение и не имеет наблюдаемых побочных действий, это очень хорошо. Такую функцию можно вызывать сколь угодно часто. Ее вызов можно переместить в другое место метода. Словом, с ней значительно меньше проблем.

Это вообще хорошая идея — различать методы с побочными действиями и методы, у которых их нет. Полезно следовать правилу, что у любого метода, возвращающего значение, не должно быть наблюдаемых побочных действий. Некоторые программисты рассматривают это правило как абсолютное [13]. Я не придерживаюсь его в 100% случаев (впрочем, как и любых других правил), но в основном стараюсь его соблюдать, и оно хорошо мне служит.

Обнаружив метод, который возвращает значение, но при этом обладает побочными действиями, следует попытаться разделить запрос и модификатор.

Возможно, вы обратили внимание на слова "*наблюдаемые* побочные действия". Распространенной оптимизацией является кеширование значения запроса в поле с тем, чтобы повторные запросы выполнялись быстрее. Хотя при этом состояние объекта с кешем изменяется, такое изменение не наблюдаемо. Любая последовательность запросов всегда возвращает одни и те же результаты для каждого запроса [13].

Техника

- Создайте запрос, возвращающий то же значение, что и исходный метод.
  - ⇛ *Посмотрите, что именно возвращает исходный метод. Если возвращается значение временной переменной, найдите место, где она получает свое значение.*

- Модифицируйте исходный метод так, чтобы он возвращал результат вызова запроса.
  - ⇒ *Все операторы* `return` *в исходном методе должны иметь вид* `return newQuery()`.
  - ⇒ *Если временная переменная использовалась в методе с единственной целью захвата возвращаемого значения, ее, скорее всего, можно удалить.*
- Выполните компиляцию и тестирование.
- Для каждого вызова замените единое обращение к исходному методу вызовом запроса. Добавьте вызов исходного метода перед строкой с вызовом запроса. Выполняйте компиляцию и тестирование после каждого изменения вызова метода.
- Объявите для исходного метода тип возвращаемого значения `void` и удалите выражения `return`.

Пример

Приведенная функция сообщает мне имя взломщика системы безопасности и посылает сигнал тревоги. Согласно имеющемуся правилу отправляется только один сигнал, даже если злодеев несколько:

```
String foundMiscreant(String[] people) {
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals("Don")) {
            sendAlert();
            return "Don";
        }
        if (people[i].equals("John")) {
            sendAlert();
            return "John";
        }
    }
    return "";
}
```

Вызов этой функции выполняется здесь:

```
void checkSecurity(String[] people) {
    String found = foundMiscreant(people);
    someLaterCode(found);
}
```

Чтобы отделить запрос от модификатора, я должен сначала создать подходящий запрос, который возвращает то же значение, что и модификатор, но не выполняет побочных действий:

```
String foundPerson(String[] people) {
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals("Don")) {
            return "Don";
        }
        if (people[i].equals("John")) {
            return "John";
        }
    }
    return "";
}
```

После этого я поочередно заменяю все `return` в исходной функции вызовами нового запроса. После каждой замены я выполняю тестирование. В результате первоначальный метод принимает следующий вид:

```
String foundMiscreant(String[] people) {
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals("Don")) {
            sendAlert();
            return foundPerson(people);
        }

        if (people[i].equals("John")) {
            sendAlert();
            return foundPerson(people);
        }
    }

    return foundPerson(people);
}
```

Теперь я изменю все вызывающие методы так, чтобы в них происходило два вызова: сначала — модификатора, а потом — запроса:

```
void checkSecurity(String[] people) {
    foundMiscreant(people);
    String found = foundPerson(people);
    someLaterCode(found);
}
```

Проделав это для всех вызовов, я могу установить для модификатора тип возвращаемого значения `void`:

```
void foundMiscreant(String[] people) {
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals("Don")) {
            sendAlert();
            return;
        }
```

```
        if (people[i].equals("John")) {
            sendAlert();
            return;
        }
    }
}
```

Теперь, пожалуй, лучше изменить имя оригинала:

```
void sendAlert(String[] people) {
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals("Don")) {
            sendAlert();
            return;
        }
        if (people[i].equals("John")) {
            sendAlert();
            return;
        }
    }
}
```

Конечно, в этом случае дублируется много кода, потому что модификатор использует для своей работы тело запроса. Теперь я могу применить к модификатору рефакторинг "Замена алгоритма" (с. 159):

```
void sendAlert(String[] people) {
    if (! foundPerson(people).equals(""))
        sendAlert();
}
```

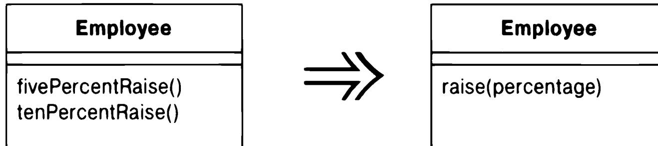
Вопросы многопоточности

Работающие в многопоточной среде знают, что выполнение операций проверки и установки в качестве единого действия является важной идиомой. Приводит ли это к конфликту с рассматриваемым рефакторингом? Я обсуждал эту проблему с Дугом Ли (Doug Lea) и пришел к выводу, что нет, но выполнить некоторые дополнительные действия необходимо. По-прежнему полезно разделение операций запроса и модификации, однако нужно сохранить и третий метод, осуществляющий и то, и другое одновременно. Операция "запрос и модификация" должна вызывать отдельные методы запроса и модификации и быть синхронизированной. Если операции запроса и модификации не синхронизированы, можно ограничить их видимость пакетом или сделать закрытыми. Благодаря этому получается безопасная синхронизированная операция, разделенная на два легких для понимания метода. Эти два метода более низкого уровня могут применяться для других целей.

Параметризация метода (Parameterize Method)

Несколько методов выполняют сходные действия, но с разными значениями, содержащимися в теле метода.

Создайте один метод, который использует для задания разных значений параметр.



Мотивация

Иногда встречаются два метода, выполняющие сходные действия, отличающиеся лишь несколькими значениями. В этом случае можно упростить ситуацию, заменив разные методы одним, который справляется с этими различиями с помощью параметров. При таком изменении устраняется дублирование кода и возрастает гибкость, потому что в результате добавления параметров можно обрабатывать и другие варианты.

Техника

- Создайте параметризованный метод, которым можно заменить каждый повторяющийся метод.
- Выполните компиляцию.
- Замените старый метод вызовом нового.
- Выполните компиляцию и тестирование.
- Повторите эти действия для каждого метода, выполняя тестирование после каждой замены.

Иногда можно выполнить такой рефакторинг не для всего метода, а только для его части. В таком случае сначала выделите фрагмент в отдельный метод, а затем параметризуйте его.

Пример

Вот простейший случай таких методов:

```
class Employee {
    void tenPercentRaise() {
        salary *= 1.1;
    }
    void fivePercentRaise() {
        salary *= 1.05;
    }
```

Их можно заменить одним методом:

```
void raise(double factor) {
    salary *= (1 + factor);
}
```

Конечно, этот случай настолько прост, что его заметит каждый. Но вот менее очевидный случай:

```
protected Dollars baseCharge() {
    double result = Math.min(lastUsage(), 100) * 0.03;
    if (lastUsage() > 100) {
        result += (Math.min(lastUsage(), 200) - 100) * 0.05;
    };
    if (lastUsage() > 200) {
        result += (lastUsage() - 200) * 0.07;
    };
    return new Dollars(result);
}
```

Этот код можно заменить следующим:

```
protected Dollars baseCharge() {
    double result = usageInRange(0, 100) * 0.03;
    result += usageInRange(100, 200) * 0.05;
    result += usageInRange(200, Integer.MAX_VALUE) * 0.07;
    return new Dollars(result);
}
protected int usageInRange(int start, int end) {
    if (lastUsage() > start) return Math.min(lastUsage(), end) - start;
    else return 0;
}
```

Сложность в том, чтобы обнаружить повторяющийся код с несколькими значениями, которые можно передать в качестве параметров.

Замена параметра явными методами (Replace Parameter with Explicit Methods)

Имеется метод, выполняющий разный код в зависимости от значения параметра перечислимого типа.

Создайте отдельный метод для каждого значения параметра.

```
void setValue(String name, int value) {
    if (name.equals("height")) {
        _height = value;
        return;
    }
    if (name.equals("width")) {
        _width = value;
        return;
    }
    Assert.shouldNeverReachHere();
}
```

```
void setHeight(int arg) {
    _height = arg;
}
void setWidth(int arg) {
    _width = arg;
}
```

Мотивация

Данный рефакторинг является обратным по отношению к рефакторингу “Параметризация метода” (с. 300). Типичная ситуация для применения данного рефакторинга возникает в случае, когда есть параметр с дискретными значениями, которые проверяются в условной инструкции, и в зависимости от результатов проверки выполняется разный код. Вызывающий код должен решить, что требуется делать, и передать в качестве параметра соответствующее значение; поэтому можно изначально создать различные методы и избавиться от условной инструкции. Так удается избежать условного поведения и получить возможность проверки времени компиляции. Кроме того, интерфейс в этом случае становится более прозрачным. Если используется параметр, то программисту, применяющему такой метод, приходится не только рассматривать имеющиеся в классе методы, но и определять правильное значение параметра (эти значения нередко очень плохо документированы).

Прозрачность интерфейса уже может быть достаточным основанием для проведения рефакторинга, даже если проверка времени компиляции не приносит особой пользы. Код `Switch.beOn()` значительно понятнее, чем `Switch.setState(true)`, даже если все выполняемое действие заключается всего лишь в установке внутреннего логического поля.

Если значения параметра могут изменяться значительно, применять данный рефакторинг не стоит. Если это происходит и переданный параметр при этом просто присваивается полю, воспользуйтесь простым методом установки значения поля. Если требуется некоторое условное поведение, лучше применить рефакторинг "Замена условной инструкции полиморфизмом" (с. 271).

Техника

- Создайте явный метод для каждого значения параметра.
- Для каждой ветви условной инструкции вызовите соответствующий новый метод.
- Выполняйте компиляцию и тестирование после изменения каждой ветви.
- Замените каждый вызов условного метода вызовом соответствующего нового метода.
- Выполните компиляцию и тестирование.
- После изменения всех вызовов удалите условный метод.

Пример

Я хочу создать подкласс класса работника на основе значения переданного параметра:

```
static final int ENGINEER = 0;
static final int SALESMAN = 1;
static final int MANAGER = 2;
static Employee create(int type) {
    switch (type) {
        case ENGINEER:
            return new Engineer();
        case SALESMAN:
            return new Salesman();
        case MANAGER:
            return new Manager();
        default:
            throw new IllegalArgumentException(
                    "Неверное значение кода типа");
    }
}
```

Поскольку это фабричный метод, я не могу воспользоваться рефакторингом “Замена условной инструкции полиморфизмом” (с. 271), так как я еще не создал объект. Я предполагаю, что новых классов будет немного, поэтому имеет смысл наличие явных интерфейсов. Сначала я создаю новые методы:

```
static Employee createEngineer() {
    return new Engineer();
}
static Employee createSalesman() {
    return new Salesman();
}
static Employee createManager() {
    return new Manager();
}
```

Затем я по одному заменяю ветви инструкции оператора switch вызовами явных методов:

```
static Employee create(int type) {
    switch (type) {
        case ENGINEER:
            return Employee.createEngineer();
        case SALESMAN:
            return new Salesman();

        case MANAGER:
            return new Manager();

        default:
            throw new IllegalArgumentException(
                    "Неверное значение кода типа");
    }
}
```

Я выполняю компиляцию и тестирование после изменения каждой ветви конструкции switch, пока не заменю их все:

```
static Employee create(int type) {
    switch (type) {
        case ENGINEER:
            return Employee.createEngineer();
        case SALESMAN:
            return Employee.createSalesman();
        case MANAGER:
            return Employee.createManager();
        default:
            throw new IllegalArgumentException(
                    "Неверное значение кода типа");
    }
}
```

Теперь я перехожу к вызовам старого метода `create` и заменяю код вида

```
Employee kent = Employee.create(ENGINEER)
```

следующим:

```
Employee kent = Employee.createEngineer()
```

После того как я выполню такую замену для всех вызовов метода `create`, я могу удалить метод `create`. Возможно, мне удастся избавиться и от констант.

Сохранение всего объекта (Preserve Whole Object)

Вы получаете от объекта несколько значений, которые затем передаете при вызове метода в качестве параметров.

Передавайте вместо этого весь объект.

```
int low     = daysTempRange().getLow();
int high    = daysTempRange().getHigh();
withinPlan = plan.withinRange(low, high);
```

```
withinPlan = plan.withinRange(daysTempRange());
```

Мотивация

Подобная ситуация возникает, когда объект передает несколько значений данных из одного объекта в качестве параметров вызова метода. Проблема в том, что, если вызываемому объекту в дальнейшем потребуются новые данные, придется искать и изменять все вызовы этого метода. Но этого можно избежать, если передавать методу весь объект, от которого поступают данные, целиком. Тогда вызываемый объект может запрашивать любые необходимые ему данные от переданного объекта.

Помимо повышения устойчивости списка параметров к изменениям, данный рефакторинг часто улучшает удобочитаемость кода. Работать с длинными списками параметров трудно, потому что как вызывающий, так и вызываемый коды должны помнить, какие именно значения должны быть в этом списке. Длинные списки параметров способствуют также дублированию кода, потому что при этом вызываемый объект не в состоянии воспользоваться никакими другими методами переданного объекта для вычисления промежуточных значений.

У этой медали есть и оборотная сторона. При передаче значений вызванный объект зависит только от значений, но не от объекта, из которого эти значения получены. Передача объекта устанавливает зависимость между ним и вызываемым объектом. Если все это приводит к усилению зависимостей и усложнению структуры зависимостей, данный рефакторинг лучше не использовать.

Я слышал еще об одной причине, по которой не стоит применять описываемый рефакторинг. Она заключается в том, что если вызывающему объекту нужно только одно значение из запрашиваемого объекта, то лучше передавать это значение, а не весь объект. Я с такой точкой зрения не согласен. Передача одного значения и одного объекта равнозначны, по крайней мере в плане понятности (передача параметров по значению может потребовать дополнительных накладных расходов). Главной проблемой в этом случае является проблема зависимости.

То, что вызываемому методу нужно много значений другого объекта, говорит о том, что в действительности этот метод должен быть определен в объекте, который поставляет эти значения. Поэтому при применении данного рефакторинга рассмотрите в качестве возможной альтернативы рефакторинг “Перенос метода” (с. 162).

Может оказаться и так, что необходимый целый объект пока не определен. Тогда вам нужно прибегнуть к рефакторингу “Введение объекта параметра” (с. 312).

Зачастую вызывающий объект передает в качестве параметров несколько значений *своих* данных. В этом случае можно вместо значений передать в вызове сам вызывающий объект, если у вас имеются необходимые методы получения значений и нет возражений против возникающей зависимости.

Техника

- Создайте новый параметр для передачи всего объекта, от которого поступают данные.
- Выполните компиляцию и тестирование.
- Определите, какие параметры должны быть получены от объекта в целом.
- Возьмите один параметр и замените ссылки на него в теле метода вызовами соответствующего метода объекта-параметра.
- Удалите ненужный параметр.
- Выполните компиляцию и тестирование.
- Повторите эти действия для каждого параметра, который можно получить от передаваемого объекта.
- Удалите из вызывающего метода код, который получает удаленные параметры.
  ⇛ *Конечно, если этот код не использует эти параметры в каком-нибудь другом месте.*
- Выполните компиляцию и тестирование.

Пример

Рассмотрим объект, представляющий помещение и регистрирующий самую высокую и самую низкую температуры в течение суток. Он должен сравнивать этот диапазон с диапазоном в заранее установленном плане обогрева:

```
class Room...
    boolean withinPlan(HeatingPlan plan) {
        int low = daysTempRange().getLow();
        int high = daysTempRange().getHigh();
        return plan.withinRange(low, high);
    }
class HeatingPlan...
    boolean withinRange(int low, int high) {
        return (low >= _range.getLow() && high <= _range.getHigh());
    }
    private TempRange _range;
```

Вместо распаковки данных диапазона для их передачи я могу передать целиком объект Range. В этом простом случае такую передачу можно осуществить за один шаг. Если бы в вызове участвовало больше параметров, могло бы потребоваться действовать небольшими шагами. Сначала я добавляю в список параметров объект, от которого буду получать необходимые данные:

```
class HeatingPlan...
    boolean withinRange(TempRange roomRange, int low, int high) {
        return (low >= _range.getLow() && high <= _range.getHigh());
    }
class Room...
    boolean withinPlan(HeatingPlan plan) {
        int low = daysTempRange().getLow();
        int high = daysTempRange().getHigh();
        return plan.withinRange(daysTempRange(), low, high);
    }
```

Затем я использую метод всего объекта вместо одного из параметров:

```
class HeatingPlan...
    boolean withinRange(TempRange roomRange, int high) {
        return (roomRange.getLow() >= _range.getLow() &&
                high <= _range.getHigh());
    }
class Room...
    boolean withinPlan(HeatingPlan plan) {
        int low = daysTempRange().getLow();
        int high = daysTempRange().getHigh();
        return plan.withinRange(daysTempRange(), high);
    }
```

Я продолжаю работать таким образом и дальше, пока не заменю все, что требуется:

```
class HeatingPlan...
    boolean withinRange(TempRange roomRange) {
        return (roomRange.getLow()  >= _range.getLow() &&
               roomRange.getHigh() <= _range.getHigh());
    }
class Room...
    boolean withinPlan(HeatingPlan plan) {
        int low = daysTempRange().getLow();
        int high = daysTempRange().getHigh();
        return plan.withinRange(daysTempRange());
    }
```

Временные переменные мне больше не нужны:

```
class Room...
    boolean withinPlan(HeatingPlan plan) {
        int low = daysTempRange().getLow();
        int high = daysTempRange().getHigh();
        return plan.withinRange(daysTempRange());
    }
```

Такое применение целых объектов очень быстро приводит к пониманию, что можно переместить поведение в объект, чтобы с ним было легче работать.

```
class HeatingPlan...
    boolean withinRange(TempRange roomRange) {
        return (_range.includes(roomRange));
    }
class TempRange...
    boolean includes(TempRange arg) {
        return arg.getLow()  >= this.getLow() &&
               arg.getHigh() <= this.getHigh();
    }
```

Замена параметра вызовом метода (Replace Parameter with Method)

Объект вызывает метод, а затем передает полученный результат в качестве параметра метода. Получатель значения тоже может вызывать этот метод.

Уберите параметр и заставьте получателя вызывать этот метод.

```
int basePrice = _quantity * _itemPrice;
discountLevel = getDiscountLevel();
double finalPrice = discountedPrice(basePrice, discountLevel);
```

```
int basePrice = _quantity * _itemPrice;
double finalPrice = discountedPrice(basePrice);
```

Мотивация

Если метод может получать значение, передаваемое в качестве параметра, некоторым другим способом, он должен поступать именно так. Длинные списки параметров затрудняют понимание, и их следует по возможности сокращать.

Один из способов сократить список параметров — проверить, не может ли рассматриваемый метод получить необходимые ему параметры другим путем. Если объект вызывает собственный метод и вычисление параметра не обращается ни к каким параметрам вызывающего метода, то вы должны иметь возможность удалить параметр путем выполнения вычислений в этом собственном методе. Это справедливо и в случае, когда вы вызываете метод другого объекта, в котором есть ссылка на вызывающий объект.

Нельзя удалить параметр, если вычисление зависит от параметра в вызывающем методе, потому что этот параметр в разных вызовах может иметь различные значения (если, конечно, вы не заменяете его методом). Нельзя удалять параметр и тогда, когда у получателя нет ссылки на отправителя и вы не предоставляете ее.

Иногда параметр присутствует в расчете на будущую параметризацию метода. В такой ситуации я все равно от него избавился бы. Вы сможете заняться параметризацией тогда, когда это потребуется; может оказаться, что необходимого параметра вообще не будет. Исключение из этого правила я сделал бы только для случая, когда изменения в интерфейсе могут иметь существенные последствия для всей программы, например потребуют длительной компиляции или изменения большого объема существующего кода. Если вас тревожит этот вопрос, прикиньте, каких усилий потребует внесение такого изменения. Следует также выяснить, нельзя ли уменьшить зависимости, из-за которых такое изменение требует столько работы. Устойчивые интерфейсы, конечно, — благо, но не следует замораживать плохие интерфейсы.

Техника

- При необходимости выделите вычисление параметра в отдельный метод.
- Замените обращения к параметру в телах методов ссылками на метод.
- Выполняйте компиляцию и тестирование после каждой замены.
- Примените к параметру рефакторинг "Удаление параметра" (с. 294).

Пример

Еще один маловероятный вариант скидки при заказе выглядит следующим образом:

```
public double getPrice() {
    int basePrice = _quantity * _itemPrice;
    int discountLevel;
    if (_quantity > 100) discountLevel = 2;
    else discountLevel = 1;
    double finalPrice = discountedPrice(basePrice, discountLevel);
    return finalPrice;
}

private double discountedPrice(int basePrice, int discountLevel) {
    if (discountLevel == 2) return basePrice * 0.1;
    else return basePrice * 0.05;
}
```

Я могу начать с извлечения вычисления уровня скидки `discountLevel`:

```
public double getPrice() {
    int basePrice = _quantity * _itemPrice;
    int discountLevel = getDiscountLevel();
    double finalPrice = discountedPrice(basePrice, discountLevel);
    return finalPrice;
}

private int getDiscountLevel() {
    if (_quantity > 100) return 2;
    else return 1;
}
```

Затем я заменяю обращения к параметру в `discountedPrice`:

```
private double discountedPrice(int basePrice, int discountLevel) {
    if (getDiscountLevel() == 2) return basePrice * 0.1;
    else return basePrice * 0.05;
}
```

Теперь я могу применить рефакторинг “Удаление параметра” (с. 294):

```
public double getPrice() {
    int basePrice = _quantity * _itemPrice;
    int discountLevel = getDiscountLevel();
    double finalPrice = discountedPrice(basePrice);
    return finalPrice;
}

private double discountedPrice(int basePrice) {
    if (getDiscountLevel() == 2) return basePrice * 0.1;
    else return basePrice * 0.05;
}
```

Теперь я могу убрать временную переменную:

```
public double getPrice() {
    int basePrice = _quantity * _itemPrice;
    double finalPrice = discountedPrice(basePrice);
    return finalPrice;
}
```

Теперь пришло время избавиться от другого параметра и его временной переменной. В конечном итоге останется код

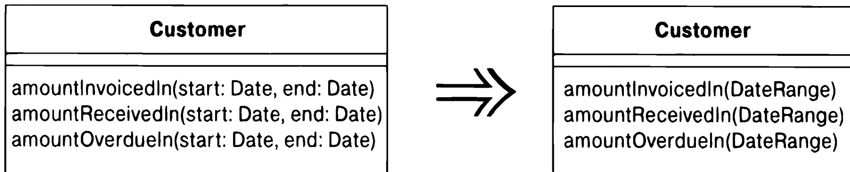
```
public double getPrice() {
    return discountedPrice();
}

private double discountedPrice() {
    if (getDiscountLevel() == 2) return getBasePrice() * 0.1;
    else return getBasePrice() * 0.05;
}

private double getBasePrice() {
    return _quantity * _itemPrice;
}
```

так что я могу применить к `discountedPrice` рефакторинг “Встраивание метода” (с. 139):

```
private double getPrice() {
    if (getDiscountLevel() == 2) return getBasePrice() * 0.1;
    else return getBasePrice() * 0.05;
}
```

Введение объекта параметра (Introduce Parameter Object)

Есть группа параметров, естественным образом связанных между собой.
Замените их объектом.



Мотивация

Часто встречается вызов метода, в котором совместно передается некоторая взаимосвязанная группа параметров. Эта группа может использоваться несколькими методами одного или нескольких классов. Такая группа классов представляет собой совокупность данных (data clump) и может быть заменена единым объектом, хранящим все эти данные. Имеет смысл сгруппировать такие данные и собрать все эти параметры в один объект. Такой рефакторинг полезен, поскольку он сокращает размер списка параметров и тем самым облегчает понимание вызова. Определяемые в новом объекте методы доступа делают код более согласованным и последовательным, благодаря чему его проще понимать и модифицировать.

Однако получаемая от этого рефакторинга выгода еще существеннее, поскольку после группировки параметров вы сталкиваетесь с поведением, которое можно переместить в новый класс. Зачастую в телах методов производятся одинаковые действия со значениями параметров. Перемещая это поведение в новый объект, можно избавиться от значительного количества дублируемого кода.

Техника

- Создайте новый класс для представления группы заменяемых параметров. Сделайте этот класс неизменяемым.
- Выполните компиляцию.
- Примените к этой новой группе данных рефакторинг “Добавление параметра” (с. 292). Во всех вызовах метода используйте в качестве значения данного параметра `null`.

  ⇛ *Если мест вызова много, можно сохранить старую сигнатуру и вызывать в ней новый метод. Начните с применения рефакторинга к старому методу. После этого можно будет поочередно изменять все вызовы и в конце концов полностью убрать старый метод.*

- Для каждого параметра в совокупности данных выполните его удаление из сигнатуры. Модифицируйте вызывающий код и тело метода так, чтобы они использовали вместо этого значения новый объект.
- Выполняйте компиляцию и тестирование после удаления каждого параметра.
- Когда вы уберете все параметры, поищите поведение, которое можно было бы переместить в новый объект с помощью рефакторинга "Перенос метода" (с. 162).

  ⇛ *Это поведение может быть целым методом или какой-то его частью. Если поведение представляет собой часть метода, сначала примените к нему рефакторинг "Извлечение метода" (с. 132), а затем переместите этот новый метод.*

Пример

Начну с бухгалтерского счета и проводок по нему. Проводки представляют собой простые объекты данных:

```
class Entry...
    Entry(double value, Date chargeDate) {
        _value = value;
        _chargeDate = chargeDate;
    }
    Date getDate() {
        return _chargeDate;
    }
    double getValue() {
        return _value;
    }
    private Date _chargeDate;
    private double _value;
```

Мое внимание сосредоточено на счете, который хранит коллекцию проводок и предоставляет метод, определяющий движение по счету в период между двумя датами:

```
class Account...
    double getFlowBetween(Date start, Date end) {
        double result = 0;
        Enumeration e = _entries.elements();

        while (e.hasMoreElements()) {
            Entry each = (Entry) e.nextElement();

            if (each.getDate().equals(start) ||
                each.getDate().equals(end) ||
               (each.getDate().after(start) &&
```

```
                each.getDate().before(end))) {
                result += each.getValue();
            }
        }

        return result;
    }
    private Vector _entries = new Vector();
```

```
Код клиента...
    double flow = anAccount.getFlowBetween(startDate, endDate);
```

Я не знаю, в который раз сталкиваюсь с парами значений, представляющих диапазон (например, даты начала и конца или верхние и нижние числовые границы). Понятно, почему это происходит, — в конце концов, раньше я сам допускал подобные вещи. Но с тех пор как я увидел проектный шаблон диапазона [7], я всегда стараюсь использовать не пары, а диапазоны. Мой первый шаг состоит в объявлении простого объекта данных для диапазона:

```
class DateRange {
    DateRange(Date start, Date end) {
        _start = start;
        _end = end;
    }

    Date getStart() {
        return _start;
    }

    Date getEnd() {
        return _end;
    }

    private final Date _start;
    private final Date _end;
}
```

Я сделал класс диапазона дат неизменяемым; это означает, что все значения для диапазона дат определены как `final` и устанавливаются в конструкторе (поэтому в классе нет методов модификации значений). Этот мудрый шаг сделан для того, чтобы избежать ошибок псевдонимов. Поскольку в Java параметры передаются по значению, придание классу свойства неизменяемости имитирует способ работы параметров Java, так что это корректное допущение для данного рефакторинга.

Затем я добавляю диапазон дат в список параметров метода `getFlowBetween`:

```
class Account...
    double getFlowBetween(Date start, Date end, DateRange range) {
    double result = 0;
    Enumeration e = _entries.elements();
```

```
    while (e.hasMoreElements()) {
        Entry each = (Entry) e.nextElement();

        if (each.getDate().equals(start) ||
                each.getDate().equals(end) ||
                (each.getDate().after(start) &&
                 each.getDate().before(end))) {
            result += each.getValue();
        }
    }

    return result;
}

Код клиента...
    double flow = anAccount.getFlowBetween(startDate,
                                           endDate, null);
```

Сейчас мне достаточно только выполнить компиляцию, потому что я еще не менял никакое поведение.

На следующем шаге я удаляю один из параметров, место которого занимает новый объект. Для этого я удаляю параметр `start` и модифицирую метод и его вызов так, чтобы они использовали новый объект:

```
class Account...
    double getFlowBetween(Date end, DateRange range) {
        double result = 0;
        Enumeration e = _entries.elements();

        while (e.hasMoreElements()) {
            Entry each = (Entry) e.nextElement();

            if (each.getDate().equals(range.getStart()) ||
                    each.getDate().equals(end) ||
                    (each.getDate().after(range.getStart()) &&
                     each.getDate().before(end))) {
                result += each.getValue();
            }
        }

        return result;
    }

Код клиента...
double flow =
    anAccount.getFlowBetween(endDate,
                             new DateRange(startDate,null));
```

После этого я удаляю конечную дату:

```
class Account...
    double getFlowBetween(DateRange range) {
    double result = 0;
    Enumeration e = _entries.elements();

    while (e.hasMoreElements()) {
        Entry each = (Entry) e.nextElement();

        if (each.getDate().equals(range.getStart()) ||
                each.getDate().equals(range.getEnd()) ||
                (each.getDate().after(range.getStart()) &&
                 each.getDate().before(range.getEnd()))) {
            result += each.getValue();
        }
    }

    return result;
}

Код клиента...
double flow =
    anAccount.getFlowBetween(new DateRange(startDate,
                                            endDate));
```

Я ввел объект параметра; однако я могу получить от этого рефакторинга еще больше пользы, если перемещу в новый объект поведение из других методов. В данном случае я могу взять код из условия и применить к нему рефакторинги “Извлечение метода” (с. 132) и “Перенос метода” (с. 162) и получить следующее:

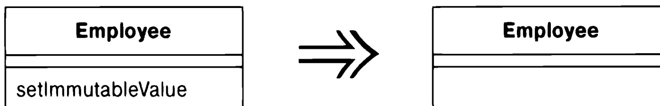
```
class Account...
    double getFlowBetween(DateRange range) {
        double result = 0;
        Enumeration e = _entries.elements();
        while (e.hasMoreElements()) {
            Entry each = (Entry) e.nextElement();
            if (range.includes(each.getDate())) {
                result += each.getValue();
            }
        }
        return result;
    }

class DateRange...
    boolean includes(Date arg) {
        return (arg.equals(_start) ||
                arg.equals(_end)   ||
                (arg.after(_start) && arg.before(_end)));
    }
```

Обычно такие простые извлечения и перемещения, как здесь, я выполняю за один шаг. Если возникнет ошибка, можно откатиться назад и модифицировать код за два маленьких шага.

Удаление метода установки значения (Remove Setting Method)

Значение поля должно быть установлено в момент создания объекта и в дальнейшем не изменяться.

Удалите методы, устанавливающие значение этого поля.



Мотивация

Наличие метода установки значения говорит о том, что значение поля может изменяться. Если вы не хотите, чтобы значение поля могло изменяться после создания объекта, не предоставляйте метод установки (а поле объявите с модификатором `final`). Благодаря этому ваш замысел становится ясным и устраняется всякая возможность изменения поля.

Такая ситуация часто встречается, когда программисты слепо пользуются косвенным обращением к переменным [6]. Такие программисты применяют методы установки даже в конструкторе. Думаю, что это можно объяснить стремлением к последовательности и согласованности, но путаница, которую метод установки вызовет впоследствии, перечеркивает все благие намерения.

Техника

- Если поле не объявлено как `final`, сделайте это.
- Выполните компиляцию и тестирование.
- Убедитесь, что метод установки вызывается только в конструкторе или методе, вызываемом конструктором.
- Модифицируйте конструктор или вызываемый им метод так, чтобы он обращался к полю непосредственно.

  ⇛ *Это не получится, если есть подкласс, устанавливающий закрытые поля суперкласса. В таком случае надо попытаться предоставить*

защищенный метод суперкласса (в идеале — конструктор), устанавливающий эти значения. Как бы вы ни поступили, не давайте методу суперкласса имя, которое можно спутать с именем метода установки значения.

- Выполните компиляцию и тестирование.
- Удалите метод установки.
- Выполните компиляцию.

Пример

Вот простой пример:

```
class Account {
    private String _id;
    Account(String id) {
        setId(id);
    }
    void setId(String arg) {
        _id = arg;
    }
```

Этот код можно заменить следующим:

```
class Account {
    private final String _id;
    Account(String id) {
        _id = id;
    }
```

У нас могут возникнуть проблемы нескольких видов. Первая из них — выполнение вычислений над аргументом:

```
class Account {
    private String _id;
    Account(String id) {
        setId(id);
    }
    void setId(String arg) {
        _id = "ZZ" + arg;
    }
```

Если изменение такое простое, как показано здесь, и конструктор только один, можно внести изменения в конструктор. Если изменение сложное или необходимо выполнять вызовы из различных методов, я должен предоставить метод для этого. В таком случае надо присвоить этому методу имя, делающее ясным его предназначение:

```
class Account {
    private final String _id;
    Account(String id) {
        initializeId(id);
    }
    void initializeId(String arg) {
        _id = "ZZ" + arg;
    }
```

Еще одна действительно неприятная ситуация возникает, когда есть подклассы, инициализирующие закрытые переменные суперкласса:

```
class InterestAccount extends Account...
    private double _interestRate;
    InterestAccount(String id, double rate) {
        setId(id);
        _interestRate = rate;
    }
```

Дело в том, что нельзя обратиться к `id`, чтобы присвоить ему значение, непосредственно. В этой ситуации наилучшим решением будет использование конструктора суперкласса:

```
class InterestAccount...
    InterestAccount(String id, double rate) {
        super(id);
        _interestRate = rate;
    }
```

Если это невозможно, то лучше воспользоваться методом с хорошим названием:

```
class InterestAccount...
    InterestAccount(String id, double rate) {
        initializeId(id);
        _interestRate = rate;
    }
```

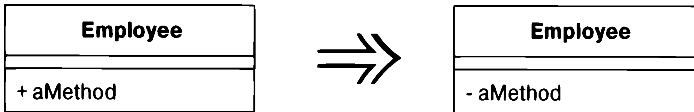
Еще один требующий рассмотрения вариант — присвоение значения коллекции:

```
class Person {
    Vector getCourses() {
        return _courses;
    }
    void setCourses(Vector arg) {
        _courses = arg;
    }
    private Vector _courses;
```

Здесь я хочу заменить метод установки с помощью операций добавления и удаления. Об этом я говорил при описании рефакторинга “Инкапсуляция коллекции” (с. 226).

Сокрытие метода (Hide Method)

Метод не используется никаким другим классом.
Сделайте этот метод закрытым.



Мотивация

Рефакторинг часто заставляет менять решения о видимости тех или иных методов. Обнаружить случаи, когда следует расширить видимость метода, легко: если метод нужен другому классу, ограничения видимости ослабляются. Несколько сложнее определить, не является ли метод излишне видимым. В идеале некоторый инструмент должен проверять все методы и определять, нельзя ли их скрыть. Если у вас нет такого средства, вы сами должны регулярно проводить данную проверку.

Очень часто потребность в сокрытии методов получения и установки значений возникает в связи с разработкой более богатого интерфейса, предоставляющего дополнительное поведение, в особенности если вы начинали с класса, практически ничего не добавлявшего к простой инкапсуляции данных. По мере добавления нового поведения в класс может выясниться, что в открытых методах получения и установки значений больше нет надобности, так что их можно скрыть. Если сделать метод получения или установки значений закрытым и использовать прямой доступ к переменным, такой метод можно удалить.

Техника

- Регулярно проверяйте возможность сделать метод более закрытым.
  - ⇛ *Используйте инструментарий в духе lint, делайте проверки вручную регулярно, а также после удаления обращения к такому методу из другого класса.*
  - ⇛ *В особенности ищите такие ситуации для методов установки значений полей.*

- Делайте каждый метод как можно более закрытым.
- Выполняйте компиляцию после проведения сокрытий групп методов.

  ⇛ *Компилятор проверяет возможность обращений естественным образом, так что необходимости в компиляции после внесения каждого изменения нет. Если что-то идет не так, ошибка легко обнаруживается.*

Замена конструктора фабричным методом (Replace Constructor with Factory Method)

При создании объекта вы хотите делать нечто большее, чем простое конструирование.

Замените конструктор фабричным методом.

```
Employee(int type) {
    _type = type;
}
```

```
static Employee create(int type) {
    return new Employee(type);
}
```

Мотивация

Самая очевидная мотивация данного рефакторинга связана с заменой кода типа подклассами. Имеется объект, который обычно создается с кодом типа, но теперь он требует подклассов. Конкретный подкласс определяется кодом типа. Однако конструкторы могут возвращать только экземпляр объекта запрашиваемого типа. Поэтому надо заменить конструктор фабричным методом [9].

Фабричные методы можно использовать и в других ситуациях, когда оказывается недостаточно возможностей конструкторов. Они важны при рефакторинге “Замена значения ссылкой” (с. 198). Они могут также применяться для задания различных режимов создания, выходящих за рамки количества и типов параметров.

Техника

- Создайте фабричный метод. В его теле должен вызываться текущий конструктор.

- Замените все вызовы конструктора вызовами фабричного метода.
- Выполняйте компиляцию и тестирование после каждой замены.
- Объявите конструктор закрытым.
- Выполните компиляцию.

Пример

Быстрый, но скучный и избитый пример дает система зарплаты служащих. Пусть класс служащего имеет следующий вид:

```
class Employee {
    private int _type;
    static final int ENGINEER = 0;
    static final int SALESMAN = 1;
    static final int MANAGER = 2;
    Employee(int type) {
        _type = type;
    }
```

Я хочу создать подклассы `Employee`, соответствующие кодам типов, поэтому мне нужен фабричный метод:

```
static Employee create(int type) {
    return new Employee(type);
}
```

После этого я изменяю все вызовы данного конструктора так, чтобы вызывался новый фабричный метод, и делаю конструктор закрытым:

```
client code...
    Employee eng = Employee.create(Employee.ENGINEER);

class Employee...
    private Employee(int type) {
        _type = type;
    }
```

Пример: создание подклассов с помощью строки

Пока что выигрыш невелик: все преимущества состоят в том, что я отделил получателя вызова создания объекта от класса создаваемого объекта. Если позднее я применю рефакторинг “Замена кода типа подклассами” (с. 241), чтобы преобразовать коды в подклассы `Employee`, я смогу скрыть эти подклассы от клиентов, используя фабричный метод:

```
static Employee create(int type) {
    switch (type) {
        case ENGINEER:
            return new Engineer();
        case SALESMAN:
            return new Salesman();
        case MANAGER:
            return new Manager();
        default:
            throw new IllegalArgumentException(
                    "Неверное значение кода типа");
    }
}
```

Беда в том, что при этом остается инструкция switch. Если нам потребуется добавить новый подкласс, придется обязательно вспомнить об обновлении этой инструкции... а я склонен к забывчивости.

Хороший способ обойти проблемы — воспользоваться Class.forName. Первое, что надо сделать, — это изменить тип параметра (что, в сущности, является разновидностью рефакторинга “Переименование метода” (с. 290)). Сначала я создаю новый метод, принимающий в качестве аргумента строку:

```
static Employee create(String name) {
    try {
        return (Employee) Class.forName(name).newInstance();
    } catch (Exception e) {
        throw new IllegalArgumentException(
                "Невозможно инстанцировать" + name);
    }
}
```

После этого я преобразую принимающий целое значение метод create так, чтобы он использовал этот новый метод:

```
class Employee {
    static Employee create(int type) {
        switch (type) {
            case ENGINEER:
                return create("Engineer");
            case SALESMAN:
                return create("Salesman");
            case MANAGER:
                return create("Manager");
            default:
                throw new IllegalArgumentException(
                        "Неверное значение кода типа");
        }
    }
```

После этого можно работать с кодом, вызывающим `create`, заменяя выражения вида

```
Employee.create(ENGINEER)
```

выражением

```
Employee.create("Engineer")
```

По завершении работы можно полностью удалить версию метода с целочисленным параметром.

Этот подход хорош тем, что устраняет необходимость в обновлении метода `create` при добавлении новых подклассов `Employee`. Однако в нем не хватает проверки на этапе компиляции: орфографическая ошибка легко приводит к ошибке времени выполнения. Если это важно, я использую явный метод (см. ниже), но тогда я должен добавлять новый метод каждый раз при введении нового подкласса. Это компромисс между гибкостью и безопасностью типов. К счастью, при принятии неверного решения его можно изменить, если воспользоваться рефакторингом “Параметризация метода” (с. 300) или “Замена параметра явными методами” (с. 302).

Еще одна причина не прибегать к `class.forName` связана с тем, что этот вариант открывает клиентам имена подклассов. Это не так уж страшно, поскольку можно использовать другие строки и осуществлять другое поведение с помощью фабричного метода. Это основание не использовать рефакторинг “Встраивание метода” (с. 139) для удаления фабрики.

Пример: создание подклассов явными методами

Можно применить другой подход и скрыть подклассы с помощью явных методов. Это полезно, когда есть лишь несколько подклассов, которые не изменяются. Так, у меня может быть абстрактный класс `Person` (лицо) с подклассами `Male` (мужчина) и `Female` (женщина). Начну с определения в суперклассе фабричного метода для каждого подкласса:

```
class Person...
    static Person createMale() {
        return new Male();
    }
    static Person createFemale() {
        return new Female();
    }
```

Теперь вызовы вида

```
Person kent = new Male();
```

можно заменить такими:

```
Person kent = Person.createMale();
```

В результате суперкласс обладает знаниями о подклассах. Если это нежелательно, нужна более сложная схема, например схема "product trader" [2]. Впрочем, обычно такие сложности не нужны, и описанный подход отлично работает.

Инкапсуляция нисходящего приведения типа (Encapsulate Downcast)

Метод возвращает объект, к которому вызывающий код должен применить нисходящее приведение типа.

Переместите нисходящее приведение внутрь метода.

```
Object lastReading() {
    return readings.lastElement();
}
```

```
Reading lastReading() {
    return (Reading) readings.lastElement();
}
```

Мотивация

Нисходящее приведение типа — одна из самых неприятных вещей, которыми приходится заниматься в строго типизированных объектно-ориентированных языках. Оно неприятно, потому что кажется ненужным: вы сообщаете компилятору то, что он должен бы сообразить и сам. Но поскольку сообразить ему бывает довольно сложно, приходится делать это самостоятельно. Это особенно распространено в Java, где отсутствие шаблонов означает, что приведение типа требуется выполнять всякий раз, когда объект выбирается из коллекции.

Нисходящее приведение типа может быть неизбежным злом, но применять его следует как можно реже. Если вы возвращаете значение из метода и знаете, что тип этого значения более специализирован, чем гласит сигнатура метода, то вы возлагаете лишнюю работу на своих клиентов. Вместо того чтобы заставлять их выполнять приведение типа, следует передавать им насколько это возможно узко специализированный тип.

Часто такая ситуация возникает для методов, возвращающих итератор или коллекцию. Лучше разберитесь, для чего программисты используют этот итератор, и создайте соответствующий метод.

Техника

- Поищите случаи, когда приходится выполнять приведение типа результата, возвращаемого методом.
  ⇛ *Такие ситуации часто возникают с методами, возвращающими коллекции или итераторы.*
- Переместите приведение типа в метод.
  ⇛ *Для методов, возвращающих коллекции, примените рефакторинг "Инкапсуляция коллекции" (с. 226).*

Пример

У меня есть метод с именем `lastReading`, возвращающий последний элемент (показание прибора) из вектора:

```
Object lastReading() {
    return readings.lastElement();
}
```

Я должен заменить его следующим:

```
Reading lastReading() {
    return (Reading) readings.lastElement();
}
```

Это стоит делать там, где имеются классы коллекций. Пусть, например, коллекция из `Reading` находится в классе `Site`, и я встречаю код наподобие следующего:

```
Reading lastReading = (Reading) theSite.readings().lastElement()
```

Я могу избежать приведения типа и скрыть используемую коллекцию с помощью следующего кода:

```
Reading lastReading = theSite.lastReading();

class Site...
    Reading lastReading() {
        return (Reading) readings().lastElement();
    }
```

Изменение метода, в результате которого возвращается подкласс, меняет сигнатуру, но сохраняет работоспособность имеющегося кода, потому что компилятор умеет использовать подкласс вместо суперкласса. Конечно, при этом следует гарантировать, что подкласс не делает ничего, нарушающего контракт суперкласса.

Замена кода ошибки исключением (Replace Error Code with Exception)

Метод возвращает код, указывающий происшедшую ошибку.

Генерируйте вместо этого исключение.

```
int withdraw(int amount) {
    if (amount > _balance)
        return -1;
    else {
        _balance -= amount;
        return 0;
    }
}
```

```
void withdraw(int amount) throws BalanceException {
    if (amount > _balance) throw new BalanceException();
    _balance -= amount;
}
```

Мотивация

В компьютерах, как и в жизни, иногда случаются неприятности, на которые надо как-то реагировать. Проще всего остановить программу и вернуть код ошибки. Но это — компьютерный эквивалент самоубийства из-за опоздания на самолет. Хотя я и пытаюсь шутить, такое решение имеет свои преимущества. Если потери от аварийного завершения программы невелики, а пользователь терпелив, можно просто прекратить работу программы. Однако в важных программах должны приниматься более существенные меры.

Проблема в том, что та часть программы, которая обнаружила ошибку, не всегда является одновременно той частью, которая может с этой ошибкой справиться. Когда некоторая процедура обнаруживает ошибку, она должна сообщить о ней вызвавшему ее коду, а тот, в свою очередь, может переслать ошибку вверх по цепочке. Во многих языках для отображения ошибки выделяется специальное устройство вывода. В системах Unix и языках программирования на основе C из

функций традиционно возвращается код, указывающий на успешное или неудачное выполнение программы.

В Java есть лучший способ — исключения. Их преимущество в том, что они ясно отделяют нормальную работу программы от обработки ошибок. Благодаря этому облегчается понимание программ, а создание понятных программ, как вы знаете, — это достоинство, которое уступает только благочестию.

Техника

- Определите, каким должно быть исключение: проверяемым или непроверяемым.
  - ⇛ *Если вызывающий код отвечает за проверку условия перед вызовом, сделайте исключение непроверяемым.*
  - ⇛ *Если исключение проверяемое, либо создайте новое исключение, либо используйте существующее.*
- Найдите все места вызова и модифицируйте их для использования исключений.
  - ⇛ *Если исключение непроверяемое, измените места вызова так, чтобы перед обращением к методу проводилась соответствующая проверка. Выполняйте компиляцию и тестирование после каждой модификации.*
  - ⇛ *Если исключение проверяемое, модифицируйте места вызова таким образом, чтобы вызов метода происходил в блоке* `try`.
- Измените сигнатуру метода, чтобы она отражала его новое использование.

Если точек вызова много, может потребоваться слишком много изменений. Можно обеспечить постепенный переход, выполняя следующие шаги.

- Определите, каким будет исключение: проверяемым или не проверяемым.
- Создайте новый метод, использующий данное исключение.
- Модифицируйте прежний метод так, чтобы он вызывал новый.
- Выполните компиляцию и тестирование.
- Модифицируйте все места вызова старого метода так, чтобы в них вызывался новый метод. Выполняйте компиляцию и тестирование после каждой модификации.
- Удалите прежний метод.

Пример

Не удивительно ли, что в учебниках по программированию часто утверждается, что со счета нельзя снять больше, чем остаток на нем, в то время как в реальной жизни это зачастую возможно?

```
class Account...
    int withdraw(int amount) {
        if (amount > _balance)
            return -1;
        else {
            _balance -= amount;
            return 0;
        }
    }
    private int _balance;
```

Если я хочу изменить этот код так, чтобы использовать исключение, я сначала должен решить, будет ли оно проверяемым или непроверяемым. Решение зависит от того, будет ли проверка остатка на счете перед снятием с него некоторой суммы входить в обязанности вызывающего кода или процедуры, выполняющей снятие. Если проверка остатка на счете вменяется в обязанность вызывающего кода, то вызов процедуры снятия суммы, превышающей остаток, будет ошибкой, допущенной при программировании. Поскольку это программная ошибка, я должен использовать непроверяемое исключение. Если же проверка остатка на счету возлагается на процедуру снятия, я должен объявить исключение в интерфейсе. Таким образом я сообщаю вызывающему коду, что он должен ожидать возможного исключения и быть готовым принять надлежащие меры.

Пример: непроверяемое исключение

Начнем со случая непроверяемого исключения. Я ожидаю, что необходимую проверку выполнит вызывающий код. Сначала я рассматриваю места вызова. В данном случае код возврата не должен использоваться нигде — это было бы ошибкой программиста. Увидев код наподобие

```
if (account.withdraw(amount) == -1)
    handleOverdrawn();
else doTheUsualThing();
```

я должен заменить его, например, таким:

```
if (!account.canWithdraw(amount))
    handleOverdrawn();
else {
```

```
    account.withdraw(amount);
    doTheUsualThing();
}
```

После каждого изменения я могу выполнить компиляцию и тестирование.

Теперь нужно удалить код ошибки и добавить генерацию исключения в случае ошибки. Поскольку поведение (по определению) является исключительным, я должен использовать защитную конструкцию для проверки условия:

```
void withdraw(int amount) {
    if (amount > _balance)
        throw new IllegalArgumentException("Сумма слишком велика");
    _balance -= amount;
}
```

Поскольку это программная ошибка, о ней следует сигнализировать еще более явно с помощью утверждения:

```
class Account...
    void withdraw(int amount) {
        Assert.isTrue("средств достаточно", amount <= _balance);
        _balance -= amount;
    }

class Assert...
    static void isTrue(String comment, boolean test) {
        if (! test) {
            throw new RuntimeException("Ошибка assert: " + comment);
        }
    }
```

Пример: проверяемое исключение

В случае проверяемого исключения я действую несколько иначе. Сначала я создаю подходящий новый класс исключения (или использую имеющийся):

```
class BalanceException extends Exception {}
```

Затем я модифицирую обращения к методу, чтобы они выглядели следующим образом:

```
try {
    account.withdraw(amount);
    doTheUsualThing();
} catch (BalanceException e) {
    handleOverdrawn();
}
```

Теперь я модифицирую метод, снимающий сумму со счета, используя в нем исключение:

```
void withdraw(int amount) throws BalanceException {
    if (amount > _balance) throw new BalanceException();
    _balance -= amount;
}
```

Неудобство такой процедуры в том, что я вынужден изменить все обращения к методу и сам метод за один шаг, иначе компилятор нас накажет. Если мест вызова много, то придется выполнять большую модификацию без промежуточных компиляций и тестирований.

В таких случаях можно применять временный промежуточный метод. Начну с того же, что и раньше:

```
if (account.withdraw(amount) == -1)
    handleOverdrawn();
else doTheUsualThing();

class Account ...
    int withdraw(int amount) {
        if (amount > _balance)
            return -1;
        else {
            _balance -= amount;
            return 0;
        }
    }
```

Первый шаг заключается в создании нового метода `withdraw`, в котором используется исключение.

```
void newWithdraw(int amount) throws BalanceException {
    if (amount > _balance) throw new BalanceException();
    _balance -= amount;
}
```

Затем я изменяю текущий метод `withdraw` таким образом, чтобы он использовал новый метод:

```
int withdraw(int amount) {
    try {
        newWithdraw(amount);
        return 0;
    } catch (BalanceException e) {
        return -1;
    }
}
```

Сделав это, я могу выполнить компиляцию и тестирование. После этого можно заменить каждый вызов прежнего метода вызовом нового:

```
try {
    account.newWithdraw(amount);
    doTheUsualThing();
} catch (BalanceException e) {
    handleOverdrawn();
}
```

При наличии обоих (и старого, и нового) методов можно выполнять компиляцию и тестирование после каждой модификации. Завершив изменения, я удаляю старый метод и применяю рефакторинг "Переименование метода" (с. 290), чтобы дать новому методу старое имя.

Замена исключения проверкой (Replace Exception with Test)

Генерация исключения при выполнении условия, которое может быть проверено вызывающим кодом.

Измените вызывающий код так, чтобы он сначала выполнял проверку.

```
double getValueForPeriod(int periodNumber) {
    try {
        return _values[periodNumber];
    } catch (ArrayIndexOutOfBoundsException e) {
        return 0;
    }
}
```

```
double getValueForPeriod(int periodNumber) {
    if (periodNumber >= _values.length) return 0;
    return _values[periodNumber];
}
```

Мотивация

Исключения представляют собой важную возможность языков программирования. Они позволяют избежать написания сложного кода с помощью рефакторинга "Замена кода ошибки исключением" (с. 327). Как и многими другими удовольствиями, исключениями иногда злоупотребляют, и они перестают быть

приятными. Исключения должны использоваться для локализации исключительного поведения, связанного с неожиданной ошибкой. Они не должны служить заменой для проверок выполнения некоторых условий. Если разумно ожидать от вызывающего кода проверки выполнения условия перед операцией, то вам следует предоставить эту проверку, а вызывающий код должен ею пользоваться.

Техника

- Поместите в начале кода проверку и скопируйте код из блока `catch` в соответствующую ветвь оператора `if`.
- Добавьте в блок `catch` утверждение, которое будет уведомлять вас о выполнении этого блока.
- Выполните компиляцию и тестирование.
- Удалите блок `catch`, а также, если других блоков `catch` нет, блок `try` полностью.
- Выполните компиляцию и тестирование.

Пример

Для этого примера я беру объект, управляющий ресурсами, создание которых обходится дорого, но они могут использоваться повторно. Хорошим примером могут служить соединения с базами данных. У диспетчера соединений есть два пула, в одном из которых находятся ресурсы, доступные для использования, а в другом — уже выделенные. Когда клиенту нужен ресурс, диспетчер предоставляет его из пула доступных и переводит в пул выделенных. Когда клиент освобождает ресурс, диспетчер возвращает его обратно. Если клиент запрашивает ресурс, когда свободных ресурсов нет, диспетчер создает новый ресурс.

Метод выдачи ресурсов может выглядеть таким образом:

```
class ResourcePool
    Resource getResource() {
    Resource result;
    try {
        result = (Resource) _available.pop();
        _allocated.push(result);
        return result;
    } catch (EmptyStackException e) {
        result = new Resource();
        _allocated.push(result);
        return result;
    }
}
Stack _available;
Stack _allocated;
```

В данном случае нехватка ресурсов не является неожиданной ситуацией, поэтому использовать исключение не следует.

Чтобы удалить исключение, я сначала добавляю необходимую предварительную проверку, в которой отсутствует поведение:

```
Resource getResource() {
    Resource result;

    if (_available.isEmpty()) {
        result = new Resource();
        _allocated.push(result);
        return result;
    } else {
        try {
            result = (Resource) _available.pop();
            _allocated.push(result);
            return result;
        } catch (EmptyStackException e) {
            result = new Resource();
            _allocated.push(result);
            return result;
        }
    }
}
```

В этом случае исключение никогда не должно возникать. Чтобы проверить это, можно добавить утверждение:

```
Resource getResource() {
    Resource result;

    if (_available.isEmpty()) {
        result = new Resource();
        _allocated.push(result);
        return result;
    } else {
        try {
            result = (Resource) _available.pop();
            _allocated.push(result);
            return result;
        } catch (EmptyStackException e) {
            Assert.shouldNeverReachHere("Ресурс исчерпан");
            result = new Resource();
            _allocated.push(result);
            return result;
        }
    }
}
```

```
class Assert...
    static void shouldNeverReachHere(String message) {
        throw new RuntimeException(message);
    }
```

Теперь я могу выполнить компиляцию и тестирование. Если все пройдет хорошо, я полностью удалю блок `try`:

```
Resource getResource() {
    Resource result;
    if (_available.isEmpty()) {
        result = new Resource();
        _allocated.push(result);
        return result;
    } else {
        result = (Resource) _available.pop();
        _allocated.push(result);
        return result;
    }
}
```

Обычно после этого оказывается возможным привести в порядок условный код. В данном случае я могу применить рефакторинг “Консолидация дублирующихся условных фрагментов” (с. 260).

```
Resource getResource() {
    Resource result;
    if (_available.isEmpty())
        result = new Resource();
    else
        result = (Resource) _available.pop();
    _allocated.push(result);
    return result;
}
```

Глава 11

Работа с обобщенностью

С обобщением связана собственная группа рефакторингов, в основном занимающаяся перемещением методов по иерархии наследования. Рефакторинги "Подъем поля" (с. 338) и "Подъем метода" (с. 339) перемещают функцию вверх по иерархии, а "Опускание метода" (с. 345) и "Опускание поля" (с. 346) перемещают функцию вниз по иерархии. Поднимать конструкторы — задача несколько более трудная; соответствующие проблемы решаются с помощью рефакторинга "Подъем тела конструктора" (с. 342). Вместо опускания конструктора часто более удобным оказывается применение рефакторинга "Замена конструктора фабричным методом" (с. 321).

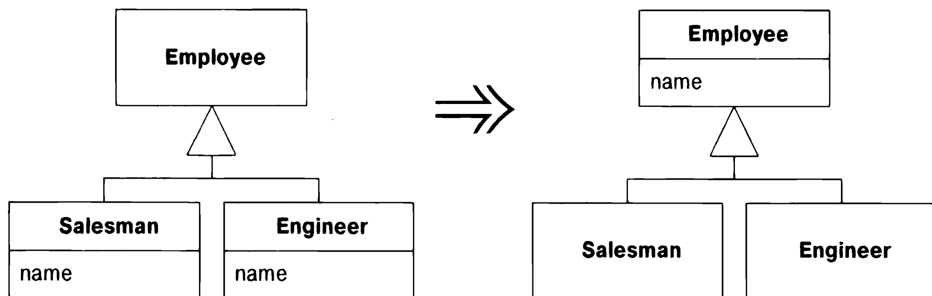
При наличии методов со сходным построением тела, но различающихся деталями, можно воспользоваться рефакторингом "Формирование шаблонного метода" (с. 361) для отделения отличий от сходства.

В дополнение к перемещениям функций по иерархии можно изменять саму иерархию, создавая новые классы. Рефакторинги "Извлечение подкласса" (с. 347), "Извлечение суперкласса" (с. 352) и "Извлечение интерфейса" (с. 357) осуществляют такое изменение путем формирования новых элементов из различных мест кода. Рефакторинг "Извлечение интерфейса" (с. 357) особенно важен тогда, когда вы хотите отметить небольшую часть функции для системы типов. Если в иерархии обнаруживаются ненужные классы, их можно удалить с помощью рефакторинга "Свертывание иерархии" (с. 360).

Иногда выясняется, что наследование — не лучший способ моделирования ситуации, и вместо него лучше применять делегирование. Рефакторинг "Замена наследования делегированием" (с. 369) позволяет выполнить это изменение. Но иногда ситуация оказывается обратной, и тогда приходится применять рефакторинг "Замена делегирования наследованием" (с. 371).

Подъем поля (Pull Up Field)

В двух подклассах есть одинаковое поле.
Переместите поле в суперкласс.



Мотивация

Если подклассы разрабатываются независимо или объединяются при выполнении рефакторинга, в них часто оказываются возможности, дублирующие одна другую. В частности, могут дублироваться некоторые поля. Иногда у таких полей даже оказываются одинаковые имена, но это не обязательно так. Единственный способ разобраться, что происходит, — внимательно рассмотреть поля и понять, как они используются другими методами. Если они используются более-менее аналогично, их можно обобщить.

В результате дублирование окажется сниженным сразу в двух отношениях. Будут удалены дублируемые объявления данных, и появится возможность переместить поведение, использующее поля, из подклассов в суперкласс.

Техника

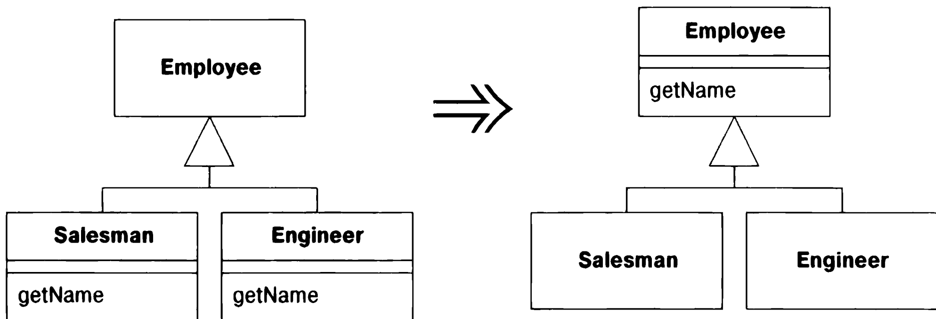
- Рассмотрите все применения полей — кандидатов на перемещение и убедитесь, что эти применения одинаковы.
- Если у полей разные имена, переименуйте их, дав имя, которое вы считаете подходящим для поля в суперклассе.
- Выполните компиляцию и тестирование.
- Создайте новое поле в суперклассе.

  ⇛ *Если поля закрытые, поле суперкласса следует делать защищенным, чтобы подклассы могли к нему обращаться.*
- Удалите поля из подклассов.

- Выполните компиляцию и тестирование.
- Рассмотрите возможность применения к новому полю рефакторинга “Самоинкапсуляция поля” (с. 190).

Подъем метода (Pull Up Method)

В подклассах есть методы с идентичными результатами работы.
Переместите их в суперкласс.



Мотивация

Устранение дублирования поведения — важная работа. Хотя два продублированных метода могут прекрасно работать в том виде, в котором они есть, они представляют собой питательную среду для возникновения ошибок в будущем. При наличии дублирования всегда есть риск, что при внесении изменений в один метод второй будет забыт и пропущен. Находить дубликаты обычно достаточно трудно.

Простейший случай применения данного рефакторинга осуществляется, когда тела двух методов одинаковы настолько, что это указывает на выполненные копирование и вставку. Конечно, далеко не всегда ситуация столь тривиальна. Можно просто выполнить рефакторинг и посмотреть, успешно ли пройдут тесты, но тогда придется полностью положиться на них. Я обычно считаю, что стоит поискать различия в методах; часто они указывают на поведение, которое я забыл протестировать.

Зачастую рассматриваемый рефакторинг применяется после внесения других изменений. Можно обнаружить два метода в разных классах, которые можно параметризовать так, что это, по сути, приведет к одному и тому же методу. В этом случае наименьшим шагом является отдельная параметризация каждого метода

в отдельности, после чего выполняется их обобщение. Если вы чувствуете себя достаточно уверенно, можете выполнить всю работу в один присест.

Особый случай для применения рефакторинга "Подъем метода" возникает, когда имеется некоторый метод подкласса, который перекрывает метод суперкласса, выполняя те же самые действия.

Самое неприятное в данном рефакторинге — это то, что в теле методов могут быть обращения к сущностям, находящимся в подклассе, а не в суперклассе. Если такая сущность представляет собой метод, можно либо обобщить его, либо создать в суперклассе соответствующий абстрактный метод. Чтобы это сработало, может потребоваться изменить сигнатуру метода или создать делегирующий метод.

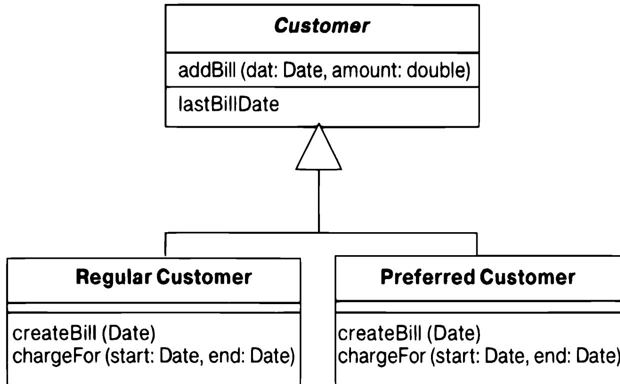
Если есть два похожих, но разных метода, можно попробовать прибегнуть к рефакторингу "Формирование шаблонного метода" (с. 361).

Техника

- Изучите методы и убедитесь, что они идентичны.
  - ⇛ *Если методы представляются решающими одну и ту же задачу, но не идентичными, примените к одному из них замену алгоритма, чтобы сделать методы полностью одинаковыми.*
- Если у методов разные сигнатуры, замените их той, которую хотите видеть в суперклассе.
- Создайте в суперклассе новый метод, скопируйте в него тело одного из методов, соответствующим образом настройте его и скомпилируйте код.
  - ⇛ *Если вы работаете со строго типизированным языком и метод вызывает другой метод, присутствующий в обоих подклассах, но не в суперклассе, объявите в суперклассе абстрактный метод.*
  - ⇛ *Если метод использует поле подкласса, воспользуйтесь рефакторингом "Подъем поля" (с. 338) или "Самоинкапсуляция поля" (с. 190) и объявите и используйте абстрактный метод получения значения.*
- Удалите один метод подкласса.
- Выполните компиляцию и тестирование.
- Продолжайте удаление методов подклассов и тестирование, пока не останется только метод суперкласса.
- Посмотрите, кто вызывает этот метод, и выясните, нельзя ли заменить тип этого объекта суперклассом.

Пример

Рассмотрим класс клиента с двумя подклассами: с обычным и с привилегированным клиентами.



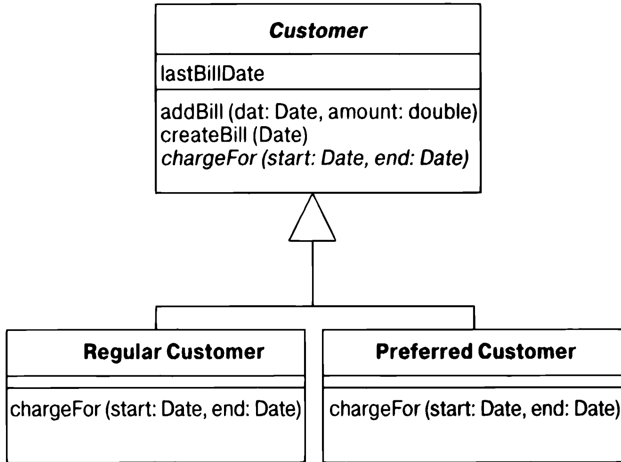
Метод `createBill` идентичен в обоих классах:

```
void createBill(date Date) {
    double chargeAmount = chargeFor(lastBillDate, date);
    addBill(date, charge);
}
```

Я не могу переместить метод в суперкласс, потому что метод `chargeFor` различный в каждом подклассе. Сначала я должен объявить его в суперклассе как абстрактный:

```
class Customer...
    abstract double chargeFor(date start, date end)
```

После этого я могу скопировать `createBill` из одного из подклассов. Я компилирую, сохраняя его на месте, а затем удаляю метод `createBill` из одного из подклассов, компилирую и тестирую. Затем я удаляю его и из другого подкласса, вновь компилирую и тестирую.



Подъем тела конструктора (Pull Up Constructor Body)

Имеются конструкторы подклассов с практически идентичными телами.

Создайте конструктор суперкласса; вызывайте его из методов подклассов.

```
class Manager extends Employee...
    public Manager(String name, String id, int grade) {
        _name = name;
        _id = id;
        _grade = grade;
    }
```

```
public Manager(String name, String id, int grade) {
    super(name, id);
    _grade = grade;
}
```

Мотивация

Конструкторы — штука непростая. Это не совсем обычные методы, поэтому при работе с ними имеется больше ограничений, чем при работе с обычными методами.

Если вы встречаете методы подклассов с одинаковым поведением, первой мыслью должно быть выделение общего поведения в отдельный метод и его подъем

в суперкласс. Однако для конструкторов общим поведением часто является построение экземпляра. В таком случае нужен конструктор суперкласса, вызываемый подклассами. Зачастую в этом и состоит все тело конструктора. Прибегнуть к рефакторингу "Подъем метода" (с. 339) в этом случае нельзя, потому что конструкторы не могут наследоваться (правда, очень жаль?).

Если данный рефакторинг становится слишком сложным, подумайте о том, чтобы вместо него воспользоваться рефакторингом "Замена конструктора фабричным методом".

Техника

- Определите конструктор суперкласса.
- Переместите общий код в начале тела из подкласса в конструктор суперкласса.

  ⇛ *Может оказаться, что это весь код.*

  ⇛ *Попробуйте переместить общий код в начало конструктора.*
- Вызовите конструктор суперкласса в качестве первого шага в конструкторе подкласса.

  ⇛ *Если весь код является общим, этот вызов будет единственной строкой в конструкторе подкласса.*
- Выполните компиляцию и тестирование.

  ⇛ *Если имеется общий код, располагающийся далее в теле класса, примените рефакторинг "Извлечение метода" (с. 132) для извлечения общего кода и примените к нему рефакторинг "Подъем метода" (с. 339).*

Пример

Рассмотрим пример классов менеджера и служащего:

```
class Employee...
    protected String _name;
    protected String _id;

class Manager extends Employee...
    public Manager(String name, String id, int grade) {
        _name = name;
        _id = id;
        _grade = grade;
    }
    private int _grade;
```

Поля в `Employee` должны быть установлены в конструкторе этого класса. Я определяю конструктор и делаю его защищенным, что должно предупреждать о том, что его должны вызывать подклассы:

```
class Employee
    protected Employee(String name, String id) {
        _name = name;
        _id = id;
    }
```

Затем я вызываю его из подкласса:

```
public Manager(String name, String id, int grade) {
    super(name, id);
    _grade = grade;
}
```

В другом варианте общий код появляется несколько дальше. Допустим, есть такой код:

```
class Employee...
    boolean isPriviliged() {
        ..
    }
    void assignCar() {
        ..
    }

class Manager...
    public Manager(String name, String id, int grade) {
        super(name, id);
        _grade = grade;

        if (isPriviliged()) assignCar(); // Выполняется
    }                                   // каждым подклассом
    boolean isPriviliged() {
        return _grade > 4;
    }
```

Я не могу переместить поведение `assignCar` в конструктор суперкласса, потому что оно должно выполняться после присваивания значения `grade` полю, так что нужно выполнить рефакторинги "Извлечение метода" (с. 132) и "Подъем метода" (с. 339).

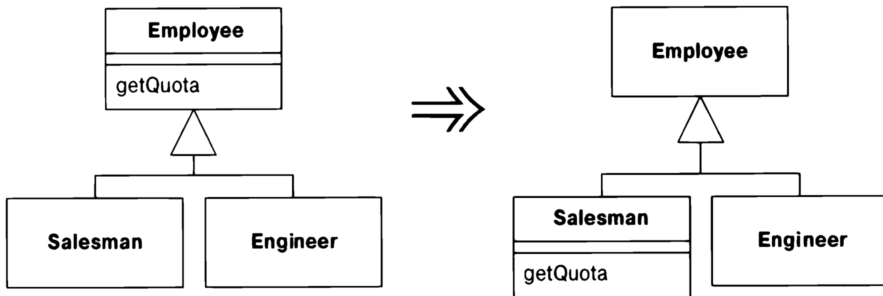
```
class Employee...
    void initialize() {
        if (isPriviliged()) assignCar();
    }
```

```
class Manager...
    public Manager(String name, String id, int grade) {
        super(name, id);
        _grade = grade;
        initialize();
    }
```

Опускание метода (Push Down Method)

В суперклассе имеется поведение, относящееся только к некоторым из его подклассов.

Переместите это поведение в соответствующие подклассы.



Мотивация

Данный рефакторинг решает задачу, противоположную рефакторингу "Подъем метода" (с. 339). Я применяю его, когда мне нужно переместить поведение из суперкласса в конкретный подкласс, обычно потому, что оно имеет смысл только в нем. Это часто происходит при использовании рефакторинга "Извлечение подкласса" (с. 347).

Техника

- Объявите метод во всех подклассах и скопируйте его тело в каждый подкласс.

  ⇛ *Может потребоваться объявить поля как защищенные, чтобы метод мог к ним обращаться. Обычно это делается, если вы намерены позже опустить поле. В противном случае используйте метод доступа к значению в суперклассе. Если этот метод доступа закрыт, необходимо объявить его защищенным.*

- Удалите метод из суперкласса.

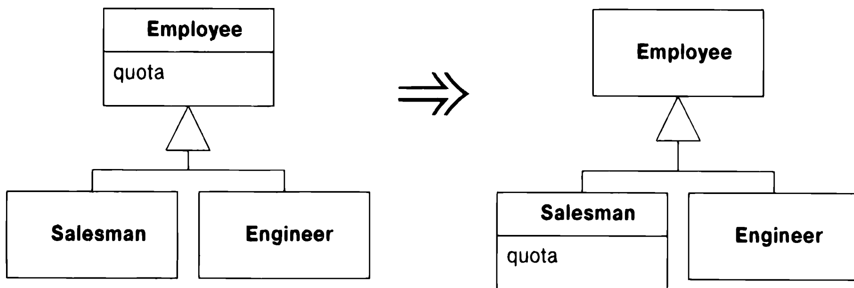
  ⇛ *Может потребоваться модифицировать вызывающий его код так, чтобы использовать подкласс в объявлениях переменных и параметров.*

  ⇛ *Если есть смысл в обращении к методу через переменную суперкласса, не планируется удаление метода из каких-либо подклассов и суперкласс является абстрактным, в суперклассе можно объявить метод как абстрактный.*

- Выполните компиляцию и тестирование.
- Удалите метод из всех подклассов, в которых он не нужен.
- Выполните компиляцию и тестирование.

Опускание поля (Push Down Field)

Имеется поле, используемое лишь некоторыми подклассами.
Переместите поле в эти подклассы.



Мотивация

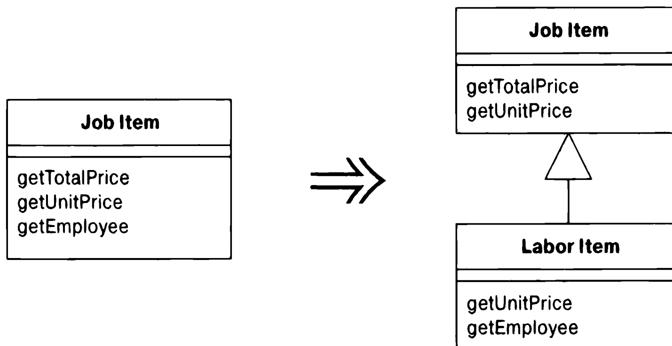
Данный рефакторинг решает задачу, обратную задаче рефакторинга “Подъем поля” (с. 338). Он применяется, когда поле требуется не в суперклассе, а в подклассе.

Техника

- Объявите поле во всех подклассах.
- Удалите поле из суперкласса.
- Выполните компиляцию и тестирование.
- Удалите поле из всех подклассов, в которых оно не нужно.
- Выполните компиляцию и тестирование.

Извлечение подкласса (Extract Subclass)

В классе есть возможности, используемые только в некоторых экземплярах.
Создайте подкласс для этого подмножества возможностей.



Мотивация

Главным побудительным мотивом применения данного рефакторинга является понимание того, что в классе есть поведение, используемое одними экземплярами и не используемое другими. Иногда об этом свидетельствует код типа, и тогда можно обратиться к рефакторингу "Замена кода типа подклассами" (с. 241) или "Замена кода типа состоянием/стратегией" (с. 245). Но вы не обязаны работать с кодом типа, чтобы использовать подклассы.

Основной альтернативой данному рефакторингу служит рефакторинг "Извлечение класса" (с. 169). Вам нужно делать выбор между делегированием и наследованием. Описываемый здесь рефакторинг обычно легче осуществить, но он имеет ряд ограничений. После того как объект создан, нельзя изменить его поведение, основанное на его классе. Изменить это поведение можно с помощью рефакторинга "Извлечение класса" (с. 169), просто подключая разные компоненты. Можно также использовать только подклассы для представления одного набора вариантов. Если вы хотите, чтобы поведение класса изменялось несколькими разными способами, для всех из них, кроме одного, вы должны применить делегирование.

Техника

- Определите новый подкласс исходного класса.
- Создайте для нового подкласса конструкторы.

  ⇛ *В простых случаях скопируйте аргументы суперкласса и вызовите конструктор суперкласса с помощью* `super`.

⇒ *Если вы хотите скрыть использование подкласса от клиентов, можете воспользоваться рефакторингом "Замена конструктора фабричным методом" (с. 321).*

- Найдите все вызовы конструкторов суперкласса. Если им нужен подкласс, замените их вызовом нового конструктора.

⇒ *Если конструктору подкласса нужны другие аргументы, воспользуйтесь рефакторингом "Переименование метода" (с. 290) для его изменения. Если некоторые параметры конструктора суперкласса больше не нужны, примените тот же рефакторинг и к нему.*

⇒ *Если суперкласс больше не может быть инстанцирован непосредственно, объявите его абстрактным.*

- Поочередно используйте рефакторинги "Опускание метода" (с. 345) и "Опускание поля" (с. 346) для перемещения функциональности в подкласс.

⇒ *В противоположность рефакторингу "Извлечение класса" (с. 169) обычно проще работать сначала с методами, а потом с данными.*

⇒ *При работе с открытым методом, чтобы вызывался новый метод, может потребоваться переопределить вызывающую переменную или тип параметра. Компилятор легко обнаруживает такие случаи.*

- Найдите все поля, определяющие информацию, на которую теперь указывает иерархия (обычно это булева величина или код типа). Устраните их с помощью рефакторинга "Самоинкапсуляция поля" (с. 190) и замены метода получения значения полиморфными константными методами. Все пользователи этого поля должны пройти рефакторинг "Замена условной инструкции полиморфизмом" (с. 271).

⇒ *Если вне класса есть методы, использующие метод доступа к значению, попробуйте перенести метод в класс с использованием рефакторинга "Перенос метода" (с. 162); затем примените рефакторинг "Замена условной инструкции полиморфизмом" (с. 271).*

- Выполните компиляцию и тестирование после каждого спуска.

Пример

Начну с класса выполняемых работ, который определяет цены операций, выполняемых в местном гараже:

```
class JobItem ...
    public JobItem(int unitPrice, int quantity, boolean isLabor,
                   Employee employee) {
        _unitPrice = unitPrice;
        _quantity = quantity;
        _isLabor = isLabor;
```

```
        _employee = employee;
    }
    public int getTotalPrice() {
        return getUnitPrice() * _quantity;
    }
    public int getUnitPrice() {
        return (_isLabor) ?
                _employee.getRate() :
                _unitPrice;
    }
    public int getQuantity() {
        return _quantity;
    }
    public Employee getEmployee() {
        return _employee;
    }
    private int _unitPrice;
    private int _quantity;
    private Employee _employee;
    private boolean _isLabor;

class Employee...
    public Employee(int rate) {
        _rate = rate;
    }
    public int getRate() {
        return _rate;
    }
    private int _rate;
```

Я выделяю из этого класса подкласс `LaborItem`, поскольку определенное поведение и данные требуются только в этом случае. Я начинаю с создания нового класса:

```
class LaborItem extends JobItem {}
```

Прежде всего мне нужен конструктор, потому что у `JobItem` нет конструктора без аргументов. Для этого я копирую сигнатуру конструктора суперкласса:

```
public LaborItem(int unitPrice, int quantity, boolean isLabor,
                 Employee employee) {
    super(unitPrice, quantity, isLabor, employee);
}
```

Этого достаточно, чтобы скомпилировать новый подкласс. Однако этот конструктор неаккуратен: некоторые аргументы для `LaborItem` нужны, а некоторые — нет. Я займусь этим позже.

На следующем шаге я выполняю поиск вызовов конструктора `JobItem` и случаев, когда вместо него должен быть вызван конструктор `LaborItem`. Поэтому инструкции вида

```
JobItem j1 = new JobItem (0, 5, true, kent);
```

приобретают следующий вид:

```
JobItem j1 = new LaborItem (0, 5, true, kent);
```

На этом этапе я не изменял тип переменной, а изменил лишь тип конструктора. Дело в том, что я хочу использовать новый тип только там, где это необходимо. В данный момент у меня нет специфического для подкласса интерфейса, так что я не хочу пока что объявлять какие-либо иные варианты.

Теперь самое время очистить списки параметров конструктора. К каждому из них я применяю рефакторинг “Переименование метода” (с. 290). Сначала я работаю с суперклассом. Я создаю новый конструктор и делаю старый защищенным (он по-прежнему нужен подклассу):

```
class JobItem...
    protected JobItem(int unitPrice, int quantity, boolean isLabor,
                      Employee employee) {
        _unitPrice = unitPrice;
        _quantity = quantity;
        _isLabor = isLabor;
        _employee = employee;
    }
    public JobItem(int unitPrice, int quantity) {
        this(unitPrice, quantity, false, null)
    }
```

Вызовы извне теперь используют новый конструктор:

```
JobItem j2 = new JobItem (10, 15);
```

После компиляции и тестирования я применяю рефакторинг “Переименование метода” (с. 290) к конструктору подкласса:

```
class LaborItem
    public LaborItem(int quantity, Employee employee) {
        super(0, quantity, true, employee);
    }
```

Пока что я продолжаю использовать защищенный конструктор суперкласса.

Теперь я могу начать опускать в подкласс функциональность JobItem. Начну с методов. Я применяю рефакторинг “Опускание метода” (с. 345) к getEmployee:

```
class LaborItem...
    public Employee getEmployee() {
        return _employee;
    }

class JobItem...
    protected Employee _employee;
```

Поскольку поле _employee позже будет опущено в подкласс, пока что я объявляю его защищенным.

После того как поле _employee защищено, я могу привести в порядок конструкторы так, чтобы поле _employee инициализировалось только в подклассе, куда оно опускается:

```
class JobItem...
    protected JobItem(int unitPrice, int quantity, boolean isLabor) {
        _unitPrice = unitPrice;
        _quantity = quantity;
        _isLabor = isLabor;
    }

class LaborItem ...
    public LaborItem(int quantity, Employee employee) {
        super(0, quantity, true);
        _employee = employee;
    }
```

Поле _isLabor применяется для указания информации, которая теперь присуща иерархии, так что это поле можно удалить. Лучше всего это осуществить, применив рефакторинг “Самоинкапсуляция поля” (с. 190), а затем изменив метод доступа к значению, чтобы применить полиморфный константный метод. Полиморфный константный метод — это такой метод, посредством которого каждая реализация возвращает (свое) фиксированное значение:

```
class JobItem...
    protected boolean isLabor() {
        return false;
    }

class LaborItem...
    protected boolean isLabor() {
        return true;
    }
```

Теперь я могу избавиться от поля isLabor.

Сейчас можно бросить взгляд на пользователей методов isLabor. Они должны быть подвергнуты рефакторингу “Замена условной инструкции полиморфизмом” (с. 271). Я беру метод

```
class JobItem...
    public int getUnitPrice() {
        return (isLabor()) ?
               _employee.getRate() :
               _unitPrice;
    }
```

и заменяю его следующим кодом:

```
class JobItem...
    public int getUnitPrice() {
        return _unitPrice;
    }

class LaborItem...
    public int getUnitPrice() {
        return _employee.getRate();
    }
```

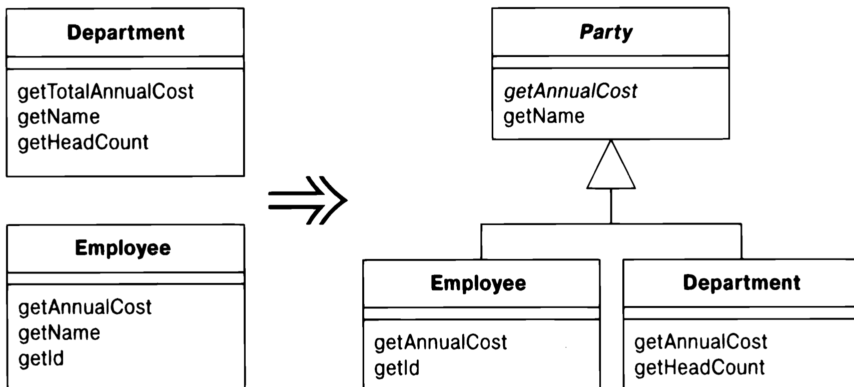
После того как группа методов, которая использует некоторые данные, перемещена в подкласс, можно воспользоваться рефакторингом “Опускание поля” (с. 346). То, что я не могу использовать его, так как метод использует данные, является сигналом для дальнейшей работы с методами либо с помощью рефакторинга “Опускание метода” (с. 345), либо с помощью рефакторинга “Замена условной инструкции полиморфизмом” (с. 271).

Поскольку цена узла (`unitPrice`) используется только элементами, которые не представляют трудовые затраты, я могу снова применить рефакторинг “Извлечение подкласса” (с. 347) к `JobItem` и создать класс, представляющий запчасти. В итоге класс `JobItem` станет абстрактным.

Извлечение суперкласса (Extract Superclass)

Имеются два класса с аналогичной функциональностью.

Создайте общий суперкласс и переместите в него общую функциональность.



Мотивация

Дублирование кода представляет собой один из главных недостатков программных систем. Если одно и то же действие выполняется в нескольких местах, то при замене этих действий какими-то иными придется редактировать больше мест, чем можно было бы.

Одним из видов дублирования кода является наличие двух классов, выполняющих сходные задачи — не важно, одинаковым способом или разными. Объекты предоставляют встроенный механизм для упрощения этой ситуации с использованием наследования. Однако часто общность оказывается незамеченной до тех пор, пока не будут созданы какие-то классы, и тогда структуру наследования придется создавать не сразу же, а позднее.

Альтернативой данному рефакторингу является рефакторинг "Извлечение класса" (с. 169). По сути, выбор осуществляется между наследованием и делегированием. Наследование представляется более простым, если у двух классов одинаковы и интерфейс, и поведение. Если выбор сделан неправильно, в дальнейшем можно будет применить рефакторинг "Замена наследования делегированием" (с. 369).

Техника

- Создайте пустой абстрактный суперкласс; сделайте исходные классы подклассами этого суперкласса.
- По очереди выполняйте рефакторинги "Подъем поля" (с. 338), "Подъем метода" (с. 339) и "Подъем тела конструктора" (с. 342), чтобы переместить общие элементы в суперкласс.

  ⇛ *Обычно проще сначала переместить поля.*

  ⇛ *Если в подклассах есть методы с разными сигнатурами, но с одинаковым предназначением, воспользуйтесь рефакторингом "Переименование метода" (с. 290) для того, чтобы дать им одинаковые имена, а затем выполните рефакторинг "Подъем метода" (с. 339).*

  ⇛ *Если есть методы с одинаковыми сигнатурами, но различными телами, объявите общую сигнатуру как абстрактный метод суперкласса.*

  ⇛ *Если есть методы с разными телами, которые решают одну и ту же задачу, можно попробовать применить рефакторинг "Замена алгоритма" (с. 159), чтобы скопировать тело одного метода в другой. Если это сработает, можно будет воспользоваться рефакторингом "Подъем метода" (с. 339).*

- После каждого подъема в суперкласс выполняйте компиляцию и тестирование.
- Рассмотрите оставшиеся в подклассах методы. Посмотрите, нет ли в них общих участков; если есть — к ним можно применить рефакторинг “Извлечение метода” (с. 132) с последующим рефакторингом “Подъем метода” (с. 339) для этих общих частей. Если похож общий поток команд, то может оказаться возможным рефакторинг “Формирование шаблонного метода” (с. 361).
- После подъема в суперкласс всех общих элементов проверьте каждый клиент подклассов. Если они используют только общий интерфейс, то можно заменить требуемый им тип суперклассом.

Пример

В качестве примера у меня есть служащий (employee) и отдел (department):

```
class Employee...
    public Employee(String name, String id, int annualCost) {
        _name = name;
        _id = id;
        _annualCost = annualCost;
    }
    public int getAnnualCost() {
        return _annualCost;
    }
    public String getId() {
        return _id;
    }
    public String getName() {
        return _name;
    }
    private String _name;
    private int _annualCost;
    private String _id;

public class Department...
    public Department(String name) {
        _name = name;
    }
    public int getTotalAnnualCost() {
        Enumeration e = getStaff();
        int result = 0;

        while (e.hasMoreElements()) {
            Employee each = (Employee) e.nextElement();
            result += each.getAnnualCost();
        }
```

```
        return result;
    }
    public int getHeadCount() {
        return _staff.size();
    }
    public Enumeration getStaff() {
        return _staff.elements();
    }
    public void addStaff(Employee arg) {
        _staff.addElement(arg);
    }
    public String getName() {
        return _name;
    }
    private String _name;
    private Vector _staff = new Vector();
```

Имеется пара обладающих общностью областей кода. Во-первых, и у служащих, и у отделов есть имя или название (`_name`). Во-вторых, у обоих есть годовой бюджет (annual cost), правда, методы его расчета немного разные у разных классов. Эти общие возможности я выделяю в суперкласс. На первом этапе я создаю новый суперкласс, а имеющиеся суперклассы определяю как его подклассы:

```
abstract class Party {}
    class Employee extends Party...
    class Department extends Party...
```

Теперь я начинаю поднимать функциональность в суперкласс. Обычно проще начать с выполнения рефакторинга "Подъем поля" (с. 338):

```
class Party...
    protected String _name;
```

Затем я могу применить рефакторинг "Подъем метода" (с. 339) к методам доступа к значениям полей:

```
class Party {
    public String getName() {
        return _name;
    }
```

Я предпочитаю делать поля закрытыми. Чтобы сделать это, мне нужно выполнить рефакторинг "Подъем тела конструктора" (с. 342), чтобы присваивать имя:

```
class Party...
    protected Party(String name) {
        _name = name;
    }
    private String _name;
```

```
class Employee...
    public Employee(String name, String id, int annualCost) {
        super(name);
        _id = id;
        _annualCost = annualCost;
    }

class Department...
    public Department(String name) {
        super(name);
    }
```

Методы Department.getTotalAnnualCost и Employee.getAnnualCost имеют одинаковое предназначение, так что они должны иметь одинаковые имена. Сначала я выполняю рефакторинг "Переименование метода" (с. 290), чтобы дать им одно и то же имя:

```
class Department extends Party {
    public int getAnnualCost() {
        Enumeration e = getStaff();
        int result = 0;
        while (e.hasMoreElements()) {
            Employee each = (Employee) e.nextElement();
            result += each.getAnnualCost();
        }
        return result;
    }
```

Тела этих методов все еще различны, поэтому я не могу применить рефакторинг "Подъем метода" (с. 339), но я могу объявить абстрактный метод в суперклассе:

```
abstract public int getAnnualCost()
```

Выполнив эти очевидные изменения, я рассматриваю клиенты обоих классов, чтобы понять, что можно изменить так, чтобы они использовали новый суперкласс. Одним из клиентов этих классов является сам класс Department, содержащий коллекцию классов служащих. Метод getAnnualCost использует только метод подсчета годового бюджета, который теперь объявлен в Party:

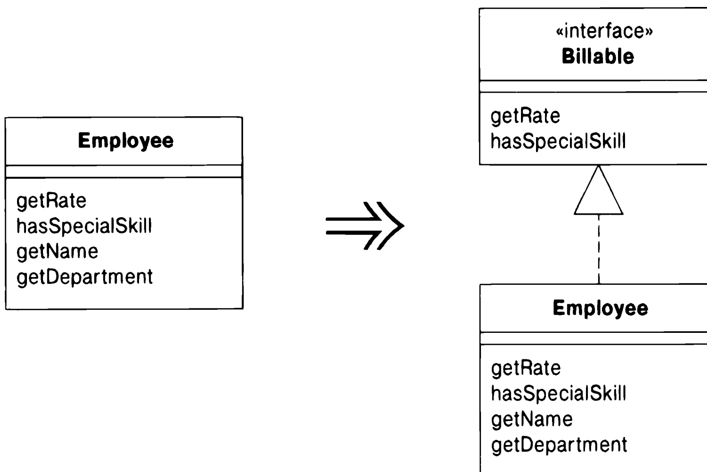
```
class Department...
    public int getAnnualCost() {
        Enumeration e = getStaff();
        int result = 0;
        while (e.hasMoreElements()) {
            Party each = (Party) e.nextElement();
            result += each.getAnnualCost();
        }
        return result;
    }
```

Это поведение предлагает новую возможность. Я могу трактовать отдел и служащего как *композит* [9]. Это позволит мне включать один отдел в другой. В результате получается новая функциональность, так что, строго говоря, назвать это рефакторингом нельзя. Если бы мне требовался композит, я мог бы получить его, изменяя имя поля, чтобы картина была нагляднее. Это изменение повлекло бы за собой соответствующее изменение имени `addStaff` и изменение параметра на `Party`. Последнее изменение делало бы метод `headCount` рекурсивным. Я мог бы сделать это, создав метод `headCount` для `Employee`, который просто возвращал бы значение 1, и применив рефакторинг "Замена алгоритма" (с. 159) для соответствующего метода отдела, который просто суммировал бы значения своих компонентов.

Извлечение интерфейса (Extract Interface)

Несколько клиентов используют одно и то же подмножество интерфейса класса, или в двух классах часть интерфейса является общей.

Выделите это подмножество в интерфейс.



Мотивация

Классы используют один другой разными способами. Использование класса часто означает обращение ко всей области ответственности класса. В других случаях группа клиентов использует только некоторое подмножество предоставляемой классом функциональности. Бывает и так, что класс должен работать с любым классом, который может обрабатывать определенные запросы.

Для двух последних случаев часто полезно заставить подмножество ответственности выступать от собственного имени, чтобы сделать его применение в системе более ясным. Это позволяет легче увидеть разделение ответственности в системе. Если для поддержки такого подмножества необходимы новые классы, то будет проще понять, что именно входит в это подмножество.

В ряде объектно-ориентированных языков такая возможность поддерживается с помощью множественного наследования. Для каждого сектора поведения создаются классы, которые затем объединяются в реализацию. В Java наследование одиночное, но такого рода требование можно сформулировать и реализовать с помощью интерфейсов. Интерфейсы оказали большое влияние на подход программистов к проектированию приложений Java. Даже программирующие на Smalltalk полагают, что интерфейсы являются большим шагом вперед!

Имеется определенное сходство между данным рефакторингом и рефакторингом "Извлечение суперкласса" (с. 352). Рефакторинг "Извлечение интерфейса" позволяет выявлять только общие интерфейсы, но не общий код. Применяя его, можно получить запах дублируемого кода. Решить проблему можно с помощью рефакторинга "Извлечение класса" (с. 169), помещая поведение в компонент и выполняя делегирование. Если объем общего поведения достаточно велик, то проще применить рефакторинг "Извлечение суперкласса" (с. 352), но так вы получите только один суперкласс.

Интерфейсы хорошо использовать, когда в разных ситуациях классы играют разные роли. Воспользуйтесь рассматриваемым здесь рефакторингом для каждой роли. Еще один полезный случай возникает, когда вы хотите описать выходной интерфейс класса, т.е. операции, которые класс выполняет на своем сервере. Если в будущем предполагается использование серверов других видов, все они должны реализовывать данный интерфейс.

Техника

- Создайте пустой интерфейс.
- Объявите в интерфейсе общие операции.
- Объявите соответствующие классы как реализующие интерфейс.
- Обновите объявления типов клиентов так, чтобы в них использовался этот интерфейс.

Пример

Класс табеля учета отработанного времени генерирует начисления для служащих. Для этого ему необходимо знать ставку оплаты служащего и наличие у него особых навыков:

```
double charge(Employee emp, int days) {
    int base = emp.getRate() * days;
    if (emp.hasSpecialSkill())
        return base * 1.05;
    else return base;
}
```

У служащего, помимо информации о ставке оплаты и специальных навыках, есть много других характеристик, но в данном приложении требуются только они. Тот факт, что для работы требуется только это подмножество, можно подчеркнуть, определив для него интерфейс:

```
interface Billable {
    public int getRate();
    public boolean hasSpecialSkill();
}
```

После этого можно объявить класс `Employee` как реализующий этот интерфейс:

```
class Employee implements Billable ...
```

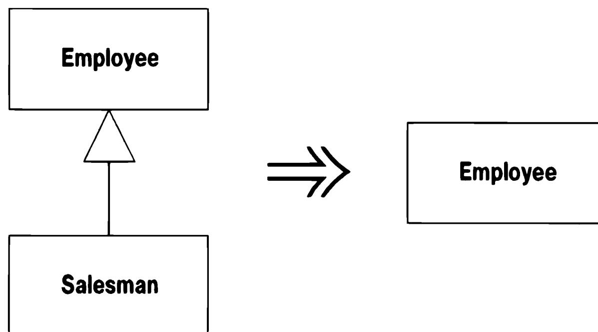
После этого можно изменить объявление `charge` так, чтобы показать, что используется только эта часть поведения `Employee`:

```
double charge(Billable emp, int days) {
    int base = emp.getRate() * days;
    if (emp.hasSpecialSkill())
        return base * 1.05;
    else return base;
}
```

В данном случае получено скромное преимущество, заключающееся в документированности кода. Для единственного метода такое преимущество не стоит затраченного труда, но было бы полезно, если бы несколько классов стали применять интерфейс `Billable`. Существенное преимущество появится тогда, когда я, например, захочу выставлять счета еще и для компьютеров. Все, что мне для этого будет нужно, — реализовать в них интерфейс `Billable`, и тогда можно будет смело включать компьютеры в табель учета времени.

Свертывание иерархии (Collapse Hierarchy)

Суперкласс и подкласс не слишком различаются.
Объедините их в один класс.



Мотивация

После некоторого времени работы с иерархией классов она может постепенно становиться все более запутанной. При проведении рефакторинга иерархии методы и поля зачастую перемещаются вверх или вниз. Так что по завершении работы вполне может обнаружиться подкласс, не имеющий никакой ценности, поэтому лучшим выходом может оказаться объединение классов.

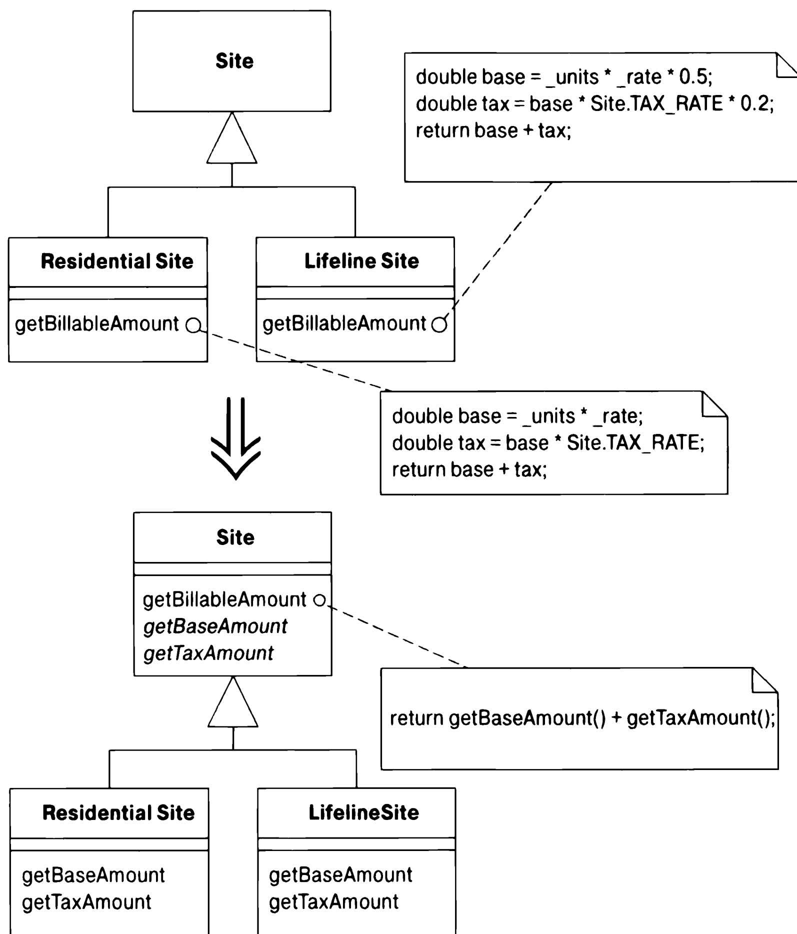
Техника

- Выберите, какой из классов будет удален — суперкласс или подкласс.
- Воспользуйтесь рефакторингами "Подъем поля" (с. 338) и "Подъем метода" (с. 339) (или рефакторингами "Опускание метода" (с. 345) и "Опускание поля" (с. 346)) для перемещения поведения и данных удаляемого класса в класс, с которым он будет слит воедино.
- Выполняйте компиляцию и тестирование после каждого перемещения.
- Замените обращения к удаляемому классу обращениями к объединенному классу. Это влияет на объявления переменных, типы параметров и конструкторы.
- Удалите пустой класс.
- Выполните компиляцию и тестирование.

Формирование шаблонного метода (Form Template Method)

В подклассах есть два метода, выполняющие аналогичные шаги в одинаковом порядке, однако сами шаги различны.

Образуйте из этих шагов методы с одинаковой сигнатурой, чтобы исходные методы стали одинаковыми. После этого их можно перенести в суперкласс.



Мотивация

Наследование является мощным инструментом устранения дублирования поведения. Встретив два аналогичных метода в подклассах, мы хотим объединить их в суперклассе. Но что если они совпадают не в точности? Как поступить в этом

случае? Мы по-прежнему должны устранить как можно больше дублирования, но при этом сохранить существенные различия.

Зачастую возникает ситуация, когда два метода выполняют во многом аналогичные шаги в одинаковой последовательности, но шаги эти разные. В таком случае можно переместить последовательность шагов в суперкласс и позволить полиморфизму обеспечить выполнение различных действий. Такой прием называется *шаблонным методом* [9].

Техника

- Выполните декомпозицию методов так, чтобы все выделенные методы полностью совпадали или полностью различались.
- С помощью рефакторинга "Подъем метода" (с. 339) переместите идентичные методы в суперкласс.
- К различающимся методам примените рефакторинг "Переименование метода" (с. 290) так, чтобы сигнатуры всех методов на каждом шаге были одинаковы.

  ⇛ *Это делает исходные методы одинаковыми в том смысле, что они выполняют одну и ту же последовательность вызовов методов, но подклассы по-разному обрабатывают эти вызовы.*
- Выполните компиляцию и тестирование после каждого изменения сигнатуры.
- Примените рефакторинг "Подъем метода" (с. 339) к одному из исходных методов. Определите сигнатуры различных методов как абстрактные методы суперкласса.
- Выполните компиляцию и тестирование.
- Удалите прочие методы и выполните компиляцию и тестирование после каждого удаления.

Пример

Закончу начатое в главе 1, "Первый пример рефакторинга". В ней у меня был класс `Customer` с двумя методами вывода выписок счетов клиента. Метод `statement` выводит выписку в виде ASCII:

```
public String statement() {
    Enumeration rentals = _rentals.elements();
    String result = "Прокат " + getName() + "\n";
```

```
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        // Вывод результатов для каждого проката
        result += "\t" + each.getMovie().getTitle() + "\t" +
                  String.valueOf(each.getCharge()) + "\n";
    }

    // Добавление колонтитула
    result += "Сумма задолженности " +
              String.valueOf(getTotalCharge()) + "\n";
    result += "Вы заработали " +
              String.valueOf(getTotalFrequentRenterPoints()) +
              " бонусных очков";
    return result;
}
```

в то время как `htmlStatement` создает выписку в HTML:

```
public String htmlStatement() {
    Enumeration rentals = _rentals.elements();
    String result = "<H1>Прокат <EM>" + getName() +
                    "</EM></H1><P>\n";

    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        // Вывод результатов для каждого проката
        result += each.getMovie().getTitle() + ": " +
                  String.valueOf(each.getCharge()) + "<BR>\n";
    }

    // Добавление колонтитула
    result += "<P>Сумма задолженности <EM>" +
              String.valueOf(getTotalCharge()) + "</ EM><P>\n";
    result += "Вы заработали <EM>" +
              String.valueOf(getTotalFrequentRenterPoints()) +
              "</EM> бонусных очков<P>";
    return result;
}
```

Прежде чем использовать рефакторинг “Формирование шаблонного метода”, я должен сделать так, чтобы эти два метода появились в подклассах некоторого общего суперкласса. Я делаю это с помощью объекта методов [6], создавая отдельную иерархию стратегий для вывода выписок (рис. 11.1).

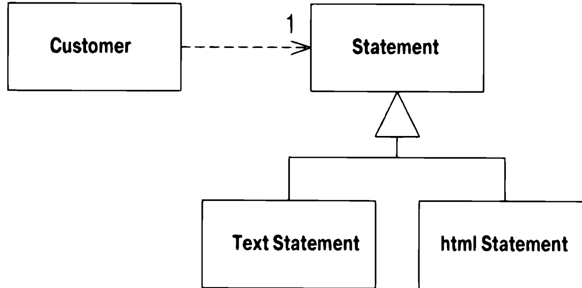


Рис. 11.1. *Применение стратегии к выпискам*

```
class Statement {}
class TextStatement extends Statement {}
class HtmlStatement extends Statement {}
```

Теперь с помощью рефакторинга “Перенос метода” (с. 162) я переношу эти два метода в подклассы:

```
class Customer...
    public String statement() {
        return new TextStatement().value(this);
    }
    public String htmlStatement() {
        return new HtmlStatement().value(this);
    }

class TextStatement {
    public String value(Customer aCustomer) {
        Enumeration rentals = aCustomer.getRentals();
        String result = "Прокат " + aCustomer.getName() + "\n";

        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            // Вывод результатов для каждого проката
            result += "\t" + each.getMovie().getTitle() + "\t" +
                      String.valueOf(each.getCharge()) + "\n";
        }

        // Добавление колонтитула
        result += "Сумма задолженности " +
                  String.valueOf(aCustomer.getTotalCharge()) + "\n";
        result += "Вы заработали " +
                  String.valueOf(
                      aCustomer.getTotalFrequentRenterPoints()) +
                  " бонусных очков";
        return result;
    }
```

```
class HtmlStatement {
    public String value(Customer aCustomer) {
        Enumeration rentals = aCustomer.getRentals();
        String result = "<H1> Прокат <EM>" +
                        aCustomer.getName() + "</EM></ H1><P>\n";

        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            // Вывод результатов для каждого проката
            result += each.getMovie().getTitle() + ": " +
                      String.valueOf(each.getCharge()) + "<BR>\n";
        }

        // Добавление колонтитула
        result += "<P>Сумма задолженности <EM>" +
                  String.valueOf(aCustomer.getTotalCharge()) +
                  "</EM><P>\n";
        result += "Вы заработали <EM>" +
                  String.valueOf(
                    aCustomer.getTotalFrequentRenterPoints())+
                  "</EM> бонусных очков<P>";
        return result;
    }
```

Перемещая методы, я переименовал их для лучшего соответствия стратегии. Я дал им одинаковые имена, потому что различие между ними теперь находится в классе, а не в методе. (Для тех, кто повторяет этот пример: мне пришлось также добавить метод `getRentals` в `Customer` и ослабить видимость `getTotalCharge` и `getTotalFrequentRenterPoints`.)

Наличие двух аналогичных методов в подклассах позволяет применить рефакторинг "Формирование шаблонного метода". Ключ к этому рефакторингу лежит в разделении различающегося и совпадающего кодов путем применения рефакторинга "Извлечение метода" (с. 132) для извлечения тех фрагментов, которые в двух методах различны. При каждом извлечении я создаю методы с различными телами, но одной и той же сигнатурой.

Первый пример — вывод заголовка. В обоих методах информация запрашивается из `Customer`, но результирующая строка форматируется по-разному. Я могу вынести форматирование этой строки в отдельные методы с одинаковой сигнатурой:

```
class TextStatement...
    String headerString(Customer aCustomer) {
        return "Прокат " + aCustomer.getName() + "\n";
    }
    public String value(Customer aCustomer) {
        Enumeration rentals = aCustomer.getRentals();
        String result = headerString(aCustomer);
```

```
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            // Вывод результатов для каждого проката
            result += "\t" + each.getMovie().getTitle() + "\t" +
                      String.valueOf(each.getCharge()) + "\n";
        }

        // Добавление колонтитула
        result += "Сумма задолженности " +
                  String.valueOf(aCustomer.getTotalCharge()) + "\n";
        result += "Вы заработали " +
                  String.valueOf(
                      aCustomer.getTotalFrequentRenterPoints()) +
                  " бонусных очков";
        return result;
    }

class HtmlStatement...
    String headerString(Customer aCustomer) {
    return "<H1>Прокат <EM>" + aCustomer.getName() + "</
        EM></H1><P>\n";
}
public String value(Customer aCustomer) {
    Enumeration rentals = aCustomer.getRentals();
    String result = headerString(aCustomer);

    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        // Вывод результатов для каждого проката
        result += each.getMovie().getTitle() + ": " +
                  String.valueOf(each.getCharge()) + "<BR>\n";
    }

    // Добавление колонтитула
    result += "<P>Сумма задолженности <EM>" +
              String.valueOf(aCustomer.getTotalCharge())+"</EM><P>\n";
    result += "Вы заработали <EM>" +
              String.valueOf(
                  aCustomer.getTotalFrequentRenterPoints()) +
              "</EM> бонусных очков<P>";
    return result;
}
```

Теперь я выполняю компиляцию и тестирование, а затем продолжаю работу с другими элементами. Я выполнил все шаги за один раз и получил следующий результат:

```
class TextStatement...
    public String value(Customer aCustomer) {
        Enumeration rentals = aCustomer.getRentals();
        String result = headerString(aCustomer);
```

```
        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            result += eachRentalString(each);
        }

        result += footerString(aCustomer);
        return result;
    }
    String eachRentalString(Rental aRental) {
        return "\t" + aRental.getMovie().getTitle() + "\t" +
               String.valueOf(aRental.getCharge()) + "\n";
    }
    String footerString(Customer aCustomer) {
        return "Сумма задолженности " +
               String.valueOf(aCustomer.getTotalCharge()) + "\n" +
               "Вы заработали " +
               String.valueOf(
                   aCustomer.getTotalFrequentRenterPoints()) +
               " бонусных очков ";
    }

class HtmlStatement...
    public String value(Customer aCustomer) {
        Enumeration rentals = aCustomer.getRentals();
        String result = headerString(aCustomer);

        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            result += eachRentalString(each);
        }

        result += footerString(aCustomer);
        return result;
    }
    String eachRentalString(Rental aRental) {
        return aRental.getMovie().getTitle() + ": " +
               String.valueOf(aRental.getCharge()) + "<BR>\n";
    }
    String footerString(Customer aCustomer) {
        return "<P>Сумма задолженности <EM>" +
               String.valueOf(aCustomer.getTotalCharge()) +
               "</EM><P>\n" + "Вы заработали <EM>" +
               String.valueOf(aCustomer.
getTotalFrequentRenterPoints()) +
               "</EM> бонусных очков <P>";
    }
```

После внесения описанных изменений оба метода `value` выглядят очень похожими. Поэтому я применяю к одному из них рефакторинг “Подъем метода” (с. 339), произвольно выбирая версию исходного текста. При подъеме метода я должен объявить методы подклассов как абстрактные:

```
class Statement...
    public String value(Customer aCustomer) {
        Enumeration rentals = aCustomer.getRentals();
        String result = headerString(aCustomer);

        while (rentals.hasMoreElements()) {
            Rental each = (Rental) rentals.nextElement();
            result += eachRentalString(each);
        }

        result += footerString(aCustomer);
        return result;
    }
    abstract String headerString(Customer aCustomer);
    abstract String eachRentalString(Rental aRental);
    abstract String footerString(Customer aCustomer);
```

Далее я удаляю метод `value` из текстового варианта, компилирую и тестирую. Если все в порядке, я удаляю метод `value` из HTML-варианта, компилирую и тестирую снова. Полученный результат показан на рис. 11.2.

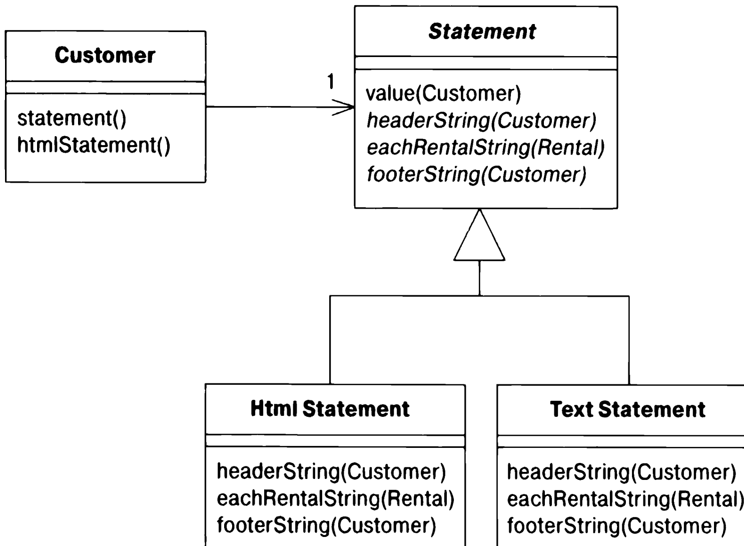


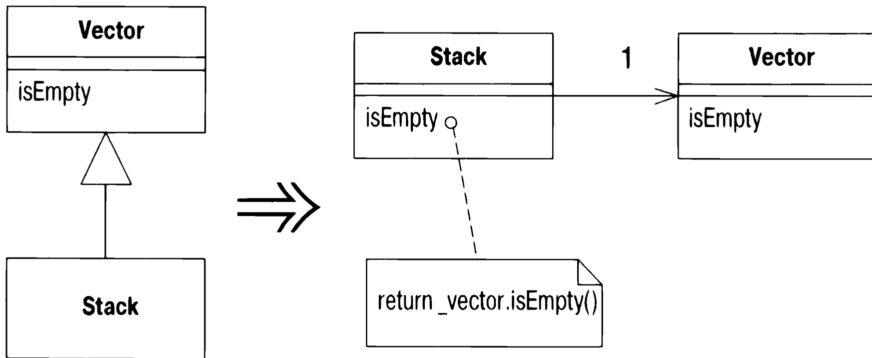
Рис. 11.2. *Классы после формирования шаблонного метода*

После выполнения этого рефакторинга становится легко добавлять новые виды вывода. Для этого требуется лишь создать подкласс, в котором будут перекрыты три абстрактных метода.

Замена наследования делегированием (Replace Inheritance with Delegation)

Подкласс использует только часть интерфейса суперкласса или не желает наследовать данные.

Создайте поле для суперкласса, настройте методы так, чтобы они делегировали выполнение суперклассу, и удалите подклассы.



Мотивация

Наследование — замечательная вещь, но иногда оно является не совсем тем, что нам требуется. Зачастую, наследуя класс, позже мы обнаруживаем, что многие из операций суперкласса для подкласса в действительности оказываются неприемлемы. В этом случае имеющийся интерфейс неправильно отражает то, что делает класс. Может оказаться, что вы наследуете целую группу данных, непригодных для подкласса. Может также обнаружиться, что в суперклассе есть защищенные методы, не имеющие особого смысла в подклассе.

Конечно, можно оставить все как есть и считать, что это подкласс, просто в нем используется лишь часть функций суперкласса. Но такое решение приводит к коду, который говорит одно, в то время как ваши намерения — совсем другое. Вы получаете беспорядок, который необходимо ликвидировать любой ценой.

Применяя вместо наследования делегирование, мы открыто заявляем, что используем делегируемый класс лишь частично. Мы управляем тем, какую часть интерфейса берем, а какую игнорируем. Цена такого решения — дополнительные делегирующие методы, писать которые скучно, но зато очень просто, так что это трудно сделать неправильно.

Техника

- Создайте в подклассе поле, ссылающееся на экземпляр суперкласса. Инициализируйте его значением `this`.
- Измените все методы, определенные в подклассе, таким образом, чтобы они использовали поле делегата. После изменения каждого метода выполните компиляцию и тестирование.

  ⇛ *Вы не можете заменить методы, вызывающие метод* `super`, *определенный в подклассе, иначе можно получить бесконечную рекурсию. Эти методы можно будет заменить только после того, как будет разорвано наследование.*
- Удалите объявление суперкласса и замените присвоение делегата присвоением новому объекту.
- Для каждого метода суперкласса, используемого клиентом, добавьте простой делегирующий метод.
- Выполните компиляцию и тестирование.

Пример

Одним из классических примеров плохого наследования является стек, объявленный как подкласс вектора. Java 1.1 поступает так в своих утилитах (как не стыдно!), но сейчас я использую упрощенную форму стека:

```
class MyStack extends Vector {

    public void push(Object element) {
        insertElementAt(element, 0);
    }

    public Object pop() {
        Object result = firstElement();
        removeElementAt(0);
        return result;
    }
}
```

Глядя на пользователей этого класса, я вижу, что клиенты выполняют над стеком лишь четыре операции: `push`, `pop`, `size` и `isEmpty`. Последние две операции наследуются из `Vector`.

Я начинаю делегирование с создания поля для вектора. Я связываю это поле с `this`, чтобы иметь возможность использовать и делегирование, и наследование, пока выполняю этот рефакторинг:

```
private Vector _vector = this;
```

Теперь я начинаю заменять методы так, чтобы они использовали делегирование. Начну с push:

```
public void push(Object element) {
    _vector.insertElementAt(element, 0);
}
```

Сейчас можно выполнить компиляцию и тестирование, и все должно корректно работать. Займемся теперь методом pop:

```
public Object pop() {
    Object result = _vector.firstElement();
    _vector.removeElementAt(0);
    return result;
}
```

Завершив работу с этими методами подкласса, я должен разорвать связь с суперклассом:

```
class MyStack extends Vector
    private Vector _vector = new Vector();
```

После этого я добавляю простые делегирующие методы для методов суперкласса, используемых клиентами:

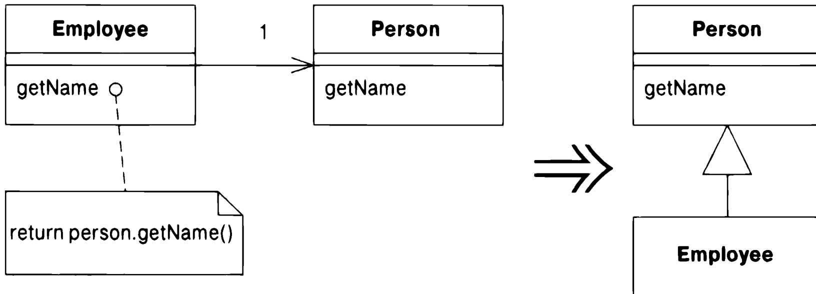
```
public int size() {
    return _vector.size();
}
public boolean isEmpty() {
    return _vector.isEmpty();
}
```

Теперь я могу выполнить компиляцию и тестирование. Если я забыл добавить делегирующий метод, компилятор напомнит мне об этом.

Замена делегирования наследованием (Replace Delegation with Inheritance)

Вы используете делегирование и часто пишете много простых делегирующих методов для всего интерфейса.

Сделайте делегирующий класс подклассом делегата.



Мотивация

Это оборотная сторона рефакторинга “Замена наследования делегированием” (с. 369). Если выясняется, что используются все методы делегата, и вам надоело писать эти простые методы делегирования, можно легко и просто вернуться к наследованию.

Но при этом нужно помнить о некоторых опасностях. Если вы используете не все методы делегата, то применять данный рефакторинг не следует, так как подкласс всегда должен следовать интерфейсу суперкласса. Если делегирующие методы вас утомляют, можно воспользоваться другими вариантами. Так, можно разрешить клиентам обращаться к делегируемому классу самостоятельно, применив рефакторинг “Удаление посредника” (с. 179). Можно воспользоваться рефакторингом “Извлечение суперкласса” (с. 352), чтобы отделить общий интерфейс, а затем наследовать новый класс. Аналогично можно также использовать рефакторинг “Извлечение интерфейса” (с. 357).

Еще одна ситуация, о которой необходимо знать, — это когда делегат совместно используется несколькими объектами и может быть изменен. В таком случае заменить делегирование наследованием нельзя, потому что данные больше не смогут использоваться совместно. Совместное использование данных — это ответственность, которую нельзя вернуть в наследование. Если объект неизменяем, совместное использование данных не вызывает проблем, поскольку можно обойтись простым копированием, и никто ничего не скажет.

Техника

- Сделайте делегирующий объект подклассом делегата.
- Выполните компиляцию.

  ⇛ *В этот момент может обнаружиться коллизия имен методов; методы могут иметь одинаковые имена, но разные типы возвращаемых*

значений, генерируемые исключения или видимость. Внесите необходимые исправления с помощью рефакторинга "Переименование метода" (с. 290).

- Присвойте полю делегата значение самого объекта.
- Удалите простые методы делегирования.
- Выполните компиляцию и тестирование.
- Замените все прочие делегирования обращениями к самому объекту.
- Удалите поле делегата.

Пример

Пусть простой класс `Employee` выполняет делегирование не менее простому классу `Person`:

```
class Employee {
    Person _person = new Person();

    public String getName() {
        return _person.getName();
    }
    public void setName(String arg) {
        _person.setName(arg);
    }
    public String toString() {
        return "Emp: " + _person.getLastName();
    }
}

class Person {
    String _name;

    public String getName() {
        return _name;
    }
    public void setName(String arg) {
        _name = arg;
    }
    public String getLastName() {
        return _name.substring(_name.lastIndexOf(' ') + 1);
    }
}
```

Первый шаг состоит в объявлении подкласса:

```
class Employee extends Person
```

В этот момент компиляция предупредит о возможных конфликтах имен методов. Они возникают, когда методы с одинаковым именем возвращают значения различных типов или генерируют разные исключения. Все проблемы такого рода исправляются с помощью рефакторинга “Переименование метода” (с. 290). В нашем простом примере таких проблем нет.

Следующий шаг состоит в том, что поле делегата ссылается на сам объект. Я должен удалить все простые делегирующие методы, такие как `getName` и `setName`. Если их оставить, я получу ошибку переполнения стека, вызванную бесконечной рекурсией. В данном случае нужно удалить методы `getName` и `setName` из `Employee`.

После того как я получаю работающий класс, я могу изменить те методы, которые используют делегирующие методы. Я переключаю их на непосредственное использование вызовов:

```
public String toString() {
    return "Emp: " + getLastName();
}
```

Теперь, когда я избавился от всех методов, которые использовали делегирующие методы, я могу полностью отказаться от поля `_person`.

Глава 12

Крупномасштабные рефакторинги

Кент Бек и Мартин Фаулер

В предыдущих главах были показаны отдельные "ходы" рефакторинга. Но чего не хватает — так это представления об "игре" в целом. Рефакторинг проводится с какой-то целью, а не просто потому, что программисту нравится это слово или хочется на время приостановить разработку (по крайней мере, обычно рефакторинг проводится с некоторой конкретной целью). Так как же выглядит вся игра в целом?

Природа игры

В последующем изложении вы, несомненно, обратите внимание на далеко не столь точное и детальное описание конкретных шагов, как в предыдущих рефакторингах. Дело в том, что при проведении крупномасштабных рефакторингов ситуация кардинально меняется. Мы не можем точно сказать вам, что именно нужно делать, потому что не можем точно знать, что именно будет у вас перед глазами, когда вы приступите к работе. Если вы вводите в метод новый параметр, то техника этого действия очевидна, поскольку ясна область видимости. Когда же вы разбираетесь с запутанным наследованием, в каждом случае вы имеете свою ситуацию.

Кроме того, следует отдавать себе отчет в том, что описываемые здесь рефакторинги отнимают много времени. Любой рефакторинг из глав с 6 по 11 можно выполнить за считанные минуты, в крайнем случае — часы. Над некоторыми же крупными рефакторингами в действующих системах мы работали в течение месяцев или даже лет. Когда есть работающая система и нужно расширить ее функциональность, практически нереально убедить руководство, что надо остановиться на пару месяцев, пока вы будете приводить в порядок код. Вместо этого вы должны поступать, как Гансель и Гретель, откусывая по кусочку — сегодня здесь, завтра — в другом месте.

Выполняя такую работу, руководствуйтесь потребностью в каких-то дополнительных действиях. Выполняйте рефакторинги, когда нужно добавить функцию или исправить ошибки. Начав рефакторинг, не обязательно доводить его до конца. Делайте столько, сколько необходимо, чтобы выполнить реально стоящую перед вами задачу. Рефакторинг всегда можно продолжить на следующий день.

Эта философия отражена в приводимых примерах. Полная демонстрация любого приведенного далее рефакторинга легко может занять сотню страниц. Мы знаем это, так как Мартин уже пытался это сделать. Так что мы сжали примеры до нескольких схематичных диаграмм.

Из-за больших затрат времени крупные рефакторинги не приносят мгновенного удовлетворения полностью выполненной работой, как рефакторинги, описанные в других главах. Вам придется довольствоваться лишь верой, что с каждым днем вы делаете жизнь своей программы немного безопаснее.

Крупные рефакторинги требуют также согласия во всей команде программистов, которое не критично для маленьких рефакторингов. Масштабные рефакторинги определяют направление множества вносимых в код изменений. Вся команда должна понимать, что в игре находится один из крупномасштабных рефакторингов, и поступать соответственно. Вряд ли вы захотите оказаться в положении тех двух парней, машина которых застряла около вершины холма. Они вылезают и начинают толкать машину, каждый со своего конца. После безуспешных попыток сдвинуть ее тот, который впереди, говорит: "Никогда не думал, что столкнуть машину вниз так тяжело". На что второй отвечает: "Что ты имеешь в виду, говоря «вниз»?"

Важность крупномасштабных рефакторингов

Если в крупных рефакторингах не хватает множества тех качеств, которые составляют ценность маленьких рефакторингов (таких, как предсказуемость, заметный прогресс, немедленное вознаграждение), то почему мы решили включить их в эту книгу? Потому что без них вы рискуете потратить время и силы на изучение рефакторинга, а затем выполнить рефакторинг на практике — и не получить никаких преимуществ. Это было бы очень неприятно. Мы не хотим даже думать о таком варианте.

Если говорить серьезно, то рефакторингом занимаются не потому, что это весело, а потому, что после проведения рефакторинга вы надеетесь суметь сделать со своими программами то, чего без него вы просто не смогли бы сделать.

Накопление малопонятных проектных решений для программы подобно водорослям для заросшего ими канала. Благодаря рефакторингу вы гарантируете, что полное понимание того, как должна быть спроектирована программа, всегда находит в ней свое отражение. Как сорные водоросли проворно распространяют свои усики, так и не до конца понятые проектные решения быстро распространяют свое действие по всей программе. Ни одного, ни даже десяти отдельных конкретных действий не будет достаточно, чтобы устранить проблему.

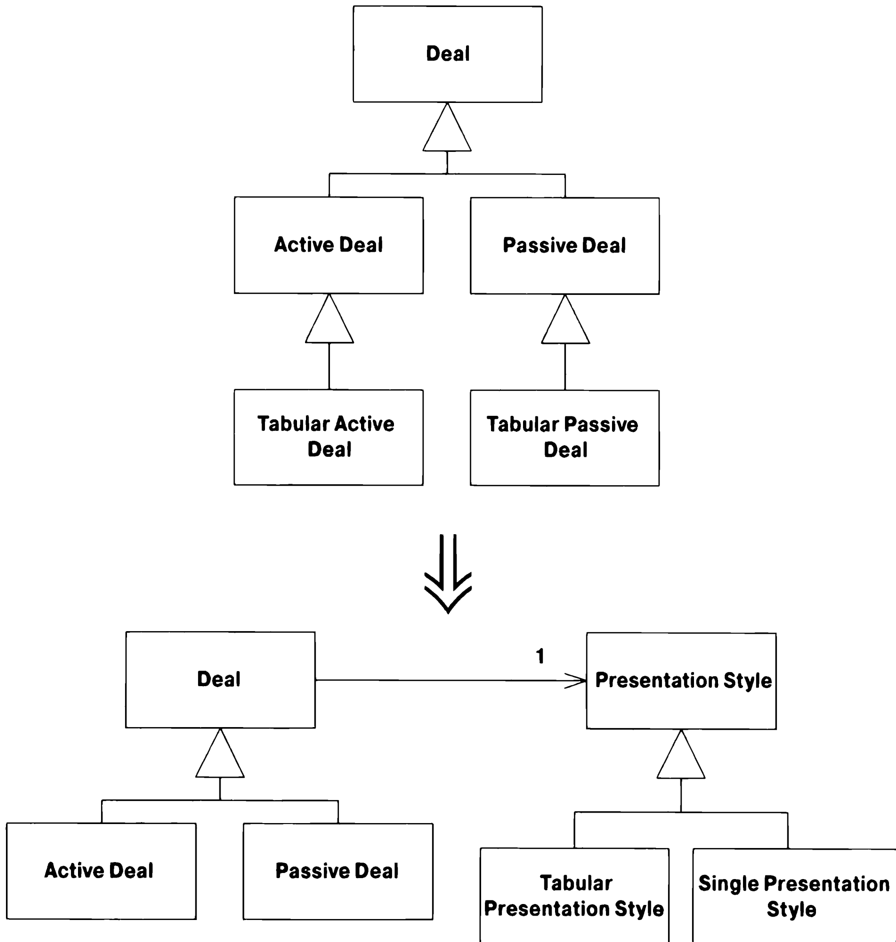
Четыре крупномасштабных рефакторинга

В этой главе мы описываем примеры четырех крупномасштабных рефакторингов. Это не попытки объять необъятное, а примеры, показывающие, о чем идет речь. Исследования и практика рефакторингов до сих пор были сосредоточены на небольших рефакторингах. Предстоящий разговор о крупномасштабных рефакторингах необычен и основан преимущественно на опыте Кента, у которого в крупных проектах его больше, чем у любого другого.

Рефакторинг “Разделение наследования” (с. 378) работает с запутанной иерархией наследования, в которой различные варианты кажутся скомбинированными вместе так, что это сбивает с толку. Рефакторинг “Преобразование процедурного проекта в объектный” (с. 383) помогает в решении классической проблемы — как поступать с процедурным кодом. Множество программистов использует объектно-ориентированные языки программирования без глубокого понимания объектов, поэтому данный вид рефакторинга достаточно распространен. Если вы увидите код, написанный с классическим двухуровневым подходом к пользовательским интерфейсам и базам данных, то вам может понадобиться рефакторинг “Отделение предметной области от представления” (с. 385), позволяющий отделить бизнес-логику от кода пользовательского интерфейса. Опытные разработчики в области объектно-ориентированных языков программирования понимают, что такое разделение жизненно важно для долго живущих и процветающих систем. Рефакторинг “Выделение иерархии” (с. 390) упрощает чрезмерно сложные классы, превращая их в группы подклассов.

Разделение наследования (Tease Apart Inheritance)

Имеется иерархия наследования, которая решает одновременно две задачи. *Создайте две иерархии и используйте делегирование для вызова одной из другой.*



Мотивация

Наследование — великая вещь. Оно помогает писать чрезвычайно сжатый код в подклассах. Отдельные методы могут приобретать значение, не пропорциональное своим размерам просто благодаря своему местоположению в иерархии.

Неудивительно, что такой мощный механизм часто применяется неправильно. А это неправильное употребление зачастую может постепенно накапливаться.

Сначала вы добавляете небольшой класс для решения небольшой задачи. На следующий день для решения той же задачи в других местах иерархии вводятся другие подклассы. И все — через неделю (месяц, год) вы уже плаваете в "макаронном коде". Без лодки и весел.

Запутанное наследование представляет собой источник неприятностей, потому что ведет к бичу программистов — дублированию кода. Оно усложняет внесение изменений в код, потому что стратегии решения проблемы определенного рода оказываются разбросанными по всей программе. Наконец, понимать получающийся в результате код существенно сложнее. Вы не можете просто сказать "Вот иерархия, она вычисляет результаты". Вы должны говорить "Да, она вычисляет результаты, а вот это подклассы для табличных версий, и у каждого из них есть подклассы для каждой из стран".

В таком коде зачастую можно обнаружить одну иерархию наследования, которая решает две задачи. Если у каждого подкласса на определенном уровне иерархии есть подклассы, имена которых начинаются с одинаковых прилагательных, то, вероятно, вы выполняете две работы с помощью одной иерархии.

Техника

- Определите различные задания, выполняемые иерархией. Создайте двумерную сетку (или трех-, четырехмерную, если у вас найдется такая замечательная бумага) и пометьте оси разными заданиями. Мы считаем, что для более чем двух измерений данный рефакторинг следует применять многократно (естественно, по одному измерению за раз).
- Определите, какая задача более важна и должна быть сохранена в текущей иерархии, а какую следует переместить в другую иерархию.
- Примените рефакторинг "Извлечение класса" (с. 169) к общему суперклассу, чтобы создать объект для вспомогательного задания, и добавьте переменную экземпляра для хранения этого объекта.
- Создайте подклассы извлеченного объекта для каждого из подклассов исходной иерархии. Инициализируйте переменную экземпляра, созданную на предыдущем шаге, экземпляром этого подкласса.
- Примените рефакторинг "Перенос метода" (с. 162) к каждому из этих подклассов для переноса поведения из подкласса в выделенный объект.
- Когда в подклассе не останется больше кода, удалите его.
- Продолжайте эту работу, пока не исчезнут все подклассы. Посмотрите, нельзя ли применить к новой иерархии дополнительные рефакторинги, например такие, как "Подъем поля" (с. 338) или "Подъем метода" (с. 339).

Примеры

Рассмотрим пример запутанной иерархии (рис. 12.1).

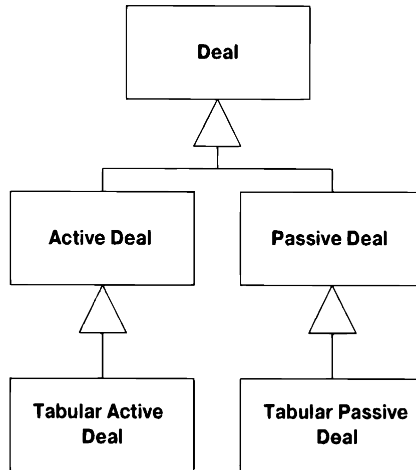


Рис. 12.1. *Запутанная иерархия*

Данная иерархия приобрела такой вид потому, что класс Deal первоначально использовался для вывода только одной сделки. Затем кому-то пришла в голову мысль выводить таблицу сделок. Поэкспериментировав со сделанным на скорую руку классом Active Deal, можно убедиться, что он без особого труда действительно способен отображать таблицу. Вы говорите, что нужна еще и таблица пассивных сделок? Без проблем, еще один небольшой подкласс, и все готово.

Двумя месяцами позже код таблицы становится сложнее, найти для него подходящее место непросто, время поджимает — словом, все как обычно. Теперь становится непросто добавить новый вид сделки — потому что теперь логика сделки тесно переплетена с логикой представления.

Следуя нашему рецепту, сначала надо определить задания, которые выполняет иерархия. Одно задание состоит в фиксации вариаций в соответствии с типом сделки. Второе — фиксировать вариации в соответствии со стилем представления. Поэтому наша сетка будет иметь следующий вид:

Deal	Active Deal	Passive Deal
Tabular Deal		

Следующий шаг состоит в принятии решения, какое из заданий важнее. Связь объекта со сделкой гораздо важнее стиля представления, поэтому мы оставим в покое Deal и извлечем стиль представления в собственную иерархию.

С практической точки зрения, пожалуй, следует оставлять на месте то задание, с которым связан больший объем кода (чтобы осуществлять меньше перемещений).

Затем мы должны применить рефакторинг “Извлечение класса” (с. 169) для создания стиля представления (рис. 12.2).

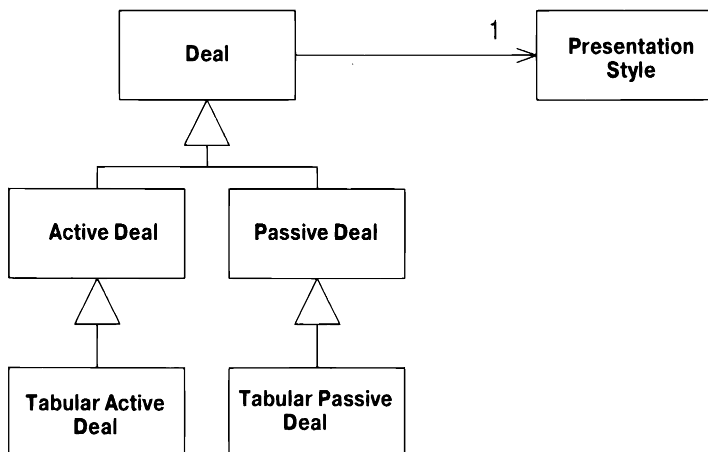


Рис. 12.2. *Добавление стиля представления*

Далее мы должны создать подклассы извлеченного класса для каждого из подклассов исходной иерархии (рис. 12.3) и инициализировать переменную экземпляра соответствующим подклассом:

```
ActiveDeal constructor
    ...presentation = new SingleActivePresentationStyle();...
```

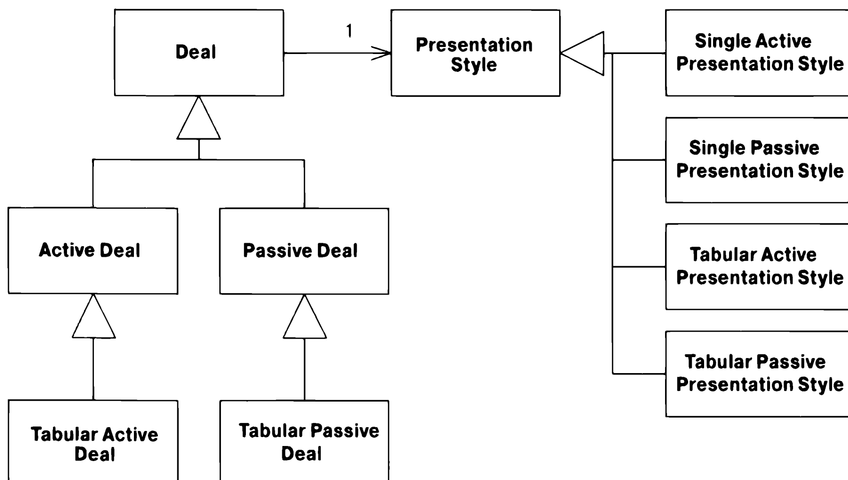


Рис. 12.3. *Добавление подклассов стиля представления*

Вы можете задать вопрос “Разве нам стало легче от того, что теперь у нас больше классов, чем было?” Что ж, иногда перед двумя шагами вперед приходится делать шаг назад. В случаях запутанной иерархии, как рассматриваемый здесь, иерархия извлеченного объекта почти всегда может быть существенно упрощена после извлечения. Однако более безопасно выполнять рефакторинг пошагово, чем перескочить сразу на десять шагов вперед к уже упрощенному проекту.

Теперь мы используем рефакторинги “Перенос метода” (с. 162) и “Перенос поля” (с. 166) для того, чтобы перенести методы и переменные, относящиеся к представлению, в подклассы стиля представления. У нас нет возможности хорошо показать этот процесс на приведенном примере, так что просто вообразите себе, как это происходит. По окончании работы в классах Tabular Active Deal и Tabular Passive Deal кода остаться не должно, поэтому мы их удаляем (рис. 12.4).

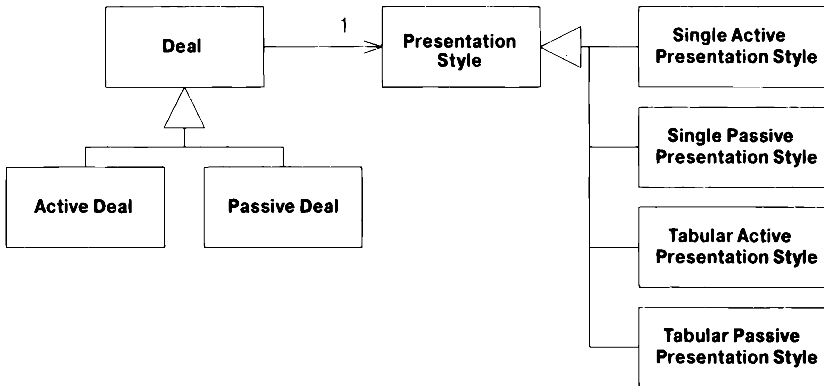


Рис. 12.4. *Табличные подклассы Deal удалены*

Теперь, когда мы разделили две задачи, можно отдельно поработать над упрощением каждой из них. Когда этот рефакторинг завершен, появляется возможность значительно упростить извлеченный класс, а часто и упростить исходный объект еще больше. На следующем шаге мы избавимся от разделения на активный/пассивный в стиле представления, показанном на рис. 12.5.

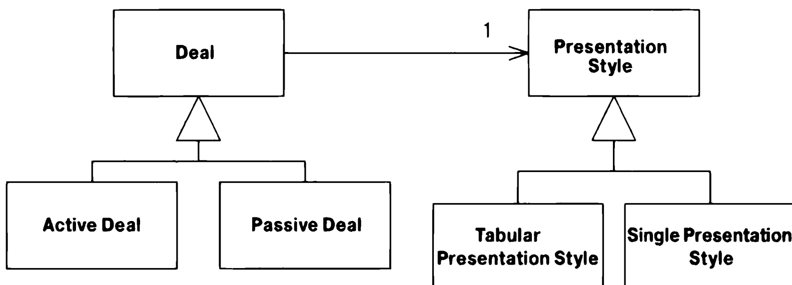


Рис. 12.5. *Теперь иерархии разделены*

Даже различие между одиночным и табличным представлениями может быть зафиксировано в значениях нескольких переменных. Подклассы становятся вообще ненужными (рис. 12.6).

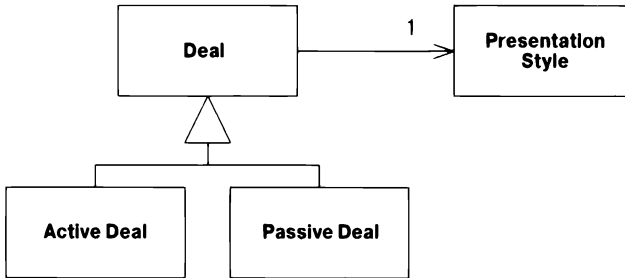
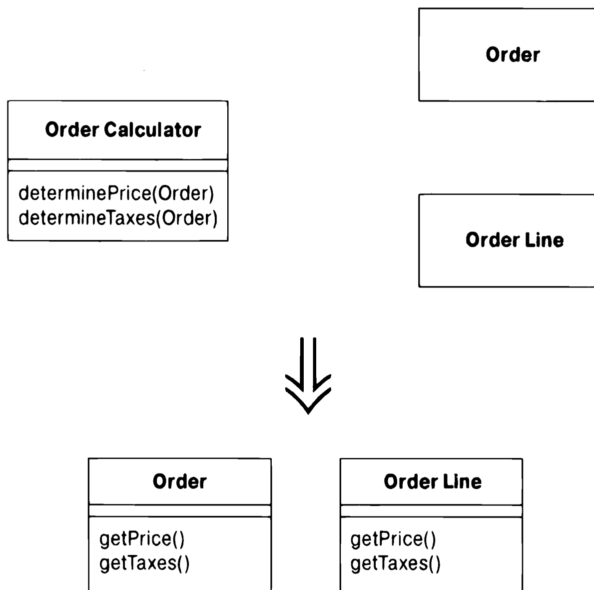


Рис. 12.6. *Различия в представлении могут обрабатываться парой переменных*

Преобразование процедурного проекта в объектный (Convert Procedural Design to Objects)

Имеется код, написанный в процедурном стиле.

Преобразуйте записи данных в объекты, отделите поведение и переместите его в объекты.



Мотивация

Однажды клиент потребовал от нас при запуске нового проекта, чтобы разработчики неукоснительно придерживались двух принципов: 1) использовали Java и 2) не использовали объекты.

Конечно, можно посмеяться, но, хотя Java и является объектно-ориентированным языком, применение объектов — это нечто большее, чем вызов конструктора. Требуется затратить время на обучение правильному применению объектов. Зачастую приходится сталкиваться с ситуацией, когда процедурно-ориентированный код надо сделать более объектно-ориентированным. Типичная ситуация — длинные процедурные методы в классе, хранящем мало данных, а также "немые" (dumb) объекты данных, в которых нет ничего, кроме методов доступа к значениям полей. Если требуется преобразование чисто процедурной программы, то может не оказаться и "немых" объектов, но это неплохая отправная точка.

Мы не утверждаем, что объектов с поведением и малым количеством данных (или полным их отсутствием) не должно быть вовсе. Мы сами часто пользуемся маленькими объектами стратегий, когда нам требуется изменить поведение. Однако такие процедурные объекты обычно невелики и применяются, когда возникает особая потребность в гибкости.

Техника

- Преобразуйте все типы записей и превратите их в "немые" объекты данных с методами доступа к значениям полей.

  ⇛ *При работе с реляционной базой данных преобразуйте каждую таблицу в немой объект данных.*

- Поместите весь процедурный код в один класс.

  ⇛ *Можно сделать такой класс синглтоном (для простоты реинициализации) или объявить методы статическими.*

- К каждой длинной процедуре примените рефакторинг "Извлечение метода" (с. 132) и сопутствующие ему, чтобы разбить эту процедуру на части. Разложив процедуры на части, примените рефакторинг "Перенос метода" (с. 162) для их переноса в соответствующий немой класс данных.

- Продолжайте выполнять описанные действия, пока из первоначального класса не будет удалено все поведение. Если исходный класс был чисто процедурным, его можно с большим удовольствием удалить.

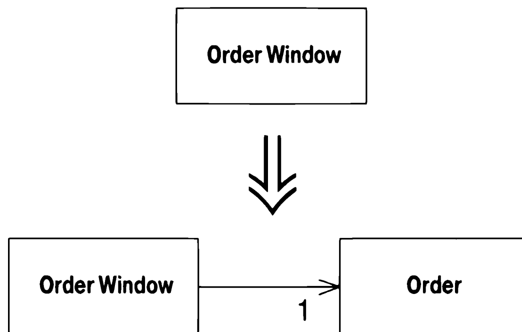
Пример

Пример, приведенный в главе 1, "Первый пример рефакторинга", хорошо иллюстрирует необходимость в данном рефакторинге, в особенности на первом этапе, когда выполняется разложение на части и распределение по классам метода `statement`. По завершении данной работы можно применить к ставшим теперь интеллектуальными объектам данных другие рефакторинги.

Отделение предметной области от представления (Separate Domain from Presentation)

Имеются классы графического интерфейса пользователя, содержащие логику предметной области.

Выделите логику предметной области в отдельные классы предметной области



Мотивация

Если послушать разговоры программистов об объектах, то можно услышать о "модели-представлении-контроллере" (model-view-controller — MVC). Эта идея послужила основой для связи между графическим интерфейсом пользователя и объектами предметной области в Smalltalk-80.

В основе идеи MVC лежит разделение кода пользовательского интерфейса (представления, называвшегося раньше *view*, а сейчас чаще *presentation*) и логики предметной области (модели — *model*). Классы представления содержат только ту логику, которая нужна для работы с интерфейсом пользователя. Объекты предметной области не содержат код визуализации, зато содержат всю бизнес-логику. В результате сложная программа разделяется на части, которые проще сопровождать и изменять. Кроме того, появляется возможность иметь несколько

представлений для одной и той же бизнес-логики. Те, кто имеют опыт работы с объектами, используют такое разделение инстинктивно, и такой подход доказал свою ценность.

Тем не менее многие работающие с графическим интерфейсом пользователя не придерживаются такого проекта. В большинстве сред с графическим интерфейсом пользователя "клиент/сервер" реализован двухуровневый подход: данные находятся в базе данных, а логика — в классах представления. Среда часто навязывает такой стиль проектирования, усложняя перемещение логики в другое место.

Java представляет собой корректную объектно-ориентированную среду, так что вы можете создавать невизуальные объекты предметной области, содержащие бизнес-логику. Однако зачастую можно столкнуться с кодом, написанным в двухуровневом стиле.

Техника

- Создайте для каждого окна класс предметной области.
- Если у вас имеется сетка, создайте класс, представляющий строки этой сетки. Используйте для окна коллекцию класса предметной области, которая будет хранить соответствующие объекты строки.
- Изучите данные в окне. Если они используются только для графического интерфейса пользователя, оставьте их в окне. Если они задействованы в логике предметной области, но фактически не отображаются в окне, примените рефакторинг "Перенос метода" (с. 162), чтобы перенести их в объект предметной области. Если же они используются как графическим интерфейсом пользователя, так и логикой предметной области, воспользуйтесь рефакторингом "Дублирование видимых данных" (с. 207), чтобы они находились в обоих местах и были синхронизированы.
- Изучите логику класса представления. Воспользуйтесь рефакторингом "Извлечение метода" (с. 132) для отделения логики представления от логики предметной области. Отделив логику предметной области, примените рефакторинг "Перенос метода" (с. 162) для ее переноса в объект предметной области.
- По окончании работы у вас будут классы представления, обрабатывающие графический интерфейс пользователя, и объекты предметной области, содержащие всю бизнес-логику. Объекты предметной области будут недостаточно структурированы, но эту проблему можно решить с помощью дальнейших рефакторингов.

Пример

Пусть имеется программа, позволяющая пользователям вводить информацию о заказах и получать их стоимость. Ее графический интерфейс пользователя показан на рис. 12.7. Класс представления взаимодействует с реляционной базой данных, структура которой показана на рис. 12.8.

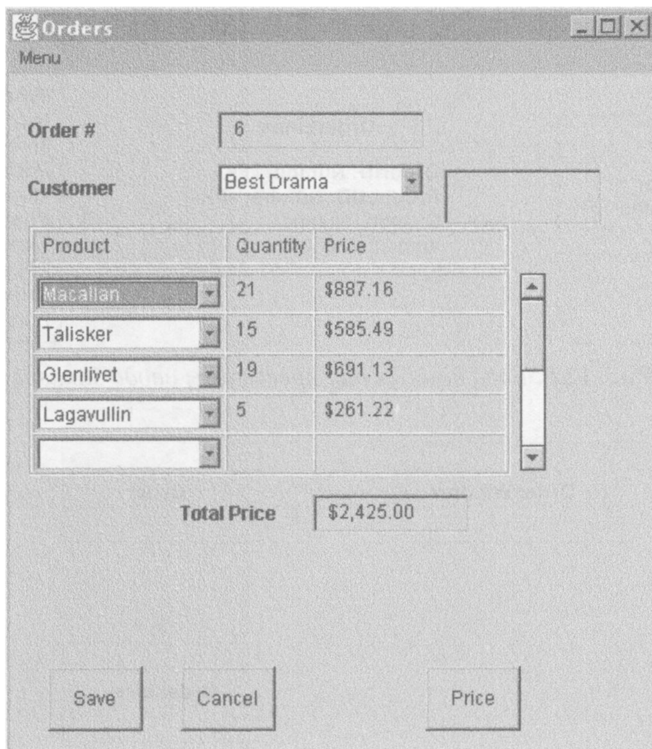


Рис. 12.7. *Интерфейс пользователя исходной программы*

Все поведение, как графического интерфейса пользователя, так и определения стоимости заказа, находится в одном классе, Order Window.

Начнем с создания подходящего класса заказа. Затем мы свяжем его с окном заказа, как показано на рис. 12.9. Поскольку в окне есть сетка для вывода строк заказа, мы также создадим класс строки заказа (order line) для строк сетки.

Мы начинаем работу с окна, а не с базы данных. Основывать первоначальную модель предметной области на базе данных — разумная стратегия, но наибольший риск заключается в смешении логики представления с логикой предметной области, поэтому мы разделяем их на основе окна, а остальное подвергнем рефакторингу позднее.

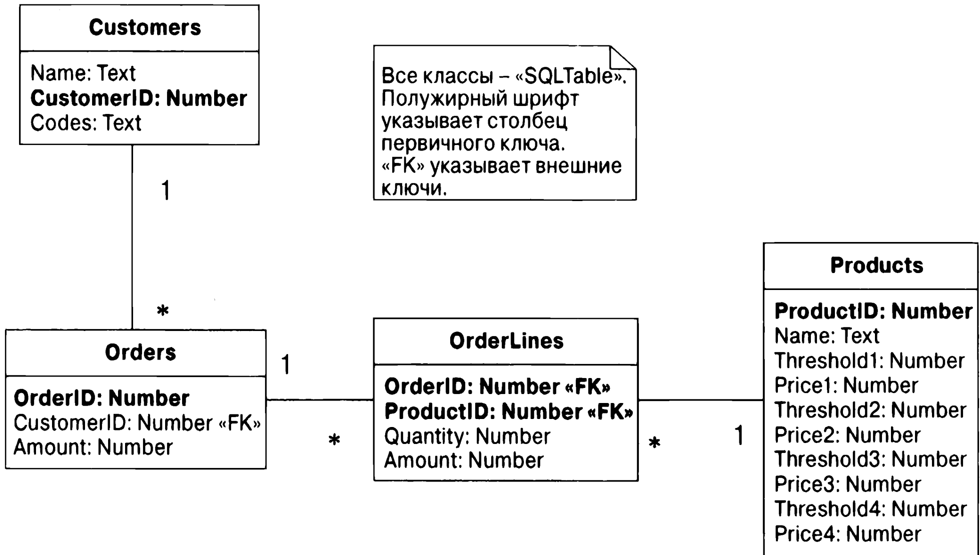


Рис. 12.8. *База данных для программы ввода заказов*

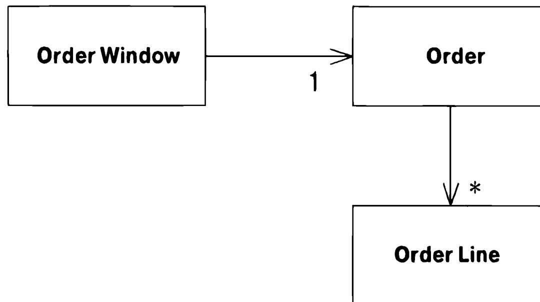


Рис. 12.9. *Классы окна заказа и заказа*

В таких программах полезно рассмотреть SQL-запросы, внедренные в окно. Данные, получаемые запросами SQL, являются данными предметной области.

Проще всего разобраться в данных предметной области, непосредственно в графическом интерфейсе пользователя не отображаемых. В рассматриваемом примере в базе данных есть поле кодов в таблице пользователей. Эти коды непосредственно в графическом интерфейсе пользователя не отображаются; они преобразуется в более удобочитаемый текст. Такое поле представляет собой простой класс, например такой, как строка, а не компонент AWT. Чтобы перенести это поле в класс предметной области, можно применить рефакторинг “Перенос поля” (с. 166).

С другими полями нам повезло меньше. Они содержат компоненты AWT, которые отображаются в окне и используются в объектах предметной области. К ним следует применить рефакторинг “Дублирование видимых данных” (с. 207). Это поместит поле предметной области в класс заказа, а соответствующее поле AWT — в окно заказа.

Это медленный процесс, но в итоге таким образом можно поместить все поля, связанные с логикой предметной области, в класс предметной области. Неплохим решением будет попытка переместить все запросы SQL в класс предметной области. Это можно сделать для того, чтобы переместить как логику базы данных, так и данные предметной области в класс предметной области. После удаления импорта `java.sql` из окна заказа вы ощутите приятное чувство завершенности данной работы. Но до этого надо выполнить немало рефакторингов “Извлечение метода” (с. 132) и “Перенос метода” (с. 162).

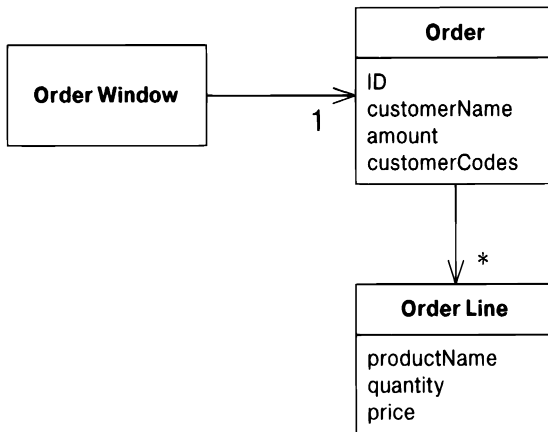


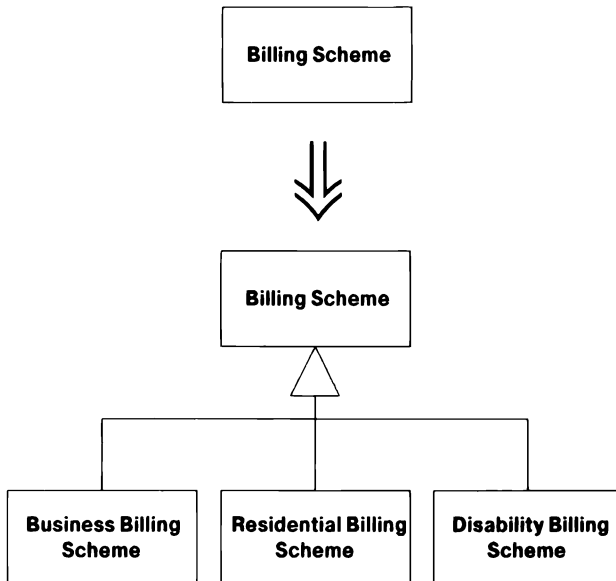
Рис. 12.10. *Распределение данных по классам предметной области*

Полученные классы (показанные на рис. 12.10) далеки от идеала, но уже в этой модели логика предметной области полностью отделена. При проведении данного рефакторинга надо внимательно следить за возможными опасностями. Если наибольшую угрозу представляет смешение логики представления и логики предметной области, полностью разделите их, прежде чем делать что-либо еще. Если важнее другие вещи — например, стратегия получения цены товара, — уберите логику этой важной части из окна и займитесь ее реорганизацией, чтобы получить структуру, пригодную для применения в области высокого риска. Весьма вероятно, что из окна заказа потребуется убрать всю логику предметной области. Если можно выполнить рефакторинг и оставить часть логики в окне, поступайте таким образом, чтобы сначала решить проблему с наивысшим уровнем риска.

Выделение иерархии (Extract Hierarchy)

Имеется класс, выполняющий слишком большую работу, как минимум частично реализованную с помощью многочисленных условных инструкций.

Создайте иерархию классов, в которой каждый подкласс представляет частный случай.



Мотивация

При эволюционном проектировании часто некоторый класс рассматривается как реализация некоторой идеи, но впоследствии оказывается, что на самом деле он реализует две или три идеи, а то и целый десяток. Вначале вы создаете простой класс. Через пару дней или недель вы замечаете, что если добавить в него один флаг и пару проверок, то можно использовать его еще в одном месте. Через месяц обнаруживается еще одна такая возможность. Через год наступает полная неразбериха: всюду раскиданы флаги и условные инструкции.

Столкнувшись с таким "швейцарским армейским ножом", который умеет открывать консервные банки, подрезать деревья, работать в качестве лазерной указки на презентациях (ах да, надо надеяться, что резать он тоже что-то может), нужно найти стратегию, позволяющую распутать этот клубок. Описанная в данном разделе стратегия работает только при условии, что условная логика класса является статичной в течение всего срока жизни объекта. В противном случае,

прежде чем начать отделять одни случаи от других, может потребоваться провести рефакторинг “Извлечение класса” (с. 169).

Пусть вас не пугает то, что данный рефакторинг нельзя завершить за день. Для распутывания действительно сложного запутанного проекта могут потребоваться недели или месяцы. Сделайте то, что просто и очевидно, а затем на некоторое время прервитесь. Займитесь другой работой, приносящей видимый результат. Научившись чему-то, вернитесь к рефакторингу и выполните еще несколько простых и очевидных действий.

Техника

Мы приводим здесь два набора технических правил. В первом случае вы не знаете, с какими вариантами вы можете встретиться. В таком случае надо действовать пошагово, как описано ниже.

- Определите один из вариантов.

  ⇛ *Если варианты могут меняться в течение срока жизни объекта, примените рефакторинг “Извлечение класса” (с. 169) для выделения данного аспекта в отдельный класс.*

- Создайте подкласс для этого частного случая и примените к оригиналу рефакторинг “Замена конструктора фабричным методом” (с. 321). Модифицируйте фабричный метод так, чтобы он возвращал экземпляр подкласса, где это уместно.
- Поочередно скопируйте в подкласс методы, содержащие условную логику, а затем упростите их, исходя из того, что можно сказать об экземплярах подкласса, но не об экземплярах суперкласса.

  ⇛ *При необходимости примените к суперклассу рефакторинг “Извлечение метода” (с. 132) для разделения условных и безусловных частей методов.*

- Продолжайте выделение частных случаев до тех пор, пока не сможете объявить суперкласс абстрактным.
- Удалите тела перегруженных во всех подклассах методов суперкласса и сделайте соответствующие объявления в суперклассе абстрактными.

Если же варианты совершенно ясны с самого начала, можно воспользоваться другой стратегией.

- Создайте подкласс для каждого варианта.
- Примените рефакторинг “Замена конструктора фабричным методом” (с. 321) для получения фабричного метода, который должен возвращать соответствующий подкласс для каждого варианта.

⇛ *Если варианты помечены кодом типа, примените рефакторинг “Замена кода типа подклассами” (с. 241). Если варианты могут меняться в течение времени жизни класса, воспользуйтесь рефакторингом “Замена кода типа состоянием/стратегией” (с. 245).*

- Примените к методам, содержащим условную логику, рефакторинг “Замена условной инструкции полиморфизмом” (с. 271). Если метод в целом не изменяется, отделите варьируемую часть с помощью рефакторинга “Извлечение метода” (с. 132).

Пример

Пример представляет собой неочевидный случай. Чтобы посмотреть, как действовать в очевидном случае, просто проследите выполнение рефакторингов “Замена кода типа подклассами” (с. 241), “Замена кода типа состоянием/стратегией” (с. 245) и “Замена условной инструкции полиморфизмом”(с. 271).

Начнем с программы, которая выписывает счета за электричество. Исходные объекты показаны на рис. 12.11.

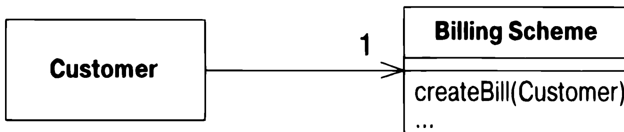


Рис. 12.11. *Схема выписки счетов клиентам*

Схема выписки счетов содержит много условной логики для действий в различных обстоятельствах. Зимой и летом применяются разные тарифы, различные схемы начисления используются для жилых домов, для малых предприятий, для лиц, получающих социальную страховку, и для инвалидов. Итоговая логика делает класс Billing Scheme достаточно сложным.

На первом шаге мы выберем тот аспект, который постоянно проявляется в условной логике. Это могут быть различные условия, зависящие от того, распространяется ли на клиента схема оплаты для инвалидов. Это может быть флаг в Customer, Billing Scheme или где-либо еще.

Создадим подкласс для этого варианта. Чтобы можно было пользоваться этим подклассом, следует обеспечить его создание и применение. Так что мы смотрим на конструктор класса Billing Scheme. Сначала мы применяем рефакторинг “Замена конструктора фабричным методом” (с. 321), а потом смотрим на полученный фабричный метод и выясняем, как логика зависит от инвалидности. После этого мы создаем конструкцию, которая при необходимости возвращает схему начисления оплаты для инвалидов.

Затем мы рассматриваем разные методы в Billing Scheme и ищем те из них, в которых имеется условная логика, зависящая от инвалидности. Одним из таких методов является CreateBill, поэтому мы копируем его в подкласс (рис. 12.12).

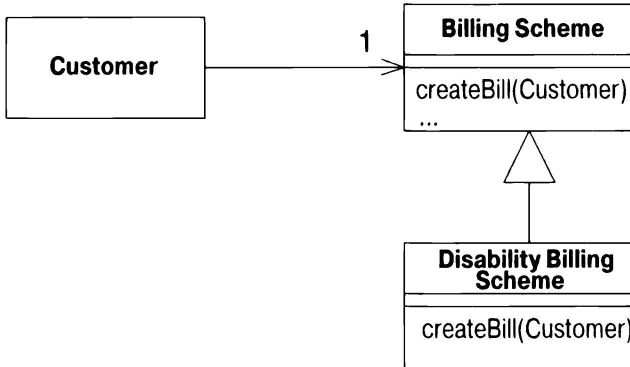


Рис. 12.12. *Добавление подкласса для инвалидов*

Теперь мы изучаем копию createBill в подклассе и упрощаем его, поскольку знаем, что теперь он находится в контексте схемы начисления оплаты для инвалидов. Таким образом, код вида

```
if (disabilityScheme()) doSomething
```

можно заменить кодом

```
doSomething
```

Если инвалидность исключается из схемы для предприятий, можно удалить условно выполняемый код в схеме для предприятий.

Сделав это, мы хотим обеспечить отделение варьируемого кода от остающегося неизменным. Для этого мы применяем рефакторинги "Извлечение метода" (с. 132) и "Декомпозиция условного оператора" (с. 256). Мы продолжаем выполнять эти действия с различными методами Billing Scheme до тех пор, пока не почувствуем, что справились с большинством условных инструкций, связанных с инвалидностью. Потом выбираем другой вариант, например социальное страхование, и то же самое проделываем с ним.

При работе над вторым вариантом мы сравниваем варианты для социального страхования с вариантами для инвалидности. Желательно выявить методы, имеющие одинаковый смысл, но выполняемые в каждом случае по-разному. В этих двух случаях таковыми могут быть варианты начисления налогов. Мы хотим гарантировать, что эти два метода в подклассах имеют одинаковые сигнатуры.

Это может означать изменение класса для инвалидов, чтобы привести в соответствие два подкласса. Обычно выясняется, что чем с большим количеством вариантов мы работаем, тем больше стабилизируются шаблоны аналогичных и различающихся методов. Это существенно облегчает работу по дополнению вариантов.

Глава 13

Рефакторинг, повторное использование и реальность

Уильям Опдайк

С Мартином Фаулером я познакомился в Ванкувере во время конференции OOPSLA 92. Несколькими месяцами ранее я закончил работу над диссертацией по рефакторингу объектно-ориентированных каркасов [1][1] в Университете штата Иллинойс. Изучая возможности продолжения своих исследований в области рефакторинга, я рассматривал и другие варианты, например работу в медицинской информатике. Мартин в то время работал над приложением в области медицинской информатики, что обеспечило нас темой для разговора за завтраком в Ванкувере. Как упоминается в начале книги, мы в течение некоторого времени обсуждали мои исследования в области рефакторинга. Тогда интерес Мартина к данной теме был невелик, но, как вы теперь знаете, с того времени он значительно возрос.

На первый взгляд, может показаться, что рефакторинг возник в академических исследовательских лабораториях. Но в действительности он возник в практике разработки программного обеспечения, когда программисты, применяющие объектно-ориентированный подход и использовавшие в то время язык программирования Smalltalk, столкнулись с потребностью в технологиях, обеспечивающих улучшенную поддержку процесса разработки, или, говоря более общо, улучшенную поддержку процесса внесения изменений в код. Это породило ряд исследований, которые вскоре дошли до того уровня, когда их результатами мог воспользоваться широкий круг профессиональных программистов.

[1] В данной главе используются отдельные ссылки на библиографические источники; их список приведен в конце главы.

Когда Мартин предложил мне написать главу для этой книги, у меня сразу же возник ряд мыслей. Я мог бы описать ранние исследования рефакторинга — те времена, когда Ральф Джонсон (Ralph Johnson) и я, имея совершенно разную техническую подготовку, объединились, чтобы заняться вопросами поддержки процесса модификации объектно-ориентированного программного обеспечения. Я мог бы обсудить вопросы создания инструментария для автоматизации рефакторинга — что и составляет область моих исследований, весьма далекую от тематики этой книги. Я мог бы поделиться некоторыми полученными мною уроками, касающимися отношений между рефакторингом и повседневными проблемами разработчиков программного обеспечения, в особенности работающих в промышленности над большими проектами. Понимание, достигнутое мной в процессе исследования рефакторинга, часто приносило пользу в достаточно широком диапазоне областей — при оценке программных технологий и формулировании стратегий развития продуктов, в разработке прототипов и программ в области телекоммуникаций и при обучении и консультировании групп разработчиков программного обеспечения.

Я решил бегло обратиться ко всем вопросам, имеющим отношение к этим областям. Как вытекает из названия главы, глубокое понимание рефакторинга часто имеет широкое применение в таких областях, как повторное использование программного обеспечения, развитие продукта и выбор платформы. Хотя отдельные части этой главы кратко касаются некоторых более интересных теоретических аспектов рефакторинга, основное внимание в ней уделено практическим проблемам реального мира и способам их решения.

Если вы хотите более глубоко изучить рефакторинг, обратитесь к разделу "Ресурсы и ссылки, относящиеся к рефакторингу" далее в этой главе.

Проверка в реальных условиях

Перед тем как заняться своими исследованиями, я несколько лет проработал в Bell Labs. Большую часть этого времени я провел в подразделении компании, которое разрабатывало электронные системы коммутации. К таким продуктам предъявляются очень высокие требования — как в отношении надежности, так и в плане скорости обработки телефонных звонков. В разработку и развитие таких систем были вложены тысячи человеко-лет, а срок жизни созданных программных систем достигал десятилетий. В результате большая часть стоимости разработки этих систем была обусловлена не разработкой первоначального изделия, а постоянной адаптацией и модификацией систем. Средства упрощения и удешевления внесения таких модификаций принесли бы компании большую прибыль.

Поскольку мои диссертационные исследования финансировала Bell Labs, я хотел выбрать область не только интересную в техническом отношении, но и связанную с практическими потребностями фирмы. В конце 1980-х годов объектно-ориентированные технологии только начали выходить из исследовательских лабораторий. Когда Ральф Джонсон (Ralph Johnson) предложил мне тему исследований, которая охватывала как объектно-ориентированные технологии, так и поддержку процесса модификации и эволюции программного обеспечения, я тут же ухватился за нее.

Мне говорили, что когда люди заканчивают работу над диссертацией, то редко бесстрастно относятся к выбранной теме. Одни, устав от нее, быстро переключаются на что-то иное; другие же сохраняют в отношении этой темы прежний энтузиазм. Я оказался в последнем лагере.

Когда я вернулся в Bell Labs после получения ученой степени, произошла странная вещь. Тех, с кем я работал, рефакторинг волновал гораздо меньше, чем меня.

Я хорошо помню свое выступление в начале 1993 года на конференции по технологиям для сотрудников AT&T Bell Labs и NCR (в то время мы все входили в одну компанию). Мне было выделено 45 минут для сообщения о рефакторинге. Сначала доклад проходил успешно. Мой энтузиазм в отношении темы передался слушателям. Тем не менее по окончании доклада было задано очень мало вопросов. Один из участников подошел ко мне позднее с дополнительными вопросами; ему надо было писать дипломную работу, и он искал тему для исследований. Я же надеялся на то, что программисты, работающие над крупными проектами, проявят желание применить рефакторинг в своей работе. Но если у них и возникло такое желание, в тот момент они никак его не проявили.

По-моему, они просто не поняли, с чем имеют дело.

Ральф Джонсон (Ralph Johnson) преподал мне важный урок, касающийся исследований: когда кто-то (рецензент статьи, участник конференции) утверждает, что он не понимает, в чем дело, — это *наша* вина. *Наша* обязанность — постараться развить и донести до него свои идеи.

В течение нескольких последующих лет у меня были многочисленные возможности рассказать о рефакторинге как на внутренних совещаниях в AT&T Bell Labs, так и на конференциях и семинарах в других местах. По ходу своих бесед с программистами-практиками я начал понимать, почему мои прежние обращения к слушателям не были ими восприняты. Отсутствие понимания частично было вызвано новизной объектно-ориентированных технологий. Лишь немногие из тех, кто работал с ними, продвинулись сколь-нибудь глубоко в эту область и потому просто еще не сталкивались с серьезными проблемами, которые помогает решить рефакторинг. Это типичная дилемма исследователя — когда современные достижения науки опережают практику. Однако была еще одна тревожная

причина отсутствия контакта с программистами. По ряду причин разработчики, основываясь на здравом смысле, *не желали* применять рефакторинг к своим программам, даже осознавая выгоды, которые он сулит. Чтобы рефакторинг был принят сообществом разработчиков, необходимо было решить эти проблемы.

Почему разработчики не хотят применять рефакторинг к своим программам

Представьте, что вы — разработчик программного обеспечения. Если ваш проект начинается "с нуля" (т.е. у вас нет проблемы совместимости с предыдущими версиями) *и* вам понятна задача, для решения которой предназначена система, *и* тот, кто финансирует ваш проект, готов поддерживать его, пока *вы* не будете удовлетворены результатами, то вам крупно повезло. Это идеальный сценарий применения объектно-ориентированных технологий, но большинство из нас может о нем только мечтать.

Куда чаще от вас требуют расширить возможности уже имеющегося программного обеспечения. При этом у вас далеко не полное понимание того, что вы делаете. Вы поставлены в жесткие временн*ы*е рамки. Что же можно предпринять в такой ситуации?

Можно переписать программу. Вы примените весь свой опыт проектировщика, исправите все прежние огрехи, будете работать творчески и с удовольствием. Но кто оплатит эти расходы? И можно ли быть уверенным, что новая программа сможет делать все то, что делала старая?

Можно скопировать и модифицировать некоторые части существующей системы, чтобы расширить ее возможности. Это может показаться оправданным и даже будет рассматриваться как демонстрация повторного использования кода. При этом вам даже не нужно разбираться в том, что именно вы используете повторно. Однако с течением времени происходит размножение ошибок, программы "разбухают", их архитектура разрушается и нарастают дополнительные издержки на внесение изменений.

Рефакторинг находится посередине между этими двумя крайностями. Он дает возможность изменить имеющееся программное обеспечение, сделав более ясным понимание конструкции, развить каркасы и извлечь повторно используемые компоненты, сделать архитектуру программы более понятной, а также создать условия, облегчающие добавление новой функциональности. Рефакторинг может способствовать извлечению выгоды из сделанных ранее инвестиций, уменьшить дублирование и рационализировать программу.

Допустим, что для вас как для разработчика эти преимущества очень привлекательны. Вы согласны с Фредом Бруксом (FredBrooks) [2] в том, что модернизация является "внутренней неотъемлемой сложностью" разработки программного обеспечения. Вы согласны, что теоретически рефакторинг дает указанные преимущества.

Почему же, тем не менее, вы не применяете рефакторинг к *своим* программам? На то есть четыре возможные причины.

1. Вы можете не понимать, как правильно выполнять рефакторинг.
2. Если выгоды ощутимы лишь в далекой перспективе, вы можете не захотеть тратить силы здесь и сейчас. Ведь в долгосрочной перспективе, когда придет время пожинать плоды, вы можете оказаться вне проекта.
3. Рефакторинг кода является непроизводительной деятельностью, а вам платят только за *новую* функциональность.
4. Рефакторинг может нарушить работу уже имеющейся работоспособной программы.

Все это — обычные причины для беспокойства. Мне приходилось выслушивать их от сотрудников коммуникационных и высокотехнологических компаний, как программистов, так и администраторов. Все эти причины должны быть тщательно рассмотрены, прежде чем разработчики придут к решению о применении рефакторинга к своему программному обеспечению. Давайте поочередно рассмотрим каждую из них.

Как и когда применять рефакторинг

Как можно научиться рефакторингу? Каковы его инструменты и технологии? В каких сочетаниях они могут принести пользу? Когда нужно их применять? В этой книге описано несколько десятков рефакторингов, которыми Мартин пользуется в своей работе, и приведены примеры применения рефакторингов для внесения важных изменений в программы.

В проекте Software Refactory, осуществлявшемся в Университете штата Иллинойс, мы выбрали минималистский подход. Мы определили небольшой набор рефакторингов [1,3] и показали, как следует их применять. Отбор рефакторингов был основан на нашем личном опыте программирования. Мы рассмотрели эволюцию структур нескольких объектно-ориентированных каркасов, в основном связанных с C++, и поговорили с несколькими опытными разработчиками на языке программирования Smalltalk, а также прочитали их обзоры. Большинство наших

рефакторингов относится к низкому уровню, например они создают или удаляют класс, переменную или функцию, изменяют атрибуты переменных и функций (например, такие как права доступа `public` или `protected`) и аргументы функций, либо перемещают переменные и функции между классами. Небольшая группа рефакторингов более высокого уровня используется для таких операций, как создание абстрактного суперкласса, упрощение класса посредством создания подклассов или упрощение условных инструкций, либо выделение части существующего класса с созданием нового класса для повторно используемого компонента (часто с взаимным преобразованием наследования в делегирование или агрегирование и наоборот). Более сложные рефакторинги определяются на языке рефакторингов низкого уровня. Наш подход зиждился на соображениях поддержки автоматизации и надежности, о чем будет сказано ниже.

Какие виды рефакторинга следует применять к имеющейся работающей программе? Это зависит от того, какие задачи вы ставите перед собой. Одним из распространенных оснований для проведения рефакторинга, которое постоянно находится в центре внимания этой книги, — облегчение добавления новой функциональности (краткосрочная цель). Этот вопрос обсуждается в следующем разделе. Однако имеются и другие основания для проведения рефакторинга.

Опытные программисты, использующие объектно-ориентированный подход, и программисты, имеющие подготовку в области шаблонов и технологий проектирования, знают, что имеется ряд желательных структурных характеристик и свойств программ, необходимых для поддержки расширяемости и повторного использования кода [4–6]. Такие объектно-ориентированные технологии проектирования, как CRC [7], сосредоточены на определении классов и их протоколов. И пусть основной упор делается на предварительное проектирование, имеющиеся программы также можно оценивать исходя из их соответствия этим руководящим принципам.

Для выявления структурных недостатков программ, таких как функции с чрезмерно большим количеством аргументов или слишком длинные функции, можно использовать автоматизированный инструментарий. Такие функции являются кандидатами на проведение рефакторинга. С помощью автоматизированных средства можно также выявлять структурное сходство, служащее признаком избыточности. Например, если есть две очень похожие функции (что случается, когда одна функция создается путем копирования другой с последующим внесением изменений), то это сходство может быть обнаружено подобным инструментарием, и могут быть предложены рефакторинги, перемещающие общий код в одно место. Если в разных местах программы есть две переменные с одинаковыми

именами, то иногда можно заменить их одной переменной, которая наследуется в обоих местах. Это всего лишь несколько простейших примеров. С помощью автоматического инструментария можно выявить и исправить много других, более сложных случаев. Такие структурные аномалии или структурные сходства часто (пусть и не всегда) означают, что данному коду не помешало бы применение рефакторинга.

Значительная часть работы над проектными шаблонами сосредоточена на хорошем стиле программирования и полезных шаблонах взаимодействия между частями программы, которые могут быть отображены на структурные характеристики и в рефакторинг. Например, раздел применимости шаблонного метода [8] ссылается на наш рефакторинг абстрактного суперкласса [9].

В [1] я перечислил некоторые эвристики, которые могут помочь выделить кандидатов на рефакторинг в программе, написанной на C++. Джон Брант и Дон Робертс [10, 11] создали инструментарий, применяющий обширный набор эвристик для автоматического анализа программ на языке программирования Smalltalk. Их средства предлагают выполнение рефакторингов, способных улучшить архитектуру программы, и указывают, где именно их применять.

Применение такого инструмента для анализа программы отчасти аналогично применению инструмента lint к программе на C или C++. Это не настолько интеллектуальный инструмент, чтобы он мог понимать смысл программы. Выполнения достойны лишь очень немногие из предложений, которые он выдвигает на основе структурного анализа программы. Вы как программист сохраняете право выбора. Решать, какие рекомендации нужно применить к своей программе, а от каких отказаться, должны вы сами. Вносимые изменения должны улучшить структуру программы и способствовать облегчению внесения предстоящих модификаций.

Прежде чем программисты убедятся в том, что их код следует подвергнуть рефакторингу, они должны понять, как и где следует применять рефакторинг. Ничто не в состоянии заменить практический опыт. Мы применяли в своем исследовании знания опытных программистов, применяющих объектно-ориентированный подход, чтобы получить набор полезных рефакторингов и понимание, где именно их следует применять. Автоматизированные инструменты могут анализировать структуру программы и предлагать рефакторинги, способные улучшить эту структуру. Как и в большинстве дисциплин, инструменты и методики могут быть полезными, *но только если вы их используете*. По мере того как программисты выполняют рефакторинг своего кода, растет их понимание самого процесса рефакторинга.

Рефакторинг программ на языке программирования C++

Когда мы с Ральфом Джонсоном (Ralph Johnson) начали изучать рефакторинг в 1989 году, язык программирования C++ вовсю развивался и становился очень популярным в сообществе объектно-ориентированных разработчиков. Впервые важность рефакторинга была понята в сообществе программистов на языке программирования Smalltalk. Мы почувствовали, что демонстрация применимости рефакторинга к программам C++ должна заинтересовать более широкое сообщество объектно-ориентированных разработчиков.

В C++ есть ряд свойств, в особенности статическая проверка типов, которые упрощают ряд задач анализа программ и рефакторинга. С другой стороны, C++ — весьма сложный язык, в значительной мере из-за своей истории и происхождения от языка программирования C. Некоторые стили программирования, допускаемые в C++, затрудняют рефакторинг и развитие программ, написанных на нем.

Особенности языков и стилей программирования, способствующие рефакторингу

Имеющиеся в C возможности статического анализа типов позволяют относительно легко сузить возможные ссылки на часть программы, к которой желательно применить рефакторинг. Возьмем простой, но распространенный случай, когда требуется переименовать функцию-член класса C++. Чтобы правильно осуществить переименование, нужно изменить объявление функции и все обращения к ней. Такой поиск и замена обращений могут быть сложным заданием в программе большого размера.

По сравнению со Smalltalk в C++ есть такие особенности наследования классов и управления доступом (`public`, `protected`, `private`), которые облегчают выявление обращений к переименовываемой функции. Если такая функция объявлена в классе как закрытая, то обращения к ней могут выполняться только в самом этом классе или классах, объявленных дружественными к нему. Если функция объявлена защищенной, то обращения к ней можно найти только в самом классе, его подклассах (и их потомках) и среди дружественных классов. Если же функция объявлена открытой (наиболее слабый режим доступа), то анализ все равно ограничен списком классов, действующим для защищенных функций, и операциями над экземплярами класса, который содержит функцию, его подклассами и их потомками.

В некоторых больших программах в разных местах могут быть объявлены функции с одинаковыми именами. В ряде случаев две или более одноименные функции лучше заменить одной; для таких изменений есть часто применяемые рефакторинги. С другой стороны, иногда нужно переименовать только одну из функций, а другую сохранить. В проекте, над которым работает несколько человек, несколько программистов могут дать одно и то же имя разным функциям, никак не связанным между собой. В C++ при изменении имени одной из таких функций почти всегда легко определить, какие обращения выполняются

к переименовываемой функции, а какие — к другой. Выполнить такой анализ в Smalltalk куда сложнее.

В C++ для реализации подтипов используются подклассы, поэтому область видимости переменной или функции обычно можно расширить или сузить, перемещая их по иерархии наследования. Действия по анализу программы и рефакторингу довольно просты.

Несколько хороших принципов программирования, примененных в начале проектирования и в течение всего процесса разработки, облегчают выполнение рефакторинга и упрощают развитие программного обеспечения. Определение данных-членов и большинства функций-членов как закрытых или защищенных представляет собой способ абстрагирования, часто упрощающий рефакторинг внутреннего устройства класса, и минимизирует необходимость изменений в других частях программы. Использование наследования для моделирования иерархий обобщения и специализации (естественных для C++) позволяет в будущем довольно легко расширять или сужать область действия переменных-членов или функций-членов путем рефакторинга, перемещающего их внутри иерархии наследования.

Возможности сред C++ поддерживают рефакторинг. Если во время проведения рефакторинга программы программист вносит ошибку, компилятор C++ зачастую сразу сообщает об этом. Многие среды разработки программного обеспечения C++ обеспечивают мощные возможности создания перекрестных ссылок и просмотра кода.

Особенности языков и стилей программирования, усложняющие рефакторинг

Совместимость C++ с C является, как хорошо знает большинство программистов, палкой о двух концах. На языке программирования C написано много программ, и работать на C умеет очень большое количество программистов, что (по крайней мере, на первый взгляд) делает переход на C++ более легким, чем на другие объектно-ориентированные языки. Однако C++ поддерживает множество стилей программирования, далеко не все из которых соответствуют принципам качественного проектирования.

Программы, использующие такие возможности C++, как указатели, преобразования типов и `sizeof(object)`, трудно поддаются рефакторингу. Указатели и операции преобразования типа приводят к псевдонимам имен, затрудняющим обнаружение всех обращений к объекту, к которому требуется применить рефакторинг. Каждая из этих возможностей показывает внутреннее представление объекта, что нарушает принципы абстрагирования.

Например, в C++ используется механизм таблиц для представления переменных-членов в выполняемой программе. Сначала идет группа унаследованных переменных-членов, за которыми следуют локально определенные переменные-члены. Один из в общем случае безопасных рефакторингов состоит в перемещении переменной в суперкласс. Поскольку в результате переменная наследуется, а не определена локально в подклассе, физическое

местоположение переменной в выполнимом модуле после рефакторинга изменяется. Если все обращения к переменным в программе производятся через интерфейсы классов, такое изменение физического местонахождения переменной не должно отразиться на поведении программы.

Однако если обращение к переменной производилось с помощью арифметики указателей (например, у программиста был указатель на объект, и он, зная, что переменная находится в пятом байте, присваивал значение пятому байту с помощью арифметики указателей), то перемещение переменной в суперкласс, скорее всего, повлечет изменение поведения программы. Аналогично, если программист написал условный оператор вида `if (sizeof(object)==15))` и произвел рефакторинг программы, чтобы удалить из класса переменные, на которые нет ссылок, размер экземпляров этого класса уменьшится, и условие, бывшее раньше истинным, станет ложным.

Предположение, что есть кто-то, кто пишет программы, выполняющие условные действия в зависимости от размера объекта или использующие арифметику указателей, в то время как C++ предоставляет значительно более понятный интерфейс для обращения к переменным класса, может показаться абсурдным. Но раз такие возможности (и другие, зависящие от физического расположения объектов) в C++ есть, то есть и программисты, которые ими воспользуются, например просто по привычке из языка программирования C. Переход с C на C++ не делает объектно-ориентированных программистов или проектировщиков из процедурных.

Из-за сложности C++ (по сравнению со Smalltalk и в меньшей степени с Java) значительно труднее создавать такие представления структуры программы, которые удобны для автоматизации проверки безопасности рефакторинга и для его выполнения.

Поскольку большинство ссылок C++ разрешает на этапе компиляции, рефакторинг программы обычно требует полной перекомпиляции (или по крайней мере ее части) и компоновки выполнимого модуля перед проверкой результата модификации. Smalltalk же и CLOS (Common Lisp Object System), напротив, предоставляют среду интерпретации и инкрементной компиляции. В то время как выполнение (а возможно, и откат) ряда инкрементных рефакторингов довольно естественно выполняется в Smalltalk и CLOS, стоимость одной итерации (измеряемая повторными компиляцией и тестированием) для программ C++ выше, поэтому программисты менее охотно вносят небольшие изменения.

Во многих приложениях используются базы данных. Изменения в структуре объектов программы C++ могут потребовать внесения соответствующих изменений в схему базы данных. (Многие из идей, применявшихся мной при практическом рефакторинге, появились в результате исследований эволюции объектно-ориентированной схемы базы данных.)

Еще одно ограничение, которое, впрочем, заинтересует скорее исследователей, а не практиков, состоит в том, что C++ не предоставляет поддержки анализа и изменения программ на метауровне. В нем нет ничего аналогичного протоколу метаобъектов, имеющемуся в CLOS. Например, протокол метаобъектов CLOS поддерживает рефакторинг превращения отдельных экземпляров класса в

экземпляры другого класса с автоматическим перенаправлением всех ссылок со старых объектов на новые (который иногда оказывается очень полезным). К счастью, ситуации, когда мне требовались такие возможности, были довольно редкими.

Завершающие комментарии

Методы рефакторинга могут применяться (и реально применялись) к программам С++ в различных контекстах. Программы на С++ часто предполагается развивать в течение нескольких лет. В процессе такого развития программного обеспечения легче всего увидеть выгоды рефакторинга. Этот язык предоставляет некоторые возможности, упрощающие рефакторинг, тогда как использование других возможностей языка делает задачу рефакторинга более трудной. К счастью, широко распространено мнение, что такими возможностями языка, как арифметика указателей, лучше не пользоваться, и большинство хороших объектно-ориентированных программистов действительно их избегает.

Я очень признателен Ральфу Джонсону (Ralph Johnson), Мику Мерфи (Mick Murphy), Джеймсу Роскинду (James Roskind) и другим, кто познакомил меня с мощью и сложностью С++ в отношении рефакторинга.

Рефакторинг как средство получения скорейших выгод

Довольно просто описать среднесрочные и долгосрочные выгоды от применения рефакторинга. Но многие организации все чаще оцениваются инвесторами (и не только ими) по краткосрочным результатам. Может ли рефакторинг принести скорейшую выгоду?

Опытными объектно-ориентированными разработчиками рефакторинг успешно применяется более десяти лет. Многие из этих программистов приобрели опыт при работе со Smalltalk, в котором ценятся ясность и простота кода и принято его повторное использование. В такой культуре программисты тратят время на рефакторинг, потому что, с их точки зрения, это правильный подход. Язык Smalltalk и его реализации сделали возможным проведение рефакторинга способами, недоступными для большинства прежних языков и сред разработки программного обеспечения. В прошлом программированием на Smalltalk в большинстве случаев занимались группы исследователей в Xerox и PARC либо небольшие группы программистов в технически передовых компаниях и консультационных фирмах. В таких группах имеется своя шкала ценностей, несколько отличающаяся от свойственных многим группам промышленной разработки программного обеспечения. Мы с Мартином понимаем, что для принятия рефакторинга основной массой сообщества разработчиков программного обеспечения хотя бы некоторые его выгоды должны проявляться немедленно.

Наша исследовательская группа описала [3, 9, 12–15] несколько примеров того, как рефакторинги могут чередоваться с развитием программы так, чтобы достигались как краткосрочные, так и долгосрочные выгоды. Одним из наших примеров является каркас файловой системы Choices. Первоначально этот каркас реализовывал формат файловой системы BSD (Berkeley Software Distribution) Unix. Позднее он была расширен для поддержки UNIX System V, MS-DOS, постоянных и распределенных файловых систем. Файловые системы System V несут в себе много сходства с файловыми системами BSD Unix. Подход, выбранный разработчиком каркаса, заключался в том, чтобы сначала клонировать части реализации BSD Unix, а затем модифицировать их для поддержки System V. Полученная реализация была работоспособной, но в ней было много дублированного кода. После добавления нового кода разработчик выполнил рефакторинг, создав абстрактный суперкласс, содержащий общее поведение двух реализаций файловой системы Unix. Общие переменные и функции были перемещены в суперкласс. Когда соответствующие функции двух реализаций файловой системы оказывались почти идентичными, но не совпадали полностью, в каждом из подклассов были определены новые функции, содержащие обнаруженные отличия, а в исходных функциях эти фрагменты кода были заменены вызовами новых функций. Пошагово код в двух подклассах был приведен к большему сходству. Если же функции оказывались идентичны, они перемещались в общий суперкласс.

Такие рефакторинги предоставляют несколько преимуществ, достижимых в короткие и средние сроки. Через короткий срок достигнуто преимущество, заключающееся в том, что ошибки, найденные при тестировании в общем коде, нужно исправлять только *в одном месте*. Уменьшился общий объем кода. Поведение, специфичное для конкретной файловой системы, оказывалось отделенным от кода, общего для обеих файловых систем. Это облегчило поиск и исправление ошибок, специфичных для каждого формата. Среднесрочной выгодой от рефакторинга была возможность применения полученных абстракций для определения новых файловых систем. Конечно же, общее для двух имеющихся форматов файловых систем поведение могло не подходить для третьего формата полностью, но являлось удобной отправной точкой. Чтобы выявить действительно общее поведение, можно было применять дополнительные рефакторинги. Постепенно группа разработчиков каркаса обнаружила, что с течением времени требовалось все меньше усилий для добавления нового формата файловой системы. Несмотря на то, что новые форматы оказывались сложнее, их разработку могли вести и менее опытные программисты.

Можно было описать и другие примеры краткосрочных и среднесрочных преимуществ рефакторинга, но Мартин уже сделал это до меня. Так что вместо расширения его списка я лучше проведу аналогию с близким и дорогим для подавляющего большинства из нас физическим здоровьем.

Во многих отношениях рефакторинг подобен физическим упражнениям и правильной диете. Многие знают, что надо больше заниматься физкультурой и есть более сбалансированную пищу. Некоторые из нас живут в среде, активно поощряющей здоровый образ жизни. Мы можем некоторое время не придерживаться этих правильных привычек без видимого ущерба. Всегда можно найти себе оправдание, но, продолжая игнорировать правильное поведение, мы только обманываем себя.

Некоторых из нас подстегивают перспективы ближайших выгод, приносимых упражнениями и соблюдением диеты, например повышенная бодрость, гибкость, более высокая самооценка и т.п. Почти каждому известно, что эти краткосрочные выгоды *вполне* реальны. Многие, хотя и далеко не все, предпринимают хотя бы спорадические усилия в этом направлении. Остальные же не чувствуют мотивации что-то делать, пока не достигнут критической точки.

Конечно, необходимо соблюдать меры предосторожности. По поводу физических упражнений и диеты следует проконсультироваться с врачом. В отношении рефакторинга следует обратиться к имеющимся ресурсам, таким как эта книга и статьи, упоминаемые в разных местах этой главы. Имеющие опыт рефакторинга специалисты в состоянии предоставить более целенаправленную помощь.

Мне встречались люди, которые могут служить образцами для подражания в отношении как физической формы, так и рефакторинга. Я восхищаюсь их энергией и продуктивностью. Отрицательные же примеры являются наглядными результатами пренебрежения здоровым образом жизни и здоровым образом программирования. Их собственное будущее, как и будущее создаваемых ими программных систем, отнюдь не кажется безоблачным.

Рефакторинг может приносить выгоды в кратчайшие сроки и облегчать модификацию и сопровождение программного обеспечения. Но рефакторинг представляет собой средство, а не конечную цель. Это всего лишь часть более широкого контекста разработки и сопровождения своих программ отдельными программистами или группами программистов [3].

Уменьшение стоимости рефакторинга

Еще одно возражение — "Рефакторинг кода является непроизводительной деятельностью. Мне платят за создание новых функций, приносящих доход". Мой ответ на него вкратце таков.

- Существуют инструменты и технологии, с помощью которых рефакторинг можно осуществлять быстро и относительно безболезненно.
- По опыту многих объектно-ориентированных программистов затраты на рефакторинг более чем компенсируются сокращением затрат и сроков других фаз разработки программ.

- Хотя поначалу рефакторинг может показаться несколько трудноватым и требующим лишних затрат сил и времени, по мере того как он становится частью режима разработки программного обеспечения, он перестает восприниматься как нечто внешнее и дополнительное и становится неотъемлемой частью процесса.

Пожалуй, наиболее зрелый инструмент для выполнения автоматического рефакторинга был разработан для Smalltalk группой Software Refactory Университета штата Иллинойс (см. главу 14, "Инструментарий для выполнения рефакторинга"). Хотя средства для выполнения рефакторинга для других языков программирования не столь общедоступны, многие из приемов, описанных в наших работах и этой книге, относительно просто применять, используя простой текстовый редактор или браузер. Интегрированные среды разработки программного обеспечения и браузеры получили за последние годы существенное развитие. Мы надеемся на появление в будущем все большего количества средств для выполнения рефакторинга.

Кент Бек и Уорд Каннингем (Ward Cunningham), опытные программисты на Smalltalk, сообщали на конференциях OOPSLA и других форумах, как рефакторинг позволил им быстро разрабатывать программы для таких предметных областей, как торговля ценными бумагами. Аналогичные свидетельства мне приходилось слышать и от разработчиков программ на C++ и CLOS. В данной книге Мартин описывает преимущества рефакторинга на примере программ на языке программирования Java. Мы надеемся услышать новые свидетельства от тех, кто прочтет эту книгу и применит изложенные принципы.

Мой опыт свидетельствует, что рефакторинг по мере врастания в стандартный процесс разработки программного обеспечения перестает восприниматься как дополнительные траты времени и сил. Конечно, такие заявления легко делать, но трудно доказывать. Обращаясь к скептикам среди читателей, я советую им просто попробовать рефакторинг без каких-либо обязательств, а уж потом принимать для себя решение. Дайте только рефакторингу немного времени, чтобы показать себя.

Безопасный рефакторинг

Важным требованием, особенно в организациях, разрабатывающих и развивающих крупные системы, является надежность. Для многих приложений имеются важные финансовые, юридические и этические основания для предоставления непрерывного, надежного и не допускающего ошибок обслуживания. Во многих организациях проводят широкое обучение и стараются управлять процессом разработки так, чтобы гарантировать надежность создаваемых продуктов.

Однако многие программисты часто не слишком заботятся о надежности. Есть определенная ирония в том, что многие из нас исповедуют безопасность как главное требование в отношении своих детей и родственников, но требуют свободы для программиста как какого-то гибрида вооруженного бандита с Дикого Запада и подростка за рулем. Дайте нам свободу, дайте нам деньги и отпустите на волю. Разве можно лишать нашу организацию плодов наших творческих возможностей ради какой-то там согласованности и следования нормам?

В этом разделе я рассматриваю подходы к безопасности выполнения рефакторинга. Я сосредоточиваю свое внимание на подходе, который, по сравнению с описываемым в этой книге Мартином, более структурирован, но зато позволяет избежать многих ошибок, которые могли бы появиться в результате рефакторинга.

Безопасность — понятие трудно определимое. Интуитивно можно считать, что безопасным является такой рефакторинг, который не нарушает работу программы. Поскольку рефакторинг имеет целью реорганизацию программы без изменения ее поведения, программа после рефакторинга должна выполняться так же, как и до него.

Как же обеспечить безопасность рефакторинга? Есть несколько вариантов.

- Надеяться на свое умение писать код.
- Надеяться на свой компилятор в поиске пропущенных вами ошибок.
- Надеяться на свой комплект тестов в поиске ошибок, не обнаруженных ни вами, ни вашим компилятором.
- Надеяться на анализ кода, который выявит ошибки, не замеченные ни вами, ни вашим компилятором, ни вашим набором тестов.

Мартин при проведении рефакторинга сосредоточивается на первых трех вариантах. В средних и крупных организациях эти шаги часто дополняются анализом кода.

Несмотря на ценность компиляторов, комплектов тестов и дисциплинированного стиля кодирования, все эти подходы имеют определенные ограничения.

- Программисты, даже такие крутые, как вы (не говоря уж обо мне), могут делать ошибки.
- Существуют скрытые (и не очень скрытые) ошибки, которые компиляторы не умеют обнаруживать, особенно в областях, связанных с наследованием [1].
- В работе [16] показано, что хотя общепринято считать (или, по крайней мере, было общепринято), что тестирование проще, когда при реализации используется наследование, в действительности часто требуется огромный набор тестов, чтобы охватить все случаи, когда операции, ранее работавшие с экземплярами класса, теперь работают с экземплярами всех его подклассов. Если разработчик тестов не всеведущ или не уделяет повышенного

внимания деталям, наверняка останутся случаи, не охватываемые комплектом тестов. Тестирование всех возможных путей выполнения программы является вычислительно неразрешимой задачей. Другими словами, комплект тестов не может гарантировать охват всех возможных случаев.

- Рецензенты кода, как и программисты, могут допускать ошибки. Кроме того, они могут быть слишком заняты своей основной работой, чтобы досконально изучать чужой код.

Другой подход, который я применял в своих исследованиях, заключается в определении инструмента проведения рефакторинга, который проверяет возможность безопасного рефакторинга программы и, в случае положительного ответа, выполняет его. В результате удается избежать многих ошибок, появляющихся в результате человеческого фактора.

Здесь я прилагаю высоуровневое описание моего подхода к безопасному рефакторингу. Возможно, это — самый ценный материал данной главы. Более подробные сведения можно найти в моей диссертации [1] и других работах, указанных в конце главы; см. также главу 14, “Инструментарий для выполнения рефакторинга”. Если данный раздел покажется вам слишком техническим, можете его пропустить (за исключением нескольких последних абзацев).

Мой инструментарий для выполнения рефакторинга включает анализатор программ, представляющий собой программу, анализирующую структуру другой программы (в данном случае — программу на языке программирования C++, к которой желательно применить рефакторинг). Этот инструмент может ответить на ряд вопросов, касающихся областей видимости, типов и семантики программы (смысла или замысла действий программы). Проблемы областей видимости, связанные с наследованием, делают такой анализ сложнее, чем для многих исходных текстов, не являющихся объектно-ориентированными программами; однако в C++ есть такие особенности языка, как статическая типизация, которые делают этот анализ проще, чем, скажем, в случае Smalltalk.

Рассмотрим, например, рефакторинг, удаляющий из программы переменную. Описываемый инструмент может определить, в каких других частях программы происходят обращения к этой переменной (если таковые есть). Если обращения имеются, то после удаления переменной останутся “висячие ссылки”, и рефакторинг, таким образом, не будет безопасным. Пользователь, запросивший у инструмента выполнение такого рефакторинга, получит сообщение об ошибке. В результате пользователь может решить, что проведение данного рефакторинга вообще было плохой идеей, либо внесет изменения в те части программы, которые обращаются к переменной, и выполнит рефакторинг, удаляющий ее. Имеется множество других проверок, в большинстве своем столь же простых, но иногда и куда более сложных.

В своем исследовании я определил безопасность на языке свойств программы (связанных с такими действиями, как определение области видимости и типизация), которые должны сохраняться после рефакторинга. Многие из этих свойств аналогичны ограничениям целостности, которые должны сохраняться при изменении схемы базы данных [17]. С каждым рефакторингом связан набор необходимых предварительных условий, истинность которых гарантирует сохранение свойств программы. Данный инструмент выполняет рефакторинг только тогда, когда убедится, что его выполнение безопасно.

К счастью, безопасность рефакторинга часто устанавливается тривиально, в особенности для рефакторингов нижнего уровня, которых на практике оказывается подавляющее большинство. Чтобы обеспечить безопасность сложных рефакторингов более высокого уровня, мы определили их на языке рефакторингов низкого уровня. Например, рефакторинг, создающий абстрактный суперкласс, определен посредством шагов, каждый из которых представляет собой более простой рефакторинг, такой, например, как создание и перемещение переменных и методов. Показав, что каждый шаг сложного рефакторинга безопасен, мы тем самым показываем безопасность всего рефакторинга.

Встречаются (впрочем, относительно редко) случаи, когда на самом деле рефакторинг вполне безопасен, но инструмент в этом не уверен. В таких случаях инструмент перестраховывается и запрещает выполнение рефакторинга. Обратимся вновь к случаю, когда вы хотите удалить из программы переменную, но в некотором месте программы имеется ссылка на нее. Возможно, эта ссылка присутствует во фрагменте кода, который никогда не выполняется. Например, она находится в условной инструкции наподобие `if-then`, условие которой не может быть истинным. Если условие гарантированно никогда не окажется истинным, проверку можно убрать, как и код, ссылающийся на переменную или функцию, которую вы хотите удалить. После этого можно безопасно удалить переменную или функцию. В общем случае точно знать, что некоторое условие всегда будет ложным, невозможно. (Допустим, вы унаследовали код, разработанный кем-то другим. Насколько можно быть уверенным в безопасности удаления этого кода?)

Инструмент для выполнения рефакторинга может сообщить о наличии ссылки и предупредить пользователя. Пользователь же может решить оставить код в покое. Если когда-нибудь у пользователя возникнет уверенность в том, что код со ссылкой не будет выполнен ни при каких условиях, он сможет удалить код и применить рефакторинг. Инструментарий ставит пользователя в известность о возможных последствиях наличия ссылок, а не слепо осуществляет модификацию.

Может показаться, что это очень сложно. Но хотя для диссертации это может быть даже хорошо (ведь ее главная аудитория — оценивающая диссертацию комиссия, которая желает, чтобы внимание было уделено, в первую очередь, теоретическим проблемам), насколько это практично для реального рефакторинга?

Вся проверка безопасности может выполняться инструментарием, поддерживающим проведение рефакторинга, "за сценой", невидимо для программиста. Программист, которому требуется провести рефакторинг, просто просит инструментарий о проверке кода и (если это безопасно) выполнении рефакторинга. Мой инструмент был всего лишь прототипом, предназначенным для исследований. Дон Робертс, Джон Брант, Ральф Джонсон (Ralph Johnson) и я [10] реализовали гораздо более надежный и функционально более богатый инструментарий (см. главу 14, "Инструментарий для выполнения рефакторинга") в ходе нашего исследования рефакторинга программ Smalltalk.

К рефакторингу может быть применено много уровней безопасности. Некоторые из них легко применимы, но не гарантируют высокого уровня безопасности. Применение инструментария для выполнения рефакторинга имеет много преимуществ. Такой инструмент может выполнить множество простых, но утомительных проверок и заранее сообщить о возможных проблемах, без решения которых корректная работа программы в результате рефакторинга будет нарушена.

Хотя применение инструментария для выполнения рефакторинга позволяет избежать множества ошибок, обнаружить которые можно было бы во время компиляции, тестирования и анализа кода, используемые приемы сохраняют свою значимость, в особенности при разработке или развитии систем реального времени. Зачастую программы выполняются не изолированно, а в составе большой сети связанных между собой систем. Некоторые рефакторинги могут не только привести в порядок код, но и повысить производительность программы. Но повышение скорости работы одной программы может привести к возникновению узких мест в других частях системы. Это аналогично тому, как после замены микропроцессора более новым, ускоряющим работу отдельных частей системы, требуются настройка и тестирование работы всей системы. И наоборот, ряд рефакторингов может привести к определенному снижению общей производительности, хотя обычно в целом такие влияния на производительность минимальны.

Безопасный подход должен гарантировать, что рефакторинг не вносит в программу *новые* ошибки. Он не обнаруживает и не исправляет ошибки, которые имелись в программе до выполнения рефакторинга. Однако рефакторинг может облегчить обнаружение и исправление таких ошибок.

Проверка в реальных условиях

На практике выполнение рефакторинга требует решения реально беспокоящих профессиональных разработчиков программного обеспечения вопросов. Чаще других выдвигаются следующие четыре возражения против рефакторинга.

- Программисты могут не понимать, как выполнять рефакторинг.
- Если выгоды ощутимы лишь в далекой перспективе, зачем тратить силы сейчас? В долгосрочной перспективе, когда придет время пожинать плоды, можно оказаться вне проекта.
- Рефакторинг представляет собой непроизводительную деятельность, а программистам платят за новую функциональность.
- Рефакторинг может нарушить работу имеющейся программы.

В этой главе я кратко остановлюсь на каждом из перечисленных возражений и дам ссылки для тех, кто хочет изучить эти проблемы более глубоко.

В некоторых проектах возникают следующие проблемы.

- Что делать, если подлежащий рефакторингу код находится в коллективной собственности нескольких программистов? Во многих случаях подходят традиционные механизмы смены управления. В других случаях, когда программное обеспечение правильно спроектировано и структурировано, подсистемы оказываются достаточно разъединенными, чтобы большинство рефакторингов воздействовало лишь на небольшую часть кода.
- Что делать при наличии нескольких версий исходного кода? В одних случаях имеет смысл проведение рефакторингов для всех версий кода, и тогда все они должны проверяться на безопасность перед применением рефакторинга. В других случаях рефакторинг должен затрагивать лишь некоторые версии, что упрощает проверку и рефакторинг кода. Управление модификациями нескольких версий часто требует применения множества приемов традиционного управления версиями. Рефакторинг может оказаться полезным при объединении вариантов или версий в модифицированный основной код, что может упростить управление версиями.

Резюмируя, должен отметить, что убеждать профессиональных разработчиков программного обеспечения в практической ценности рефакторинга надо совсем не так, как ученый совет в том, что диссертация заслуживает степени кандидата наук. Чтобы оценить эту разницу, мне потребовалось немалое время по завершении моего исследования.

Ресурсы и ссылки, относящиеся к рефакторингу

Я надеюсь, что, раз уж вы добрались до этого места книги, вы планируете применять рефакторинг в своей повседневной практике и уговариваете коллег присоединиться к вам. Тем же, кто все еще пребывает в нерешительности, возможно, следует воспользоваться приводимой мной библиографией или связаться

с Мартином (`Fowler@acm.org`), со мной или с другими разработчиками, у которых есть опыт применения рефакторинга.

Для тех, кто хочет изучить рефакторинг детальнее, я приведу список книг, которые, может быть, следует приобрести. Как уже говорил Мартин, данная книга — не первый печатный труд по рефакторингу, но, я надеюсь, она донесет идеи и преимущества рефакторинга до широкой аудитории. Хотя моя диссертация была первой крупной печатной работой по этой тематике, большинству читателей, интересующихся ранними основополагающими работами по рефакторингу, следует начать со статей [3, 9, 12, 13]. Рефакторинг был темой конференций OOPSLA 95 и OOPSLA 96 [14, 15]. Для тех, кого интересуют как проектные шаблоны, так и рефакторинг, будет хорошей отправной точкой работа "*Lifecycle and Refactoring Patterns That Support Evolution and Reuse*" [3] (*Жизненный цикл и шаблоны, поддерживающие развитие и повторное использование*), которую Брайан Фут (Brian Foote) и я представили на PLoP'94 и которая вышла в первом томе серии Addison-Wesley *Pattern Languages of Program Design*. Мое исследование рефакторинга в большой степени основано на работе Ральфа Джонсона (Ralph Johnson) и Брайана Фута (Brian Foote), посвященной каркасам объектно-ориентированных приложений и проектированию повторно используемых классов [4]. Последующие исследования рефакторинга, которые проводили Джон Брант, Дон Робертс и Ральф Джонсон в Университете штата Иллинойс, были посвящены рефакторингу программ на языке Smalltalk [10, 11]. На их веб-сайте (`http://st-www.cs.uiuc.edu`) можно найти некоторые из их последних работ. Возросший интерес к рефакторингу отмечается и в сообществе исследователей объектно-ориентированного программирования. Несколько работ на эту тему было представлено на конференции OOPSLA 96 в секции *Refactoring and Reuse* (*Рефакторинг и повторное использование кода*) [18].

Следствия повторного использования программного обеспечения и передачи технологий

Практические проблемы, о которых говорилось выше, касаются не только рефакторинга. В более широком плане они применимы ко всему развитию и повторному использованию программного обеспечения.

Несколько последних лет я много занимался проблемами, связанными с повторным использованием программного обеспечения, платформами, каркасами, проектными шаблонами и эволюцией старых систем, в которых часто имеется большое количество кода, не использующего объекты. Помимо работы над проектами в Lucent и Bell Labs, я участвовал в совещаниях в других организациях, которые столкнулись с аналогичными проблемами [19–22].

Практические вопросы, касающиеся повторного использования программ, аналогичны относящимся к рефакторингу.

- Технический персонал может не понимать, что и как повторно использовать.
- Технический персонал может оказаться незаинтересованным в применении повторного использования, если при этом нельзя быстро добиться выгоды.
- Чтобы повторное использование было принято персоналом, надо решить проблемы оплаты дополнительной работы и обучения.
- Принятие повторного использования кода не должно приводить к срыву проекта. Может оказываться сильное давление в пользу применения имеющихся средств или реализаций, хотя и с некоторыми действующими ограничениями. Новые реализации должны взаимодействовать или быть обратно совместимыми с существующими системами.

Джеффри Мур (Geoffrey Moore) [23] описал процесс принятия технологии как колоколообразную кривую, на переднем фронте которой находятся новаторы и те, кто первым принял технологию, большой горб посредине содержит запоздалое большинство, а в хвостовой части плетутся самые нерасторопные. Иными словами, многие идеи, притягательные для новаторов, оказываются неудачными просто потому, что так и не доходят до большинства. Отсутствие контакта в основном обусловлено разными факторами мотивации в этих группах. Новаторов и первых пользователей привлекают новые технологии, предвидение смены парадигм и технологические прорывы. Большинство озабочено в основном зрелостью технологии, ее стоимостью и наличием поддержки и постоянно присматривается, насколько успешно новая идея или продукт применяются теми, перед кем стоят аналогичные задачи.

Чтобы произвести впечатление и убедить исследователей в области программного обеспечения, нужны совсем другие средства. Исследователями в области программного обеспечения чаще всего являются те, кого Мур называет новаторами. Разработчики программного обеспечения, а особенно менеджеры, входят в основном в раннее и позднее большинство. Чтобы добиться внимания каждой из этих групп, следует понимать указанные различия. Для повторного использования программного обеспечения, как и для рефакторинга, важно предлагать профессиональным разработчикам то, что их заинтересует.

В Lucent/Bell Labs я обнаружил, что для поощрения повторного использования кода необходимо установить контакт с рядом заинтересованных кругов. Это потребовало формулировки стратегии для администраторов, организации совещаний руководителей среднего звена, консультаций с разработчиками проектов и освещения преимуществ этих технологий для широкой аудитории исследователей и разработчиков посредством семинаров и публикаций. Всюду было важно

обучить персонал основным принципам, показать краткосрочные выгоды, продемонстрировать сокращение издержек и заботу о безопасном внедрении этих технологий. Понимание всего этого я приобрел из опыта исследований рефакторинга.

Как отметил, просматривая черновик этой главы, Ральф Джонсон (Ralph Johnson), бывший руководителем моей диссертационной работы, эти принципы применимы не только к рефакторингу и повторному использованию программного обеспечения; это общие проблемы передачи технологии. Если вам придется когда-либо убеждать людей в достоинствах рефакторинга (или какой-либо иной технологии или практики), сосредоточьтесь на упомянутых проблемах и убеждайте людей с учетом их интересов. Передать технологию трудно, но вполне возможно.

Завершающее замечание

Спасибо, что вы нашли время прочесть эту главу. Я попытался коснуться многих из ваших опасений, которые могли возникнуть в отношении рефакторинга, и постарался показать, что многие из них не только касаются рефакторинга, а имеют более прямое отношение к развитию и повторному использованию программного обеспечения. Я надеюсь, что эта глава придаст вам энтузиазма в применении рассмотренных идей в своей работе. Желаю вам прогресса в решении стоящих перед вами задач разработки программного обеспечения!

Библиография

1. Opdyke, William F. “Refactoring Object-Oriented Frameworks.” Ph.D. diss., University of Illinois at Urbana-Champaign. Also available as Technical Report UIUCDCS-R-92-1759, Department of Computer Science, University of Illinois at Urbana-Champaign.
2. Brooks, Fred. “No Silver Bullet: Essence and Accidents of Software Engineering.” In *Information Processing 1986: Proceedings of the IFIP Tenth World Computing Conference*, edited by H.-L. Kugler. Amsterdam: Elsevier, 1986.
3. Foote, Brian, and William F. Opdyke. “Lifecycle and Refactoring Patterns That Support Evolution and Reuse.” In *Pattern Languages of Program Design*, edited by J. Coplien and D. Schmidt. Reading, Mass.: Addison-Wesley, 1995.
4. Johnson, Ralph E., and Brian Foote. “Designing Reusable Classes.” *Journal of Object-Oriented Programming* 1(1988): 22–35.

5. Rochat, Roxanna. “In Search of Good Smalltalk Programming Style.” Technical report CR-86-19, Tektronix, 1986.
6. Lieberherr, Karl J., and Ian M. Holland. “Assuring Good Style For Object- Oriented Programs.” *IEEE Software* (September 1989) 38–48.
7. Wirfs-Brock, Rebecca, Brian Wilkerson, and Luaren Wiener. *Design Object-Oriented Software*. Upper Saddle River, N.J.: Prentice Hall, 1990.
8. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison-Wesley, 1985.
9. Opdyke, William F., and Ralph E. Johnson. “Creating Abstract Superclasses by Refactoring.” In *Proceedings of CSC '93: The ACM 1993 Computer Science Conference*. 1993.
10. Roberts, Don, John Brant, Ralph Johnson, and William Opdyke. “An Automated Refactoring Tool.” In *Proceedings of ICAST 96: 12th International Conference on Advanced Science and Technology*. 1996.
11. Roberts, Don, John Brant, and Ralph E. Johnson. “A Refactoring Tool for Smalltalk.” *TAPOS* 3(1997) 39–42.
12. Opdyke, William F., and Ralph E. Johnson. “Refactoring: An Aid in Designing Application Frameworks and Evolving Object-Oriented Systems.” In *Proceedings of SOOPPA '90: Symposium on Object-Oriented Programming Emphasizing Practical Applications*. 1990.
13. Johnson, Ralph E., and William F. Opdyke. “Refactoring and Aggregation.” In *Proceedings of ISOTAS '93: International Symposium on Object Technologies for Advanced Software*. 1993.
14. Opdyke, William, and Don Roberts. “Refactoring.” Tutorial presented at OOPSLA 95: 10th Annual Conference on Object-Oriented Program Systems, Languages and Applications, Austin, Texas, October 1995.
15. Opdyke, William, and Don Roberts. “Refactoring Object-Oriented Software to Support Evolution and Reuse.” Tutorial presented at OOPSLA 96: 11th Annual Conference on Object-Oriented Program Systems, Languages and Applications, San Jose, California, October 1996.
16. Perry, Dewayne E., and Gail E. Kaiser. “Adequate Testing and Object-Oriented Programming.” *Journal of Object-Oriented Programming* (1990).
17. Banerjee, Jay, and Won Kim. “Semantics and Implementation of Schema Evolution in Object-Oriented Databases.” In *Proceedings of the ACM SIGMOD Conference*, 1987.

18. Proceedings of OOPSLA 96: Conference on Object-Oriented Programming Systems, Languages and Applications, San Jose, California, October 1996.
19. Report on WISR '97: Eighth Annual Workshop on Software Reuse, Columbus, Ohio, March 1997. *ACM Software Engineering Notes.* (1997).
20. Beck, Kent, Grady Booch, Jim Coplien, Ralph Johnson, and Bill Opdyke. "Beyond the Hype: Do Patterns and Frameworks Reduce Discovery Costs?" Panel session at OOPSLA 97: 12th Annual Conference on Object-Oriented Program Systems, Languages and Applications, Atlanta, Georgia, October 1997.
21. Kane, David, William Opdyke, and David Dikel. "Managing Change to Reusable Software." Paper presented at PLoP 97: 4th Annual Conference on the Pattern Languages of Programs, Monticello, Illinois, September 1997.
22. Davis, Maggie, Martin L. Griss, Luke Hohmann, Ian Hopper, Rebecca Joos, and William F. Opdyke. "Software Reuse: Nemesis or Nirvana?" Panel session at OOPSLA 98: 13th Annual Conference on Object-Oriented Program Systems, Languages and Applications, Vancouver, British Columbia, Canada, October 1998.
23. Moore, Geoffrey A. *Cross the Chasm: Marketing and Selling Technology Products to Mainstream Customers.* New York: HarperBusiness, 1991.

Глава 14

Инструментарий для выполнения рефакторинга

Дон Робертс и Джон Брант

Одним из самых высоких барьеров для рефакторинга кода была печальная недостаточность поддерживающего его инструментария. В языках, таких как Smalltalk, в которых рефакторинг составляет часть культуры программирования, обычно имеются мощные среды, поддерживающие многие функции, необходимые для рефакторинга кода. Но даже в них до недавнего времени эта поддержка была лишь частичной, а бо́льшая часть работы все еще осуществлялась вручную.

Рефакторинг с помощью инструментов

Рефакторинг с использованием автоматизированных средств выглядит иначе, чем ручной. Даже при наличии подстраховки в виде комплекта тестов рефакторинг, производимый вручную, требует больших затрат времени. Одно это обстоятельство зачастую удерживает программистов от рефакторинга, хотя им понятна его необходимость. Если же добиться того, чтобы рефакторинг оказывался не сложнее настройки формата кода, то приводить код в порядок можно так же, как приводится в порядок его внешний вид. Однако такая разновидность приведения кода в порядок может оказать большой положительный эффект на дальнейшую поддержку, повторное использование и легкость понимания кода. Вот что пишет Кент Бек.

Кент Бек

Refactoring Browser полностью изменяет представление о программировании. Всякие мелкие соображения наподобие "конечно, следовало бы поменять это имя, но..." уходят в прошлое: вы просто меняете это имя, поскольку для этого достаточно выбрать пункт в меню.

Начав пользоваться этим инструментом, я провел около двух часов, занимаясь рефакторингом в своем прежнем темпе. Я выполнял рефакторинг, а затем просто смотрел в пустое пространство в течение минут пяти, которые понадобились бы мне, чтобы выполнить рефакторинг вручную. Затем я выполнял другой рефакторинг и снова смотрел в пространство. Наконец я поймал себя на этом и понял, что необходимо научиться думать быстрее и более крупными рефакторингами. Сейчас я делю время между рефакторингом и вводом нового кода примерно пополам и делаю это с одинаковой скоростью.

С повышением уровня инструментальной поддержки рефакторинг все сильнее сливается с программированием. Становится практически невозможно сказать "Сейчас я программирую" или "Сейчас я занимаюсь рефакторингом". Скорее уж свои действия можно будет описать примерно как "Выделяю эту часть метода, поднимаю ее в суперкласс, затем добавляю вызов нового метода в новый подкласс, над которым работаю". Поскольку тестирование после автоматизированного рефакторинга не требуется, один вид деятельности плавно переходит в другой, и процесс переключения между ними становится малозаметным.

Рассмотрим важный вид рефакторинга — "Извлечение метода" (с. 132). Действуя вручную, нужно выполнить много проверок. С помощью Refactoring Browser вы просто отмечаете текст, который нужно извлечь, и выбираете в меню пункт Extract Method. Инструмент сам определит, можно ли извлечь помеченный текст в метод. Есть ряд причин, по которым это может оказаться невозможным. В этом фрагменте может оказаться помеченной лишь часть идентификатора, могут быть присваивания переменной, но содержаться не все обращения к ней. Обо всем этом не надо беспокоиться, потому что проверкой занимается браузер. Он также выяснит, сколько параметров должно быть передано в новый метод. Потом он предложит ввести имя нового метода и позволит задать порядок параметров в новом вызове. Когда это будет сделано, инструмент извлечет код из исходного метода и заменит его вызовом нового. После этого он создаст новый метод в том же классе, что и исходный, с именем, указанным пользователем. Весь процесс занимает 15 секунд. Сравните это со временем, необходимым для выполнения шагов, указанных в описании рефакторинга "Извлечение метода" (с. 132).

По мере удешевления рефакторинга менее дорогими становятся и ошибки проектирования. Поскольку исправление ошибок проектирования обходится

дешевле, предварительное проектирование требует меньших усилий. Предварительное проектирование основывается на прогнозировании, поскольку технические требования никогда не бывают полными. Пока кода нет, непонятно, как же следует проектировать, чтобы упростить код. До эпохи рефакторинга приходилось придерживаться изначально принятого проекта, каким бы он ни был, просто потому, что стоимость изменения проекта была слишком высока. Благодаря автоматизации рефакторинга можно позволить себе работу с более изменчивыми проектами, поскольку их изменение стало обходиться значительно дешевле. Исходя из новой расстановки стоимости можно выполнять проектирование на уровне текущей задачи, зная, что в будущем при необходимости можно будет без особых затрат расширить проект, введя в него дополнительную гибкость. Больше не нужно стараться предсказать все направления, по которым станет развиваться система. Если выяснится, что текущий проект служит причиной неуклюжести кода и запахов, описанных в главе 3, "Запах в коде", можно быстро изменить проект, чтобы сделать код понятнее и проще в сопровождении.

Применение инструментальных средств рефакторинга оказывает воздействие и на тестирование. Теперь оно требуется в значительно меньшем объеме, поскольку многие рефакторинги выполняются автоматически. Однако всегда найдутся рефакторинги, которые невозможно автоматизировать, поэтому полностью исключить этап тестирования также невозможно. Опыт показывает, что в течение дня выполняется столько же тестов, сколько и в средах без средств автоматизации рефакторинга, но зато рефакторингов выполняется больше.

Как указывал Мартин, такие вспомогательные средства нужны для программистов на Java. Мы хотим выделить ряд критериев, которым должен удовлетворять такой инструмент. Мы включили в их состав и технические критерии, но полагаем, что гораздо большее значение имеют критерии практические.

Технические критерии инструментария для рефакторинга

Основная задача инструментария для выполнения рефакторинга — позволить программисту производить рефакторинг кода без обязательного последующего тестирования программы. Тестирование требует времени, даже если оно автоматизировано, и его исключение может существенно повысить скорость рефакторинга. В данном разделе кратко описываются технические требования к инструментам для выполнения рефакторинга, необходимые для преобразования программы с сохранением ее поведения.

База данных программы

Одним из первых осознанных требований была возможность поиска различных программных объектов по всей программе; например, для некоторого метода — поиска всех вызовов, потенциально реализующих вызовы рассматриваемого метода, или поиска всех методов, выполняющих чтение или запись некоторой переменной экземпляра. В таких сильно интегрированных средах, как Smalltalk, данная информация постоянно поддерживается в виде, пригодном для поиска. Это не база данных в традиционном понимании; тем не менее это *хранилище* с возможностями поиска. Программист может выполнить поиск и найти перекрестные ссылки к любому элементу программы, в основном благодаря динамической компиляции кода. Как только в какой-либо класс вносится изменение, оно немедленно компилируется в байт-код, и происходит обновление "базы данных". В более статичных средах, таких как Java, код вносится в текстовые файлы. Обновление базы данных в этом случае должно происходить путем выполнения некоторой программы, обрабатывающей эти файлы и извлекающей из них необходимую информацию. Выполнение таких обновлений аналогично компиляции самого кода Java. В некоторых более современных средах, таких как IBM VisualAge for Java, из Smalltalk скопировано динамическое обновление базы данных программы.

Наивный подход заключается в использовании для поиска текстовых средств, таких как `grep`, но он быстро приводит к ошибкам, потому что не видит разницы, например, между переменной с именем `foo` и функцией с тем же именем. Создание базы данных требует выполнения синтаксического анализа для идентификации каждого токена программы. Это должно делаться как на уровне определения класса (для выявления определений переменных экземпляров и методов), так и на уровне метода (для выявления обращений к переменным экземпляров и методам).

Деревья синтаксического анализа

В большинстве рефакторингов приходится работать с фрагментами системы, находящимися ниже уровня метода. Обычно это обращения к изменяемым элементам программы. Например, при переименовании переменной экземпляра (простое изменение ее определения) должны быть обновлены все обращения к ней внутри методов класса и его подклассов. Другие рефакторинги целиком осуществляются ниже уровня метода, как, например, извлечение части метода в отдельный самостоятельный метод. Любое обновление метода должно иметь возможность работать со структурой метода. Для этого необходимо наличие деревьев синтаксического анализа. Дерево синтаксического анализа — это структура данных, представляющая внутреннюю структуру самого метода. В качестве простого примера рассмотрим следующий метод:

```
public void hello() {
    System.out.println("Hello World");
}
```

Соответствующее дерево синтаксического анализа показано на рис. 14.1.

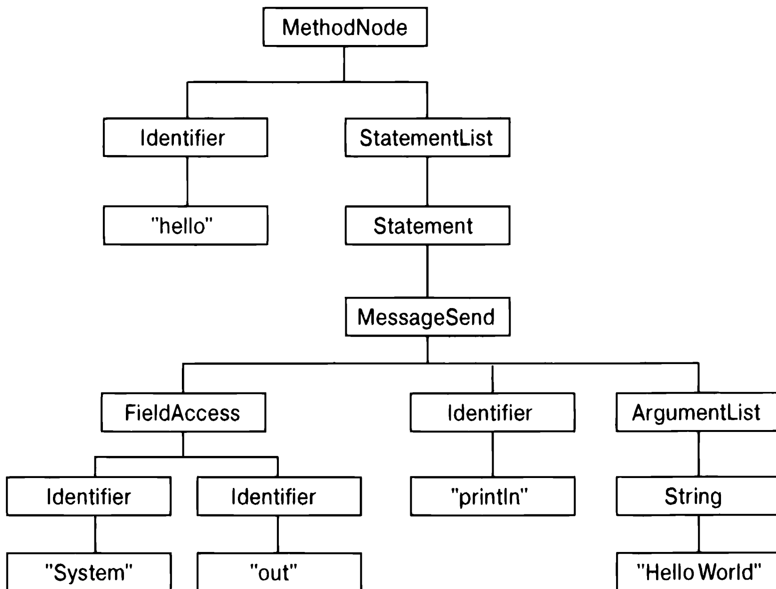


Рис. 14.1. *Дерево синтаксического анализа простейшей программы*

Точность

Рефакторинги, выполняемые автоматизированным инструментарием, должны в достаточной мере сохранять поведение программ. Абсолютного сохранения поведения достичь невозможно. Например, что делать, если в результате рефакторинга программа станет работать на несколько миллисекунд быстрее или медленнее? Обычно это несущественно, но если к программе предъявляются жесткие требования по работе в реальном времени, она может начать работать некорректно.

Можно нарушить работу даже очень простых программ. Например, если программа создает строку и использует Java Reflection API для выполнения метода с именем `String`, то в результате переименования метода программа сгенерирует исключение, которого не было в исходной программе.

Однако для большинства программ рефакторинги могут выполняться достаточно корректно. При корректной идентификации ситуаций, могущих конфликтовать с рефакторингом, программисты могут отказаться от проведения

рефакторинга или исправить вручную части программы, в которых автоматический инструментарий проведения рефакторинга бессилен.

Практические критерии инструментария

Инструменты создаются для того, чтобы помогать человеку в решении конкретных задач. Если инструмент не соответствует требованиям или способу работы человека, он не станет им пользоваться. Наиболее важные критерии связаны с интеграцией рефакторинга и других инструментов.

Скорость

Анализ и преобразования, необходимые для выполнения рефакторинга, могут потребовать большого количества времени, в особенности при высокой сложности рефакторинга. Всегда следует учитывать относительную стоимость времени и точности. Если рефакторинг выполняется слишком долго, программист не станет выполнять его автоматически, предпочтя выполнение вручную со всеми возможными последствиями. Пренебрегать скоростью выполнения нельзя. В процессе разработки Refactoring Browser мы отказались от реализации ряда рефакторингов только потому, что не могли надежно выполнять их за разумное время. Но мы неплохо справились со своей задачей, и большинство наших рефакторингов выполняется очень быстро и точно. Теоретики склонны обращать особое внимание на все предельные случаи, с которыми не справляется конкретный подход. Но на самом деле большинство программ далеки от таких предельных случаев, так что упрощенные быстрые подходы действуют на удивление хорошо.

Если анализ выполняется слишком медленно, возможное решение состоит в том, чтобы запросить нужную информацию у программиста. Так ответственность за точность перекладывается на программиста, в то же время позволяя быстро выполнить анализ. Очень часто программист располагает необходимой информацией. Конечно, при таком подходе нельзя быть уверенным в безопасности, поскольку программисту как человеку свойственно ошибаться, но теперь ответственность за ошибки ложится на него. Как ни странно, в результате программисты более склонны пользоваться таким инструментом, потому что при поиске данных им не приходится полагаться на заложенные в программе эвристики.

Отмена модификаций

Автоматический рефакторинг допускает исследовательский подход к проектированию. Можно поэкспериментировать с кодом и посмотреть, как он выглядит в рамках нового проекта. Поскольку рефакторинг должен сохранять поведение, возврат к оригиналу тоже является рефакторингом и тоже должен сохранять поведение. В ранних версиях Refactoring Browser отсутствовала возможность отката. Из-за этого рефакторинг оказывался существенно осложненным, потому что откат примененных рефакторингов, несмотря на сохранение поведения, был затруднен. Часто приходилось искать старую версию программы и начинать работу сначала, что изрядно досаждало программистам. Когда в Refactoring Browser была добавлена возможность отката, оказалось снятым еще одно ограничение. Теперь можно заниматься исследованиями, зная, что в любой момент можно вернуться к любому предшествующему варианту кода. Теперь можно очень быстро создавать классы, перемещать в них методы и смотреть, что при этом получается, а затем изменять решение и двигаться совсем в другом направлении.

Интеграция с другими инструментами

В последнее десятилетие интегрированные среды разработки (IDE) стали основой большинства разрабатываемых проектов. В интегрированной среде разработки объединены редактор, компилятор, компоновщик, отладчик и прочие инструменты, необходимые при разработке программ. Ранняя реализация Refactoring Browser для Smalltalk представляла собой инструмент, отдельный, не связанный со стандартными средствами разработки для Smalltalk. Но при этом, как выяснилось, никто с ним не работал. Даже мы сами не применяли его! Но как только мы встроили рефакторинги непосредственно в Smalltalk Browser, мы стали широко пользоваться ими. То, что теперь они всегда были под рукой, оказалось очень существенным фактором.

Краткое заключение

Мы занимались разработкой и применением Refactoring Browser в течение нескольких лет. Нередко с его помощью мы выполняем рефакторинг его собственного кода. Одна из причин нашего успеха в том, что мы сами программисты и постоянно стремимся приспособить Refactoring Browser к своим методам работы. Если мы сталкивались с рефакторингом, который приходилось выполнять вручную, но при этом чувствовали его универсальность, мы реализовывали его

и добавляли в браузер. Если какой-то рефакторинг выполнялся слишком долго, мы добивались увеличения скорости его выполнения. Если что-то делалось не слишком точно, мы старались немедленно исправить ситуацию.

Мы уверены, что средства автоматизации рефакторинга представляют собой наилучший способ справиться с ростом сложности программного проекта по мере его развития. Без инструментов, позволяющих бороться с этой сложностью, программное обеспечение становится раздутым, переполненным ошибками и хрупким. Поскольку Java гораздо проще языка, с которым имеет общий синтаксис, разрабатывать для него средства автоматизации рефакторинга значительно легче. Мы надеемся, что это произойдет, и мы сможем избежать недостатков C++.

Глава 15

Заключение

Кент Бек

Теперь у вас есть все части головоломки. Вы познакомились с рефакторингом, изучили каталог и освоили все списки необходимых действий. Вы умеете тестировать и поэтому ничего не боитесь. Теперь вам может казаться, что вы умеете выполнять рефакторинг программы. Но пока что это не совсем так.

Список методов рефакторинга — это только начало. Это ворота, через которые нужно пройти. Без этих методов вы не сможете изменять конструкцию работающих программ. С ними, впрочем, тоже — но по крайней мере это хорошая отправная точка.

Почему же все эти замечательные методы представляют собой только начало? Потому что вы пока что не знаете, когда их нужно применять, а когда — нет, когда нужно начать рефакторинг, а когда остановиться, когда двигаться вперед, а когда ждать. Рефакторинг — это мелодия, а не отдельные ноты.

Как узнать, что вы действительно начали понимать рефакторинг? Вы узнаете об этом, когда на вас снизойдет спокойствие. Когда вы почувствуете абсолютную уверенность, что, как бы хитро ни был запутан код, который вам достался, вы сумеете настолько его улучшить, что сможете легко работать над ним дальше.

Но главный признак того, что вы освоили рефакторинг, — умение вовремя остановиться. Это самый сильный ход из имеющихся в запасе у применяющего рефакторинг. Вы видите крупную цель — удалить кучу лишних подклассов. Вы начинаете двигаться к этой цели маленькими, но твердыми шагами, подкрепляя каждый шаг успешным выполнением всех тестов. Вы близки к цели. Осталось объединить пару методов в каждом из подклассов, и о них можно забыть.

И здесь ваш мотор глохнет. Может быть, просто уже поздно, и вы устали. Может быть, вы ошиблись изначально, и на самом деле избавиться от всех этих подклассов нельзя. Может быть, у вас недостаток тестов. Что бы ни было причиной, вы потеряли уверенность. Вы не можете уверенно, не сомневаясь сделать следующий шаг. Вряд ли произойдет что-то неприятное... но полной уверенности в этом у вас нет.

Здесь вам надо остановиться. Если код уже улучшился, соберите все сделанное вами в одно целое и выпускайте готовую версию. Если код не стал лучше, оставьте

его. Выбросьте свой рефакторинг и забудьте. Жаль, что фокус не удался, но зато у вас есть опыт. Что там у вас в планах на завтра?

Завтра (или через день, или через месяц, или через год, мой личный рекорд составляет девять лет перед тем, как приступить ко второй части рефакторинга) придет озарение. Вы либо поймете, в чем были не правы, либо сообразите, что были правы. И в том, и в другом случаях следующий шаг очевиден. И вы сделаете его с такой же уверенностью, с какой начинали рефакторинг. Может быть, вам будет даже несколько неловко оттого, что вы не сообразили раньше. Не надо стесняться — с кем не бывает...

Рефакторинг напоминает поход по узкой тропке над высоким обрывом. Пока светло, можно двигаться вперед осторожно, но уверенно. Однако когда солнце село, лучше остановиться. Вы укладываетесь спать с уверенностью, что утром солнце взойдет снова и можно будет продолжить свой путь.

Вам это кажется мистическим и туманным? В определенном смысле так оно и есть, потому что вы вступили со своей программой в новые отношения. Если вы действительно понимаете рефакторинг, то архитектура системы становится такой же подвижной, пластичной и формируемой для вас, как и отдельные символы в файле с исходным кодом. Вы ощущаете всю конструкцию в целом. Вы видите, как она может изгибаться и изменяться — чуть-чуть в одну сторону, немножко в другую.

Но здесь нет никакой мистики и туманности. Рефакторинг — это искусство, которому можно научиться. Вы уже начали это делать, прочтя о составляющих его действиях в этой книге и начав их изучать. Вы соединяете мелкие приемы в одно целое и "полируете" их. После этого разработка представляется вам в новом свете.

Я сказал, что это искусство, которому можно научиться. Как это сделать?

- **Учитесь выбирать цель**. В ряде мест вашего кода есть запах. Решитесь избавиться от него. После этого двигайтесь к выбранной цели. Вы занимаетесь рефакторингом не в поисках истины и красоты (во всяком случае, это не единственная и не главная ваша задача). Вы стараетесь, чтобы программу было проще понять, чтобы восстановить контроль над программой, которая живет своей жизнью.
- **Остановитесь, почувствовав неуверенность**. По мере продвижения к цели может наступить момент, когда вы не можете уверенно сказать себе и другим, что ваши действия сохранят семантику программы. Остановитесь. Если код уже стал лучше, выпускайте версию, которую удалось получить. Если нет, отмените изменения.
- **Возвращайтесь к прежней версии**. Дисциплину проведения рефакторинга трудно осваивать. Перспективу легко потерять из виду, отвлекшись даже

на мгновение. Я по-прежнему теряю перспективу чаще, чем хотелось бы. Я выполняю по три-четыре рефакторинга подряд, не выполняя постоянно тестирования. Все обойдется, я же чувствую себя уверенно, у меня же большой опыт! Бум! Тест не прошел, и я даже не знаю, какое из изменений стало источником проблемы.

В этот момент может возникнуть сильный соблазн пойти путем отладки. В конце концов, есть тесты, которые должны выполняться. Так ли уж трудно будет добиться, чтобы они снова стали выполняться? Но остановитесь. Вы потеряли контроль над ситуацией, и у вас нет представления о том, чего будет стоить его восстановление при продвижении вперед. Лучше вернитесь к своей последней успешной конфигурации. Воспроизводите модификации по одной и выполняйте тесты после каждой из них.

Это звучит очевидно, когда вы удобно сидите в кресле и читаете эту книгу. Но когда вы работаете с кодом и чувствуете, что до существенного упрощения рукой подать, сложнее всего становится остановиться и возвратиться назад. Но подумайте об этом сейчас, на ясную голову. Если вы занимались рефакторингом час, то сможете воспроизвести сделанное за десять минут. Так что вы гарантированно сможете вновь выйти на достигнутые рубежи за десять минут; попытавшись же двигаться вперед через отладку, вы можете потратить на нее пять секунд, а можете и несколько часов.

Мне легко объяснять, что нужно делать. Но очень тяжело действительно поступать таким образом. То, что я не последовал собственному совету, обошлось мне, пожалуй, часа в четыре потерянного времени в трех отдельных попытках. Я потерял контроль, вернулся назад, медленно пошел вперед, снова потерял контроль, и еще раз — и так в течение четырех мучительных часов. Это не шутки. Вот почему вам нужна помощь.

- **Дуэты**. Старайтесь выполнять рефакторинг вместе с кем-нибудь. Есть много преимуществ работы парами на всех этапах разработки. При проведении рефакторинга преимущества парной работы оказываются особенно существенными. Рефакторинг вознаграждает тщательную и методичную работу. Ваш напарник заставит вас неспешно двигаться шаг за шагом, а вы — его. При проведении рефакторинга вознаграждается способность увидеть как можно более отдаленные последствия. С помощью напарника вы увидите то, чего не смогли увидеть сами, и понять то, чего не поняли сами. При проведении рефакторинга вознаграждается и умение вовремя остановиться. Если ваш напарник не понимает, что вы делаете, это верный признак того, что вы и сами этого не понимаете. Больше всего при проведении рефакторинга вознаграждается спокойная уверенность. Партнер подбодрит вас, когда вы уже будете готовы остановиться.

Еще одной положительной стороной работы с партнером является возможность обсуждения. Вам постоянно нужно обсуждать выполняемые действия, что, по вашему мнению, должно произойти, чтобы вы оба думали в одном направлении. Вы должны обсуждать, что, по вашему мнению, должно произойти в результате ваших действий, чтобы увидеть опасность как можно раньше; обсуждать неприятности, которые только что произошли, чтобы в следующий не встретиться с ними. Все эти обсуждения способствуют закреплению в мозгу понимание того, как отдельные рефакторинги вписываются в ритм проведения общего рефакторинга.

Вполне можно увидеть в своем коде новые возможности, даже после того, как вы проработаете с ним многие годы, — если знать, какие бывают запахи и как с ними бороться. У вас может даже возникнуть желание взять и привести в порядок все видимые вам недостатки. Но не спешите делать это. Ни один руководитель не обрадуется, услышав, что команда должна прервать работу на три месяца, просто чтобы убрать весь беспорядок, который она натворила. Да это и не нужно. Крупный рефакторинг — это способ навлечь на себя катастрофу.

Сколь бы ужасно ни выглядел беспорядок, приучите себя ограничиваться только границами проблем. Если надо добавить некоторую новую функцию в определенной области, потратьте несколько минут и приведите эту область в порядок. Если требуется добавить некоторые тесты, чтобы быть уверенным в корректности программы после наведения порядка, добавьте их. Вы не пожалеете об этом. После предварительного рефакторинга добавление нового кода менее опасно. Поработав с кодом, вы заодно вспомните, как он работает. В результате вы быстрее закончите работу и получите удовлетворение, понимая, что, когда вы вернетесь к этому коду в следующий раз, он будет выглядеть лучше, чем до вашей работы с ним.

Не забывайте менять шляпы. Во время рефакторинга вы неизбежно столкнетесь с неправильной работой кода. Вы будете абсолютно уверены в этом. Но не поддавайтесь соблазну! При проведении рефакторинга ваша задача состоит в том, чтобы новый код выдавал точно те же ответы, что и до того, как вы за него взялись. Не больше и не меньше. Ведите список (у меня всегда рядом с компьютером для этого лежит лист бумаги) того, что надо будет сделать позднее — добавить или изменить контрольные примеры, выполнить не относящиеся к делу рефакторинги, составить документацию, нарисовать графики... Так вы не забудете свои мысли, но и не позволите им мешать выполняемой в данный момент работе.

Библиография

1. Auer, Ken. "Reusability through Self-Encapsulation". In *Pattern Languages of Program Design 1*, edited by J.O. Coplien and D.C. Schmidt. Reading, Mass.: Addison-Wesley, 1995.
   Работа по проектным шаблонам, связанным с идеей самоинкапсуляции.
2. Bäumer, Dirk, and Dirk Riehle. "Product Trader." In *Pattern Languages of Program Design 3*, edited by R. Martin, F. Buschmann, and D. Riehle. Reading, Mass.: Addison-Wesley, 1998.
   Проектный шаблон гибкого создания объектов без знания о том, какому классу они должны принадлежать.
3. Beck, Kent and Andres, Cynthia. *eXtreme Programming eXplained: Embrace Change, 2nd edition*. Pearson Education, Inc., 2005.
4. Beck, Kent, and Erich Gamma. *JUnit Open-Source Testing Framework*. Доступно по адресу `http://ourworld.compuserve.com.homepages/Martin_Fowler`.
   Важный инструмент для работы с Java. Простой каркас, помогающий писать, организовывать и выполнять модульные тесты. Аналогичные каркасы есть для Smalltalk и C++.
5. Beck, Kent. "Make it Run, Make it Right: Design Through Refactoring." *The Smalltalk Report*, 6: (1997b): 19–24.
   Это первая печатная работа, демонстрирующая смысл процесса рефакторинга. Источник множества идей для главы 1, "Первый пример рефакторинга".
6. Beck, Kent. *Smalltalk Best Practice Patterns*. Upper Saddle River, N.J.: Prentice Hall, 1997.
   Важнейшая книга для всех, кто программирует на Smalltalk, и чрезвычайно полезная для всех разработчиков объектно-ориентированных программ.
7. Fowler, M. *Analysis Patterns: Reusable Object Models*. Reading, Mass.: Addison-Wesley, 1997.
   Книга о проектных шаблонах в разных предметных областях.
8. Fowler, M., with K. Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Reading, Mass.: Addison-Wesley, 1997.
   Полное руководство по универсальному языку моделирования (UML), используемому в ряде схем этой книги.

9. Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Reading, Mass.: Addison-Wesley, 1995.
Пожалуй, самая важная книга по объектно-ориентированному проектированию. Ныне решительно невозможно делать вид, будто понимаешь что-то в объектах, если не умеешь с умным видом говорить о стратегиях, синглтонах и цепочке ответственности.
10. Gosling, James, Bill Joy, and Guy Steele. *The Java Language Specification*. Reading, Mass.: Addison-Wesley, 1996.
Ответы на любые вопросы о Java, но желательно использовать последнее издание. (См., например, Джеймс Гослинг, Билл Джой, Гай Стил, Гилад Брача, Алекс Бакли. *Язык программирования Java SE 8. Подробное описание.* — М.: И.Д. ООО "Вильямс". — 2016.)
11. Lea, Doug. *Concurrent Programming in Java: Design Principles and Patterns*, Reading, Mass.:Addison-Wesley, 1997.
Компилятор должен останавливать каждого, кто реализует `Runnable`, предварительно не прочитав эту книгу.
12. McConnell, Steve. *Code Complete: A Practical Handbook of Software Construction*. Redmond, Wash.: Microsoft Press, 1993.
Отличное руководство по стилям программирования и разработке программного обеспечения. Книга написана до создания языка программирования Java, но почти все приводимые в ней советы сохранили свою ценность.
13. Meyer, Bertrand. *Object Oriented Software Construction*. 2 ed. Upper Saddle River, N.J.: Prentice Hall, 1997.
Очень хорошая, хотя и очень большая, книга по объектно-ориентированному проектированию. Содержит исчерпывающее обсуждение проектирования по контракту.
14. Opdyke, William F. "*Refactoring Object-Oriented Frameworks.*" Ph.D. diss., University of Illinois at Urbana-Champaign, 1992.
См. `ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z`. Это первый большой труд, посвященный рефакторингу. В нем рефакторинг рассматривается с несколько академической и ориентированной на инструментальные средства точки зрения (это все же тематическая диссертация); будет интересен всем, кто интересуется теорией рефакторинга.
15. Woolf, Bobby. "Null Object." In *Pattern Languages of Program Design 3*, edited by R. Martin, F. Buschmann, and D. Riehle. Reading, Mass.: Addison- Wesley, 1998.
Обсуждение проектного шаблона "Нулевой объект".

Примечания

С. 36 *Когда выясняется, что в программу необходимо добавить новую функциональность, но при этом код программы не структурирован способом, удобным для добавления этой функциональности, сначала выполните рефакторинг, упрощающий внесение изменений, и только после этого приступайте к самим изменениям.*

С. 37 *Перед тем как приступать к рефакторингу, убедитесь, что у вас есть комплект надежных самопроверяющихся тестов.*

С. 41 *Рефакторинг подразумевает изменение программы небольшими порциями, что облегчает обнаружение ошибок.*

С. 43 *Написать код, понятный компилятору, сможет любой дурак. Но лишь немногие смогут написать код, понятный людям.*

С. 69 *Рефакторинг: изменения внутренней структуры программного обеспечения, направленные на облегчение понимания его работы и упрощение модификации без изменения наблюдаемого поведения.*

С. 70 *Рефакторинг: внесение изменений в структуру программного обеспечения, применяя ряд преобразований, не затрагивающих поведение модифицируемого программного обеспечения.*

С. 74 *Начинайте рефакторинг после трех повторов.*

С. 82 *Не спешите с публикацией интерфейсов. Измените стратегию в отношении вопросов собственности на код так, чтобы она не мешала рефакторингу.*

С. 108 *Почувствовав необходимость написать комментарий, попробуйте сначала изменить структуру кода так, чтобы любые комментарии стали ненужными.*

С. 110 *Тесты должны быть полностью автоматизированы и проверять свои результаты.*

С. 110 *Набор тестов является мощным детектором ошибок, резко уменьшающим время их поиска.*

С. 115 *Выполняйте тесты почаще. Выполняйте их при каждой компиляции, как минимум каждый тест ежедневно.*

С. 118 *Получив сообщение об ошибке, начинайте с написания модульного теста, выявляющего эту ошибку.*

С. 119 *Лучше написать и выполнить неполные тесты, чем не выполнить полные тесты.*

С. 120 *Подумайте о граничных условиях, которые могут быть неправильно обработаны, и сосредоточьте на них свои усилия.*

С. 122 *Не забывайте проверять генерацию исключений в случае возникновения проблем.*

С. 123 *Опасения по поводу того, что тестирование не выявит все ошибки, не должно мешать писать тесты, которые выявят большинство ошибок.*

Список рефакторингов

С. 181 Введение внешнего метода
С. 183 Введение локального расширения
С. 276 Введение нулевого объекта
С. 312 Введение объекта параметра
С. 145 Введение поясняющей переменной
С. 284 Введение утверждения
С. 140 Встраивание временной переменной
С. 174 Встраивание класса
С. 139 Встраивание метода
С. 390 Выделение иерархии
С. 256 Декомпозиция условного оператора
С. 292 Добавление параметра
С. 207 Дублирование видимых данных
С. 159 Замена алгоритма
С. 267 Замена вложенных условных операторов граничным оператором
С. 141 Замена временной переменной запросом
С. 219 Замена двунаправленной связи однонаправленной
С. 371 Замена делегирования наследованием
С. 235 Замена записи классом данных
С. 194 Замена значения данных объектом
С. 198 Замена значения ссылкой
С. 332 Замена исключения проверкой
С. 327 Замена кода ошибки исключением
С. 236 Замена кода типа классом
С. 241 Замена кода типа подклассами
С. 245 Замена кода типа состоянием/стратегией
С. 321 Замена конструктора фабричным методом
С. 223 Замена магического числа символической константой
С. 204 Замена массива объектом
С. 155 Замена метода объектом методов
С. 369 Замена наследования делегированием
С. 216 Замена однонаправленной связи двунаправленной
С. 308 Замена параметра вызовом метода
С. 302 Замена параметра явными методами

С. 250 Замена подкласса полями
С. 202 Замена ссылки значением
С. 271 Замена условной инструкции полиморфизмом
С. 357 Извлечение интерфейса
С. 169 Извлечение класса
С. 132 Извлечение метода
С. 347 Извлечение подкласса
С. 352 Извлечение суперкласса
С. 226 Инкапсуляция коллекции
С. 325 Инкапсуляция нисходящего приведения типа
С. 224 Инкапсуляция поля
С. 260 Консолидация дублирующихся условных фрагментов
С. 258 Консолидация условного выражения
С. 345 Опускание метода
С. 346 Опускание поля
С. 385 Отделение предметной области от представления
С. 300 Параметризация метода
С. 290 Переименование метода
С. 162 Перенос метода
С. 166 Перенос поля
С. 339 Подъем метода
С. 338 Подъем поля
С. 342 Подъем тела конструктора
С. 383 Преобразование процедурного проекта в объектный
С. 296 Разделение запроса и модификатора
С. 378 Разделение наследования
С. 149 Расщепление временной переменной
С. 190 Самоинкапсуляция поля
С. 360 Свертывание иерархии
С. 176 Сокрытие делегирования
С. 320 Сокрытие метода
С. 305 Сохранение всего объекта
С. 317 Удаление метода установки значения
С. 294 Удаление параметра
С. 179 Удаление посредника
С. 152 Удаление присваиваний параметрам
С. 262 Удаление управляющего флага
С. 361 Формирование шаблонного метода

Список запахов

Запах	Применяемые рефакторинги
Дублируемый код	*Извлечение метода (с. 132), Извлечение класса (с. 169), Подъем метода (с. 339), Формирование шаблонного метода (с. 361)*
Длинный метод	*Извлечение метода (с. 132), Замена временной переменной запросом (с. 141), Замена метода объектом методов (с. 155), Декомпозиция условного оператора (с. 256)*
Большой класс	*Извлечение класса (с. 169), Извлечение подкласса (с. 347), Извлечение интерфейса (с. 357), Замена значения данных объектом (с. 194)*
Длинный список параметров	*Замена параметра вызовом метода (с. 308), Введение объекта параметра (с. 312), Сохранение всего объекта (с. 305)*
Расходящиеся изменения	*Извлечение класса (с. 169)*
Стрельба дробью	*Перенос метода (с. 162), Перенос поля (с. 166), Встраивание класса (с. 174)*
Завистливые функции	*Перенос метода (с. 162), Перенос поля (с. 166), Извлечение метода (с. 132)*
Группы данных	*Извлечение класса (с. 169), Введение объекта параметра (с. 312), Сохранение всего объекта (с. 305)*
Одержимость примитивами	*Замена значения данных объектом (с. 194), Извлечение класса (с. 169), Введение объекта параметра (с. 312), Замена массива объектом (с. 204), Замена кода типа классом (с. 236), Замена кода типа подклассами (с. 241), Замена кода типа состоянием/стратегией (с. 245)*
Инструкции switch	*Замена условной инструкции полиморфизмом (с. 271), Замена кода типа подклассами (с. 241), Замена кода типа состоянием/стратегией (с. 245), Замена параметра явными методами (с. 302), Введение нулевого объекта (с. 276)*
Параллельные иерархии наследования	*Перенос метода (с. 162), Перенос поля (с. 166)*

Запах	Применяемые рефакторинги
Ленивый класс	*Встраивание класса (с. 174), Свертывание иерархии (с. 360)*
Теоретическая общность	*Свертывание иерархии (с. 360), Встраивание класса (с. 174), Удаление параметра (с. 294), Переименование метода (с. 290)*
Временное поле	*Извлечение класса (с. 169), Введение нулевого объекта (с. 276)*
Цепочки сообщений	*Сокрытие делегирования (с. 176)*
Посредник	*Удаление посредника (с. 179), Встраивание метода (с. 139), Замена делегирования наследованием (с. 371)*
Неуместная близость	*Перенос метода (с. 162), Перенос поля (с. 166), Замена двунаправленной связи однонаправленной (с. 219), Замена наследования делегированием (с. 176), Сокрытие делегирования (с. 176)*
Альтернативные классы с разными интерфейсами	*Переименование метода (с. 290), Перенос метода (с. 162)*
Неполный библиотечный класс	*Введение внешнего метода (с. 181), Введение локального расширения (с. 183)*
Классы данных	*Перенос метода (с. 162), Инкапсуляция поля (с. 224), Инкапсуляция коллекции (с. 226)*
Отказ от наследства	*Замена наследования делегированием (с. 369)*
Комментарии	*Извлечение метода (с. 132), Введение утверждения (с. 284)*

Предметный указатель

B

break 263

C

C++ 290, 401, 402
continue 263

I

instanceof 284

J

Java 26, 111, 287, 290, 314, 325, 328, 358, 384
  рефлексия 128
JUnit 112, 118

R

Refactoring Browser 42, 90, 420, 425

S

Smalltalk 27, 89, 395
Software Refactory 399

U

UML 76, 125

А

Автозамена 127
Алгоритм 159
Анализ кода 75

Б

База данных 80
Безопасность 409
Большой класс 96

В

Возврат нескольких значений 138
Временная переменная 48, 51, 95, 103, 131, 140, 142, 149
Выходная точка 262

Г

Гибкость 85
Граничный оператор 268

Д

Данные 99, 106
  группы данных 100, 312
Декомпозиция 37
Делегирование 44, 104, 139, 177, 179, 347, 369, 372
Длинный метод 48, 94
Длинный список параметров 97, 294, 309
Дублирование 72, 94, 208, 338, 339, 353, 379, 398

З

Зависимость 80, 177, 215, 220, 273, 306, 309
Запросы и модификаторы 290, 296

И

Иерархия 360, 390
Иерархия наследования 102, 107, 273
Именование 95, 132, 291
Инкапсуляция 81, 104, 177, 225, 226
Интерфейс 81, 289, 309, 357
  опубликованный 81
  пользовательский 385
  тестирующий 283
Исключение 82, 116, 287, 290, 328, 332

К

Кеширование 296
Класс
  абстрактный 102
  большой 96, 170
  данных 106
  ленивый 102
  немой 225
  оболочка 184, 186
  переполненный функциональностью 390
Код
  ошибки 327
  типа 100, 101, 190, 236, 242, 245, 347
Комментарии 107

Композит 357
Конструктор 342
Копирование и вставка 36
Косвенность 78, 80, 139, 191

Л

Локальное расширение 183

М

Магические числа 189, 224
Массив 205, 234
Метод
  возврат нескольких значений 138
  декомпозиция 156
  длинный 48, 94, 155
  именование 132
  константный 250
  параметризация 300
  перенос 162
  сокрытие 320
  установки значения 317
  фабричный 321
Модификаторы и запросы 290, 296
Модульные тесты 117

Н

Надежность 408
Наследование 60, 347, 353, 369, 378
  запутанное 379
Нисходящее приведение 325

О

Обратный указатель 216
Объект
  данных 236
  значение 198
  методов 156, 363
  немой 384
  нулевой 277
  параметра 312
  связи 217
  ссылка 198
Оптимизация 56, 70
Отладка 73, 75

П

Параметр 152, 289
  добавление 293
  передача по значению 154, 314
  удаление 294
Переменная
  временная 48, 51, 95, 103, 131, 136, 140, 142, 149
  глобальная 97
  локальная 135, 136
  поясняющая 146
Поиск и замена 127
Полиморфизм 60, 101, 242, 272, 277
Посредник 179
Программирование
  объектно-ориентированное 26, 384
  параллельное 129, 173, 299
  парами 76
  процедурное 67, 384
  экстремальное 76, 85, 89, 111
Проектирование 79, 84
  предварительное 84
Проектный шаблон 129, 401
  "Визитер" 99
  "Диапазон" 314
  "Количество" 127
  "Нулевой объект" 278, 432
  "Самоделегирование" 99
  "Синглтон" 384
  "Состояние" 60, 61, 66
  "Стратегия" 99
  "Частный случай" 284
  "Шаблонный метод" 362
Производительность 56, 70, 87
Профилирование 88
Публикация интерфейсов 82

Р

Работа парами 429
Расходящиеся изменения 98
Рефакторинг 24, 427
  "Введение внешнего метода" 106, 161
  "Введение локального расширения" 182
  "Введение нулевого объекта" 276

“Введение объекта параметра” 312
“Введение поясняющей переменной” 145
“Введение утверждения” 284
“Встраивание временной переменной” 140
“Встраивание класса” 174
“Встраивание метода” 139
“Выделение иерархии” 390
“Декомпозиция условного оператора” 256
“Добавление параметра” 292
“Дублирование видимых данных” 207
“Замена алгоритма” 159
“Замена вложенных условных операторов граничным оператором” 267
“Замена временной переменной запросом” 141
“Замена двунаправленной связи однонаправленной” 219
“Замена делегирования наследованием” 371
“Замена записи классом данных” 235
“Замена значения данных объектом” 194
“Замена значения ссылкой” 198
“Замена исключения проверкой” 332
“Замена кода ошибки исключением” 327
“Замена кода типа классом” 236
“Замена кода типа подклассами” 241
“Замена кода типа состоянием/стратегией” 245
“Замена конструктора фабричным методом” 321
“Замена магического числа символической константой” 223
“Замена массива объектом” 204
“Замена метода объектом методов” 155
“Замена наследования делегированием” 369
“Замена однонаправленной связи двунаправленной” 216
“Замена параметра вызовом метода” 308
“Замена параметра явными методами” 302
“Замена подкласса полями” 250
“Замена ссылки значением” 202
“Замена условной инструкции полиморфизмом” 271
“Извлечение интерфейса” 357
“Извлечение класса” 169
“Извлечение метода” 132
“Извлечение подкласса” 347
“Извлечение суперкласса” 352
“Инкапсуляция коллекции” 226
“Инкапсуляция нисходящего приведения типа” 325
“Инкапсуляция поля” 224
“Консолидация дублирующихся условных фрагментов” 260
“Консолидация условного выражения” 258
“Опускание метода” 345
“Опускание поля” 346
“Отделение предметной области от представления” 385
“Параметризация метода” 300
“Переименование метода” 290
“Перенос метода” 162
“Перенос поля” 166
“Подъем метода” 339
“Подъем поля” 338
“Подъем тела конструктора” 342
“Преобразование процедурного проекта в объектный” 383
“Разделение запроса и модификатора” 296
“Разделение наследования” 378
“Расщепление временной переменной” 149
“Самоинкапсуляция поля” 190
“Свертывание иерархии” 360
“Сокрытие делегирования” 176
“Сокрытие метода” 320
“Сохранение всего объекта” 305
“Удаление метода установки значения” 317
“Удаление параметра” 294
“Удаление посредника” 179

“Удаление присваиваний параметрам” 152
“Удаление управляющего флага” 262
“Формирование шаблонного метода” 361
автоматизация 401, 419
и администрация проекта 77, 399
и проектирование 84
и производительность 87
крупномасштабный 375
определение 69
проблемы 79, 398
программ на C++ 402
ресурсы и ссылки 413
ритм 67
стоимость 407
цель 70
Рефлексия 128
Роли разработчика 70

С

Самоинкапсуляция 168, 191
Самотестирование 110
Синтаксический анализ 422
Слушатель 215

Т

Теоретическая общность 102
Тестирование 37, 109, 216, 421
граничных условий 120, 434
добавление тестов 118, 216
и наследование 123
модульное 117
управляемое рисками 118
функциональное 117
Транзакция 173

У

Указатель 404
Управляющий флаг 262
Условное выражение 255
вложенное 268, 270
обращение 270

Ф

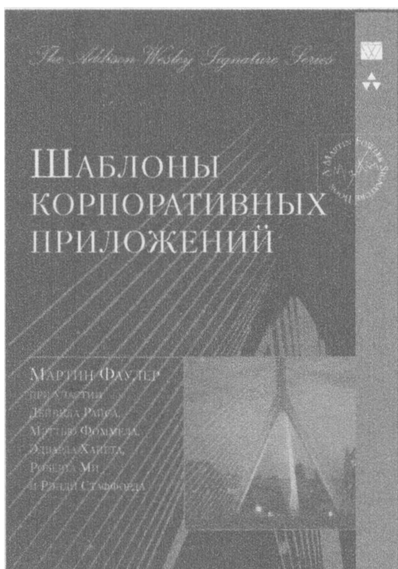
Функциональность
добавление 36, 170, 183, 400
перенос 161
расширение 375
Функциональные тесты 117

Ц

Цепочка сообщений 103
Цикл 143

ШАБЛОНЫ КОРПОРАТИВНЫХ ПРИЛОЖЕНИЙ

Мартин Фаулер



www.williamspublishing.com

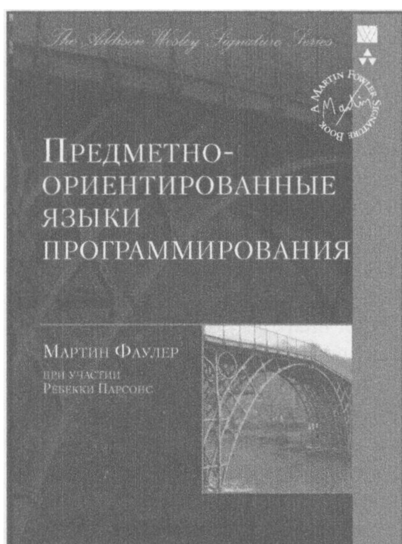
Книга дает ответы на трудные вопросы, с которыми приходится сталкиваться всем разработчикам корпоративных систем. Автор, известный специалист в области объектно-ориентированного программирования, заметил, что с развитием технологий базовые принципы проектирования и решения общих проблем остаются неизменными, 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.

ISBN 978-5-8459-1611-2

в продаже

ПРЕДМЕТНО-ОРИЕНТИРОВАННЫЕ ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Мартин Фаулер



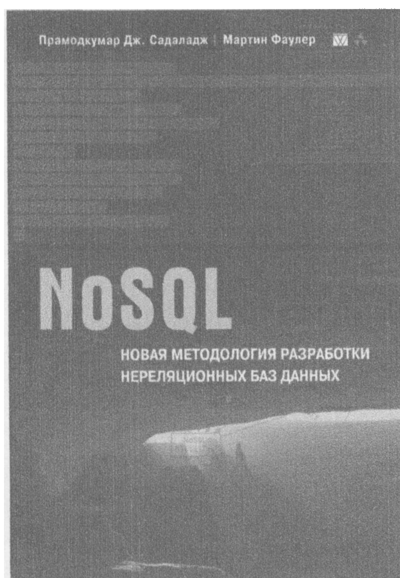
www.williamspublishing.com

В этой книге известный эксперт в области разработки программного обеспечения Мартин Фаулер предоставляет читателям всю информацию, необходимую для того, чтобы принять решение об использовании предметно-ориентированных языков в своих разработках и при положительном решении — эффективно применять методы создания таких языков. Каждая из методик сопровождается не только детальным пояснением ее работы, но и советами, когда именно ее стоит применять, а когда лучше прибегнуть к иным методам. Кроме того, здесь представлены примеры практического применения этих методик.

ISBN 978-5-8459-1738-6 **в продаже**

NoSQL: новая методология разработки нереляционных баз данных

Прамодкумар Дж. Садаладж
Мартин Фаулер



Необходимость обрабатывать все большие объемы данных является одним из факторов, стимулирующих появление альтернатив реляционным базам данных, использующим язык SQL. Одной из таких альтернатив является технология NoSQL.
В книге Фаулера рассматриваются основные концепции NoSQL, включая неструктурированные модели данных, агрегаты, новые модели распределения, теорему CAP и отображение-свертку, а также исследованы архитектурные и проектные вопросы, связанные с реализацией баз данных NoSQL.

Книга предназначена для разработчиков баз данных, корпоративных приложений, а также для студентов.

www.williamspublishing.com

ISBN 978-5-8459-1920-5 **в продаже**

DOMAIN-DRIVEN DESIGN ПРЕДМЕТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ

структуризация сложных программных систем

Эрик Эванс

Классическая книга Э. Эванса посвящена наиболее общим вопросам объектно-ориентированной разработки программного обеспечения: структуризации знаний о предметных областях, применению архитектурных шаблонов, построению и анализу моделей, проектированию программных объектов, организации крупномасштабных структур, выработке общего языка и стратегии коммуникации в группе.

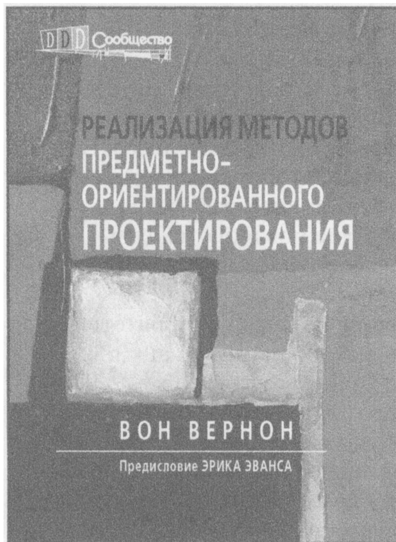
Книга предназначена для повышения квалификации программистов, в частности, по методикам экстремального программирования и agile-разработки. Может быть полезна студентам соответствующих специальностей.

www.williamspublishing.com

ISBN 978-5-8459-1597-9 **в продаже**

РЕАЛИЗАЦИЯ МЕТОДОВ ПРЕДМЕТНО-ОРИЕНТИРОВАННОГО ПРОЕКТИРОВАНИЯ

Вон Вернон



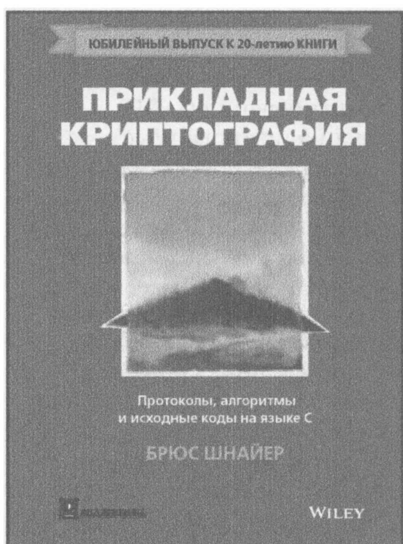
www.williamspublishing.com

В книге описаны реализация методов предметно-ориентированного проектирования (DDD) и специализированные подходы к реализации систем на основе современной архитектуры, а также показана важность ориентации на предметную область с учетом технических ограничений. Автор описывает методы DDD на реалистичных примерах, написанных на языке Java. Все примеры объединены в рамках единого сценария: разработки системы управления гибким проектированием SaaS для многоарендной среды на основе методологии Scrum. Книга представляет собой ценный источник знаний по методам предметно-ориентированного проектирования и предназначена для программистов всех уровней.

ISBN 978-5-8459-1881-9 **в продаже**

ПРИКЛАДНАЯ КРИПТОГРАФИЯ
ВТОРОЕ ИЗДАНИЕ
ПРОТОКОЛЫ, АЛГОРИТМЫ И ИСХОДНЫЕ КОДЫ НА ЯЗЫКЕ С

Брюс Шнайер



www.williamspublishing.com

Книга представляет собой юбилейный выпуск второго издания, подготовленный к 20-летнему юбилею всемирно признанного бестселлера. В ней приведен энциклопедический обзор криптографических алгоритмов и протоколов по состоянию на 1995 год. Многие годы эта книга остается самым авторитетным источником информации о криптографических протоколах и алгоритмах, которые легли в основу многих современных компьютерных технологий защиты данных. Книга предназначена для всех специалистов, использующих или разрабатывающих криптографические средства.

ISBN 978-5-9908462-4-1 **в продаже**

Объектные технологии/Инженерия ПО

По мере распространения объектных технологий – в особенности с использованием языка программирования Java – для сообщества разработчиков программного обеспечения становилась все более и более актуальной новая проблема. Недостаточно опытные разработчики создали множество плохо спроектированных программ, оказавшихся в результате малопроизводительными приложениями, которые трудно поддерживать и расширять. Все чаще разработчики программного обеспечения сталкиваются с тем, насколько сложно работать с такими доставшимися им в наследство приложениями. В течение нескольких лет объектные программисты-эксперты использовали постоянно растущую коллекцию методов повышения структурной целостности и эффективности таких существующих программ. Известные как "рефакторинг", эти практики так и оставались достоянием экспертов, потому что не предпринималось никаких попыток донести эти знания до широких масс разработчиков в доступной форме. В данной книге известный эксперт в области объектных технологий Мартин Фаулер открывает перед сообществом разработчиков новые горизонты, рассказывая о практиках, применяемых экспертами, и демонстрируя, какие значительные преимущества от их применения может получить любой разработчик.

При надлежащей подготовке квалифицированный проектировщик систем может взять плохо спроектированную программу и превратить ее в хорошо продуманный, надежный код. В книге Мартин Фаулер показывает читателям, где обычно можно найти возможности для оптимизации и как превратить плохой проект в хороший. Каждый шаг рефакторинга прост – даже, казалось бы, слишком прост, чтобы его стоило выполнять. Оптимизация может включать перемещение поля из одного класса в другой или извлечение некоторого кода из метода с тем, чтобы превратить его в отдельный метод, или даже перенос некоторого кода вверх или вниз по иерархии классов. Хотя эти отдельные шаги могут показаться элементарными, кумулятивный эффект таких небольших изменений может радикально улучшить проект программы. Рефакторинг кода – проверенный способ предотвращения распада программного обеспечения.

Помимо описания различных методов рефакторинга, автор приводит подробный каталог более чем с семьюдесятью рефакторингами и полезными указаниями, которые научат вас, когда их следует применять. Книга содержит подробное описание свыше 70 методов рефакторинга, причем не только теоретическое их описание, но и практические примеры на языке программирования Java. Следует учесть, что изложенные в книге идеи применимы к любому объектно-ориентированному языку программирования.

Об авторах

Мартин Фаулер – независимый консультант, который применяет объектные технологии для решения насущных проблем бизнеса около сорока лет. Он давал консультации по программным системам в таких областях, как здравоохранение, торговля и финансы. Среди его клиентов были Chrysler, Citibank, Министерство здравоохранения Великобритании, Andersen Consulting и Netscape Communications. Кроме того, Фаулер часто делает доклады по объектным технологиям, унифицированному языку моделирования UML и проектным шаблонам.

Кент Бек – известный программист, тестировщик ПО, специалист по рефакторингу, автор книг и игрок на банджо.

Джон Брант и **Дон Робертс** – авторы инструментария для автоматизированного выполнения рефакторингов Refactoring Browser for Smalltalk. Они также являются консультантами, около тридцати лет изучавшими практические и теоретические аспекты рефакторинга.

Диссертационная работа Уильяма Опдайка "Исследования по оптимизации объектно-ориентированных структур", выполненная в Университете Иллинойса, привела к первой важной публикации по этой теме. В настоящее время он работает в Lucent Technologies/Bell Laboratories.

ISBN 978-5-9909445-1-0

www.williamspublishing.com